



TALLER DE PROGRAMACIÓN
(TA045) CURSO DEIMONNAZ

Proyecto Aerolíneas Rústicas



9 de diciembre de 2024

Thiago Pacheco
111003

Matias Bartellone
110484

Ivan Maximoff
110868

Índice

Introducción	5
Node	5
1 Funcionalidades	5
1.1 Requests	5
1.2 Queries	6
1.3 Funcionalidad Interna	6
2 Implementación	7
2.1 Inicialización	7
2.1.1 Manejo de Clientes	7
2.2 Parsers	8
2.2.1 Startup Parser	8
2.2.2 Options Parser	8
2.2.3 Auth Response Parser	8
2.2.4 Query Parser	8
2.2.5 Prepare Parser	9
2.2.6 Execute Parser	9
2.3 Query Lexer	9
2.3.1 Standarize	9
2.3.2 Tokenize	10
2.4 Query Parsers	14
2.5 Query	15
2.5.1 Insert	16
2.5.2 Update	16
2.5.3 Delete	16
2.5.4 Select	16
2.5.5 Create Table	16
2.5.6 Create Keyspace	16
2.5.7 Drop Table	16
2.5.8 Drop Keyspace	16
2.5.9 Alter Table	17
2.5.10 Use	17
2.5.11 Where Clause	17
2.5.12 Order By Clause	17
2.5.13 Set Clause	17
2.5.14 If Clause	18
2.6 Executables	18
2.6.1 Startup Executable	18
2.6.2 Options Executable	18
2.6.3 Auth Response Executable	18
2.6.4 Query Executable	19
2.6.5 Prepare Executable	19
2.6.6 Execute Executable	19
2.7 Data	19
2.7.1 Meta Data	19
2.7.2 Data Access	21
2.7.3 Locks	21
2.8 Query Delegation	23
2.8.1 Query Delegator	23
2.8.2 Query Receiver	23
2.8.3 Query Serializer	24
2.9 Read Repair	24

2.9.1	Read Repair(estructura)	28
2.9.2	Response Manager	29
2.9.3	Row Comparer	29
2.9.4	Row Response	30
2.9.5	DataResponse	30
2.9.6	Repair Row	30
2.10	Gossip	31
2.10.1	Gossip Emitter	31
2.10.2	Gossip Listener	32
2.10.3	Seed Listener	32
2.11	Hinted Handoff	32
2.11.1	Hints Handler	32
2.11.2	Hints Receiver	33
2.11.3	Hints Sender	33
2.12	Encriptación	33
2.12.1	Conexiones Cliente-Servidor	33
2.12.2	Conexiones Servidor-Servidor	33
2.12.3	Hash de contraseñas	34
2.13	Utils	34
2.13.1	Frame	34
2.13.2	Bytes Cursor	34
2.13.3	Errors	35
2.13.4	Node Ip	36
2.13.5	Timestamp	37
2.13.6	Primary Key	38
2.13.7	Consistency	39
2.13.8	Constantes	40
2.13.9	Funciones	40
2.13.10	Response	40
3	Instrucciones de Uso	42
3.1	Configuración mediante archivo YAML	42
3.2	Configuración dinámica en terminal	42
3.3	IPs Certificadas para el Uso del Servidor	42
3.4	Ejecución del programa	43
3.4.1	Posibles errores y soluciones	44
Client		45
4	Client communication	45
4.1	Cassandra Connection	45
4.2	Cassandra Client	45
4.2.1	Características principales	45
4.2.2	Ejemplo de uso	46
4.3	Thread Pool Client	46
4.3.1	Principales Características	46
4.3.2	Metodo Execute	46
4.3.3	Ejemplo de Uso	47
4.4	Query Builder	47
4.4.1	Métodos	48
4.4.2	Ejemplo de uso	48
5	Instrucciones de uso	48
Modelacion de tablas		50
6	Keyspace	50

7	Tablas	50
7.1	Airports	50
7.2	Flights by airport	50
7.3	Relaciones entre Tablas	51
8	Ejemplos de Consultas	51
8.1	Insertar Aeropuertos	51
8.2	Insertar Datos en la Tabla flights.by_airport	51
8.3	Consultas de Selección	52
8.4	Descripción de las Consultas	52
	Flight App	53
9	Estructura	53
9.1	Paneles	53
9.1.1	Panel del Mapa	53
9.1.2	Panel de Información	53
9.2	Flights	53
9.2.1	Flight	54
9.3	Airports	54
9.3.1	Plugin de Airports	54
9.3.2	Dibujar Aeropuertos	55
9.4	Flight Selected	55
9.4.1	Estructuras Internas	55
10	Instrucciones de uso	56
	Flight Simulator	57
10.1	Insert Single Flight	57
10.2	Restart Flight	57
10.3	Update Flight	57
10.4	Flight Updates Loop	58
11	Instrucciones de uso	58
11.1	Agregar Vuelos para un Aeropuerto	58
11.2	Agregar un Solo Vuelo	58
11.3	Iniciar el Ciclo de Actualizaciones de Vuelos	59
11.4	Salir	59
	Test Client	60
12	Instrucciones de Uso	60
	Node Handler	61
13	Instrucciones de Uso	61

Introducción

Node

Esta sección describe el desarrollo e implementación de un nodo distribuido basado en el protocolo de Cassandra, un sistema de gestión de bases de datos diseñado para ofrecer alta disponibilidad, escalabilidad y consistencia eventual. El objetivo principal de este proyecto es implementar un nodo que emule el comportamiento de un sistema Cassandra real, permitiendo la gestión eficiente de datos distribuidos en un entorno controlado.

El nodo implementado aborda diversos aspectos clave de los sistemas distribuidos, como la sincronización entre nodos, la tolerancia a fallos, la replicación de datos y la ejecución de consultas. Además, se presta especial atención a los mecanismos de recuperación y a la consistencia de los datos, fundamentales para garantizar el correcto funcionamiento en un entorno de múltiples nodos que pueden experimentar caídas o desconexiones.

Este informe está estructurado para describir detalladamente las funcionalidades implementadas, así como la implementación técnica detrás de cada una, proporcionando así una visión integral de los componentes del sistema y los métodos empleados.

1. Funcionalidades

A continuación, se describen las funcionalidades implementadas en el nodo, organizadas en categorías principales:

1.1. Requests

El sistema soporta la mayoría de las solicitudes del protocolo de Cassandra, con excepción de Register, Event y Batch que devolverán error al no estar implementadas. Todas las requests se gestionan utilizando el [frame](#) protocolar de Cassandra.

Las requests implementadas son:

- **Startup**
- **Options**
- **Query**
- **Prepare**
- **Execute**
- **Auth Response**

A parte de las request que el usuario puede llegar a mandar, las que puede devolver el servidor son:

- **Error**
- **Authenticate**
- **Auth Challenge**
- **Auth Success**
- **Result (Void)**
- **Result (Schema Change)**
- **Result (Rows)**

1.2. Queries

El soporte para consultas (queries) es amplio, permitiendo trabajar con diversas operaciones y variaciones de las mismas. Las queries admitidas incluyen:

- **Insert**
- **Update**
- **Delete**
- **Select**
- **Use**
- **Create Table**
- **Create Keyspace**
- **Drop Table**
- **Drop Keyspace**
- **Alter Table**
- **Alter Keyspace**

Adicionalmente, el sistema soporta:

- **Cláusulas WHERE:** Todas las opciones posibles están implementadas.
- **Cláusula ORDER BY:** Permite ordenar los resultados según los criterios definidos.
- **Cláusula IF:** Compatible con verificaciones condicionales en las operaciones de modificación de datos.
- **Cláusula SET:** Todas las variaciones admitidas están soportadas.

1.3. Funcionalidad Interna

A nivel interno, el sistema cuenta con las siguientes implementaciones clave:

- **Protocolo Gossip:** Para la comunicación y sincronización entre nodos.
- **Hinted Handoff:** Gestión de datos pendientes en nodos caídos.
- **Delegación de Queries:** Reenvío de consultas a otros nodos según sea necesario.
- **Inicialización flexible:** Soporte para múltiples métodos de inicialización.
- **Read Repair:** Reparación de datos inconsistentes durante las lecturas.
- **Consistencia:** Implementación de niveles de consistencia ajustables.
- **Gestión de estados de nodo:** Monitoreo y soporte ante caídas de nodos.
- **Nodos semilla:** Implementación de nodos semilla para facilitar la inicialización y descubrimiento de nodos dentro del clúster.

2. Implementación

2.1. Inicialización

La inicialización del nodo consta de cinco etapas principales:

1. **Configuración inicial:** Se solicitan los datos necesarios al usuario, quien puede proporcionar la información de tres maneras diferentes (detalladas en la sección [Instrucciones de Uso](#)).
2. **Armado del nodo y configuración del clúster:** Una vez creado el nodo, se configura el clúster. Si no es el primer nodo, este debe conectarse a un nodo semilla para adquirir la información de los demás nodos en el clúster.
3. **Modo booting:** Si la dirección IP:puerto del nodo estaba marcada como inactiva en el clúster, el nodo entra en *modo booting*. En esta etapa, recibe las *hints* generadas durante su período de inactividad (detallado en la sección [Hinted Handoff](#)).
4. **Configuración de listeners:** Se establecen listeners en distintos *threads* para gestionar las siguientes funciones: acceso a datos, delegación de queries, protocolo Gossip y, si corresponde, el listener del [nodo semilla](#).
5. **Inicio de Gossip y listener principal:** Finalmente, se inicia el proceso de [Gossip](#), que se ejecuta cada segundo, seguido por la inicialización del listener principal. Este listener utiliza un *threadpool* para limitar la cantidad de clientes que pueden conectarse simultáneamente.

2.1.1. Manejo de Clientes

El ciclo de manejo de clientes funciona de la siguiente manera:

1. **Conexión del cliente:** Cuando se recibe una conexión de un cliente, se asigna un *thread* del *threadpool*. Este *thread* ejecuta un bucle que permanece activo para escuchar continuamente requests hasta que el cliente se desconecte.
2. **Procesamiento del request:**
 - Al recibir un [frame](#) de request, se lee su encabezado (*header*) para identificar el *parser* correspondiente según el opcode.
 - El cuerpo (*body*) del frame es procesado por el *parser*, lo que genera un objeto ejecutable.
 - Al ejecutar este objeto, se genera automáticamente un *frame* de respuesta listo para ser enviado al cliente.

Este diseño asegura que el nodo pueda manejar múltiples clientes de manera eficiente y procesar requests de forma modular y estructurada.

```
//Codigo del ciclo de procesamiento de la request:
fn execute_request(bytes: Vec<u8>) -> Result<Vec<u8>, Errors> {
    let frame = Frame::parse_frame(bytes.as_slice())?;
    let parser = ParserFactory::get_parser(frame.opcode)?;
    let mut executable = parser.parse(frame.body.as_slice())?;
    let frame = executable.execute(frame)?;
    Ok(frame.to_bytes())
}
```

2.2. Parsers

Los parsers son responsables de interpretar el cuerpo del request `frame`, asegurando su validez y creando el `Executable` correspondiente para ejecutar la solicitud. Todos los parsers implementan el trait `Parser`, que contiene una única función, `parse()`. Esto permite extender fácilmente la funcionalidad agregando nuevos parsers que cumplan con la interfaz definida.

El objetivo principal de los parsers es validar la estructura del body usando herramientas como `BytesCursor` y generar el `Executable` adecuado. Los parsers más complejos son aquellos que procesan queries, ya que incluyen la implementación de un lexer y parsers específicos para cada tipo de query.

A continuación, se presenta la función `get_parser`, que actúa como una factory para obtener el parser correspondiente según el opcode recibido:

```
pub fn get_parser(opcode: u8) -> Result<Box<dyn Parser>, Errors> {
    match opcode {
        STARTUP => Ok(Box::new(StartupParser)),
        OPTIONS => Ok(Box::new(OptionsParser)),
        QUERY => Ok(Box::new(QueryParser)),
        PREPARE => Ok(Box::new(PrepareParser)),
        EXECUTE => Ok(Box::new(ExecuteParser)),
        REGISTER => Ok(Box::new(RegisterParser)),
        EVENT => Ok(Box::new(EventParser)),
        BATCH => Ok(Box::new(BatchParser)),
        AUTH_RESPONSE => Ok(Box::new(AuthResponseParser)),
    }
}
```

A continuación, se describen los parsers implementados:

2.2.1. Startup Parser

Este parser valida que la configuración inicial proporcionada por el cliente sea correcta. Verifica, por ejemplo, que la versión utilizada sea **3.0.0** y que el compresor especificado sea compatible con nuestro nodo.

2.2.2. Options Parser

El `OptionsParser` es sencillo, ya que solo valida que el cuerpo del `frame` esté vacío, dado que no se requiere información adicional en este caso.

2.2.3. Auth Response Parser

Este parser valida las credenciales enviadas por el cliente. Comprueba que se haya utilizado el esquema de autenticación esperado (`PasswordAuthenticator`), en el formato `user:password`, y prepara los datos necesarios para generar el ejecutable correspondiente.

2.2.4. Query Parser

Este es el parser más complejo, ya que debe validar tanto la consistencia proporcionada como la query enviada. La query pasa primero por un `lexer` para su tokenización y, posteriormente, por un `query parser` específico para analizar su estructura y semántica.

A continuación se presenta una representación del código del parser, abstraído de las implementaciones específicas:


```
fn parse(&self, body: &[u8]) -> Result<Box<dyn Executable>, Errors> {  
    let mut cursor = BytesCursor::new(body);  
    let string = cursor.read_long_string()?;  
    let tokens = query_lexer(string)?;  
    let query = query_parser(tokens)?;  
    let consistency = cursor.read_short()?;  
    let executable = QueryExecutable::new(query, consistency);  
    Ok(Box::new(executable))  
}
```

2.2.5. Prepare Parser

El **PrepareParser** funciona de manera similar al **QueryParser**, utilizando el lexer y el query parser correspondiente. Sin embargo, su objetivo principal es procesar y guardar la query en un archivo para futuras ejecuciones.

2.2.6. Execute Parser

El **ExecuteParser** valida la consistencia indicada por el cliente y también verifica la existencia del identificador de la query a ejecutar. Este parser prepara los datos necesarios para el ejecutable de manera sencilla y eficiente.

2.3. Query Lexer

El lexer puede dividirse en dos etapas: la **estandarización** de la consulta y la **tokenización**.

En la primera etapa, se realiza un análisis sintáctico de la consulta para generar una versión limpia, correctamente espaciada, sin comentarios ni caracteres inválidos. Esta etapa tiene como objetivo garantizar que la consulta esté en un formato adecuado para su procesamiento posterior.

En la segunda etapa, con la consulta ya estandarizada, se procede a la clasificación de tokens. Esta clasificación puede realizarse a nivel de palabras individuales o grupos de palabras, tales como listas de nombres de columnas, condiciones, entre otros elementos estructurales de la consulta.

2.3.1. Standarize

El lexer debe considerar todas las formas posibles en que una consulta puede ser escrita. Esto incluye la presencia de comentarios dentro de la consulta, ya sea como líneas completas, partes de líneas comentadas de diversas maneras o bloques de comentarios. Además, debe manejar casos de cantidad indeterminada de espacios entre palabras, e incluso situaciones en las que ciertas palabras no estén separadas por espacios. Por ejemplo, en expresiones como `columna1=titulo1`, el lexer debe ser capaz de reconocer y separar correctamente los elementos involucrados, como operandos, nombres de columnas y valores.

Asimismo, el lexer debe gestionar aperturas y cierres de paréntesis, llaves y corchetes, así como la presencia de comas que puedan estar adyacentes a otros elementos, sin espacios. Otro aspecto importante es el manejo de comillas simples, dobles y símbolos de dólar (\$), ya que, en el contexto del lenguaje de Cassandra, es fundamental preservar el formato de los valores encerrados por estos delimitadores. Esto significa que todo contenido dentro de estas delimitaciones no debe ser modificado, incluyendo espacios, caracteres especiales, tabulaciones y saltos de línea.

Finalmente, estas secciones delimitadas se tratarán de manera distinta durante el proceso de tokenización, respetando las reglas previamente mencionadas.

Entrada al estandarizador:

```
-- Selecciona informacin de vuelos desde la tabla 'flights'
SELECT
    flightCode, -- Cdigo del vuelo
    departureAirport, -- Aeropuerto de salida
    arrivalAirport, -- Aeropuerto de llegada
    departureTime, -- Hora de salida
    arrivalTime, -- Hora de llegada
    altitude -- Altitud en la que vuela
FROM aviation.flightInfo -- Desde la tabla de informacin de vuelos
WHERE departureAirport = 'CAI' -- Filtrar por vuelos que salen de CAI
    AND arrivalAirport = 'EZE' -- Y que llegan a EZE
    AND departureTime >= '2023-10-01T00:00:00' -- Salidas a partir de una fecha
ORDER BY departureTime ASC; -- Ordenar por hora de salida ascendente
```

Salida del estandarizador:

```
[
    "SELECT",
    "flightCode",
    "departureAirport",
    "arrivalAirport",
    "departureTime",
    "arrivalTime",
    "altitude",
    "FROM", "aviation.flightInfo",
    "WHERE", "departureAirport", "=", "'CAI'",
    "AND", "arrivalAirport", "=", "'EZE'",
    "AND", "departureTime", ">=", "'2023-10-01T00:00:00'",
    "ORDER", "BY", "departureTime", "ASC"
]
```

2.3.2. Tokenize

El proceso de tokenización clasifica cada palabra o símbolo dentro de una consulta CQL en diferentes tipos de tokens. Estos tokens están organizados en una estructura jerárquica, que permite representarlos de manera eficiente para su posterior análisis sintáctico. A continuación se presentan los diferentes tipos de tokens utilizados en este proceso:

```
pub enum Token {
    Identifier(String),
    Term(Term),
    Reserved(String),
    DataType(DataType),
    ParenList(Vec<Token>),
    IterateToken(Vec<Token>),
    BraceList(Vec<Token>),
    Symbol(String),
}
```

Descripción de los tipos de tokens

- **Identifier:** Este tipo de token representa identificadores de objetos o entidades en la consulta, tales como el nombre de columnas, tablas, keyspaces, etc. Los identificadores pueden estar entre comillas dobles, lo que preserva su formato exacto, incluyendo mayúsculas, espacios y caracteres especiales que de otro modo no serían tolerados. Por ejemplo, "flightCode" es un identificador que hace referencia a una columna, y si se escribe entre comillas dobles, se mantiene su formato exacto.
- **Reserved:** Son las palabras reservadas del lenguaje CQL que no pueden ser utilizadas como identificadores. Estas palabras son insensibles a las mayúsculas (case-insensitive). Entre ellas se incluyen términos como **SELECT**, **WITH**, **ORDER**, **KEY**, **PRIMARY**, entre otras. La tokenización detecta estas palabras y las clasifica como reservadas si corresponden al conjunto predefinido de palabras clave.
- **Term:** Los términos representan componentes utilizados dentro de operaciones dentro de la consulta. Un **Term** puede ser un literal, un operador aritmético o un operador booleano. Los detalles de cada uno de estos tipos se describen a continuación.

```
pub enum Term {  
    Literal(Literal),  
    ArithMath(ArithMath),  
    BooleanOperations(BooleanOperations),  
}
```

- **Literal:** Los literales representan valores específicos que se utilizan en las consultas. Cada literal está asociado con un tipo de dato, como **Int**, **Text**, **Date**, entre otros. El formato y la forma de escribir los literales dependen del tipo de dato que representen. Por ejemplo, un literal de tipo cadena de texto podría escribirse entre comillas simples, como 'CAI'.
- **BooleanOperations:** Los operadores booleanos se utilizan para realizar operaciones lógicas y de comparación dentro de una consulta. Estos operadores se dividen en dos categorías: los operadores lógicos y los operadores de comparación. Ambos tipos se describen a continuación.

```
pub enum BooleanOperations {  
    Logical(LogicalOperators),  
    Comparison(ComparisonOperators),  
}
```

- **LogicalOperators:** Incluye operadores como **OR**, **AND** y **NOT**. Estos operadores no son sensibles a las mayúsculas y minúsculas, por lo que **AnD** se interpretará como válido.

- **ComparisonOperators:** Incluye operadores de comparación como **<**, **=**, **!=**, **>**, **>=** y **<=**. Estos operadores permiten comparar valores dentro de una consulta..

- **ArithMath:** Los operadores aritméticos son utilizados para realizar operaciones matemáticas básicas dentro de la consulta. Estos operadores son fácilmente identificables y se reconocen mediante símbolos específicos. Los operadores disponibles son:

```
pub enum ArithMath {  
    Suma,  
    Sub,  
    Division,  
    Rest,  
    Multiplication,  
}
```

- **DataType:** Los tipos de datos soportados por la implementación incluyen `Int`, `Boolean`, `Date`, `Text`, entre otros. Estos tipos de datos se identifican por palabras clave que no distinguen entre mayúsculas y minúsculas. Por ejemplo, `Text` será reconocido correctamente como `Text`.

```
pub enum DataType {  
    Int,  
    Boolean,  
    Date,  
    Decimal,  
    Text,  
    Duration,  
    Time,  
}
```

- **Sublistas:** Las sublistas son una estructura que agrupa elementos relacionados dentro de la consulta para facilitar su análisis. Existen tres tipos principales de sublistas:
 - **ParenList:** Agrupa elementos que se encuentran dentro de paréntesis. Por ejemplo, los valores de las condiciones en una cláusula `WHERE`.
 - **BraceList:** Agrupa elementos que se encuentran dentro de llaves. Esto es útil, por ejemplo, para el análisis de las definiciones de tablas o keyspaces.
 - **IterateToken:** Agrupa secciones de longitud variable que siguen un patrón específico. Esto puede incluir las columnas en un `SELECT`, las condiciones en un `WHERE`, los valores en un `INSERT`, etc. La lógica de este tipo de token es que todo lo que se encuentra entre palabras reservadas consecutivas forma parte de una sublista. Por ejemplo, en una consulta como `SELECT iterateToken FROM table WHERE iterateToken ORDER BY iterateToken`, cada `iterateToken` agrupará diferentes elementos de la consulta.
- **Symbol:** Los símbolos son caracteres específicos que se utilizan en la consulta, como comas, puntos y coma, dos puntos, entre otros. Estos caracteres son esenciales para separar diferentes elementos en una consulta CQL.

Funcionamiento del Tokenizer

El proceso de tokenización es fundamental, ya que convierte una consulta CQL en una secuencia de tokens que pueden ser procesados por un parser para validar y analizar su estructura. Este proceso clasifica cada palabra o conjunto de palabras en un tipo de token específico, siguiendo reglas predefinidas que los parsers utilizarán para interpretar correctamente la consulta. Si un literal es ingresado incorrectamente, como por ejemplo, escribir `"5"` cuando se esperaba un valor numérico, el sistema lo identificará como un `Identifier` en lugar de un `Term::Literal`. Esto permite detectar errores de forma precisa y proporciona retroalimentación clara sobre problemas de sintaxis, indicando exactamente la ubicación del error y su naturaleza. Este enfoque facilita la depuración y mejora la experiencia del usuario al interactuar con la base de datos.

Ejemplo de Tokenización

Entrada:

```
[
  "SELECT",
  "flightCode",
  "departureAirport",
  "arrivalAirport",
  "departureTime",
  "arrivalTime",
  "altitude",
  "FROM",
  "aviation.flightInfo",
  "WHERE",
  "departureAirport", "=", "'CAI'",
  "AND", "arrivalAirport", "=", "'EZE'",
  "AND", "departureTime", ">=", "'2023-10-01T00:00:00'",
  "ORDER", "BY", "departureTime", "ASC"
]
```

Salida:

```
[
  Reserved(SELECT),
  IterateToken[
    Identifier(flightCode),
    Identifier(departureAirport),
    Identifier(arrivalAirport),
    Identifier(departureTime),
    Identifier(arrivalTime),
    Identifier(altitude),
  ],
  Reserved(FROM),
  Identifier(aviation.flightInfo),
  Reserved(WHERE),
  IterateToken[
    Identifier(departureAirport),
    ComparisonOperators(=),
    Literal('CAI'),
    Reserved(AND),
    Identifier(arrivalAirport),
    ComparisonOperators(=),
    Literal('EZE'),
    Reserved(AND),
    Identifier(departureTime),
    ComparisonOperators(>=),
    Literal('2023-10-01T00:00:00'),
  ],
  Reserved(ORDER),
  IterateToken[
    Reserved(BY),
    Identifier(departureTime),
    Reserved(ASC)
  ]
]
```

Este ejemplo muestra cómo la consulta inicial es transformada en una estructura organizada de tokens, simplificando su análisis y posterior procesamiento.

2.4. Query Parsers

Los **Query Parsers** son parsers específicos para cada tipo de query. Reciben una query previamente tokenizada por el lexer, que se encuentra en formato de un vector de tokens. El proceso de parseo se implementa utilizando un *recursive descent parser*, una técnica que resultó ser muy útil para este caso. Este enfoque emplea un iterador sobre los tokens, avanzando uno a uno y verificando que se cumplan las reglas correspondientes a cada tipo de query. Cada regla tiene una función propia, y recursivamente se invoca a la siguiente regla mientras no se detecten errores. El resultado final es la construcción de la **Query**, que será utilizada posteriormente para su ejecución.

Previo al llamado de cada parser específico, se selecciona que parser utilizar dependiendo de la palabra reservada inicial de cada tira de tokens. Ya sea **UPDATE**, **DELETE**, **SELECT**, etc. A partir de ahí se sabe que parser ejecutar, y si no es ninguno de los casos es porque la query no es admitida.

Por ejemplo, en el caso de la query **INSERT**, primero con el primer token siendo la palabra reservada **INSERT** se llama al **Insert Query Parser**. Luego, este llama a una función encargada de comprobar que el siguiente token sea **INTO**, luego que venga el nombre de la tabla y así. Este proceso continúa avanzando, validando los tokens que deben seguir de acuerdo con las reglas estrictas definidas para la sintaxis de la query.

A continuación, se presenta la lista de los parsers implementados:

- **Insert Query Parser**
- **Update Query Parser**
- **Delete Query Parser**
- **Select Query Parser**
- **Where Clause Parser**
- **Order By Clause Parser**
- **Set Clause Parser**
- **If Clause Parser**
- **Create Table Query Parser**
- **Create Keyspace Query Parser**
- **Drop Table Query Parser**
- **Drop Keyspace Query Parser**
- **Alter Table Query Parser**
- **Use Query Parser**

Este es un código ejemplo del *Use Query Parser* que es el más corto y sencillo de todos:

```
impl UseQueryParser {
    pub fn parse(&self, tokens: Vec<Token>) -> Result<UseQuery, Errors> {
        let mut use_query = UseQuery::new();
        self.keyspace_name(&mut tokens.into_iter().peekable(), &mut use_query)?;
        Ok(use_query)
    }

    fn keyspace_name(
        &self,
        tokens: &mut Peekable<IntoIter<Token>>,
        query: &mut UseQuery,
    ) -> Result<(), Errors> {
        match get_next_value(tokens)? {
            Token::Identifier(identifier) => {
                query.keyspace_name = identifier;
                Ok(())
            }
            _ => Err(Errors::SyntaxError(String::from(UNEXPECTED_TOKEN))),
        }
    }
}
```

2.5. Query

Para la ejecución de queries, utilizamos un enfoque basado en el trait **Query**. Esta abstracción facilita la extensión del sistema para agregar nuevas queries, ya que cualquier nueva implementación debe cumplir con las funciones definidas por este trait. Para cada tipo de query, definimos una estructura específica que implementa los métodos correspondientes. Estos métodos son:

- **run()**: Ejecuta la query y devuelve un **RESULT** que puede ser de tipo **VOID**, **SCHEMA CHANGE** o **ROWS**.
- **get partition()**: Devuelve los valores de la *partition key* de la consulta, o **None** si se trata de una query general. Este método es útil para determinar a qué nodos delegar la query.
- **set table()**: Define la tabla asociada a la query en formato **keyspace.table**, lo cual puede depender de si se ha ejecutado previamente una **Use Query**.

Esta implementación resulta particularmente útil ya que permite abstraernos y aprovechar el polimorfismo. Los parsers devolverán una implementación concreta de **Query**, y podemos ejecutar la query simplemente llamando a **query.run()**, sin necesidad de conocer el tipo exacto de la query. Además, la implementación es extensible: para agregar una nueva query, basta con crear una estructura que implemente el trait correspondiente.

Este enfoque también facilita el manejo de queries en archivos y la delegación de tareas, ya que permite serializar las queries y guardarlas, o enviarlas a través de una conexión TCP.

La ejecución de las queries es sencilla, ya que consiste en utilizar los datos proporcionados por el parser (atributos previamente definidos en el parser) para determinar la acción a tomar. Es un proceso en el que se llaman funciones del módulo de [Meta Data](#) y del módulo de [Data Access](#) para cumplir con el propósito de la query.

A continuación, se describen las **Queries** implementadas como también las estructuras **where**, **order by**, **set** e **if**:

2.5.1. Insert

El **Insert** verifica los datos a insertar y determina cuáles son los valores asociados a la clave primaria. Además, realiza diversas validaciones relacionadas con la [Meta Data](#) de la tabla, como asegurarse de que no se hayan ingresado columnas no definidas. Un **Insert** puede no contener todas las columnas definidas, pero debe incluir la clave primaria completa.

Para su ejecución, se construye una [Row](#) con los valores proporcionados y se hace uso de [Data Access](#) para añadirla al final de la tabla.

2.5.2. Update

El **Update** valida que los datos dentro de la **SET** clause sean correctos y que no se intente modificar la clave primaria de ninguna [Row](#). También busca los datos de la clave de partición dentro de la **WHERE** clause.

Para su ejecución, se emplea [Data Access](#), que recorre toda la tabla y actualiza los datos que cumplan con la **WHERE** clause. Las nuevas [Rows](#) se construyen dinámicamente basadas en la **SET** clause.

2.5.3. Delete

El **Delete** es uno de los más simples, ya que solo busca la clave de partición de la consulta. La ejecución es similar a la de **Update**, pero en este caso, marca las [Rows](#) que cumplen con la **WHERE** clause como **deleted**. Para la eliminación permanente de datos, decidimos que, al insertar un segundo delete, se borre la línea por completo del archivo. Así no se van acumulando líneas eliminadas y nos es muy útil para la implementación del [Read Repair](#).

2.5.4. Select

El **Select**, al igual que el **Update** y el **Delete**, busca la clave de partición en los valores de la **WHERE** clause. Hace uso de [Data Access](#) con una función similar, que filtra las filas de la tabla, las ordena y devuelve las columnas solicitadas por el cliente de las filas encontradas.

2.5.5. Create Table

El **Create Table** hace uso tanto del [Data Access](#) como del [Meta Data](#) para crear el archivo y actualizar los datos del keyspace. Como no tiene partition key devuelve None para que se ejecute en todos los nodos.

2.5.6. Create Keyspace

El **Create Keyspace** es similar al **Create Table**, pero su propósito es solo actualizar la [meta data](#) para reflejar la creación del keyspace.

2.5.7. Drop Table

El **Drop Table** es similar al **Create Table**, pero en lugar de crear una tabla, la elimina y actualiza la [meta data](#) para reflejar la eliminación de la tabla.

2.5.8. Drop Keyspace

El **Drop Keyspace** es similar al **Create Keyspace**, pero en este caso elimina el keyspace y todas las tablas asociadas a él, actualizando la [meta data](#) en consecuencia.

2.5.9. Alter Table

El **Alter Table** actualiza la **meta data** de la tabla, permitiendo modificaciones en la configuración de la tabla, como cambios en las **columnas** definidas.

2.5.10. Use

El **Use** es sencillo, ya que solo agrega el keyspace especificado a los datos del **cliente** en la **meta data**, lo que permite que no sea necesario especificarlo en cada consulta.

2.5.11. Where Clause

El **Where Clause** es una estructura recursiva diseñada para representar toda la condición lógica de una consulta. Este enfoque permite modelar diversas combinaciones de lógicas booleanas que deben cumplirse en una determinada **Row**.

```
pub enum WhereClause {
    Comparison(ComparisonExpr),
    Tuple(Vec<ComparisonExpr>),
    And(Box<WhereClause>, Box<WhereClause>),
    Or(Box<WhereClause>, Box<WhereClause>),
    Not(Box<WhereClause>),
}

pub struct ComparisonExpr {
    column_name: String,
    operator: ComparisonOperators,
    literal: Literal,
}
```

El principal beneficio de esta estructura radica en su capacidad para evaluar condiciones. Posee una función **evaluate()** que, para una **Row** específica, recorre recursivamente su propia estructura, verificando que se cumplan las comparaciones definidas. Esto permite, una vez construida la cláusula, iterar fácilmente sobre una tabla y evaluar cada línea en función de las condiciones establecidas.

2.5.12. Order By Clause

El **Order By Clause** es una estructura sencilla que contiene el nombre de una **Columna** a ordenar y el método de ordenamiento, que puede ser ascendente o descendente. En caso de que se requiera ordenar por varias columnas de referencia, se construye un vector de cláusulas que se procesa desde la menos significativa hasta la más significativa, utilizando un enfoque similar al *radix sort*.

2.5.13. Set Clause

El **Set Clause** se representa mediante un mapa que asocia el nombre de una **Columna** con un **AssignmentValue**. Este enum define los cambios que pueden aplicarse a una **Row**. Los valores posibles son:

- **Simple:** Representa una asignación directa de un valor literal a una columna (**columna = nuevo_valor**).
- **Column:** Representa una asignación basada en el valor de otra columna (**columna = otra_columna**).
- **Arithmetic:** Representa operaciones aritméticas que modifican el valor de una columna con base en su valor actual (**columna = columna operador valor**). Por ejemplo, **SET cantidad = cantidad + 2**.

```
pub enum AssignmentValue {  
    Simple(Literal),  
    Column(String),  
    Arithmetic(String, ArithMath, Literal),  
}
```

2.5.14. If Clause

El **If Clause** comparte similitudes con el **Where Clause**, ya que también utiliza una estructura recursiva y una función `evaluate()` que permite verificar condiciones.

Los valores posibles de un **IfClause** son:

- **Exist:** Verifica si existe una fila. - **Comparison:** Evalúa una comparación simple basada en una **ComparisonExpr**. - **Expresiones Booleanas:** Evalúa expresiones AND, OR y NOT de **IfClauses**

```
pub enum IfClause {  
    Exist,  
    Comparison(ComparisonExpr),  
    And(Box<IfClause>, Box<IfClause>),  
    Or(Box<IfClause>, Box<IfClause>),  
    Not(Box<IfClause>),  
}
```

2.6. Executables

Los **Executables** son estructuras asociadas a cada request que implementan el trait **Executable**. De esta manera, el trait **Parser** devuelve una implementación de **Executable**, manteniendo todo el sistema abstracto y extensible. Cada **Executable** debe cumplir con la función `execute()`, la cual utiliza los datos previamente parseados y transformados para ejecutar la request específica. Al finalizar, devuelve el frame de respuesta listo para ser entregado al cliente.

A continuación, se describen los **Executables** implementados:

2.6.1. Startup Executable

Este **Executable** es muy simple, ya que solo arma el frame **AUTHENTICATE**, cuyo cuerpo incluye el autenticador que utiliza el nodo (en nuestro caso, **PasswordAuthenticator**).

2.6.2. Options Executable

El **OptionsExecutable** devuelve el frame **SUPPORTED**, que tiene como cuerpo un string map con la versión de CQL y el compresor admitido.

2.6.3. Auth Response Executable

Este **Executable** valida las credenciales proporcionadas por el cliente. Verifica si está autorizado, comprobando que el usuario y la contraseña estén dentro del archivo de credenciales autorizadas. Dependiendo del resultado, puede devolver un **AUTH_SUCCESS** o un **AUTH_CHALLENGE**.

2.6.4. Query Executable

El **QueryExecutable** es el más complejo, ya que no solo ejecuta la **Query**, sino que también hace uso del **QueryDelegator**. Este determina a qué nodos debe enviarse la query, dependiendo de la consistencia solicitada, y las envía a los nodos correspondientes para su ejecución. Si es exitosa, devuelve un frame **RESULT** con los datos obtenidos de la query.

A continuación, se presenta una representación del código del **Executable**, abstraído de las implementaciones específicas:

```
fn execute(&mut self, request: Frame) -> Result<Frame, Errors> {
    self.query.set_table()?;
    let pk = self.query.get_partition()?;
    let delegator = QueryDelegator::new(
        pk,
        query,
        ConsistencyLevel::from_i16(self.consistency_integer)?,
    );
    let response_msg = delegator.send()?;
    let response_frame = FrameBuilder::build_response_frame(request, RESULT, response_msg)?;
    Ok(response_frame)
}
```

2.6.5. Prepare Executable

El **PrepareExecutable** recibe la **Query** previamente parseada y la guarda en un archivo interno, asignándole un ID único. Posteriormente, devuelve un frame **RESULT** con el ID asignado, de modo que el cliente pueda utilizarlo en el futuro.

2.6.6. Execute Executable

El **ExecuteExecutable** busca en el archivo donde se guardan las queries si existe una con el ID proporcionado por el cliente. Si la encuentra, la elimina del archivo y la ejecuta con la consistencia indicada. La ejecución sigue el mismo procedimiento que el **QueryExecutable**, y también devuelve un frame **RESULT**.

2.7. Data

La información que almacenamos se divide en dos secciones principales. Por un lado, tenemos la **meta data**, que guarda información relevante sobre los clientes, los nodos del clúster, y los keyspaces y tablas definidas. Por otro lado, tenemos la **data en sí**, que se gestiona mediante un archivo separado por cada tabla.

En ambas secciones decidimos utilizar archivos **.json** debido a su utilidad para la serialización y deserialización de estructuras. A continuación, se describe cómo funciona cada parte.

2.7.1. Meta Data

La **meta data** se organiza en tres áreas principales, cada una con su propia estructura para almacenar y recuperar información. Además, estas estructuras incluyen funcionalidades adicionales, como obtener la lista de **IPs** de una partición.

A continuación, se detallan cada una de estas áreas:

2.7.1.1 Client

La información de los clientes se modela mediante una estructura **Client**, donde se almacena si el cliente ha realizado el **STARTUP**, si está autenticado y el keyspace actualmente en uso (definido mediante el comando **USE**).

Para cada cliente activo, se utiliza un archivo cuyo nombre corresponde al thread actual. Esto permite identificar fácilmente al cliente asociado con cada instancia del programa. La estructura `Client` es la siguiente:

```
pub struct Client {
    id: String,
    startup: bool,
    authorized: bool,
    keyspace: Option<String>,
}
```

2.7.1.2 Node

La información de los nodos se modela con la estructura `Cluster`, que contiene datos del nodo propio y una lista con los demás nodos del clúster. Toda esta información se almacena en un único archivo y se utiliza también para la transmisión mediante [Gossip](#).

Cada nodo incluye información como su [IP](#), [Timestamp](#), y otros datos útiles relacionados con su estado. A continuación, se presentan las estructuras `Cluster` y `Node`:

```
pub struct Cluster {
    own_node: Node,
    other_nodes: Vec<Node>,
}

pub struct Node {
    pub ip: NodeIp,
    pub position: usize,
    pub is_seed: bool,
    pub state: State,
    pub timestamp: Timestamp,
}
```

2.7.1.3 Keyspace

Los keyspaces se representan mediante la estructura `Keyspace`, la cual define el **replication factor**, la estrategia de replicación, y una lista de tablas asociadas. Cada tabla incluye información sobre sus columnas y su [Primary Key](#).

Toda esta información se almacena en un único archivo que contiene una lista de keyspaces. Las estructuras `Keyspace` y `Table` son las siguientes:

```
pub struct Keyspace {
    pub tables: HashMap<String, Table>,
    pub replication_strategy: String,
    pub replication_factor: usize,
}

pub struct Table {
    pub primary_key: PrimaryKey,
    pub columns: HashMap<String, DataType>,
}
```

2.7.2. Data Access

El módulo **Data Access** define los tipos de datos almacenados en la base de datos y proporciona diversas funcionalidades utilizadas por las distintas queries. Para cada tabla, se crea un archivo único con el formato `keyspace.tabla`, garantizando así la unicidad de su identificación.

Este módulo incluye funciones tanto para la gestión de archivos como para la manipulación de datos. Entre sus principales capacidades se encuentran:

Gestión de archivos: creación, eliminación e inserción de datos.

Procesamiento basado en condiciones: operaciones como `Update`, `Delete` y `Select`, que recorren la tabla completa y actúan según un `Where Clause`. Estas funciones procesan las distintas `Rows` y construyen un nuevo archivo en un archivo temporal. Una vez finalizada la escritura, el archivo temporal reemplaza al archivo original.

A continuación, se detallan las principales estructuras utilizadas en el módulo:

2.7.2.1 Row

La **Row** es la unidad fundamental que se almacena serializada en el archivo JSON. Está compuesta por una lista de columnas, una `Primary Key`, e información sobre si ha sido eliminada. Además, incluye funciones para comparar `Rows` y obtener columnas mediante getters. Su estructura es la siguiente:

```
pub struct Row {  
    pub columns: Vec<Column>,  
    pub primary_key: Vec<String>,  
    deleted: bool,  
}
```

2.7.2.2 Column

La **Column** representa un dato dentro de una `Row` y está definida por tres elementos: el nombre de la columna, su valor (un `Literal`), y un `Timestamp` que indica la última modificación. Esta estructura permite rastrear los cambios en cada columna. A continuación, se muestra su definición:

```
pub struct Column {  
    pub column_name: String,  
    pub value: Literal,  
    pub timestamp: Timestamp,  
}
```

2.7.3. Locks

El uso de concurrencia en nuestro sistema plantea el problema de acceso simultáneo a los datos, lo que podría generar errores. Para abordar este desafío, implementamos un mecanismo de *locks* para controlar el acceso a los archivos. Diseñamos dos tipos de locks: uno para *Data Access* y otro para *Meta Data*, denominados respectivamente *Data Access Handler* y *Meta Data Handler*.

2.7.3.1. Implementación de Locks La solución se basa en un `TcpListener` que se inicializa al levantar el nodo. Este `TcpListener` escucha conexiones TCP de las distintas instancias del programa que intentan acceder a los datos. Este enfoque garantiza que solo un cliente pueda estar conectado a la vez, lo que equivale a permitir acceso exclusivo a un único hilo por vez. Una vez establecida la conexión, se obtiene una instancia de las estructuras `DataAccess` o `MetaData`, que se pueden usar mientras la conexión esté activa. Al salir del *scope*, el lock se libera automáticamente.

2.7.3.2. Motivación Optamos por esta metodología debido a las siguientes razones:

- **Compatibilidad con el polimorfismo:** Muchas funcionalidades están diseñadas para cumplir con *traits*. Si hubiéramos usado un lock convencional o un canal `rx/tx`, habríamos necesitado pasarlos como parámetros a múltiples funciones, lo cual sería desprolijo y poco manejable, especialmente si algunas funciones no los utilizan.
- **Evitar dependencias externas:** Diseñamos esta implementación sin la necesidad de importar *crates* externos, manteniendo la simplicidad y el control sobre el sistema.

2.7.3.3. Ejemplo de Código A continuación, se presenta un ejemplo de cómo funciona el acceso al lock de `Data Access` y su utilización en una *query* de tipo `Insert`:

```
pub fn use_data_access<F, T>(action: F) -> Result<T, Errors>
where
    F: FnOnce(&DataAccess) -> Result<T, Errors>,
{
    let mut meta_data_stream = DataAccessHandler::establish_connection()?;
    let data_access = DataAccessHandler::get_instance(&mut meta_data_stream)?;
    action(&data_access)
}

let applied = use_data_access(|data_access| {
    data_access.update_row(
        &self.table_name,
        &self.changes,
        where_clause,
        &self.if_clause,
    )
})?;
```

En este ejemplo:

- La función `use_data_access` establece la conexión con el `Data Access Handler` y obtiene una instancia de `DataAccess`.
- Posteriormente, una *query* de tipo `Insert` utiliza este lock para insertar filas en la tabla correspondiente.

Este enfoque asegura que las operaciones sobre los datos sean seguras y eficientes, previniendo conflictos entre hilos y manteniendo la integridad de la base de datos.

2.8. Query Delegation

Cuando un cliente se conecta a un nodo y envía una solicitud de *query*, dependiendo de la partición de esa *query*, debe delegarse a otros nodos para que la ejecuten. Para ello, diseñamos un sistema de transmisor y receptor, donde el transmisor es el nodo que está conectado al cliente y envía la *query* serializada a otros nodos.

El ciclo general de funcionamiento es el siguiente: Primero, se realiza todo el proceso de análisis (parseo) de la *query*. Al obtener la *query*, se definen los datos de la partición y se solicitan las direcciones IP de los nodos que pertenecen a esa partición. Luego, la *query* serializada se envía a todos los nodos a los que debe delegarse. Estos nodos reciben la *query* y la ejecutan, devolviendo la respuesta serializada. Finalmente, el nodo delegador debe esperar a que lleguen *n* respuestas exitosas, según la consistencia requerida.

A continuación, se detalla el funcionamiento de cada bloque:

2.8.1. Query Delegator

El *Query Delegator* es el encargado de gestionar todo lo relacionado con la delegación. Para inicializarlo, necesita la *Query*, la *Consistencia* y la *Primary Key*.

La delegación consiste en crear un hilo por cada dirección IP a la que se debe enviar la *query*. En cada hilo, se envía la *query* serializada a su *Query Receiver* correspondiente a través de una conexión TCP. Al recibir una respuesta, esta se envía a un contador gestionado por un canal *rx/tx*, donde se esperan respuestas exitosas hasta alcanzar el nivel de consistencia deseado. Si transcurre un tiempo determinado sin recibir las respuestas requeridas, se devuelve el primer error obtenido o un error de *timeout* si no se obtuvo respuesta alguna.

Una vez que se ha obtenido la lista de respuestas de todos los nodos, estas se analizan. Si las respuestas son de tipo *Select*, se debe verificar si son diferentes, ya que en tal caso podría ser necesario ejecutar un *Read Repair*.

A continuación se muestra la parte del código en la que se esperan por las respuestas:

```
/// get responses until n = consistency
let timeout = Duration::from_secs(TIMEOUT_SECS);
for _ in 0..self.consistency.get_consistency(self.get_replication())? {
    match rx.recv_timeout(timeout) {
        Ok(response) => {
            let mut res = responses.lock().unwrap();
            res.push(response);
        }
        _ => {
            return match error.lock().unwrap().take() {
                Some(e) => Err(e),
                None => Err(Errors::ReadTimeout(String::from("Timeout"))),
            }
        }
    }
}
```

2.8.2. Query Receiver

El *Query Receiver* es un *TcpListener* que permanece constantemente escuchando una *query* serializada. Su funcionamiento es sencillo: tras recibir una *query*, ejecuta *query.run()*. Al actuar como un pasamanos de datos, puede ejecutar la *query* sin problemas, incluso si fue creada en otro nodo. Finalmente, envía la respuesta de la *query* al nodo que la envió originalmente.

```
impl QueryReceiver {
    pub fn start_listening(ip: NodeIp) -> Result<(), Errors> {
        let listener = bind_listener(ip.get_query_delegation_socket());
        for incoming in listener.incoming() {
            match incoming {
                Ok(mut stream) => {
                    thread::spawn(move || -> Result<(), Errors> {
                        match handle_query(&mut stream) {
                            Ok(response) => write_to_stream(&mut stream, response.as_slice())?,
                            Err(e) => write_to_stream(&mut stream, e.serialize().as_slice())?,
                        }
                        Ok(())
                    });
                }
                Err(_) => return Err(Errors::ServerError(String::from("Error in connection"))),
            }
        }
        Ok(())
    }
}

fn handle_query(stream: &mut TcpStream) -> Result<Vec<u8>, Errors> {
    let res = read_exact_from_stream(stream)?;
    let query = QuerySerializer::deserialize(res.as_slice())?;
    query.run()
}
```

2.8.3. Query Serializer

Para transmitir *queries* entre nodos es necesario poder serializarlas y deserializarlas. Sin embargo, debido a que las *queries* están implementadas mediante un *trait*, no es posible serializarlas directamente. Por esta razón, se creó el *enum* `QueryEnum`, que encapsula la *query* dentro del *enum*. Este diseño permite convertir una *query* en un `QueryEnum` y viceversa mediante las funciones `from_query` e `into_query`. Es el `QueryEnum` el que se serializa y deserializa para su transmisión.

A continuación, se muestra la definición del *enum*:

```
#[derive(Serialize, Deserialize)]
pub enum QueryEnum {
    Insert(InsertQuery),
    Delete(DeleteQuery),
    Update(UpdateQuery),
    Select(SelectQuery),
    Use(UseQuery),
    CreateKeyspace(CreateKeyspaceQuery),
    CreateTable(CreateTableQuery),
    DropKeyspace(DropKeySpaceQuery),
    DropTable(DropTableQuery),
    AlterTable(AlterTableQuery),
}
```

2.9. Read Repair

El proceso de `read repair` es fundamental para garantizar la consistencia de los datos entre los diferentes nodos en Cassandra. Este mecanismo debe manejar diversas situaciones en las que las respuestas de los nodos no coinciden y actuar de forma adecuada para reparar dichas discrepancias. Se distinguen dos casos principales de discordancias:

1. Diferencias en los valores de las filas (rows)

Los nodos pueden devolver filas que contienen valores diferentes en algunas de sus columnas para una misma fila. Esto ocurre, por ejemplo, cuando los valores en una columna específica han sido actualizados en un nodo, pero no han sido replicados correctamente en los demás.

Nodo 1				Nodo 2			
ID	nombre	apellido	voto	ID	nombre	apellido	voto
1	thiago	pacheco	true	1	tiago	pacheco	false
2	matias	bartellone	false	2	matias	bartellone	false
3	ivan	maximoff	true	3	ivan	maximoff	false

Figura 1: Ejemplo diferencias entre Rows

2. Filas faltantes o sobrantes

Un nodo puede devolver una respuesta que contiene filas adicionales o carece de algunas filas en comparación con las respuestas de otros nodos. Esto puede suceder si un nodo no recibió correctamente las actualizaciones o si contiene filas que fueron eliminadas en otros nodos.

Nodo 1				Nodo 2			
ID	nombre	apellido	voto	ID	nombre	apellido	voto
1	thiago	pacheco	true	2	matias	bartellone	false
3	ivan	maximoff	true	3	ivan	maximoff	true

Figura 2: Ejemplo Rows Faltantes o Sobrantes

Estados de las filas y timestamps

Para comprender la solución propuesta, es esencial recordar el funcionamiento del comando **DELETE** en Cassandra. Cada fila puede tener uno de los siguientes tres estados:

- **Deleted False:** La fila está almacenada y no ha sido eliminada.
- **Deleted True:** La fila está almacenada pero se registró como eliminada en algún momento.
- **Sin registro:** La fila no está presente ni existe registro de ella, como si nunca hubiera sido insertada.

Además, cada fila posee un **timestamp** asociado a su creación o última modificación, y cada valor en una columna también tiene su propio **timestamp**. El proceso de **read repair** sigue una regla clave: aunque las respuestas de los nodos sean idénticas, si existe una fila con **Deleted True**, se aprovecha para eliminarla definitivamente del sistema. Esto asegura que cualquier inconsistencia con registros eliminados sea reparada proactivamente.

Propuesta: Generación de una best response

Para ambos casos mencionados, proponemos la creación de una respuesta ideal llamada **BEST**. Esta respuesta solo se genera si se detecta la necesidad de realizar reparaciones. La **BEST** se construye recolectando todos los valores de todas las columnas de todas las filas reportadas por los nodos, eligiendo siempre el valor con el **timestamp** más reciente.

De este modo, la **BEST** representa los datos más actualizados posibles, incluyendo todas las filas que deberían existir en la base de datos. Una vez generada, se compara la **BEST** con las respuestas de cada nodo para identificar inconsistencias. Los nodos con diferencias en valores o en la cantidad de filas deberán ser reparados.

Caso 1: Diferencias en los valores de las filas

Para este caso, se identifican los valores rotos en las filas de cada nodo y se realizan actualizaciones específicas (**UPDATE**) para corregirlos. Cada actualización se realiza fila por fila, delegándose a cada nodo afectado.

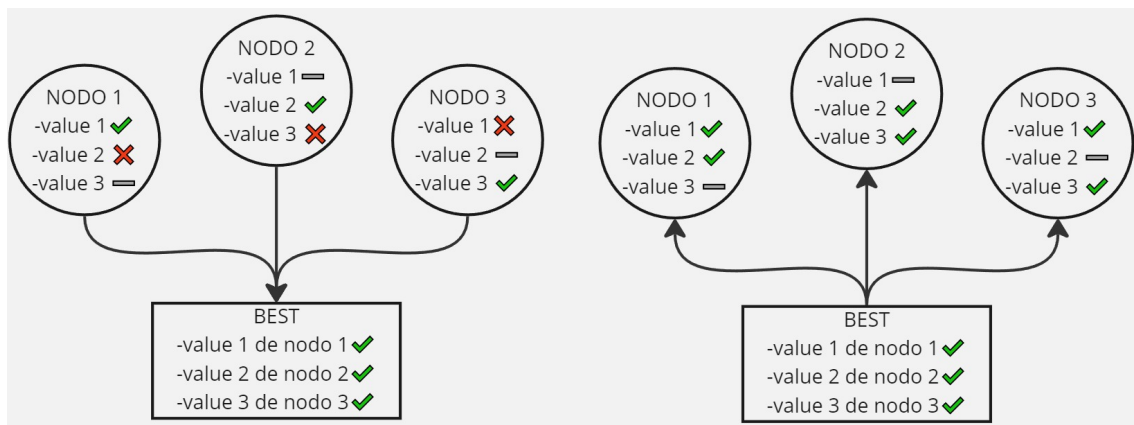


Figura 3: Ejemplo Reparación de valores

Caso 2: Filas faltantes o sobrantes

Debido a que la **BEST** recolecta todas las filas posibles, nunca habrá un nodo con más filas que la **BEST**. Por lo tanto, cualquier fila adicional estará en la **BEST** pero no en los nodos afectados. Para cada fila que falta, se evalúan los siguientes casos:

- Si la fila tiene **Deleted True** en la **BEST**, se elimina del nodo.
- Si la fila tiene **Deleted False** en la **BEST**, se inserta o actualiza en el nodo.
- Si el nodo no tiene registro alguno de la fila, se realiza una inserción para sincronizarlo o no se realiza operación alguna.

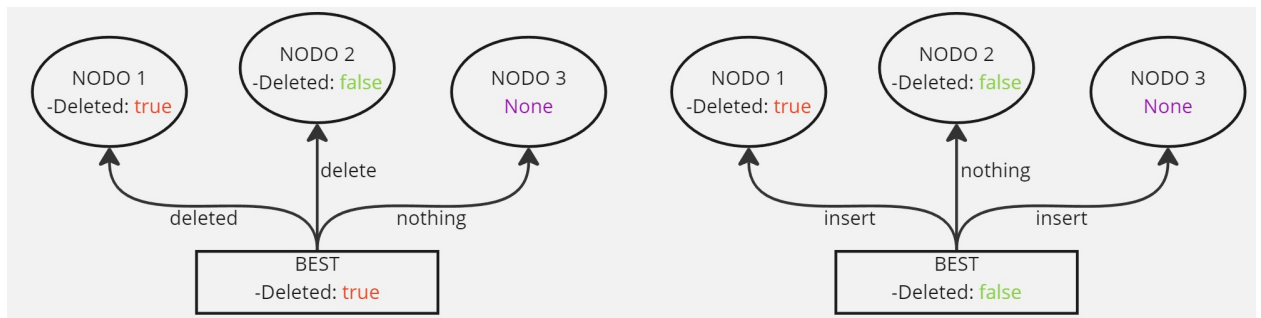


Figura 4: Ejemplo Reparación de Rows

Reglas de reparación basadas en estados

El manejo de las filas se realiza según el estado de la fila en la BEST y el nodo:

1. BEST con Deleted True:

- Nodo con Deleted True: Se envía un comando DELETE para eliminarla completamente del sistema.
- Nodo con Deleted False: Se envía un comando DELETE para actualizar su estado a Deleted True.
- Nodo sin registro: No se realiza ninguna acción.

2. BEST con Deleted False:

- Nodo con Deleted True: Se envía un comando INSERT para reinsertar la fila.
- Nodo con Deleted False: No se realiza ninguna acción.
- Nodo sin registro: Se envía un comando INSERT para insertar la fila.

Consideración especial: Evitar pérdida de registros eliminados

En el caso en que la BEST tenga el atributo Deleted True (es decir, existe un registro de la fila con el estado Deleted True y este es el más actualizado) y un nodo tenga esa misma fila con el atributo Deleted False, se toma la decisión de enviar únicamente un comando DELETE. Esto asegura que el nodo actualice su estado a Deleted True, manteniendo un registro del borrado.

La razón detrás de esta decisión es evitar situaciones en las que se pueda perder completamente el rastro de una fila borrada. En un escenario donde un nodo no sincronice correctamente el estado de una fila y esta quede registrada como Deleted False, el sistema podría asumir que la fila debe permanecer activa, ya que no existiría un Deleted True con un timestamp más reciente para corroborar el borrado. Esta pérdida de evidencia podría ocasionar inconsistencias en los datos del sistema.

Aunque esta estrategia no garantiza la eliminación inmediata de las inconsistencias, reduce significativamente las probabilidades de que ocurran. Supongamos un caso extremo en el que un nodo experimente fallas y no procese los comandos DELETE. Si asumimos que el resto de los nodos del sistema actualizan correctamente sus estados, se necesitarían al menos tres intentos fallidos de enviar el comando DELETE al mismo nodo para que se genere una inconsistencia:

Este enfoque es un compromiso entre garantizar la consistencia del sistema y evitar la pérdida total de evidencia sobre filas previamente borradas. Al mantener un registro de Deleted True, incluso en situaciones de falla prolongada en un nodo específico, el sistema conserva la capacidad de corregir inconsistencias en futuras reparaciones.

Implementación

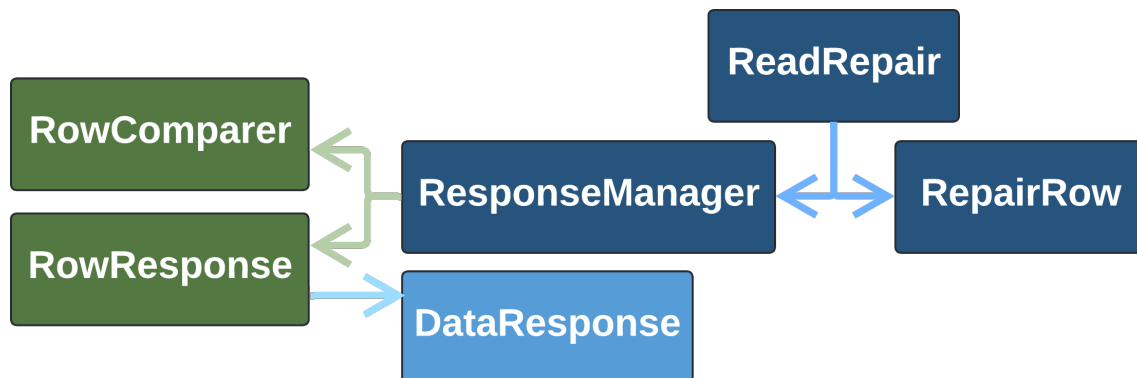


Figura 5: Diagrama Read Repair

2.9.1. Read Repair(estructura)

El componente en cuestión se conecta con el *Query Delegator* para devolver la **response** en caso de tratarse de una operación de lectura. Este proceso comienza al recibir las **response** de cada nodo participante. En primer lugar, verifica si es necesaria una reparación de los datos; si no lo es, devuelve cualquier **response**, previamente convertida al formato del protocolo **row** de Cassandra. Sin embargo, si detecta inconsistencias, procede a realizar una reparación utilizando el **ResponseManager**.

La reparación comienza construyendo una respuesta unificada denominada **BEST**, que contiene los datos más actualizados y consistentes posibles a partir de todas las **response** recibidas. Una vez obtenido el **BEST**, se inicia el proceso de reparación nodo por nodo mediante el uso del componente **RepairRow**. Este proceso se divide en dos fases:

Primera Fase: Reparación de filas coincidentes

En esta etapa, se identifican las filas de **BEST** que coinciden con las del nodo correspondiente, según la clave primaria (*primary key*). Para cada fila coincidente, se comparan los valores columna por columna entre el **BEST** y el nodo. Dependiendo de las discrepancias detectadas, se generan y envían las siguientes consultas al nodo para actualizarlo:

- **Discrepancias entre valores con Deleted False en ambos:** Se genera una consulta **UPDATE** para sincronizar los valores del nodo con los de **BEST**.
- **El BEST tiene Deleted True:** Se envía una consulta **DELETE** al nodo. Esto puede servir para corregir el estado de la fila o eliminarla definitivamente, dependiendo del estado del nodo.
- **El BEST tiene Deleted False y el nodo Deleted True:** Se genera una consulta de reinsertión (**INSERT**) para restaurar la fila en el nodo.

Segunda Fase: Inserción de filas remanentes

En esta etapa, se procesan las filas presentes en **BEST** que no coinciden con ninguna fila en el nodo (es decir, filas remanentes). Para cada una de estas filas, se toman las siguientes acciones:

- **Fila con Deleted True:** No se realiza ninguna acción, ya que no se requiere mantener la fila activa en el nodo.
- **Fila con Deleted False:** Se envía una consulta **INSERT** para añadir la fila al nodo.

Finalización del proceso

Tras completar todas las reparaciones en los nodos, el BEST se convierte al formato del protocolo `row` de Cassandra y se devuelve como la respuesta final de la consulta. Este enfoque garantiza que los datos sean consistentes y actualizados en todos los nodos, respetando el protocolo de Cassandra.

2.9.2. Response Manager

El componente descrito es responsable de manejar las diferentes respuestas proporcionadas por los nodos en el sistema. Las respuestas se dividen en dos partes principales:

- **Rows:** Representan las filas que han sido leídas de la base de datos.
- **Meta_data:** Contiene información estructural, como el *keyspace*, la tabla, las claves primarias (*primary keys*) de las tablas y los nombres de estas.

Este componente almacena la información asociada a cada nodo y es responsable de realizar diversas operaciones relacionadas con las **responses**. Entre estas operaciones se encuentran las siguientes:

- **Leer las IPs de los nodos:** Extrae las direcciones IP de los nodos participantes.
- **Verificar si es necesaria la reparación:** Determina si existen discrepancias que requieran reparación, ya sea debido a filas marcadas como `Deleted True` o a diferencias en los valores entre las filas de los nodos.
- **Calcular el BEST:** Identifica la respuesta más consistente y actualizada al comparar todas las **responses** recibidas de los nodos.
- **Leer las filas (Rows) devueltas por cada nodo:** Permite acceder a las filas específicas obtenidas como resultado de la consulta en cada nodo.
- **Leer la Meta_data devuelta por cada nodo:** Proporciona acceso a los datos estructurales que describen el contexto de la consulta.
- **Castear respuestas al protocolo de Cassandra:** Convierte las respuestas de un nodo o el BEST al formato `Protocol Row Response` requerido por Cassandra.

El cálculo del BEST se realiza utilizando un componente denominado **RowCompare**, que compara las respuestas de los nodos y selecciona los valores más recientes y consistentes. Para la lectura de las filas (**Rows**) y la metadata (**Meta_data**), se emplea el componente **RowResponse**, que facilita el acceso a los datos devueltos por los nodos.

Este diseño asegura una correcta gestión de las respuestas de los nodos, tanto para detectar inconsistencias como para consolidar la información en un formato consistente y adecuado para su retorno al cliente.

2.9.3. Row Comparer

El componente **RowCompare** es el encargado de calcular la mejor respuesta posible (BEST) entre dos respuestas proporcionadas por los nodos. Este proceso implica comparar las filas (**rows**) de ambas respuestas, evaluando cada una de ellas individualmente.

Para determinar la mejor respuesta, **RowCompare** realiza las siguientes acciones:

- **Comparación de filas:** Se compara cada fila de las dos respuestas, seleccionando los valores con el *timestamp* más reciente. Esto garantiza que se prioricen los datos más actualizados.

- **Evaluación de atributos:** Además de los valores, se comparan atributos específicos de las filas, como el estado de eliminación (**Deleted**). En este caso, si las filas difieren, se prioriza esta la fila de mejor timestamp para asegurar la coherencia en el sistema.

El cálculo de la mejor respuesta global (**BEST**) se lleva a cabo utilizando de manera iterativa el componente **RowCompare**. El procedimiento consiste en mejorar una respuesta base, comparándola con las demás respuestas, una por una. Cada iteración actualiza la respuesta base con los valores y atributos más recientes, hasta obtener una respuesta consolidada y óptima.

Este enfoque asegura que **BEST** contenga los datos más precisos y actualizados, sirviendo como referencia para la reparación de nodos y la respuesta final al cliente.

2.9.4. Row Response

La estructura **RowResponse** es utilizada para acceder y manipular tanto las filas (**rows**) como la metadata contenida en las respuestas proporcionadas por los nodos. Esta estructura actúa como una interfaz que permite extraer, interpretar y procesar la información de manera eficiente.

- **Filas (rows):** Representan los datos leídos de las tablas en respuesta a una consulta. Cada fila contiene los valores asociados a las columnas de una tabla específica.
- **Metadata:** Incluye información clave sobre el contexto de las respuestas, como el **keyspace**, el nombre de la tabla, las claves primarias (**primary keys**) y otras características estructurales de las tablas involucradas.

Esta estructura es fundamental para interactuar con las respuestas de los nodos, ya que permite aislar y trabajar de manera específica con los datos o con la metadata según lo requiera el sistema. Además, facilita la integración de estas respuestas en procesos posteriores, como el cálculo de **BEST** o la preparación de datos para el cliente.

2.9.5. DataResponse

La estructura **DataResponse** es una herramienta auxiliar diseñada para facilitar el almacenamiento y acceso eficiente a la metadata obtenida de las respuestas de los nodos. Su propósito principal es organizar y centralizar esta información, permitiendo su manejo de manera clara y estructurada en procesos posteriores.

- **Organización de la metadata:** Centraliza datos esenciales, como el **keyspace**, el nombre de las tablas, las claves primarias (**primary keys**) y cualquier otra información relevante asociada a las tablas implicadas en la respuesta.

2.9.6. Repair Row

RepairRow: Generación de Consultas para la Reparación de Filas

La estructura **RepairRow** es responsable de generar las consultas (**queries**) necesarias para reparar inconsistencias en una fila (**row**). Actúa como intermediaria entre la **Best Row** (fila que contiene los datos más actualizados y consistentes) y la **Node Row** (fila correspondiente a un nodo específico), determinando las acciones correctivas según la situación detectada.

Funcionalidades Principales

■ Reparación Basada en la Comparación de Filas:

- **Caso 1:** Si la *Best Row* tiene el atributo `deleted` configurado en `true`, se genera una consulta `DELETE` para eliminar la fila del nodo. Este proceso puede implicar:
 - Actualizar el registro para reflejar el estado `deleted`.
 - Borrar definitivamente la fila en el nodo, asegurando la consistencia global.
- **Caso 2:** Si la *Best Row* tiene `deleted = false` y la *Node Row* tiene `deleted = true`, se genera una consulta `INSERT` para reintegrar la fila en el nodo con los datos actualizados.
- **Caso 3:** Si ambas filas tienen `deleted = false`, se verifica si existen discrepancias en los valores de las columnas. En caso afirmativo, se genera una consulta `UPDATE` para corregir y sincronizar los datos en el nodo.

- **Inserción de Filas Remanentes:** En situaciones donde existen filas en la *Best Row* que no están presentes en la *Node Row*, *RepairRow* es capaz de crear consultas `INSERT` específicas para dichas filas. Este mecanismo asegura que todas las filas de la *Best Row* sean replicadas en el nodo correspondiente.

Resumen

La estructura *RepairRow* juega un papel crítico en la sincronización de datos entre nodos, permitiendo identificar y corregir inconsistencias en las filas. Su capacidad para generar consultas específicas de reparación, ya sean `DELETE`, `UPDATE` o `INSERT`, garantiza que los datos se mantengan consistentes y actualizados en todo el sistema distribuido.

2.10. Gossip

El protocolo *Gossip* es un proceso que ocurre cada segundo, durante el cual los nodos eligen aleatoriamente a otro nodo del *cluster* para intercambiar información y mantener los datos actualizados.

Este protocolo fue modelado separándolo en dos componentes principales: un emisor y un receptor. A continuación, se explican ambos en detalle.

2.10.1. Gossip Emitter

El **Gossip Emitter** es el encargado de iniciar el proceso de *Gossip* cada segundo mediante la función `start_gossip()`. Esta función selecciona aleatoriamente una *IP* del *cluster*. Al establecer conexión con otro nodo, envía la lista completa de *nodos* registrada en la *meta data*. Posteriormente, espera recibir una lista de nodos con datos actualizados si es necesario.

Por ejemplo, en un *cluster* de 8 nodos, si el nodo A se conecta aleatoriamente con el nodo B, el nodo A enviará la lista completa de los 8 nodos. Si el nodo B tiene versiones más recientes de los nodos E y F, estos serán enviados de vuelta al nodo A, que los actualizará y dará por finalizado el proceso.

```
pub fn start_gossip() -> Result<(), Errors> {
    let Some(ip) = Self::get_random_ip()? else {
        return Ok(());
    };
    if let Ok(mut stream) = TcpStream::connect(ip.get_gossip_socket()) {
        Self::send_nodes_list(&mut stream)?;
        Self::get_nodes_list(&mut stream)
    } else {
        Self::set_inactive(ip)
    }
}
```

2.10.2. Gossip Listener

El **Gossip Listener** tiene la tarea de revisar y comparar los datos del **cluster**. Recibe una lista de nodos de otro emisor y la compara con la suya propia. Si identifica diferencias en los datos o detecta la ausencia de algún nodo, debe actualizar su información. Además, debe determinar qué nodo está desactualizado para devolverle los datos necesarios.

Por cada nodo en la lista recibida, se verifica si es necesario actualizar los datos utilizando la siguiente función:

```
// 1 yes (node 1 newer)
// 0 no
// -1 yes (node 2 newer)
fn needs_to_update(node1: &Node, node2: &Node) -> i8 {
    if node1.get_pos() != node2.get_pos()
        || node1.get_ip() != node2.get_ip()
        || node1.state != node2.state
        || node1.is_seed != node2.is_seed
    {
        if node1.get_timestamp().is_newer_than(node2.get_timestamp()) {
            return 1;
        }
        return -1;
    }
    0
}
```

2.10.3. Seed Listener

El **Seed Listener** no es parte directa del protocolo *Gossip*, pero se incluye en esta sección debido a su relación con el manejo de datos de los nodos. Cuando un nodo se inicializa (excepto el primero del *cluster*), necesita obtener información principal y anunciar su incorporación al *cluster*. Para ello, se definen nodos *seed*, que están a la espera de nuevos nodos en la red.

Cuando un nuevo nodo se conecta, el *Seed Listener* guarda su **IP** y le envía la lista de todos los nodos ya existentes en el *cluster*.

```
fn handle_connection(stream: &mut TcpStream) -> Result<(), Errors> {
    Self::send_nodes_list(stream)?;
    let new_node = Self::get_new_node(stream)?;
    Self::set_new_node(new_node)
}
```

2.11. Hinted Handoff

El **Hinted Handoff** es un proceso mediante el cual, si un nodo en un **cluster** está inactivo y no puede recibir una **query**, esta se almacena temporalmente. Cuando el nodo caído se reintegra al sistema (modo *booting*), se le transmiten las *queries* acumuladas durante su inactividad. Este mecanismo fue modelado utilizando un **Sender** y un **Receiver** para la comunicación, y un *handler* para el almacenamiento y mantenimiento de las *queries*. A continuación, se explican cada uno de estos componentes.

2.11.1. Hints Handler

El **Hints Handler** se encarga del almacenamiento y mantenimiento de las **queries**. Cuando un nodo intenta guardar una *query*, proporciona también la **IP** del nodo inactivo, lo que permite crear un archivo nombrado con dicha IP. Cada *query* incluye un **timestamp**, y periódicamente se revisa el archivo correspondiente para identificar *queries* vencidas y eliminarlas.

Para evitar leer el archivo completo, las *queries* se almacenan de forma secuencial al final del archivo (es decir, se *appendean*), manteniendo el orden cronológico. Por lo tanto, solo se necesita verificar la primera *query* del archivo para determinar si está vencida. Si lo está, se continúa revisando hasta encontrar una no vencida. Todas las *queries* vencidas son eliminadas del archivo.

2.11.2. Hints Receiver

Cuando un nodo caído se reinicia, entra en modo *booting* y activa el **Hints Receiver**, que es un **TcpListener** encargado de recibir *hints* de otros nodos. Durante este proceso, el *receiver* recopila las *hints* provenientes de distintos nodos, ya que pueden estar distribuidas. Estas *hints* se almacenan temporalmente en una lista.

Una vez recopiladas todas las *hints*, se ordenan por **timestamp** para garantizar que se procesen en el orden en que llegaron al **cluster**. Finalmente, se ejecutan una a una mediante el siguiente código:

```
fn execute_queries(hints: &mut [StoredQuery]) -> Result<(), Errors> {
    hints.sort_by_key(|stored_query| stored_query.timestamp.timestamp);
    for stored in hints.iter() {
        if stored.get_query().run();
    }
    Ok(())
}
```

2.11.3. Hints Sender

Durante el proceso de **Gossip**, si un nodo detecta que otro nodo previamente inactivo ha entrado en estado *booting*, utiliza el **Hints Sender** para transmitir las *hints* almacenadas. Primero verifica si tiene *hints* guardadas para ese nodo. Si las tiene, invoca la función **send_hints()**.

Esta función lee las *queries* del archivo correspondiente y las envía una por una al receptor, esperando un *acknowledge* después de cada transmisión para garantizar que las *queries* se reciben correctamente. Al finalizar la transmisión, elimina el archivo de *hints* y el nodo continúa con su funcionalidad habitual.

2.12. Encriptación

2.12.1. Conexiones Cliente-Servidor

Se utiliza TLS para proteger las comunicaciones entre cliente y servidor, implementado con la biblioteca **rustls**. El servidor utiliza un certificado generado a partir de una clave privada (auto-certificado), que debe ser incorporado en el *root store* del cliente para ser reconocido como válido. Este certificado también incluye las direcciones IP que actuarán como dominios válidos.

2.12.2. Conexiones Servidor-Servidor

Para conexiones *Servidor-Servidor* encriptamos manualmente con **openssl**. Se usa una clave privada secreta conocida por los servidores y una IV que se genera aleatoriamente cada vez que se escriba en el *stream*. El acceso al IV por sí solo no permite descryptar los datos. Para descryptar un mensaje cifrado, un atacante también necesitaría tener acceso a la clave secreta utilizada para la encriptación. Sin embargo, un IV fijo o repetido puede crear vulnerabilidades adicionales, ya que puede permitir que los atacantes noten patrones y puedan realizar ciertos ataques, especialmente si se usan varios mensajes cifrados con el mismo IV y clave. Por eso, es importante utilizar un IV aleatorio único para cada operación de cifrado para mejorar la seguridad.

2.12.3. Hash de contraseñas

Las contraseñas se almacenan de manera segura mediante un hash generado con el algoritmo **Argon2**, que está específicamente diseñado para proteger contraseñas. Este método:

- Asegura que las contraseñas no puedan ser descifradas ni revertidas.
- Permite verificar si una contraseña ingresada coincide con su hash sin revelar la contraseña original.

2.13. Utils

2.13.1. Frame

El **Frame** es la estructura que utilizamos para representar las *requests* siguiendo el protocolo de Cassandra. Esta estructura resulta fundamental, ya que nos permite convertir una secuencia de bytes en un formato estructurado, lo que facilita su posterior análisis y *parsing*. Además, también es utilizada para construir las respuestas que se envían de vuelta al cliente.

El **Frame** cuenta con dos funciones principales: - **parse_frame()**: Convierte un vector de bytes (**Vec<u8>**) en una instancia de **Frame**. - **to_bytes()**: Realiza la operación inversa, transformando un **Frame** en una secuencia de bytes que puede enviarse a través de la red.

La definición de la estructura **Frame** es la siguiente:

```
pub struct Frame {  
    pub version: u8,  
    pub flags: u8,  
    pub stream: i16,  
    pub opcode: u8,  
    pub length: u32,  
    pub body: Vec<u8>,  
}
```

Cada campo tiene un propósito específico: - **version**: Indica la versión del protocolo de Cassandra utilizado. - **flags**: Contiene información adicional sobre la *request*, como compresión o rastreo. - **stream**: Identifica el *stream* asociado a la solicitud o respuesta. - **opcode**: Representa el tipo de operación que se está realizando. - **length**: Especifica la longitud del cuerpo del *Frame*. - **body**: Contiene el contenido principal de la *request* o respuesta en formato de bytes.

Esta estructura nos proporciona un mecanismo eficiente y claro para interactuar con el protocolo de Cassandra, permitiendo tanto la interpretación como la generación de mensajes en el formato requerido.

2.13.2. Bytes Cursor

El **Bytes Cursor** es una herramienta sumamente útil para el *parsing* de secuencias de bytes. Se trata de una estructura que toma una tira de bytes y la consume progresivamente dependiendo del tipo de dato que se esté leyendo. Además, retorna dicho tipo de dato concreto, lo que facilita la interpretación de los datos. Esta herramienta es utilizada tanto para construir el **Frame** como para analizar el contenido del *body*.

Un ejemplo práctico de uso sería validar que el *body* de una *request* comience con un `[Short]` (un entero de 2 bytes). El código correspondiente sería:

```
pub fn read_short(&mut self) -> Result<i16, Errors> {
    let buf = self.read_exact(2)?;
    Ok(i16::from_be_bytes(buf.try_into()).map_err(|_| {
        Errors::ProtocolError(String::from("Could not read bytes"))
    })?)
}
let mut cursor = BytesCursor::new(body);
let id = cursor.read_short()?;
```

El **Bytes Cursor** soporta la lectura de los siguientes tipos de datos:

- `[Int]`: Entero de 4 bytes.
- `[Long]`: Entero de 8 bytes.
- `[Short]`: Entero de 2 bytes.
- `[String]`: Un `[Short]` *n* seguido de *n* bytes representando una cadena en formato UTF-8.
- `[Long String]`: Un `[Int]` *n* seguido de *n* bytes representando una cadena en formato UTF-8.
- `[Bytes]`: Un `[Int]` *n* seguido de *n* bytes.
- `[Short Bytes]`: Un `[Short]` *n* seguido de *n* bytes.
- `[Consistency]`: Un `[Short]` que representa el nivel de consistencia.
- `[String Map]`: Un `[Short]` *n* seguido de *n* pares `<k><v>`, donde *k* y *v* son `[String]`.

El **Bytes Cursor** también incluye funciones para leer una cantidad específica de bytes, lo cual es especialmente útil para construir el `Frame`. Estas funciones incluyen:

- `[u8]`: Lectura de un byte sin signo.
- `[i16]`: Lectura de un entero de 2 bytes con signo.
- `[u32]`: Lectura de un entero de 4 bytes sin signo.
- `[read remaining bytes]`: Lectura de todos los bytes restantes en el cursor.
- `[read exact]`: Lectura de una cantidad específica de bytes.

El diseño del **Bytes Cursor** garantiza un manejo eficiente y seguro de las secuencias de bytes, permitiendo operaciones precisas en un contexto donde el orden y la estructura de los datos son fundamentales.

2.13.3. Errors

Los **Errors** están representados mediante un `enum` que define los distintos tipos de errores posibles en el sistema. Todas las funciones del nodo tienen la capacidad de devolver un error, el cual se propaga hasta el momento de generar la *response*. En caso de detectar un error, se construye un **Error Frame**, que incluye tanto el tipo de error como un mensaje descriptivo.

El **Error Frame** cuenta con una función que convierte el **Error**, junto con su descripción (`String`), en una tira de `Bytes`, lo que permite construir el *body* de la respuesta para el cliente siguiendo el protocolo de Cassandra.

El siguiente código muestra la definición del `enum`:

```
pub enum Errors {  
    ServerError(String),  
    ProtocolError(String),  
    BadCredentials(String),  
    UnavailableException(String),  
    Overloaded(String),  
    IsBootstrapping(String),  
    TruncateError(String),  
    WriteTimeout(String),  
    ReadTimeout(String),  
    SyntaxError(String),  
    Unauthorized(String),  
    Invalid(String),  
    ConfigError(String),  
    AlreadyExists(String),  
    Unprepared(String),  
}
```

A continuación, se describen los errores más frecuentes y sus posibles causas:

- **ServerError**: Indica un error interno en el servidor, generalmente causado por problemas inesperados.
- **ProtocolError**: Ocurre cuando el cliente envía una petición que no cumple con el protocolo definido.
- **BadCredentials**: Se produce cuando las credenciales proporcionadas no son válidas.
- **UnavailableException**: Se genera cuando no hay nodos suficientes disponibles para satisfacer la petición.
- **Overloaded**: Indica que el nodo está sobrecargado y no puede procesar más peticiones.
- **IsBootstrapping**: Señala que el nodo aún está en proceso de inicialización (*bootstrapping*).
- **WriteTimeout** y **ReadTimeout**: Se presentan cuando se excede el tiempo de espera al realizar operaciones de escritura o lectura, respectivamente.
- **SyntaxError**: Indica que la consulta enviada tiene errores de sintaxis.
- **Unauthorized**: Se produce cuando el cliente no tiene permisos para realizar la operación solicitada.
- **Invalid**: Representa errores en los datos o parámetros de la petición.
- **ConfigError**: Ocurre cuando hay problemas en la configuración del nodo o del clúster.
- **AlreadyExists**: Se genera cuando se intenta crear un recurso que ya existe.
- **Unprepared**: Indica que una consulta preparada no se encuentra disponible.

El diseño del **Errors** enum permite una gestión robusta y detallada de los errores en el sistema, asegurando que cada problema pueda ser identificado y reportado correctamente al cliente.

2.13.4. Node Ip

La estructura **NodeIp** se utiliza para gestionar las direcciones IP y los puertos de los nodos dentro del clúster. Proporciona una representación sencilla, pero poderosa, al incluir tanto la dirección **IpAddr** como el **port**, lo que permite una implementación versátil y funcional.

Una de las principales ventajas de esta implementación es su capacidad para manejar múltiples `TcpListeners` de manera eficiente. Al iniciar el nodo en un puerto base, los puertos subsecuentes se utilizan para listeners específicos, definidos por modificadores únicos. Esto simplifica la lógica para establecer conexiones o iniciar listeners con distintos propósitos.

Por ejemplo, el modificador para delegación de queries (`QUERY_DELEGATION_PORT_MOD`) y el modificador para acceso a datos (`DATA_ACCESS_PORT_MOD`) permiten generar `sockets` específicos sin necesidad de manejar manualmente los puertos.

A continuación, se presenta el código de la estructura junto con tres funciones clave que ilustran cómo obtener distintos `sockets`:

```
pub struct NodeIp {
    ip: IpAddr,
    port: u16,
}

impl NodeIp {
    /// Obtiene el socket estándar del nodo
    pub fn get_std_socket(&self) -> SocketAddr {
        SocketAddr::new(self.ip, self.port)
    }

    /// Obtiene el socket para la delegación de queries
    pub fn get_query_delegation_socket(&self) -> SocketAddr {
        SocketAddr::new(self.ip, self.port + QUERY_DELEGATION_PORT_MOD as u16)
    }

    /// Obtiene el socket para el acceso a datos
    pub fn get_data_access_socket(&self) -> SocketAddr {
        SocketAddr::new(self.ip, self.port + DATA_ACCESS_PORT_MOD as u16)
    }
}
```

Gracias a esta estructura, la implementación de distintos listeners o conexiones en el nodo es sencilla y escalable. Cada tipo de comunicación puede ser mapeada a un puerto distinto mediante los modificadores, lo que asegura que no haya conflictos y que la lógica sea fácil de mantener.

Este diseño fomenta la modularidad y permite que las diferentes funcionalidades del nodo operen de forma independiente mientras comparten una base común: la dirección `NodeIp`.

2.13.5. Timestamp

La estructura `Timestamp` abstrae el concepto de tiempo en el sistema, evitando el uso directo de valores enteros como `i64`. Esto permite una representación más clara del tiempo, además de facilitar la incorporación de funcionalidades específicas relacionadas con timestamps.

La principal ventaja de esta estructura es encapsular el manejo de tiempo dentro de una interfaz definida, lo que mejora la legibilidad del código y reduce la posibilidad de errores. Además, esta abstracción permite extender la funcionalidad del manejo de timestamps sin modificar directamente otras partes del sistema.

La definición básica de `Timestamp` incluye un único campo: el valor del timestamp en milisegundos desde la época Unix. También se incluye un método para inicializar un nuevo timestamp basado en el tiempo actual:

```
pub struct Timestamp {
    pub timestamp: i64,
}

impl Timestamp {
    /// Crea un nuevo Timestamp con el tiempo actual en milisegundos
    pub fn new() -> Self {
        Self {
            timestamp: Utc::now().timestamp_millis(),
        }
    }
}
```

El diseño de `Timestamp` también tiene métodos adicionales para comparar, calcular diferencias o realizar verificaciones relacionadas con el tiempo. Algunas posibles extensiones útiles incluyen:

- Comparar si un `Timestamp` es anterior o posterior a otro.
- Verificar si ha pasado un intervalo de tiempo dado.

Se muestran ejemplos de algunas de estas funcionalidades:

```
impl Timestamp {
    /// Comprueba si el timestamp actual es más antiguo que otro
    pub fn is_older_than(&self, timestamp: Timestamp) -> bool {
        self.timestamp < timestamp.timestamp
    }

    /// Verifica si paso un tiempo específico desde este timestamp
    pub fn has_perished_hours(&self, hours: i64) -> bool {
        let milliseconds = hours * 1000 * 60 * 60;
        Utc::now().timestamp_millis() > self.timestamp + milliseconds
    }
}
```

El uso de `Timestamp` es fundamental en varias partes del sistema, como:

- Determinar la antigüedad de `queries` en `Hinted Handoff`.
- Ordenar eventos basados en su tiempo de ocurrencia.
- Tener información sobre la última actualización de `columns` para la realización del `Read Repair`

Al encapsular el manejo del tiempo dentro de esta estructura, el sistema logra una mayor cohesión y modularidad, permitiendo manejar el tiempo de forma consistente y sin redundancia.

2.13.6. Primary Key

La **Primary Key** es un componente fundamental en el modelo de datos, ya que determina la unicidad de los registros y organiza la distribución y el ordenamiento de los datos dentro del sistema. Nuestra implementación modela esta clave como una estructura que combina la *partition key* y las *clustering columns*. Esto permite una fácil serialización y deserialización para su almacenamiento y recuperación, además de proporcionar métodos convenientes para acceder a su información.

La definición de `PrimaryKey` incluye dos elementos principales:

- **Partition Key:** una lista de cadenas que define cómo se distribuyen los datos en los nodos del clúster.

- **Clustering Columns:** una lista opcional de cadenas que especifica cómo se ordenan los datos dentro de una partición.

A continuación, se presenta el código que define la estructura y su constructor:

```
pub struct PrimaryKey {
    pub partition_keys: Vec<String>,
    pub clustering_columns: Vec<String>,
}

impl PrimaryKey {
    /// Crea una nueva Primary Key con una partition key y opcionalmente clustering columns.
    pub fn new(partition_key: Vec<String>, clustering_columns: Option<Vec<String>>) -> Self {
        Self {
            partition_keys: partition_key,
            clustering_columns: clustering_columns.unwrap_or_default(),
        }
    }
}
```

El diseño de PrimaryKey proporciona las siguientes ventajas:

- **Modularidad:** Separa claramente la lógica de partición y ordenamiento, facilitando el mantenimiento y extensibilidad del código.
- **Flexibilidad:** Permite crear claves primarias con o sin *clustering columns*, adaptándose a diferentes casos de uso.
- **Facilidad de serialización/deserialización:** Al representar las claves como vectores de cadenas, su conversión a y desde formatos serializados es directa.

2.13.7. Consistency

La **Consistency** (consistencia) en un sistema distribuido como Cassandra es un aspecto clave para asegurar que las lecturas y escrituras sean coherentes a través de los nodos del clúster. En este sistema, modelamos los niveles de consistencia como un **enum** con los valores más comunes en Cassandra: **One**, **Quorum**, y **All**.

Cada nivel de consistencia tiene implicaciones diferentes sobre cuántos nodos deben confirmar una operación para considerarla exitosa. Por ejemplo:

- **One** requiere que al menos un nodo confirme la operación.
- **Quorum** necesita que una mayoría de nodos (dependiendo del *Replication Factor*) confirme la operación.
- **All** requiere la confirmación de todos los nodos en el clúster.

Además, implementamos funciones auxiliares para convertir valores de consistencia en enteros y para calcular el número de nodos requeridos, en función del *Replication Factor* del clúster, lo que facilita la gestión dinámica de la consistencia a medida que cambian las configuraciones del sistema.

A continuación se muestra la definición de la estructura para la consistencia:

```
const ONE: i16 = 0x0001;
const QUORUM: i16 = 0x0004;
const ALL: i16 = 0x0005;

pub enum ConsistencyLevel {
    One,
    Quorum,
    All,
}
```

2.13.8. Constantes

Para la definición de variables que se usan a lo largo del nodo, definimos distintas constantes con distintos propósitos. Estas constantes se agrupan de la siguiente manera:

- **Constantes para el manejo de símbolos o palabras especiales** usadas durante el proceso de parseo de datos.
- **Constantes con la dirección de carpetas o archivos** utilizados para la gestión de datos y configuraciones del sistema.
- **Constantes con los modificadores para los sockets de la IP**, que se emplean para manejar los distintos puertos asignados a cada nodo o servicio.
- **Constantes que definen aspectos importantes del sistema**, como tiempos de *timeouts*, número máximo de clientes habilitados simultáneamente, entre otros.

2.13.9. Funciones

Definimos un archivo para funciones generales que se utilizan en varios módulos del sistema. Estas funciones tienen como objetivo modularizar y simplificar tareas que se repetían en distintas partes del código. Dentro de este conjunto de funciones se encuentran:

- **Getters:** Funciones que permiten acceder a los valores de las propiedades de diversas estructuras de datos.
- **Funciones relacionadas con el manejo de streams:** Incluyen operaciones para la lectura y escritura de datos en flujos de entrada y salida.
- **Funciones de serialización y deserialización:** Son responsables de convertir estructuras de datos en representaciones en bytes y viceversa, facilitando la comunicación entre nodos y la persistencia de datos.

2.13.10. Response

La estructura **response** es responsable de generar las respuestas a las consultas (**queries**) de acuerdo con el protocolo de Cassandra. Cada respuesta se representa como una secuencia de bytes, siguiendo el formato de lectura de bytes especificado en dicho protocolo, donde los primeros 4 bytes identifican el tipo de respuesta. A continuación, se describen los tipos de respuestas posibles:

- **Void** (código 0x0001): Esta respuesta no contiene datos, simplemente indica que la operación se ha realizado correctamente. Es utilizada en consultas como **INSERT**, **DELETE** y **UPDATE**, que no requieren devolver datos a la parte que hace la consulta, sino solo confirmar que la acción se completó sin errores.
- **Set Keyspace** (código 0x0003): La respuesta incluye el nombre del **keyspace** que ha sido establecido como el contexto actual. Esta respuesta es generada por la consulta **USE**, que permite cambiar el **keyspace** activo en Cassandra.
- **Schema Change** (código 0x0005): Esta respuesta indica que se ha realizado un cambio en el esquema de la base de datos. Incluye detalles sobre el tipo de cambio (como creación, alteración o eliminación de objetos de la base de datos), el objeto afectado (como tablas, índices, etc.), y un campo adicional **options**, que proporciona una breve descripción del cambio realizado. Este tipo de respuesta es generado por consultas como **ALTER**, **CREATE** y **DROP**.

- **Row** (código 0x0002): Esta respuesta contiene información sobre las filas de una tabla. Comienza con una flag, cuyo valor siempre será 0x0001 en nuestra implementación, indicando que la respuesta contiene datos de filas válidas. Luego, se incluye un entero que indica la cantidad de columnas presentes en la respuesta, seguido de un string que especifica el **keyspace** y otro string que indica el nombre de la tabla. A continuación, se listan los nombres de las columnas junto con el código correspondiente a su tipo de dato. Los códigos de tipo de dato son los siguientes:

- **boolean** → 0x0004
- **date** → 0x000B
- **decimal** → 0x0006
- **duration** → 0x000F
- **int** → 0x0009
- **text** → 0x000A
- **time** → 0x000C

Después de los detalles de las columnas, se incluye un entero que indica la cantidad de filas (**rows**) que forman parte de la respuesta. Para cada fila, se presenta un valor por cada columna correspondiente. En el caso de que una fila no tenga un valor asignado para una columna, se utilizará el valor especial **None** para indicar la ausencia de datos.

- **Row Auxiliar**: En algunos casos, Cassandra puede devolver respuestas adicionales con información extra sobre las filas, como cuáles columnas son claves primarias y los tipos de datos asociados a estas. Estas respuestas adicionales se generan durante consultas **SELECT**, y su propósito es proporcionar información detallada sobre la estructura de los datos, lo que puede ser útil para la reparación de errores o la interpretación de los resultados. Posteriormente, estas respuestas se procesan y se transforman al tipo **Row**, siguiendo el protocolo estándar de Cassandra, antes de ser enviadas al cliente.

3. Instrucciones de Uso

En esta sección se detallan las instrucciones para el uso de uno o más nodos, incluyendo cómo inicializarlos y conectarlos al mismo clúster. Siguiendo estas instrucciones, es posible levantar varios nodos que se unirán automáticamente al clúster, siempre y cuando los datos hayan sido configurados correctamente.

Aclaración importante: Si se levantan varios nodos y se quiere usar misma ip pero distinto puerto, no se pueden usar puertos consecutivos, ya que una misma ip puede llegar a usar del puerto base hasta los seis siguientes.

Para la inicialización, se requiere proporcionar ciertos datos del nodo, los cuales pueden ser configurados de dos maneras distintas:

3.1. Configuración mediante archivo YAML

La primera forma consiste en escribir los datos en un archivo `.yaml`, que puede ser el archivo predeterminado ubicado en la carpeta `/src`, o uno personalizado. La estructura de los datos es la siguiente:

```
ip:
  ip: "127.0.0.1"
  port: 9090
seed_ip:
  ip: "127.0.0.1"
  port: 9090
is_first: true
is_seed: true
```

- `ip`: Representa la dirección IP y el puerto del nodo a levantar. Puede definirse de dos maneras:
- `seed_ip`: Representa la dirección IP de un nodo semilla dentro del clúster. Este campo solo es relevante si `is_first` está en `false`.
- `is_first`: Indica si el nodo es el primero del clúster. Si está en `true`, el nodo será automáticamente un nodo semilla, ignorando los datos de `seed_ip`.
- `is_seed`: Define si el nodo será un nodo semilla o no. Puede tomar los valores `true` o `false`.

3.2. Configuración dinámica en terminal

La segunda forma consiste en proporcionar los datos dinámicamente a través de la terminal. El nodo guía al usuario sobre qué información debe ingresar. Aunque este método es más intuitivo, puede ser menos cómodo en comparación con el uso de un archivo `.yaml`.

3.3. IPs Certificadas para el Uso del Servidor

Para que una conexión entre el cliente y el servidor sea válida, debe cumplir con las condiciones de *TLS*, lo que implica que la IP a la que se conecta el servidor (nodo) debe estar incluida en el autocertificado generado, y el cliente debe confiar en dicha IP. Si se desea modificar esta configuración, se deben seguir los pasos detallados a continuación en la carpeta de certificados.

1. **Generación de la Clave Privada (opcional):** Si aún no se ha creado una clave privada, es necesario generar una para poder utilizarla en la creación del certificado. Para ello, se puede utilizar el siguiente comando de `openssl`:

```
openssl genpkey -algorithm RSA -out private_key.pem
```

Este comando genera una clave privada utilizando el algoritmo RSA y la guarda en el archivo `private_key.pem`.

2. **Modificación del Archivo de Configuración de OpenSSL:** Para agregar las IPs requeridas al autocertificado, es necesario modificar el archivo de configuración `openssl.cnf` para incluir las IPs bajo la sección `[alt_names]`. A continuación se muestra un ejemplo de cómo debe quedar esta sección:

```
[ alt_names ]
IP1 = 127.0.0.1
IP2 = 127.0.0.5
...
IPN = 100.20.50.30
```

Asegúrese de agregar todas las IPs necesarias, cada una en una línea separada, bajo la sección `[alt_names]`.

3. **Creación del Certificado:** Una vez que se haya generado la clave privada y modificado el archivo de configuración, se puede proceder a la creación del certificado utilizando el siguiente comando:

```
openssl req -new -x509 -key private_key.pem -out certificate.pem
-config openssl.cnf -days 365
```

Este comando genera un certificado autofirmado utilizando la clave privada `private_key.pem` y el archivo de configuración `openssl.cnf`. El certificado generado se guardará en el archivo `certificate.pem` y será válido durante 365 días.

3.4. Ejecución del programa

Una vez configurados los datos del nodo, el programa puede ejecutarse de las siguientes maneras:

- Si los datos están en el archivo `config.yaml` ubicado dentro de `/src`, el programa debe ejecutarse con el comando:

```
cargo run default
```

- Si los datos están en un archivo `.yaml` personalizado, debe ejecutarse con el comando:

```
cargo run path
```

donde `path` es la ruta al archivo `.yaml`.

- Si no se utiliza un archivo `.yaml`, simplemente debe ejecutarse con el comando:

```
cargo run
```

3.4.1. Posibles errores y soluciones

En caso de que se presente un error relacionado con alguna librería, como por ejemplo un problema al intentar compilar o ejecutar el programa, puede deberse a que algunas dependencias del sistema no están instaladas o configuradas correctamente. A continuación se mencionan posibles soluciones:

- Error relacionado con OpenSSL: Si el error menciona que falta una librería como ‘libssl-dev’ o alguna relacionada con OpenSSL, asegúrese de tener las librerías de desarrollo necesarias instaladas con los siguientes comandos:

```
sudo apt update
sudo apt install openssl
sudo apt install libssl-dev
```

Estos comandos actualizarán los repositorios y luego instalarán OpenSSL y las librerías de desarrollo necesarias.

Client

4. Client communication

4.1. Cassandra Connection

La **Cassandra Connection** es una capa interna del cliente que gestiona la conexión subyacente, implementada con **rustls** y **TcpStream**.

Su función principal es establecer y mantener la conexión segura con el servidor Cassandra. Utiliza certificados para garantizar la autenticidad del servidor, cargándolos desde archivos locales y agregándolos al *root store* del cliente. Además, implementa la lógica para la lectura y escritura de *frames* en el flujo de datos.

```
pub struct CassandraConnection {
    stream: StreamOwned<ClientConnection, TcpStream>,
}

impl CassandraConnection {
    /// Create a new connection to the server with the given node (ip:port)
    pub fn new(node: &str) -> Result<Self, String> {
        ...
    }
}
```

4.2. Cassandra Client

El **CassandraClient** es un componente clave que actúa como intermediario entre los clientes y la base de datos Cassandra. Su diseño abstrae los detalles de implementación de la conexión TCP y el protocolo, ofreciendo a los clientes una interfaz unificada para interactuar con la base de datos sin necesidad de conocer los detalles de bajo nivel.

```
pub struct CassandraClient {
    connection: CassandraConnection
}
```

4.2.1. Características principales

- **Abstracción de la conexión:** Utiliza **CassandraConnection**, que maneja el flujo TLS basado en **rustls**, garantizando que las comunicaciones sean seguras.
- **Protocolos y operaciones:** Proporciona métodos para enviar y recibir *Frames* según el protocolo de Cassandra, manejando automáticamente operaciones como autenticación y configuración inicial.
- **Configuración inicial:** Incluye métodos para iniciar las conexiones y autenticar al cliente.

```
pub fn start_up(&mut self) -> Result<(), String>
pub fn authenticate(&mut self, user: &str, password: &str) -> Result<(), String>
```

- **Niveles de consistencia:** Permite ejecutar consultas con diferentes niveles de consistencia, tales como:
 - **Strong:** Por ejemplo, `ConsistencyLevel::Quorum`, para garantizar una mayor confiabilidad en operaciones críticas.
 - **Weak:** Por ejemplo, `ConsistencyLevel::One`, utilizado para operaciones que toleran cierto grado de inconsistencia.

- **Uso simplificado:** Ofrece métodos de alto nivel que encapsulan las operaciones necesarias para interactuar con Cassandra, facilitando la integración con otras partes del sistema (como el simulador de vuelos y la aplicación gráfica).

4.2.2. Ejemplo de uso

A continuación se muestra un ejemplo de cómo se utiliza la función `use_aviation_keyspace` de `UIClient` para interactuar con la base de datos Cassandra:

```
impl UIClient {
    pub fn new(client: CassandraClient) -> Self {
        Self {
            client,
        }
    }
    /// Use the aviation keyspace in the cassandra database
    pub fn use_aviation_keyspace(&mut self) -> Result<(), String> {
        let frame_id = STREAM as usize;
        let query = format!("USE {}; ", KEYSPACE_AVIATION);
        self.client.execute_strong_query_without_response(&query, &frame_id)
    }
}
```

4.3. Thread Pool Client

El `ThreadPoolClient` está diseñado para manejar múltiples clientes conectados al sistema, como el simulador de vuelos. Permite ejecutar consultas y operaciones de forma concurrente, distribuyéndolas entre un conjunto de hilos preexistentes (un *pool* de hilos), en lugar de crearlos y destruirlos repetidamente para cada tarea.

4.3.1. Principales Características

- **Ejecución Concurrente:** Recibe tareas mediante el método `execute` y las asigna a los hilos disponibles en el *pool*.
- **Reutilización de Hilos:** Cada hilo permanece activo y listo para procesar nuevas tareas, reduciendo la sobrecarga.
- **Sincronización:** Permite al hilo principal esperar a que se completen todas las tareas con el método `join`.

4.3.2. Metodo Execute

Permite que el usuario pueda ejecutar un función de forma concurrente y ademas si necesita un resultado de dicha función le proporciona un `Sender` para pueda esperar la respuesta.

```
/// Send a job to the worker
pub fn execute<T, F>(&self, f: F) -> Receiver<T>
where
    T: Send + 'static,
    F: FnOnce(usize, &mut CassandraClient) -> T + Send + 'static,
{
```

4.3.3. Ejemplo de Uso

```
fn get_flights_by_airports(
    &self,
    airports_codes: &Vec<String>,
    flight_codes_by_airport: &HashMap<String, Vec<String>>,
    thread_pool: &ThreadPoolClient) -> Vec<Flight> {
    let mut receivers = Vec::new();
    for airport_code in airports_codes {
        let airport_code = airport_code.to_string();
        let flight_codes_airport = flight_codes_by_airport.get(&airport_code)
            .unwrap_or(&Vec::new()).to_vec();
        let flights_receiver = thread_pool.execute(move |frame_id, client| {
            Self.get_flights_by_airport(&airport_code, &flight_codes_airport, client, &frame_id)
        });
        receivers.push(flights_receiver);
    }

    let mut flights = Vec::new();
    for receiver in receivers {
        if let Ok(mut received_flights) = receiver.recv() {
            flights.append(&mut received_flights);
        }
    }
    flights
}
```

4.3.3.1 Descripción del Código

El método `get_flights_by_airports` toma dos parámetros: una lista de códigos de aeropuertos y un mapa que asocia cada código de aeropuerto con los códigos de vuelo correspondientes. Utiliza el `ThreadPoolClient` para distribuir las tareas de obtener los vuelos de cada aeropuerto a los hilos disponibles en el pool. Cada tarea se ejecuta de forma concurrente y, al final, todos los vuelos obtenidos se agrupan en un único vector `flights`, que se retorna como resultado.

4.4. Query Builder

El `QueryBuilder` es una estructura diseñada para facilitar la construcción dinámica de consultas CQL. Permite crear consultas de tipo `SELECT`, `INSERT`, `UPDATE` y `DELETE` de manera flexible, incorporando condiciones, operadores lógicos y parámetros adicionales como ordenamientos y cláusulas condicionales.

La estructura tiene los siguientes campos:

- `query_type`: Especifica el tipo de consulta (e.g., `SELECT`, `INSERT`).
- `table`: Nombre de la tabla sobre la que se ejecuta la consulta.
- `columns`: Lista de columnas involucradas en la consulta (selección, inserción o actualización).
- `values`: Valores correspondientes a las columnas en las consultas de inserción.
- `conditions`: Condiciones para aplicar en la consulta, como las de la cláusula `WHERE`.
- `logical_operators`: Operadores lógicos para conectar las condiciones, como `AND` o `OR`.
- `order_by`: Especifica la columna y el orden (ascendente o descendente) para ordenar los resultados.
- `if_condition`: Condición opcional utilizada en consultas `UPDATE` o `DELETE` (e.g., `IF EXISTS`).

4.4.1. Métodos

- **new:** Crea un nuevo `QueryBuilder` con el tipo de consulta y la tabla especificada.
- **select:** Define las columnas a seleccionar en una consulta `SELECT`.
- **insert:** Define las columnas y los valores a insertar en una consulta `INSERT`.
- **update:** Define las columnas y los valores a actualizar en una consulta `UPDATE`.
- **where_condition:** Agrega condiciones a la cláusula `WHERE`.
- **order_by:** Define la columna y dirección de ordenamiento.
- **if_condition:** Establece una condición `IF` para consultas `UPDATE` o `DELETE`.
- **delete:** Especifica que el tipo de consulta es `DELETE`.
- **build:** Construye la cadena SQL final, concatenando los elementos configurados.

Este diseño permite construir consultas SQL complejas de manera fluida, usando una sintaxis encadenada, lo cual es útil para aplicaciones que requieren consultas dinámicas y personalizadas.

4.4.2. Ejemplo de uso

```
let query = QueryBuilder::new("SELECT", "flight")
    .select(vec!["id", "status", "location"])
    .where_condition("status = 'delayed'", None)
    .order_by("id", Some("DESC"))
    .build();

let expected = "SELECT id, status, location FROM flight WHERE status = 'delayed' ORDER BY id DESC";
assert_eq!(query, expected);
```

5. Instrucciones de uso

Inicilización de múltiples clientes:

■ Paso 1: Número de clientes

Al ejecutar la aplicación, se solicita ingresar el número de clientes que se desean inicializar. Este número determina cuántos clientes se conectarán al servidor Cassandra.

```
let cant_clients = get_user_data("Enter the number of clients: ")
```

■ Paso 2: Ingresar IP de conexión

A continuación, para cada cliente, se solicita ingresar la dirección IP y el puerto del servidor al que debe conectarse. Esto debe hacerse uno por uno. La IP y el puerto se proporcionan en el formato: ip:puerto.

```
for _ in 0..cant_clients {
    let node = get_user_data("FULL IP (ip:port): ");
    let mut client = CassandraClient::new(&node)?;
    client.start_up()?;
    authenticate_client(&mut client);
    clients.push(client);
}
```

■ Paso 3: Inicialización de los clientes

Al momento de inicializar el cliente se pedirán las credenciales hasta que sean validas.


```
loop {  
    let user = get_user_data("Enter the user: ");  
    let password = get_user_data("Enter the password: ");  
    let Err(e) = client.authenticate(&user, &password) else {  
        break;  
    };  
}
```

Modelacion de tablas

El diseño de las tablas de la base de datos en Cassandra sigue un enfoque orientado a la eficiencia en el manejo y consulta de información de vuelos y aeropuertos, teniendo en cuenta las características de Cassandra como un sistema distribuido y de alto rendimiento. La estructura propuesta está pensada para soportar consultas rápidas y escalabilidad sin sacrificar la integridad de los datos.

6. Keyspace

```
CREATE KEYSPACE aviation WITH replication = {  
    'class': 'SimpleStrategy',  
    'replication_factor': 3  
};
```

El keyspace `aviation` se crea con un factor de replicación de 3, lo que significa que cada dato será replicado en tres nodos del clúster. Esto proporciona redundancia y disponibilidad, un principio clave en un sistema distribuido como Cassandra.

7. Tablas

7.1. Airports

La tabla `airports` almacena la información básica sobre cada aeropuerto, como su código, nombre y ubicación (latitud y longitud).

```
CREATE TABLE aviation.airports (  
    code text PRIMARY KEY,  
    name text,  
    position_lat decimal,  
    position_lon decimal  
);
```

`code` se usa como clave primaria, lo que asegura que cada aeropuerto tenga un identificador único.

7.2. Flights by airport

La tabla `flights_by_airport` almacena la información detallada de cada vuelo, incluyendo el estado, horarios de salida y llegada, información de posición, altitud, velocidad y nivel de combustible. Tiene como partition key el código del aeropuerto ya sea como aeropuerto de salida o de llegada.

```
CREATE TABLE aviation.flights_by_airport_detailed (  
    airport_code text,          -- Código del aeropuerto (salida o llegada)  
    flight_code text,           -- Código del vuelo  
    departure_airport text,     -- Aeropuerto de salida  
    arrival_airport text,      -- Aeropuerto de llegada  
    departure_time time,        -- Hora de salida  
    arrival_time time,         -- Hora de llegada  
    status text,               -- Estado del vuelo  
    position_lat decimal,      -- Latitud actual  
    position_lon decimal,      -- Longitud actual  
    altitude int,              -- Altitud actual
```

```
speed int,                -- Velocidad actual
fuel_level int,           -- Nivel de combustible actual
PRIMARY KEY ((airport_code), flight_code)
);
```

7.3. Relaciones entre Tablas

Las tablas `airports` y `flights_by_airport` están relacionadas a través del código de aeropuerto (`airport_code`), que se utiliza tanto en la tabla de aeropuertos como en la tabla de vuelos para vincular la información de los vuelos con los aeropuertos correspondientes. Esto permite realizar consultas eficientes para obtener información sobre los vuelos que salen o llegan a un aeropuerto específico. Es importante aclarar que la información de los vuelos va a estar duplicada ya que tiene que estar para ambos aeropuertos de llegada y salida pero esto nos permite acceder con una simple consulta a todos los vuelos de un aeropuerto cosa que es muy necesaria para nuestro uso específico en la aplicación de vuelos.

8. Ejemplos de Consultas

A continuación se muestran algunos ejemplos de consultas que interactúan con la base de datos de vuelos y aeropuertos:

8.1. Insertar Aeropuertos

En este ejemplo se insertan registros para dos aeropuertos en la tabla `airports`.

```
INSERT INTO aviation.airports (code, name, position_lat, position_lon)
VALUES ('EZE', 'Aeropuerto Internacional Ministro Pistarini', -34.812, -58.535);

INSERT INTO aviation.airports (code, name, position_lat, position_lon)
VALUES ('MIA', 'Miami International Airport', 25.7959, -80.2870);
```

8.2. Insertar Datos en la Tabla `flights_by_airport`

Este ejemplo inserta registros en la tabla `flights_by_airport`, asociando un vuelo con los aeropuertos de salida y llegada. Este modelo aprovecha la duplicación de datos para optimizar las consultas, permitiendo consultar vuelos por aeropuerto de forma eficiente.

```
-- Insertar un vuelo asociado al aeropuerto de salida 'EZE'
INSERT INTO aviation.flights_by_airport (
    airport_code, flight_code, position_lon, position_lat, altitude, speed,
    fuel_level, status, departure_airport, departure_time, arrival_airport,
    arrival_time, arrival_position_lon, arrival_position_lat
) VALUES (
    'EZE', 'AR1003', -34.812, -58.535, 0.0, 600.0, 100.0, 'OnTime',
    'EZE', '23:00:00', 'MIA', '05:00:00', 25.7959, -80.2870
);

-- Insertar el mismo vuelo asociado al aeropuerto de llegada 'MIA'
INSERT INTO aviation.flights_by_airport (
    airport_code, flight_code, position_lon, position_lat, altitude, speed,
    fuel_level, status, departure_airport, departure_time, arrival_airport,
    arrival_time, arrival_position_lon, arrival_position_lat
) VALUES (
    'MIA', 'AR1003', -34.812, -58.535, 0.0, 600.0, 100.0, 'OnTime',
    'EZE', '23:00:00', 'MIA', '05:00:00', 25.7959, -80.2870
);
```

8.3. Consultas de Selección

A continuación, se presentan consultas para obtener información sobre vuelos asociados a aeropuertos:

```
-- Obtener todos los vuelos asociados al aeropuerto 'EZE'
SELECT * FROM aviation.flights_by_airport
WHERE airport_code = 'EZE';

-- Seleccionar vuelos específicos asociados a 'EZE'
SELECT * FROM aviation.flights_by_airport
WHERE airport_code = 'EZE'
  AND (flight_code = 'AR1003' OR flight_code = 'AR1004' OR flight_code = 'AR1005');

-- Ordenar los vuelos por su código en el aeropuerto 'EZE'
SELECT * FROM aviation.flights_by_airport
WHERE airport_code = 'EZE'
ORDER BY flight_code;
```

8.4. Descripción de las Consultas

- **Insertar Aeropuertos:** Se inserta información básica de los aeropuertos en la tabla `airports`.
- **Insertar Vuelos:** Cada vuelo se asocia a los aeropuertos de salida y llegada. Esto permite una búsqueda eficiente, ya que los datos están duplicados según las necesidades del modelo no relacional.
- **Consultas de Selección:** Recuperan información sobre vuelos en un aeropuerto específico, permitiendo filtrar por códigos de vuelo específicos o aplicando un ordenamiento.

Flight App

El **FlightApp** es el componente principal de la interfaz de usuario (UI) del sistema. Su propósito es integrar y gestionar la información de vuelos, aeropuertos y mapas, permitiendo una interacción eficiente entre los datos del sistema y la visualización gráfica. Utiliza **egui** para la interfaz gráfica y **walkers** para la gestión del mapa.

9. Estructura

```
pub struct FlightApp {  
    pub airports: Airports,  
    pub selected_airport_code: Arc<Mutex<Option<String>>>,  
    pub flights: Flights,  
    pub selected_flight: Arc<Mutex<Option<FlightSelected>>>,  
    pub updater: AppUpdater,  
    // Map  
    pub tiles: HttpTiles,  
    pub map_memory: MapMemory  
}
```

Algunos elementos están manejados como una opción protegida por un `Arc<Mutex>` para la sincronización entre hilos.

9.1. Paneles

La interfaz de usuario se organiza en diferentes paneles, cada uno encargado de mostrar tipos específicos de información. A continuación, se describen los paneles principales de la aplicación:

9.1.1. Panel del Mapa

El **Panel del Mapa** es el encargado de mostrar la ubicación de los vuelos y aeropuertos sobre un mapa interactivo. Este panel utiliza la librería **walkers** para renderizar el mapa en tiempo real. Permite que los vuelos se muevan sobre el mapa conforme avanzan y que los usuarios interactúen con ellos de manera dinámica. Además, los vuelos y aeropuertos pueden usarse como botones para acceder a utilidades adicionales.

9.1.2. Panel de Información

El **Panel de Información** muestra información detallada sobre los vuelos y aeropuertos. Este panel permite a los usuarios obtener datos específicos acerca de los vuelos, como su estado, el origen y destino, y detalles adicionales. También ofrece la opción de deseleccionar un vuelo o aeropuerto, permitiendo al usuario modificar su selección de manera intuitiva.

9.2. Flights

La sección de **Flights** almacena y gestiona la información sobre los vuelos. Cada vuelo tiene datos como su código, hora de salida, destino, y otros detalles. Estos vuelos son actualizados constantemente y se visualizan en el panel de información para su estado y en el panel del mapa para su posición, permitiendo a los usuarios ver todos los vuelos disponibles en un aeropuerto.

```
pub struct Flights {  
    pub flights: Arc<Mutex<Vec<Flight>>>,  
    pub on_flight_selected: Arc<Mutex<Option<FlightSelected>>>,  
}
```

Implementa el trait Plugin para mostrar cada vuelo en el mapa.

```
impl Plugin for &mut Flights {  
    /// Dibuja los vuelos en la pantalla  
    fn run(&mut self, response: &Response, painter: Painter, projector: &Projector) {  
        self.draw_flights(response, &painter, projector);  
    }  
}
```

9.2.1. Flight

Cada vuelo está representado por la siguiente estructura:

```
pub struct Flight {  
    // weak consistency  
    pub position: (f64, f64),  
    pub arrival_position: (f64, f64),  
    // strong consistency  
    pub code: String,  
    pub status: FlightState,  
    pub arrival_airport: String,  
}
```

Cabe destacar que la información guardada en la estructura de Flight solo almacena la necesaria para visualizar en la pantalla y mostrar la información general. Para ver la información completa se deberá hacer uso de Flight Selected.

9.3. Airports

La sección de **Aeropuertos** gestiona la información relacionada con los aeropuertos disponibles en el sistema. Los usuarios pueden seleccionar un aeropuerto específico para visualizar los vuelos que están saliendo o llegando a dicho aeropuerto.

```
pub struct Airports {  
    pub airports: HashMap<String, Airport>,  
    pub selected_airport_code: Arc<Mutex<Option<String>>>,  
    pub selected_flight: Arc<Mutex<Option<FlightSelected>>>,  
}
```

En esta estructura, `airports` almacena un mapa de aeropuertos identificados por su código, mientras que `selected_airport_code` y `selected_flight` mantienen el estado del aeropuerto y vuelo seleccionados por el usuario.

9.3.1. Plugin de Airports

El siguiente Plugin se encarga de manejar las interacciones y actualizaciones visuales relacionadas con los aeropuertos. Existen tres posibles comportamientos:

```
impl Plugin for &mut Airports {  
    /// Could be three cases:  
    /// 1. Draw the selected flight airports
```

```
/// 2. Draw the selected airport
/// 3. Draw all airports if there is no selected airport or flight
fn run(&mut self, response: &Response, painter: Painter, projector: &Projector)
}
```

9.3.2. Dibujar Aeropuertos

La función `draw` se encarga de representar gráficamente los aeropuertos en la interfaz.

```
/// Draw the airport in the screen with its icon and information when hovering
/// If the airport is clicked, it will change the selected airport
pub fn draw(
    &self,
    response: &Response,
    painter: Painter,
    projector: &Projector,
    selected_airport_code: &Arc<Mutex<Option<String>>>,
    selected_flight: &Arc<Mutex<Option<FlightSelected>>>,
) {
    self.draw_icon_airport(painter.clone(), projector);
    self.clickeable_airport(response, projector, selected_airport_code, selected_flight);
    self.holdeable_airport(response, painter, projector);
}
```

9.4. Flight Selected

La estructura **FlightSelected** gestiona el caso en que se selecciona un vuelo. Guarda la información completa del vuelo, no solo la necesaria para mostrar en pantalla. Cuando existe un vuelo seleccionado, el panel de información listará únicamente la información de este, y el panel de mapa dibujará ambos aeropuertos de salida y llegada, además de una línea que simboliza el recorrido desde la posición actual del avión hasta el aeropuerto de llegada.

```
pub struct FlightSelected {
    // strong consistency
    pub status: FlightStatus,
    // weak consistency
    pub info: FlightTracking
}
```

9.4.1. Estructuras Internas

```
pub struct FlightStatus {
    pub code: String,
    pub status: FlightState,
    pub departure_airport: String,
    pub arrival_airport: String,
    pub departure_time: String,
    pub arrival_time: String,
}

pub struct FlightTracking {
    pub position: (f64, f64),
    pub arrival_position: (f64, f64),
    pub altitude: f64,
    pub speed: f32,
    pub fuel_level: f32,
}
```

10. Instrucciones de uso

Despues de hacer el cargo run pide todo lo necesario para inicilizar un solo cliente como haria el [Thread Pool Client](#) si le pidieras un solo cliente.

Flight Simulator

El *Flight Simulator* genera datos en tiempo real para simular vuelos en curso. Actualiza parámetros como posición, altitud, velocidad y nivel de combustible mediante ecuaciones que emulan el comportamiento real de los aviones. Los datos generados son enviados al sistema mediante comunicación TLS para su visualización en la aplicación gráfica.

Functions

10.1. Insert Single Flight

Esta función pide los datos con consistencia fuerte al usuario y luego agrega el vuelo a la base de datos.

```
let flight = get_flight_data();
simulator.insert_single_flight(&flight, thread_pool);
flights.insert(flight.get_code(), flight);
```

10.2. Restart Flight

La función *Restart Flight* permite reiniciar un vuelo desde su punto de origen, restaurando todos los parámetros del vuelo a su estado inicial. Esto es útil si se desea comenzar nuevamente una simulación o probar diferentes condiciones sin tener que reiniciar toda la aplicación.

```
/// Restart with initial values and set the position
pub fn restart(&mut self, position: (f64, f64)) {
    self.set_position(position);
    self.set_altitude(0.0);
    self.set_speed(0.0);
    self.set_fuel_level(gen_random(80.0, 100.0));
    self.set_status(FlightState::OnTime);
}
```

10.3. Update Flight

La función update progress simula el avance del vuelo basándose en el tiempo de paso (step). Primero, verifica si el vuelo ha llegado a su destino, comparando su posición actual con la posición de destino. Si el vuelo no ha llegado, se actualiza la fase de vuelo. Según la fase del vuelo, la función ajusta la altitud, la velocidad y el combustible, utilizando las funciones correspondientes. Finalmente, se actualiza la posición del vuelo y se verifica si ha alcanzado el objetivo de destino.

```
/// Update the flight progress based on the arrival position and the step time
pub fn update_progress(&mut self, step: f32) {
    if self.get_position() == self.get_arrival_position() {
        return;
    }
    self.update_phase();
    let phase = self.get_phase();
    phase.update_altitude(self, step*60.0);
    phase.update_speed(self, step*60.0);
    phase.update_fuel(self, step);
    self.update_position(step);
    self.update_status_if_target_reached();
}
```

10.4. Flight Updates Loop

El sistema actualiza continuamente los vuelos en intervalos regulares. Para cada vuelo, calcula la posición de su aeropuerto de llegada y actualiza el progreso de vuelo en paralelo utilizando un conjunto de hilos. Luego, espera a que todas las actualizaciones se completen antes de pasar al siguiente ciclo de actualización, asegurando que la interfaz de usuario permanezca responsiva mientras los datos se actualizan en segundo plano.

```
/// Ciclo que actualiza los vuelos cada intervalo de tiempo
loop {
  let flights =
    self.get_selected_flights(&airport_codes, &flight_codes_by_airport, thread_pool);
  for mut flight in flights.into_iter() {
    let _ = thread_pool.execute(move |frame_id, client| {
      flight.update_progress(step);
      let _ = Self.update_flight(client, &flight, &frame_id);
    });
  }
  thread_pool.join();
  thread::sleep(Duration::from_millis(interval));
}
```

11. Instrucciones de uso

Después de ejecutar el comando `cargo run`, el programa solicitará toda la información necesaria para la [inicialización del Thread Pool Client](#).

Una vez que la configuración inicial esté completa, el programa mostrará un menú con las siguientes opciones:

- **1. Agregar vuelos para un aeropuerto:** Se solicitará el código de un aeropuerto, y el sistema añadirá todos los vuelos relacionados con ese aeropuerto.
- **2. Agregar un solo vuelo:** Puedes agregar un vuelo individual proporcionando los detalles del vuelo (código, aeropuertos de salida y llegada, y horarios).
- **3. Iniciar el ciclo de actualizaciones de vuelos:** Esta opción comenzará un ciclo donde los vuelos seleccionados se actualizarán periódicamente según el intervalo y el paso de tiempo que determines.
- **4. Salir:** Termina la ejecución del simulador.

Para elegir una opción, ingresa el número correspondiente y presiona Enter.

11.1. Agregar Vuelos para un Aeropuerto

Si eliges la opción **1**, el programa te pedirá que ingreses el código del aeropuerto:

Enter the airport code:

Luego, el sistema recuperará todos los vuelos asociados a ese aeropuerto y los añadirá al simulador.

11.2. Agregar un Solo Vuelo

Si eliges la opción **2**, se te pedirá que ingreses los datos del vuelo de la siguiente manera:

- **Código del vuelo:** Ingresar un código único para el vuelo.
- **Código del aeropuerto de salida:** Ingresar el código del aeropuerto desde el cual sale el vuelo.
- **Código del aeropuerto de llegada:** Ingresar el código del aeropuerto de destino.
- **Hora de salida:** Ingresar la hora de salida en formato HH:MM:SS.
- **Hora de llegada:** Ingresar la hora de llegada en formato HH:MM:SS.

Una vez que ingreses estos datos, el vuelo será añadido al simulador.

11.3. Iniciar el Ciclo de Actualizaciones de Vuelos

Si eliges la opción **3**, el programa te pedirá que ingreses dos parámetros:

1. **Paso de tiempo (Step time):** Ingresar el paso de tiempo para la actualización de los vuelos (en horas).

Enter the step time:

2. **Intervalo de tiempo (Interval time):** Ingresar el intervalo de tiempo entre cada actualización de vuelo (en milisegundos).

Enter the interval time:

Después de ingresar estos valores, el sistema comenzará a actualizar los vuelos en tiempo real e ira mostrando cuales son los vuelos que van llegando a sus destinos hasta que ya no quede ninguno en vuelo y vuelva al menu de opciones principal.

11.4. Salir

Para salir del simulador, selecciona la opción **4**. Esto detendrá la ejecución del programa.

Test Client

El Test Client modela un cliente que se conecta a uno de los nodos. Se maneja mediante la terminal, donde, a través de distintos comandos, se pueden enviar requests a la base de datos.

Las requests pueden estar predefinidas, como el **STARTUP** u **OPTIONS**, o también pueden ser armadas dinámicamente, como en el caso de las queries. Sin embargo, el header del frame siempre es fijo.

El cliente funciona en un loop continuo donde se solicita al usuario que ingrese las distintas requests que desea enviar, permitiendo probar una amplia variedad de operaciones.

12. Instrucciones de Uso

Para iniciarlo, hay que correr el comando **cargo run**, y luego, cuando lo solicite, ingresar la IP completa (**ip:puerto**) del nodo al que se desea conectar. A partir de ahí, se pueden ir ingresando requests hasta finalizar la ejecución del cliente. Por pantalla se mostrará lo recibido de parte del nodo.

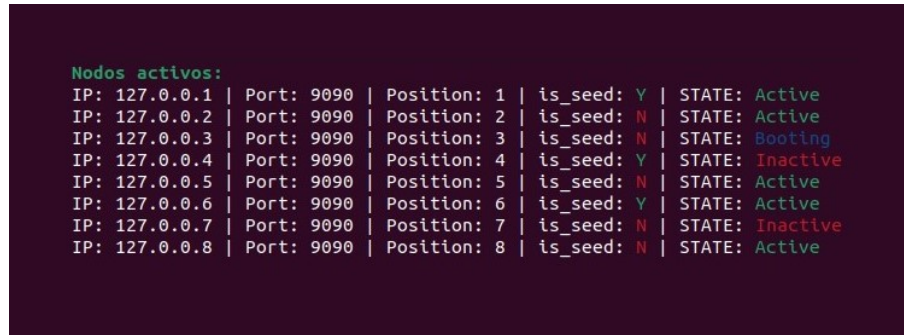
A continuación, se listan las distintas opciones de requests disponibles:

1. **startup**: Este comando enviará una **STARTUP** request al nodo.
2. **admin**: Este comando autorizará con credenciales de administrador en el nodo enviando un **AUTH_RESPONSE**.
3. **auth_response**: Este comando permite autorizarse con credenciales propias enviando un **AUTH_RESPONSE**. Al escribirlo, aparecerá un mensaje solicitando las credenciales, las cuales se deben ingresar en el formato **usuario:contraseña**.
4. **options**: Este comando enviará una **OPTIONS** request al nodo solicitando las opciones de configuración.
5. **query**: Este comando permite enviar una **QUERY** request. El sistema pedirá primero ingresar el texto de la query y luego la consistencia requerida, donde 1 es **One**, 4 es **Quorum** y 5 es **All**.
6. **queries**: Este comando permite enviar una lista de **QUERY** request. El sistema pedirá primero ingresar el path del archivo de queries del cual leera linea por linea y las ejecutara. Luego pedira la consistencia general requerida con el mismo formato explicado en query..
7. **prepare**: Este comando enviará una **PREPARE** request al nodo. Solicitará únicamente el texto de la query, de forma similar a la **QUERY** request.
8. **execute**: Este comando enviará una **EXECUTE** request al nodo. Se pedirá primero el **id** de la query a ejecutar y luego la consistencia requerida.

Node Handler

El Node Handler es una herramienta que permite la visualización en vivo del estado de todos los nodos en el cluster. Su lógica principal consiste únicamente en leer una lista de nodos y mostrarla en la terminal.

A continuación, se muestra una imagen representativa de cómo se ve corriendo:

A screenshot of a terminal window with a dark purple background. It displays a list of active nodes with their IP addresses, ports, positions, seed status, and states. The text is color-coded: green for 'Active', red for 'Inactive', and blue for 'Booting'.

```
Nodos activos:
IP: 127.0.0.1 | Port: 9090 | Position: 1 | is_seed: Y | STATE: Active
IP: 127.0.0.2 | Port: 9090 | Position: 2 | is_seed: N | STATE: Active
IP: 127.0.0.3 | Port: 9090 | Position: 3 | is_seed: N | STATE: Booting
IP: 127.0.0.4 | Port: 9090 | Position: 4 | is_seed: Y | STATE: Inactive
IP: 127.0.0.5 | Port: 9090 | Position: 5 | is_seed: N | STATE: Active
IP: 127.0.0.6 | Port: 9090 | Position: 6 | is_seed: Y | STATE: Active
IP: 127.0.0.7 | Port: 9090 | Position: 7 | is_seed: N | STATE: Inactive
IP: 127.0.0.8 | Port: 9090 | Position: 8 | is_seed: N | STATE: Active
```

Figura 6: Ejemplo de la salida del Node Handler en la terminal.

13. Instrucciones de Uso

Esta herramienta lee la metadata de uno de los nodos, actualizando así el estado del cluster.

Para ejecutarla, se debe usar el siguiente comando:

```
cargo run
```

Cuando la herramienta lo solicite, se debe proveer el path absoluto al archivo de metadata de alguno de los nodos en el cluster. El path a partir del directorio del nodo siempre es fijo, por lo que debe seguir el formato:

```
path-to-node/node/src/meta_data/nodes/metadata.json
```

Por ejemplo, si el directorio base del nodo está en el escritorio, el path podría ser:

```
Desktop/node/src/meta_data/nodes/metadata.json
```