GRUPO FERRUM: AEROLÍNEAS RÚSTICAS - 2C 2024

INFORME DEL PROYECTO: AEROLÍNEAS RÚSTICAS

1. Introducción general

En el presente trabajo práctico se desarrolló un sistema de control de vuelos global para **Aerolíneas Rústicas**, con el objetivo de gestionar eficientemente tanto vuelos nacionales como internacionales mediante la implementación de una base de datos distribuida escalable y tolerante a fallos. Este sistema fue diseñado para satisfacer las demandas de consistencia y disponibilidad necesarias en un entorno distribuido.

2. Metodología de trabajo

2.1 Investigación Inicial

El desarrollo de este proyecto fue acompañado de una investigación exhaustiva y continua, basada en diversas fuentes, tales como videos disponibles en internet, los recursos oficiales de Apache Cassandra (documentación y tutoriales) y materiales complementarios. Adoptamos un enfoque incremental, en el cual los conceptos necesarios para cada etapa del desarrollo fueron identificados y estudiados conforme avanzábamos en la implementación. Este enfoque práctico fue clave para enfrentar desafíos específicos y aplicar las soluciones correspondientes de manera eficiente.

2.1.1 Bases de datos distribuidas

La investigación comenzó con los fundamentos de las bases de datos distribuidas, enfocándonos en los conceptos esenciales de Cassandra y cómo integrarlos en nuestro sistema. Entre los puntos clave que aprendimos y aplicamos se encuentran:

- **Consistent Hashing**: Comprendimos su importancia para distribuir datos de manera uniforme entre los nodos del clúster y reducir la redistribución de datos cuando se añaden o eliminan nodos. Este concepto resultó fundamental para implementar un sistema escalable y balanceado.
- **Replication Factor**: Investigamos cómo replicar datos en múltiples nodos para asegurar alta disponibilidad y tolerancia a fallos. Implementamos estrategias de replicación adaptadas a los distintos niveles de consistencia requeridos.
- **Consistency Level**: Aprendimos sobre los diferentes niveles de consistencia (e.g., QUORUM, ONE, ALL) y cómo impactan el rendimiento y la confiabilidad del sistema. Esto nos permitió manejar estados críticos de vuelos con strong consistency y datos de tracking en tiempo real con weak consistency.
- **Read Repair**: Analizamos cómo este mecanismo corrige discrepancias entre réplicas durante las lecturas. Este aprendizaje fue aplicado para garantizar la consistencia eventual incluso cuando algunos nodos fallan.

La investigación fue apoyada por tutoriales, videos explicativos y ejemplos prácticos que nos guiaron en la implementación de estas características.

2.1.2 Cassandra Query Language (CQL)

El estudio del lenguaje de consultas CQL fue otro pilar del proyecto, ya que este define cómo interactuar con los datos almacenados en la base de datos distribuida. A través de la documentación oficial y tutoriales prácticos, aprendimos a:

- Diseñar tablas con particiones y clustering keys optimizadas para consultas frecuentes, como búsquedas por aeropuerto, fecha o estados de vuelo.
- Implementar operaciones CRUD (Create, Read, Update, Delete) mediante comandos como INSERT, SELECT y UPDATE. Esto incluye aprender a mapear los requerimientos del sistema a consultas eficientes.
- Configurar niveles de consistencia en las consultas, adaptándose a los requerimientos específicos de cada tipo de dato.

A medida que desarrollamos estas funciones, encontramos problemas relacionados con el modelado de datos distribuidos, lo que nos llevó a iterar y refinar el diseño para optimizar el rendimiento.

2.1.3 Protocolo Gossip

El protocolo Gossip, que permite la comunicación entre nodos del clúster, fue otra área crítica de investigación. Aprendimos a:

- Sincronizar estados entre nodos, compartiendo información sobre disponibilidad, réplicas y cambios en la topología del clúster.
- Identificar y manejar nodos fallidos mediante las métricas de Gossip.

Esta comprensión fue posible gracias al uso de diagramas técnicos y explicaciones detalladas en los recursos oficiales y videos que descomponen los flujos de comunicación.

2.2 Planteamiento del problema

A lo largo del desarrollo, nos encontramos con problemas específicos que nos llevaron a profundizar en áreas adicionales para encontrar soluciones:

- **Relojes y sincronización**: Tuvimos que investigar sobre el manejo de timestamps distribuidos para garantizar que las operaciones respetaran el orden lógico, especialmente en entornos donde múltiples nodos escriben datos simultáneamente.
- **Concurrencia y multithreading**: Al implementar el simulador de vuelos y el manejo de múltiples clientes en paralelo, enfrentamos problemas relacionados con el acceso concurrente a recursos compartidos. Esto nos llevó a estudiar conceptos como **locks**, estrategias de sincronización y el uso de herramientas en Rust como Mutex.
- **Consistencia eventual**: Encontramos casos en los que las réplicas podían estar desactualizadas, lo que nos llevó a profundizar en estrategias como Read Repair y cómo ajustar los niveles de consistencia para mitigar estos problemas.

- **Distribución de datos**: Durante la implementación de particiones y claves de clustering, enfrentamos retos para optimizar las consultas. Tuvimos que iterar sobre el diseño de las tablas y ajustar las claves primarias para mejorar el rendimiento.

Cada uno de estos problemas nos permitió aprender y aplicar soluciones específicas, enriqueciendo no sólo el proyecto, sino también nuestra comprensión de los sistemas distribuidos.

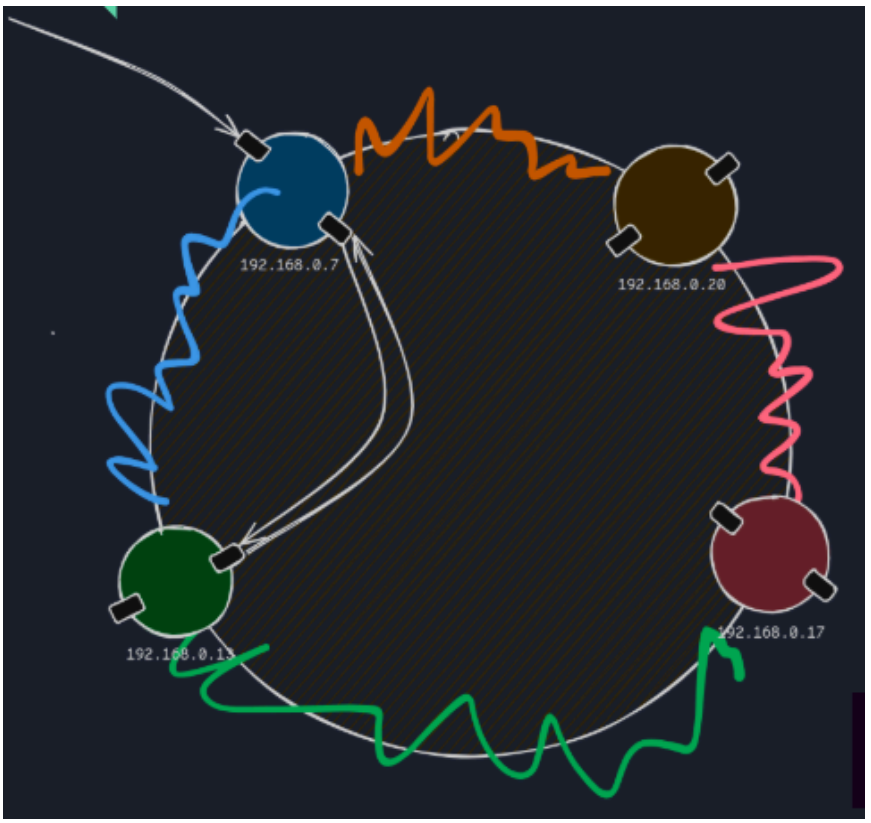
2.3 Conclusiones de la investigación inicial

- A lo largo de este proyecto, aprendimos tanto las ventajas como las desventajas de trabajar con bases de datos distribuidas. Entre las principales ventajas, destacamos la capacidad de estos sistemas para garantizar **alta disponibilidad y tolerancia a fallos**, lo que permite que los datos sigan siendo accesibles incluso ante la caída de uno o más nodos. Además, la distribución de datos en múltiples nodos mejora significativamente los tiempos de acceso, especialmente en entornos con alta concurrencia.
- También enfrentamos desafíos, como la **complejidad para diseñar esquemas de tablas** que optimicen consultas específicas, dada la necesidad de respetar las limitaciones de las claves de partición y clustering. Estas restricciones exigen un modelado cuidadoso y adaptado a los patrones de acceso esperados, lo que puede limitar la flexibilidad en algunos casos.
- Este equilibrio entre ventajas y desafíos nos permitió comprender a fondo las capacidades y limitaciones de las bases de datos distribuidas, así como su importancia en aplicaciones críticas como la gestión de vuelos.

3. Solución al problema

3.1 Anillo

En sistemas distribuidos como Cassandra, el **anillo lógico** es una representación abstracta utilizada para organizar y distribuir los nodos en un clúster. Cada nodo se identifica mediante una dirección única (en este caso, las direcciones IP como 192.168.0.7, 192.168.0.11, etc.) y se posiciona en un punto específico dentro del anillo.



La relación con **Consistent Hashing** es fundamental para lograr una distribución uniforme y eficiente de los datos en estos nodos.

Las principales características que definen al anillo son:

1. **Distribución de Nodos**:
    - Cada nodo en el clúster se posiciona en el anillo lógico según un valor hash único. Este valor se calcula utilizando un algoritmo de hash (como MD5 o MurmurHash) aplicado a su identificador.
    - El anillo permite visualizar cómo los nodos "cobijan" fragmentos de datos. Cada nodo es responsable de almacenar un rango específico de datos determinado por su posición en el anillo y el rango que hereda del nodo anterior.

2. **Distribución de Datos**:
    - Los datos se distribuyen en el anillo utilizando el mismo algoritmo de hash. Cada clave de dato genera un valor hash que define su ubicación en el anillo.
    - Los datos se asignan al nodo que sigue en el anillo al valor hash generado (esto se llama **nodo sucesor**). De esta manera, el sistema asegura que cada nodo almacene datos para un rango específico del espacio hash.

3. **Replicación**:

- Para garantizar la **tolerancia a fallos**, los datos no solo se almacenan en un nodo, sino que se replican en varios nodos consecutivos en el anillo. La cantidad de réplicas está determinada por el **Replication Factor** configurado.
- En el caso de un fallo en un nodo, los datos pueden recuperarse de las réplicas almacenadas en los nodos vecinos.

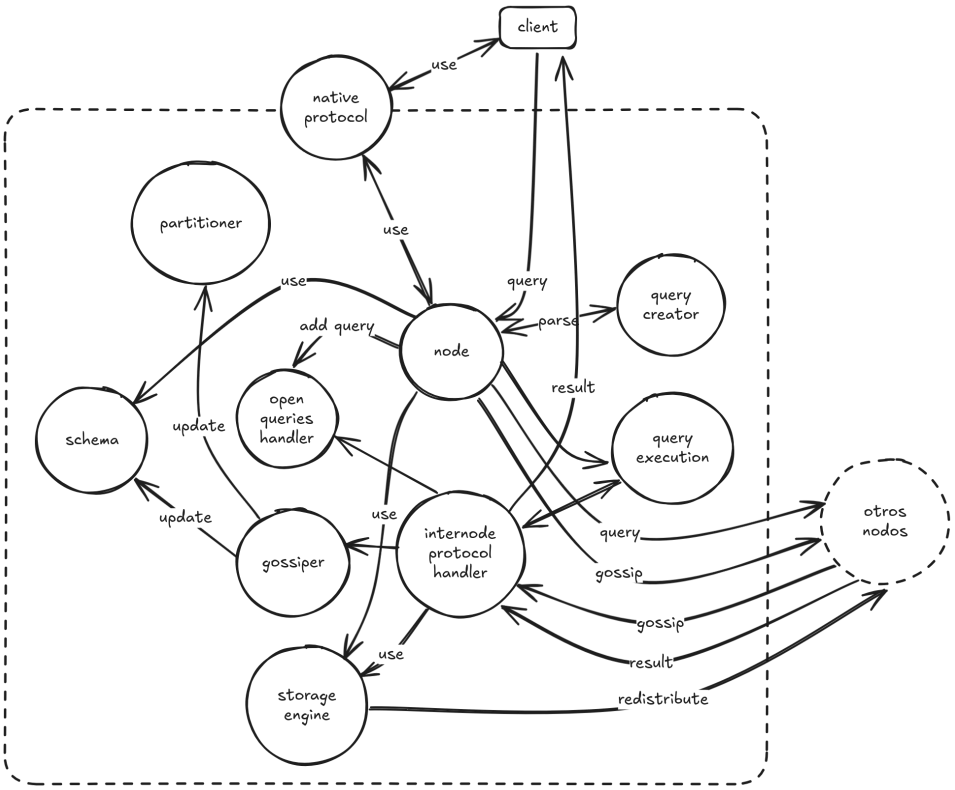
Relación con Consistent Hashing:

1. **Minimización de Redistribución de Datos**:
   - Cuando se agrega o elimina un nodo del anillo, el algoritmo de *Consistent Hashing* minimiza la cantidad de datos que deben moverse. Solo los datos que caen dentro del rango del nodo afectado necesitan ser redistribuidos, en lugar de redistribuir todos los datos del sistema.

2. **Equilibrio de Carga**:
   - Consistent Hashing asegura que los datos se distribuyan uniformemente entre los nodos, evitando que algunos nodos estén sobrecargados mientras otros están subutilizados.

3. **Escalabilidad**:
   - El sistema de anillo combinado con *Consistent Hashing* permite añadir o quitar nodos al clúster sin interrumpir el funcionamiento global. Esto es ideal para sistemas que necesitan crecer dinámicamente.

3.2 Estructura de un nodo



La imagen representa un esquema conceptual de un nodo en un sistema de base de datos distribuida inspirado en Cassandra. Cada nodo contiene varios componentes fundamentales que interactúan entre sí para manejar consultas, coordinar la comunicación con otros nodos, y gestionar el almacenamiento y recuperación de datos. A continuación, se explican los componentes mostrados (y algunos faltantes según la actualización mencionada):

3.2.1. Partitioner

El **Partitioner** es responsable de gestionar el anillo lógico del clúster. Sus principales funciones incluyen:

- **Hashing de claves**: Calcula un valor hash para cada clave de dato (por ejemplo, una clave primaria) y determina a qué nodo pertenece ese dato dentro del anillo.
- **Distribución de datos**: Decide qué nodos serán responsables de almacenar réplicas de un dato, en función del **Replication Factor** configurado.
- **Mapeo de nodos**: Mantiene un registro de qué nodos forman parte del anillo y sus respectivas responsabilidades en el espacio de claves.

3.2.2. Query Creator

Es el punto inicial de procesamiento de consultas enviadas por el cliente. Sus responsabilidades son:

- **Recepción de la consulta en formato texto (string)**: Toma la consulta cruda enviada por el cliente.
- **Parsing de la consulta**: Traduce la consulta textual a una estructura interna que puede ser procesada por los demás componentes del sistema. Esto incluye validar la sintaxis y construir un objeto "Query" con los datos estructurados.

3.2.3. Internode Protocol Handler

Este gestiona la comunicación entre nodos en el clúster. Sus funciones incluyen:

- **Procesamiento de mensajes internodales**: Decide cómo manejar mensajes que provienen de otros nodos, como solicitudes de replicación, sincronización de datos o confirmaciones.
- **Creación de mensajes internodales**: Utiliza el protocolo de comunicación definido (por ejemplo, Gossip) para generar mensajes que serán enviados a otros nodos. Estos mensajes pueden incluir datos de réplicas, actualizaciones de estado, entre otros.

3.2.4. Open Query Handler

Rastrea las consultas activas en el sistema. Este componente es crucial para gestionar la consistencia y las respuestas. Sus responsabilidades incluyen:

- **Seguimiento de consultas abiertas**: Una consulta se considera abierta desde el momento en que es recibida del cliente hasta que se alcanza el **Consistency Level** especificado o se detecta que no se podrá cumplir debido a errores.
- **Gestión de respuestas**: Almacena el estado de las respuestas recibidas de otros nodos para determinar si se alcanzaron las condiciones necesarias para cerrar la consulta.

3.2.5. Query Execution

Es el encargado de ejecutar las consultas y coordinar sus efectos en el sistema. Esto incluye:

- **Ejecución de consultas locales**: Aplica las modificaciones requeridas a las keyspaces y tablas dentro del nodo.
- **Coordinación de consultas distribuidas**: Envía consultas internodales a los nodos responsables de las réplicas, utilizando el protocolo de comunicación.
- **Actualización del estado del almacenamiento**: Trabaja junto con el **Storage Engine** para aplicar cambios en los datos.

3.2.6. Native Protocol

Es el mecanismo de comunicación entre los clientes y el clúster.
Este protocolo permite a los clientes enviar solicitudes y recibir respuestas del clúster.
Sirve como el estándar para la comunicación entre un cliente y un nodo Cassandra. Su propósito principal es proporcionar un canal eficiente para ejecutar comandos de base de datos, asegurando compatibilidad y optimización para operaciones distribuidas.
Sus funciones incluyen:

- **Recepción de solicitudes:** El protocolo maneja la conexión inicial entre el cliente y el clúster mediante un **handshake**. Este proceso incluye:
    - **Frame Startup:** Inicia la conexión e informa al servidor las capacidades del cliente, así como la versión del protocolo soportada.
    - **Frame Ready:** Confirma que el servidor está listo para aceptar solicitudes.
    - **Frames de autenticación:** Si el servidor requiere autenticación, estos frames gestionan el intercambio de credenciales entre el cliente y el nodo.
- **Creación y procesamiento de consultas:** Convierte las consultas escritas en **CQL (Cassandra Query Language)** por el cliente en un **frame Query**, un formato binario que Cassandra puede interpretar y procesar internamente.
- **Generación de respuestas:** El protocolo transforma los resultados internos de Cassandra en un **frame Result**, que el cliente puede interpretar fácilmente. Esto incluye:
    - **Resultados de SELECT:** Datos solicitados por el cliente.
    - **Mensajes de confirmación:** Para operaciones como INSERT, UPDATE o DELETE.
    - **Mensajes de error:** En caso de problemas como errores de sintaxis o fallos en la ejecución.
- **Autenticación y seguridad:**
    - Integra mecanismos para autenticar y autorizar clientes mediante frames dedicados. Esto asegura que solo usuarios autorizados puedan acceder al clúster.
    - Soporta encriptación para datos en tránsito utilizando TLS/SSL, proporcionando confidencialidad y seguridad en la comunicación.

3.2.7. Storage Engine

Es el encargado del almacenamiento físico y lógico de los datos. Este componente asegura que los datos se almacenen de manera eficiente y ordenada. Sus responsabilidades incluyen:

- **Modificación de datos**: Aplica los cambios validados por los componentes anteriores (como inserciones, actualizaciones y eliminaciones).

- **Persistencia**: Garantiza que los datos sean almacenados de forma duradera en disco, respetando las reglas de partición y replicación

- **Optimización del almacenamiento**: Organiza los datos de acuerdo con las **clustering columns**, lo que permite realizar búsquedas y consultas eficientes .Para optimizar la búsqueda en los datos almacenados en disco, se implementó un sistema de índices adicionales que aprovecha el ordenamiento natural de las tablas por la columna de clustering. Cada tabla cuenta con un archivo complementario llamado `{nombre archivo}_index`, que registra los rangos de bytes correspondientes a los valores únicos de la primera columna de clustering. Este índice permite identificar de manera precisa de qué byte a qué byte en el archivo principal se encuentran las filas asociadas a un valor específico de dicha columna. Por ejemplo, en una tabla con columnas `id` (clave primaria), `país` (primera columna de clustering) y `edad` (segunda columna de clustering), el archivo de índice contendrá únicamente los valores únicos de `país` y los rangos de bytes donde se encuentran almacenadas las filas de cada país en el archivo principal. Cuando se realiza una consulta `SELECT` filtrando por la columna `país`, se consulta primero el índice para obtener el rango de bytes asociado y luego se accede directamente a esa porción del archivo principal, evitando lecturas innecesarias y mejorando significativamente el rendimiento. Aunque esta estrategia ha incrementado el rendimiento de las consultas, también ha introducido una mayor complejidad lógica, ya que es necesario mantener actualizado el archivo de índices cada vez que se realizan operaciones como `INSERT`, `UPDATE` o `DELETE`. Esto implica llevar un control preciso de los desplazamientos en bytes asociados a cada clave para reflejar correctamente las modificaciones en el archivo principal. Este balance entre mejora de rendimiento y complejidad adicional ha sido clave para optimizar el acceso a los datos en un sistema distribuido.

3.2.8. Gossip

El protocolo **Gossip** es un mecanismo central en Cassandra utilizado para la comunicación interna entre los nodos de un clúster. Este protocolo, permite que los nodos compartan información de estado y asegurarse de que todos los nodos del clúster mantengan una versión actualizada del sistema. Gossip es crucial para garantizar la alta disponibilidad y la consistencia eventual en Cassandra.
Tiene como propósito principal facilitar la coordinación entre nodos en un entorno distribuido sin depender de un punto central.
Es clave para operaciones como:

- **Descubrimiento de nodos:** Identificar qué nodos están activos en el clúster.
- **Propagación de metadatos:** Compartir información sobre tokens, esquemas y datos replicados.
- **Detección de fallos:** Determinar qué nodos están caídos o inaccesibles.
- **Consistencia eventual:** Asegurar que todos los nodos tengan una vista sincronizada de la topología del clúster.

El protocolo Gossip utiliza un modelo descentralizado en el que los nodos intercambian información periódicamente.
Principales características:

1. **Intercambio periódico de mensajes:**
    - Cada nodo selecciona aleatoriamente a otros nodos para intercambiar información de estado.
    - Esta interacción ocurre en intervalos regulares (por defecto, cada segundo).
2. **Información propagada:**
    - **Estado del nodo:** Incluye datos como si un nodo está activo o caído.
    - **Versiones del esquema:** Asegura que todos los nodos mantienen un esquema de base de datos consistente.
3. **Algoritmo probabilístico:**
    - Gossip asegura que, después de varias rondas, la información de un nodo se propaga a todos los demás nodos del clúster. Este enfoque garantiza una comunicación eficiente incluso en clústeres grandes.
4. **Consistencia eventual:**
    - A medida que los nodos intercambian información, las discrepancias se resuelven, y todo el clúster converge hacia un estado consistente.

3.3 Implementación: Cassandra Simplificado

Nuestro sistema, aunque no es una réplica exacta de Cassandra, es una versión simplificada que mantiene y adapta las características principales del motor de base de datos distribuida. A continuación, explicamos cómo se han implementado estas características clave, detallando tanto las similitudes como las simplificaciones realizadas:

3.3.1 Consistent Hashing

La implementación de **Consistent Hashing** en nuestro sistema sigue los mismos principios que en Cassandra:

- Los datos se distribuyen uniformemente en un anillo lógico entre los nodos del clúster.
- Cada nodo es responsable de un rango específico del espacio hash, y las claves de datos se asignan al nodo sucesor del hash calculado.
- Esta característica permite escalabilidad y balance de carga, ya que al agregar o eliminar nodos solo se redistribuyen los datos que caen en el rango afectado.

Esta funcionalidad fue implementada de manera idéntica a Cassandra para garantizar la distribución eficiente de los datos.

3.3.2 Consistency Level

En nuestro sistema, el **Consistency Level** determina en qué momento se devuelve la respuesta al cliente tras una consulta. Hemos adaptado esta característica permitiendo los siguientes niveles de consistencia (entre otros):

- **ALL**: La respuesta se devuelve solo después de recibir confirmaciones de todos los nodos responsables (nodo primario y réplicas).
- **QUORUM**: La respuesta se devuelve una vez que la mayoría de los nodos responsables han respondido.
- **ONE**: La respuesta se devuelve tan pronto como un nodo haya respondido.

Si durante la ejecución de una query un nodo no responde, el sistema lo registra como una falla de comunicación y sigue procesando la consulta con los nodos restantes. Al finalizar, si se cumplen las condiciones del **Consistency Level**, la respuesta se devuelve al cliente. Si no se cumple, se informa al cliente que no se alcanzó el nivel de consistencia requerido.

3.3.3 Read Repair

La **Read Repair** en nuestro sistema se activa cuando se alcanza el nivel de consistencia configurado en una consulta de lectura. Durante este proceso:

1. Si se detecta que los datos devueltos por diferentes nodos no coinciden, el sistema identifica cuáles están desactualizados.
2. Se genera una consulta de actualización que se envía automáticamente a los nodos desincronizados para corregir sus datos.

Esto garantiza que las réplicas converjan hacia un estado consistente con el tiempo, mejorando la integridad del sistema.

3.3.4 Replication Factor

El **Replication Factor** define el número de nodos en los que se almacena una réplica de cada dato. En nuestro sistema:

- Este valor es un número configurado que influye en las decisiones del **Partitioner** al distribuir las claves de datos.
- Por ejemplo, si el **Replication Factor** es 3, el Partitioner seleccionará el nodo primario y dos nodos adicionales para almacenar réplicas de cada dato.

Esto asegura alta disponibilidad y tolerancia a fallos, ya que los datos permanecen accesibles incluso si un nodo falla.

3.3.5 Almacenamiento de Datos

En lugar de implementar el sistema complejo de almacenamiento de Cassandra, como **SSTables** y **memtables**, optamos por una solución simplificada:

- Cada tabla se almacena en un archivo **CSV**, que contiene todas las filas de la tabla.
- Cada dato tiene un timestap que representa el momento en el cual el Nodo Coordinador recibio la Query. Esto sirve para poder comparar datos y ver cual esta mas actualizado.
- Para optimizar las consultas, cada tabla tiene un archivo de índice adicional que almacena:
    - Los valores únicos de la **primera clustering column**.
    - Un mapeo que indica de qué byte a qué byte en el archivo CSV se encuentra cada valor de clustering.

- Al realizar un SELECT, el sistema consulta primero el archivo de índice, lo que permite buscar directamente en las ubicaciones relevantes del archivo principal, optimizando el tiempo de lectura.

Esta implementación simplificada permite consultas eficientes sin la complejidad de estructuras adicionales.

3.3.6 Conexiones TCP

Para reducir la sobrecarga en la comunicación entre nodos:

- Se mantienen ciertas conexiones **TCP** abiertas entre los nodos del clúster. Esto evita la necesidad de establecer y cerrar conexiones repetidamente, mejorando el rendimiento general del sistema.

Ventajas del sistema propuesto

Nuestro sistema mantiene las características principales de Cassandra, adaptándolas para simplificar la implementación sin sacrificar funcionalidad esencial. Las principales adaptaciones incluyen:

- Un almacenamiento más sencillo basado en archivos CSV e índices.
- Una gestión simplificada del protocolo Gossip para la caída y unión de nodos.
- Manejo de fallos durante una query, permitiendo que la consulta se procese con los nodos restantes y devolviendo el resultado según el **Consistency Level** alcanzado.
- Una implementación directa pero funcional de niveles de consistencia, read repair y replicación.

Estas decisiones permiten un sistema funcional que captura las ventajas de Cassandra, pero con una menor complejidad en su diseño interno.

3.4 Gossip

El **Protocolo Gossip** en nuestro sistema sigue la siguiente implementación:

- Los nodos comparten información, con 3 nodos aleatorios del clúster, periódicamente (cada 1 segundo) para mantenerse actualizados sobre el estado del clúster.
- También lo utilizamos para sincronizar los metadatos de las tablas y keyspaces, así como la topología del clúster y el estado de los nodos.

Decisiones de implementación:

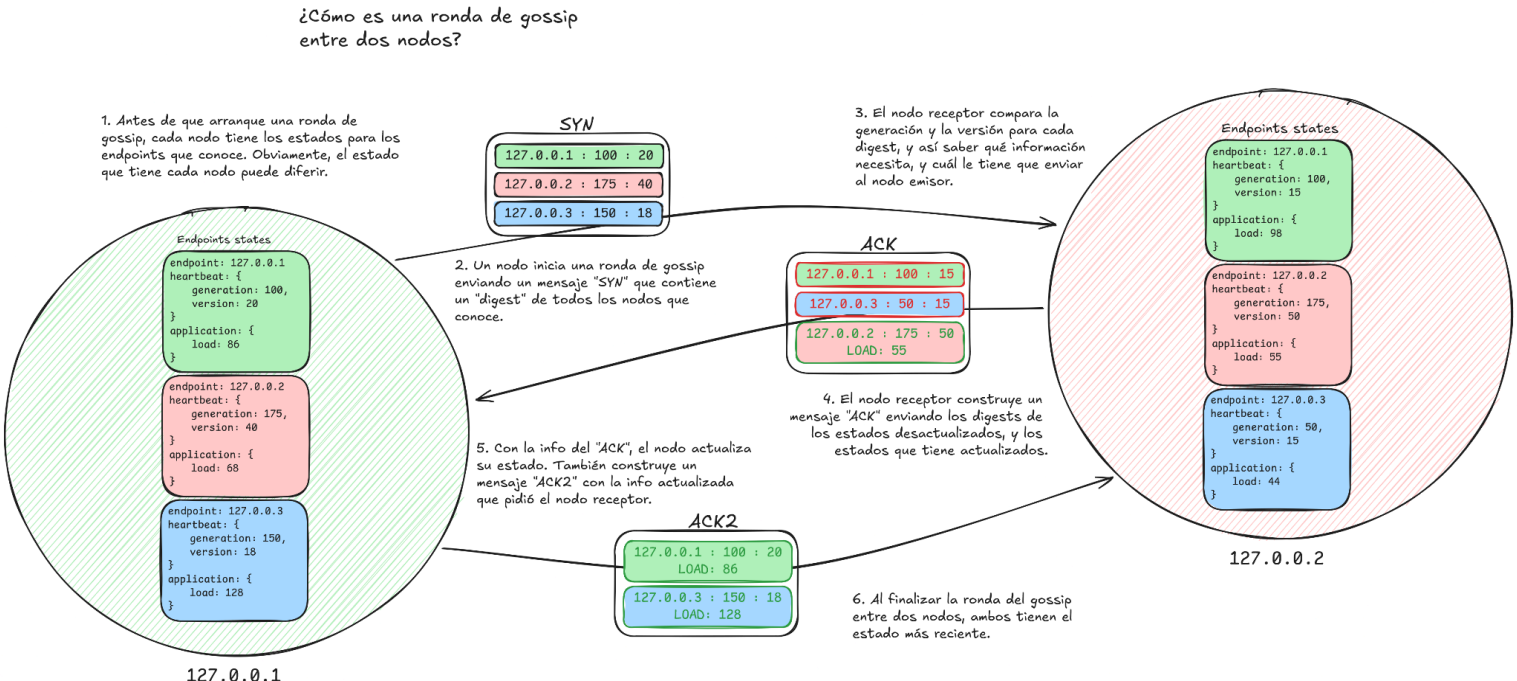
- **Caída de nodos**:
  - Si un nodo no responde durante el intercambio de mensajes Gossip con otros nodos, se registra como "inactivo".
  - Si esto ocurre repetidamente durante un número configurable de intentos, el nodo se considera "muerto" y se desconecta del clúster.
  - El nodo que no pudo comunicarse con el nodo inactivo, marca el estado de este nodo como *Dead* y empieza a propagarlo a los demás nodos del cluster (a través de gossip) para que todos se enteren.

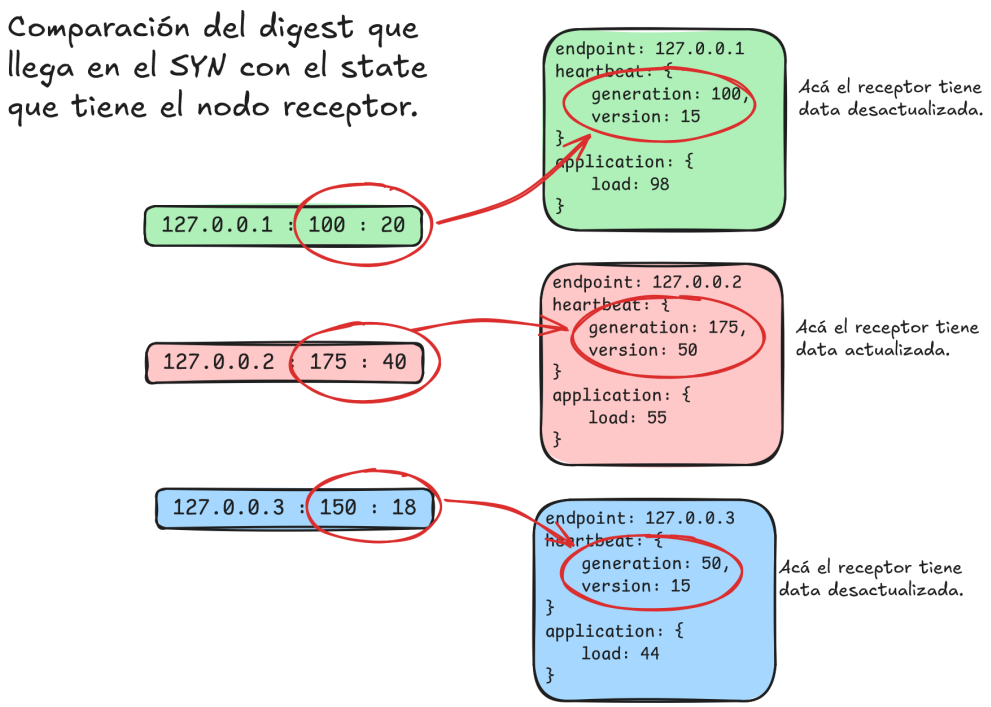
- Los datos almacenados en este nodo se redistribuyen entre los nodos restantes para asegurar que todas los datos estén disponibles.

- **Unión de nodos**:
  - Al unirse un nuevo nodo al clúster, éste hace gossip por primera vez con los *seed nodes* que se encargan de que este nuevo nodo conozca a los demás nodos del clúster.
  - Además, el sistema redistribuye automáticamente los datos para incluir al nuevo nodo en el anillo y asignarle los tokens que le corresponden.

- **Propagación del esquema**:
  - Cada vez que se realiza una modificación en el esquema de la base de datos, ya sea creando o eliminando keyspaces o tablas, el nodo coordinador que se encargó de esa query, actualiza su esquema local y luego a través de gossip informa a los demás nodos sobre el cambio en el esquema.
  - Al recibir por gossip la actualización de esquema, cada nodo actualiza su esquema local.
  - Los esquemas tiene un timestamp para saber cual es el más reciente y así los nodos pueden identificar cual es el que deben utilizar de referencia para actualizarse.

- **Ronda de Gossip**: En cada ronda de gossip, dos nodos intercambian los siguientes mensajes:
  - SYN: Es el primer mensaje que envía un nodo a otro para iniciar la ronda de gossip. En este se envía un resumen de información que este nodo tiene sobre los demás nodos del clúster (generación, versión o heartbeat y estado).
  - ACK: En respuesta a un SYN, un nodo envía un ACK que contiene:
    - Información sobre nodos que el otro nodo tiene desactualizada.
    - Resumen (o digests) de nodos, en pedido de la información más actualizada sobre ellos.
  - ACK2: En respuesta a un ACK, un nodo envía un ACK2 que contiene la información pedida en el ACK sobre nodos desactualizados.





3.7. Native Protocol

3.7.1 Integración y Procesamiento de Requests

Recepción de requests:

- Las requests de los clientes llegan a través de una conexión TCP establecida entre el cliente y uno de los nodos del clúster.
- Cada request se encapsula en un **frame** que incluye información como el tipo de operación, y cualquier dato necesario para la operación (por ejemplo, consultas CQL).

Deserialización de los frames:

- Los frames se descomponen en sus componentes utilizando las especificaciones del protocolo. Esto incluye:
    - **Headers:** Información sobre la versión del protocolo, longitud del mensaje.
    - **Cuerpo del mensaje:** Los datos reales de la operación, como la consulta o credenciales de autenticación.
- Este proceso asegura que los datos enviados por el cliente se conviertan en una estructura entendible por el sistema interno.

3.7.2 Construcción de Queries

Interpretación del frame Query:

- Las consultas enviadas por los clientes en CQL llegan encapsuladas como frames del tipo **Query**. Estos frames contienen:
    - El texto de la consulta.

- Opciones como el nivel de consistencia deseado o parámetros de consulta.

Parsing del CQL:

- El texto de la consulta se analiza y traduce a una representación interna. Esto incluye:
  - Identificación de la operación (SELECT, INSERT, UPDATE, DELETE, etc.).
  - Extracción de las tablas y columnas involucradas.
  - Validación preliminar de sintaxis y semántica.

Construcción de estructuras internas:

- Una vez procesada, la consulta se convierte en una estructura de datos optimizada para el clúster.

3.7.3 Validación y Comunicación

Validación de la consulta:

- Se revisa que la consulta cumpla con los siguientes criterios:
  - **Permisos:** Verificar que el usuario tenga los permisos necesarios para realizar la operación.
  - **Integridad del esquema:** Comprobar que las tablas, columnas y tipos de datos mencionados existan y sean consistentes con el esquema actual.
  - **Consistencia del nivel solicitado:** Asegurar que el nivel de consistencia requerido (por ejemplo, ONE, QUORUM) sea compatible con el estado del clúster.

Propagación de resultados:

- Los resultados obtenidos (datos para SELECT, confirmaciones para INSERT/UPDATE/DELETE, o errores) se encapsulan nuevamente en un **frame Result** y se devuelven al cliente.
- En caso de errores (como un nodo inaccesible o una violación de restricciones), se envían frames de tipo **Error** con los detalles correspondientes.

3.8 CQL

3.8.1 Soporte de Sintaxis de CQL

Nuestro sistema soporta la mayoría de las funcionalidades principales del **Cassandra Query Language (CQL)**, incluyendo:

- **Operaciones CRUD**:
  - INSERT: Para insertar datos en tablas.
  - SELECT: Para consultar datos, con soporte de cláusulas como WHERE, LIMIT y ciertas configuraciones de ORDER BY.

    - UPDATE: Para actualizar datos existentes, con soporte de condiciones como IF.
    - DELETE: Para eliminar filas o columnas específicas.

- **Definición de Esquema**:
    - CREATE TABLE: Para definir nuevas tablas con claves de partición y clustering keys.
    - DROP TABLE: Para eliminar tablas existentes.
    - ALTER TABLE: Para modificar esquemas, como agregar nuevas columnas.

3.8.2 Variantes y Limitaciones

- **Cláusulas soportadas**:
    - WHERE: Permite realizar filtros según claves primarias y clustering keys.
    - IF: Ofrece soporte para condiciones en operaciones como UPDATE y INSERT, garantizando que solo se ejecuten si las condiciones son verdaderas.
    - LIMIT: Para restringir el número de resultados devueltos por una consulta.
    - ORDER BY: Con soporte limitado para ordenar resultados según las clustering keys.
- **Tipos de datos**:
    - Se soporta la mayoría de los tipos simples como INT, TEXT, BOOLEAN, FLOAT, etc.
    - No se soportan tipos complejos como colecciones (list, map, set) o estructuras anidadas.
- **Características no implementadas**:
    - BATCHES: Las consultas batch para agrupar múltiples operaciones no fueron incluidas en esta implementación.
    - Algunas funciones avanzadas de CQL no se encuentran soportadas, como operaciones sobre índices secundarios o materialized views.

3.8.3 Conclusión sobre funcionalidad de CQL

Aunque nuestro sistema no implementa la totalidad de la funcionalidad avanzada de CQL, ofrece soporte suficiente para la mayoría de las operaciones necesarias en un entorno distribuido, enfocándose en un diseño eficiente y práctico para consultas comunes y manejo de datos en bases distribuidas.

3.9. Simulador de Vuelos

Esta es una aplicación de consola que permite simular múltiples vuelos de manera concurrente. El ciclo principal (`main`) escucha los comandos del usuario, como agregar vuelos, listar vuelos o aeropuertos, y actualizar la tasa de tiempo de la simulación.

3.9.1. Manejo de Estados y Concurrencia con `RwLock` y `ThreadPool`

La simulación emplea un **ThreadPool** para gestionar múltiples vuelos simultáneamente. Cada vuelo se maneja en un hilo separado, lo que permite simular el comportamiento de múltiples vuelos en paralelo. La estructura `Simulator` almacena la información de la simulación, incluyendo vuelos, aeropuertos, y la gestión del tiempo global.

- **ThreadPool:** Se utiliza un `ThreadPool` para manejar tareas concurrentes de simulación de vuelos. Tras cada tick, se agrega la tarea de actualización de cada uno de los vuelos a la estructura, de manera que se puedan simular sin bloquear el hilo principal.
- **RWLocks (Lectura/Escritura):** Se usan `RwLocks` para permitir el acceso concurrente a estructuras compartidas, como los estados de los vuelos y la lista de vuelos. Esto asegura que varios hilos puedan leer los datos de manera eficiente, pero solo uno pueda escribir al mismo tiempo, evitando condiciones de carrera.

3.9.2. Actualización del Estado de los Vuelos

Los vuelos tienen estados que reflejan su progreso, como `Scheduled`, `OnTime`, `Delayed`, `Finished`, y `Canceled`. Estos estados se actualizan constantemente en función del tiempo de simulación, lo cual ocurre cada tick (`TICK_DURATION_MILLIS`).

- **Actualización de Estado:** Cada vuelo tiene su propio hilo que simula su avance. En cada ciclo, se verifica si el vuelo ha alcanzado el tiempo de salida, si ha llegado a su destino, o si debe ser marcado como terminado o cancelado. Los estados de vuelo se actualizan de forma separada a otros datos, como la latitud y longitud, garantizando que la simulación se ejecute de manera independiente para cada vuelo.
- **Actualización de Datos de Tracking:** Además del estado, la posición del vuelo (latitud, longitud), velocidad y otros parámetros como el nivel de combustible se actualizan periódicamente. Estos datos son modificados por cada hilo asociado a un vuelo y reflejan su progreso a lo largo de la simulación.

3.9.3. Gestión del Tiempo con un Timer Global

El `Simulator` utiliza la estructura `Timer` que se actualiza en intervalos constantes. Esta actualización del tiempo es gestionada por un hilo separado que incrementa el tiempo global basado en la tasa de tiempo configurada por el usuario. Además, ejecuta una función de callback, que en este caso haría nuestra actualización, lo que permite avanzar la simulación en tiempo real. La tasa de tiempo determina la velocidad a la que avanzan los vuelos en la simulación, lo que permite acelerar o desacelerar el proceso según las preferencias del usuario.

3.9.4. Visualización en Tiempo Real

La aplicación ofrece una visualización en tiempo real de la simulación de vuelos. Cada vez que se listan los vuelos, se muestra su estado actual (en tiempo real), junto con datos de su ubicación (latitud y longitud). La visualización es dinámica, y se actualiza constantemente para reflejar el progreso de los vuelos. El usuario puede salir de la visualización de los

vuelos presionando 'q', y la consola se limpia entre cada actualización para ofrecer una vista clara de la información actualizada.

3.10. Interfaz Gráfica

3.10.1 Mapa Geográfico Mundial

Para la visualización del mapa se utilizó como proveedor a OpenStreetMaps, que provee acceso libre a un mapa político mundial. Sobre este mapa se agregaron los puntos de interés para la aplicación: aeropuertos y vuelos.

3.10.2 Visualización de Vuelo

Los vuelos consisten de dos grupos de información bien diferenciados: estado y tracking. El estado consiste de información que cambia poco. Estos son el número de vuelo, el tipo de avión, el destino y el origen, y su estado (en horario, demorado, etc). El otro conjunto de datos es más dinámico y cambia segundo a segundo (con el simulador de vuelos proveyendo la información en tiempo real). Estos son el nivel de combustible, la ubicación, la altitud, etc. Ambos grupos de información pueden visualizarse en la interfaz. La información de estado de un vuelo puede verse también desde la ventana al seleccionar un aeropuerto. La información de tracking puede verse como el movimiento del avión en tiempo real a través de la pantalla, y también al seleccionarlo.

3.10.3 Edición de Estado de Vuelos

Además de la visualización de vuelos y aeropuertos existentes, también existe la funcionalidad de insertar vuelos, y actualizar manualmente el estado de los mismos.

<u>Inserción:</u>

Para insertar un vuelo, simplemente se especifican todos los campos requeridos para iniciar un vuelo, siendo estos su número, horarios estimados de salida y llegada, el nivel de combustible con el que comienza, su velocidad promedio y la altitud a la que va a volar, además de claramente los vuelos de partida y destino.

Una limitación importante es que se asume que se agrega un vuelo que todavía no ocurrió, osea que tiene que ser a futuro, no se puede preparar un vuelo para que ocurra ayer.

<u>Actualización de estado:</u>

Otra cosa que se puede realizar, es manualmente actualizar el estado de un vuelo, lo cual permitiría, por ejemplo, que parta, se retrase, o finalice antes de tiempo. O cancelar vuelos.

Existen dos limitantes:

- No se puede “volver atrás”. Ej. Si el vuelo comenzó, no puede volver a estar esperando en el aeropuerto. Si ya aterrizó, no puede volver a estar atrás en el aire. Lo que sí, en cualquier momento un vuelo se puede dar como finalizado o cancelado, sin limitantes.

- No se puede iniciar un vuelo fuera de fechas, se puede lograr lo mismo cancelando el vuelo, y creando uno nuevo.

3.11 Seguridad

3.11.1 Gestión de Identidades y Permisos

Para la gestión de identidades y permisos se utiliza el mecanismo nativo de Cassandra, que permite la autenticación de un cliente mediante un intercambio inicial de paquetes especiales, que permiten a un cliente autenticarse para poder acceder a los recursos del cluster. Se configuró al sistema para que todos los nodos requieran autenticación por parte de los clientes, la cual se resuelve utilizando una contraseña por defecto.

3.11.2 Protección de Datos en Tránsito

Existen dos flujos de datos a proteger de actores maliciosos que intenten espiar o alterar el funcionamiento del sistema. Estos son: las comunicaciones con los clientes; y las comunicaciones entre nodos.
Para la protección de los datos en tránsito se utilizó TLS por medio de la librería rustls y certificados autofirmados con openssl.

3.12. Evaluación de Resultados

3.12.1 Pruebas Unitarias

Cada estructura implementada en el sistema tiene sus propias pruebas unitarias diseñadas para validar sus funcionalidades y casos de uso más relevantes. Estas pruebas abarcan operaciones como:

- Validación de estructuras internas (e.g., particionamiento, parseo de queries, y manejo de índices).
- Casos específicos para componentes clave como el Partitioner y el Query Creator.
- Simulación de interacciones locales en componentes como el Storage Engine.

En algunos casos, ciertas estructuras no pudieron ser completamente testeadas de forma unitaria debido a su complejidad inherente, ya que requieren un contexto más amplio (e.g., múltiples nodos o interacciones distribuidas). Estas estructuras se validaron principalmente a través de pruebas de integración.

3.12.2 Pruebas de Integración

Para validar la interacción entre los diferentes componentes del sistema, se diseñó una prueba de integración completa. Esta prueba incluye:

1. Levantamiento de un clúster de múltiples nodos.
2. Envío de diferentes tipos de queries (lectura, escritura, actualización) al sistema.
3. Verificación de las respuestas obtenidas, evaluando si cumplen con los niveles de consistencia configurados y las reglas del sistema.

Esta prueba permitió analizar la funcionalidad del sistema en condiciones reales, simulando escenarios típicos de uso.

3.12.3 Simulación de Escenarios de Fallo

Se realizaron pruebas específicas para evaluar la capacidad del sistema de tolerar fallos y adaptarse a cambios en la topología del clúster. Estas pruebas incluyeron:

- **Desconexión de nodos**: Simulación de la caída de uno o más nodos durante la ejecución de consultas, evaluando cómo el sistema redistribuye las cargas y si las queries alcanzan el Consistency Level requerido.
- **Reconexión de nodos**: Verificación de cómo el sistema reintegra un nodo previamente desconectado y redistribuye los datos en consecuencia.
- **Fallo en el protocolo Gossip**: Simulación de fallos en el intercambio de mensajes Gossip para identificar cómo el sistema maneja nodos considerados "muertos".

Estas pruebas confirmaron que el sistema puede manejar fallos en nodos individuales sin comprometer la funcionalidad global.

3.12.4 Optimización de Recursos

La optimización de recursos en el sistema se basó en varios enfoques clave:

- **Concurrencia eficiente**: Uso de estrategias concurrentes (como ThreadPool y manejo de múltiples queries) para maximizar la utilización de CPU y evitar bloqueos innecesarios.
- **Optimización de búsquedas**: Uso de índices adicionales para localizar registros específicos de manera más eficiente, reduciendo los tiempos de lectura en archivos grandes.
- **Evitar esperas innecesarias**: En ningún momento del flujo del sistema se implementaron bloqueos que dependen de respuestas interminables. Las consultas avanzan con los datos disponibles, devolviendo resultados parciales o errores si no se alcanza el nivel de consistencia requerido.

Estos esfuerzos garantizaron que el sistema sea capaz de manejar grandes volúmenes de datos y consultas simultáneas sin un consumo excesivo de recursos.

4. Extensión Proyecto

4.1 Refactor Conexiones

Se necesitó realizar un cambio en la estructura *OpenQueryHandler* ya que esta guardaba por cada query abierta, el stream perteneciente al cliente que mandó esa query para poder enviarle la respuesta al finalizarla.
Al integrar todo el sistema con TLS y utilizar sus *StreamOwned*, y estos al no ser clonables, surgieron varios problemas de integración.
Decidimos utilizar channels para solucionarlo, aprovechando las ventajas de los patrones de concurrencia con pasaje de mensajes, en lugar de estado mutable compartido. Cada query

crea un channel. El extremo de escritura se guarda en la *OpenQuery*, y en el extremo de lectura se aguarda la resolución de la *query*. Cuando la query se resuelve, en lugar de intentar utilizar directamente el stream, envía la respuesta por el *channel*, la cual será capturada en el otro extremo y enviada por la única instancia existente del stream TLS. Esto permite una mayor modularización al separar las responsabilidades de procesamiento y comunicación.

4.2 Refactor Flight Sim

Se realizaron múltiples modificaciones al Simulador de Vuelos debido a varias problemáticas que surgieron con la versión anterior:

- **Asignación de vuelos en el threadpool:** Anteriormente, los vuelos se actualizaban constantemente dentro de los threads del threadpool, lo cual hacía que solo se puedan simular una cantidad limitada de vuelos al mismo tiempo, ya que bloqueaban el thread hasta su aterrizaje. Además, cada actualización de cada vuelo requería un lockeo individual del timer. Como solución, ahora el simulador usa un único Timer, que ejecuta una función de callback. Esta función se encarga de recorrer los vuelos y agregar la tarea de cada actualización individual al ThreadPool, de forma que ningún thread se quede bloqueado esperando.
- **Posibles deadlocks y retrasos**: Además, la mayoría de operaciones (inserts, updates, cálculos y la obtención datos de la DB) hacían uso de read/write/lock, que podían introducir deadlocks cuando el cliente de Cassandra tardaba en responder. Ahora, toda operación no esencial (todo lo que no son los inserts) intenta hacer el read/write/lock y si no lo logra simplemente lo vuelve a intentar en el siguiente tick. Entonces, en el peor de los casos, puede que en la visualización pegue pequeños saltos cada cortos intervalos de tiempo.

4.3 Dockerizacion

Se utilizó Docker para ejecutar los nodos en contenedores dedicados. La configuración provista en el Dockerfile es muy sencilla: una imagen base (rust:latest) que provee un entorno para compilar y ejecutar proyectos desarrollados en Rust. Lo más interesante es la posibilidad de ejecutar un cluster de nodos con Compose, una herramienta de Docker que permite ejecutar aplicaciones con múltiples contenedores. Se provee un archivo de configuración (compose.yml), con una configuración básica que consiste de un cluster de 3 nodos, uno de los cuales es el nodo semilla, y dos réplicas. Cada “servicio” (instancia del nodo que se ejecuta en un contenedor) se configura con sus variables de entorno, utilización de la imagen de Docker básica provista en el Dockerfile, y su comando de ejecución. De este modo, con un solo comando se puede ejecutar un cluster entero. Además, en la configuración se mapean distintos puertos de la interfaz de red del host al puerto del nodo dedicado para atender conexiones de clientes (17989), por lo que se puede realizar consultas a distintos nodos utilizando la interfaz de loopback (localhost) y distintos puertos (10000, 10001, etc).

4.4 Reconfiguración Dinámica del Cluster

La reconfiguración dinámica de un clúster ocurre tanto cuando un nodo se une como cuando un nodo se retira. Este proceso es llamado desde el thread que maneja **Gossip**, mediante el cual los nodos se notifican mutuamente sobre los cambios en la estructura del

anillo, así como la modificación del partitioner. Una vez detectado el cambio, se inicia en paralelo un proceso de redistribución de datos en un hilo dedicado en cada nodo.

Durante esta redistribución, cada nodo recorre sus datos y, si es necesario, los reorganiza para garantizar que estén correctamente distribuidos según el nuevo partitioner. Este proceso se realiza en paralelo con las operaciones normales del clúster, lo que permite que las consultas de los clientes sigan siendo atendidas de manera natural. Los clientes solo notan el impacto si la caída de un nodo afecta el nivel de consistencia configurado en la consulta, ya que esto podría impedir cumplir con consistency level.

Aunque la redistribución de datos es un proceso costoso en términos de recursos, es fundamental para mantener la integridad y el equilibrio del clúster. En caso de que un nodo caído se reincorpore más tarde, el clúster no conserva información previa sobre ese nodo, tratándolo como uno nuevo. Sin embargo, si se trata del mismo nodo, la redistribución termina restaurando el estado original, logrando que el sistema se comporte como si el nodo nunca hubiera salido del clúster. Este enfoque asegura que el clúster mantenga su estructura consistente y funcional en todo momento, incluso frente a cambios dinámicos.

4.5 Logger

El Logger desarrollado tiene como propósito registrar y monitorear las actividades de los nodos en el sistema, implementado en un entorno Docker. Este componente permite registrar todos los mensajes enviados y recibidos por un nodo, facilitando tanto la visualización en tiempo real como la conservación de un registro histórico. Esto se logra mediante la escritura simultánea de los mensajes en la salida estándar (stdout) y en un archivo de log, permitiendo a los usuarios visualizar las operaciones en curso utilizando comandos como **`docker logs`**.

El Logger registra eventos clasificados en niveles de severidad, como informativos, advertencias y errores, asegurando que cada mensaje contenga un timestamp que facilita su rastreo. Además, se utiliza un sistema de colores en la salida estándar para distinguir visualmente los distintos tipos de eventos, mejorando la claridad y usabilidad en la supervisión en tiempo real. Los archivos de log se almacenan de manera estructurada y nombrados según la dirección IP del nodo, lo que permite identificar fácilmente los registros asociados a cada nodo en el sistema distribuido.

El Logger resuelve de manera eficiente el problema de supervisar el comportamiento de los nodos en un sistema distribuido, asegurando un registro persistente y una visualización inmediata. Esto facilita la detección y resolución de problemas, al tiempo que proporciona un mecanismo para auditar el funcionamiento del sistema en caso de incidentes. La solución garantiza robustez y simplicidad, permitiendo un uso directo dentro del entorno Docker sin configuraciones complejas adicionales.

5. Conclusiones y Aprendizajes

5.1 Conclusiones

A lo largo de este proyecto, aprendimos sobre los distintos temas involucrados en un proyecto de esta magnitud, incluyendo las características y ventajas de una base de datos distribuida, temas de concurrencia, conexiones entre servidores y diseño de interfaces gráficas.
Además, mejoramos significativamente nuestro trabajo en equipo, el manejo de Git, y adoptamos mejores prácticas como el uso de ramas para cada funcionalidad, revisiones de código en pull requests, y evitar pushear directamente a la rama `main`. Este enfoque colaborativo y estructurado nos permitió gestionar eficientemente las tareas y resolver problemas de manera efectiva.

6. Anexos

6.1 Repositorio de Github

https://github.com/taller-1-fiuba-rust/24C2-Ferrum