Trabajo Práctico Integrador

Protocolos y Modelo OSI

Fecha de Entrega: 22/11/2024

Joaquin Dominguez, Luca Bautista Cacciatore, Facundo Nahuel Schwab

Propuesta de Replicación para Firebird 2.5

1. Introducción

La replicación en Firebird permite mantener múltiples copias de bases de datos sincronizadas en diferentes servidores, asegurando alta disponibilidad, tolerancia a fallos y escalabilidad. En este diseño, proponemos una solución de replicación que cumple con los requisitos del trabajo, utilizando las capacidades nativas y prácticas viables en Firebird 2.5 o superior.

2. Objetivos

1. **Alta disponibilidad:** Recuperación rápida en caso de fallo de un servidor.
2. **Escalabilidad:** Balancear la carga de trabajo entre múltiples servidores.
3. **Tolerancia a fallos:** Evitar pérdida de datos en caso de caídas.
4. **Consistencia:** Asegurar que las copias estén sincronizadas.
5. **Granularidad ajustable:** Soporte para replicación a nivel de base de datos, tabla o tupla.

3. Arquitectura propuesta

Modelo Maestro-Esclavo

- **Descripción**: Un servidor principal (Maestro) realiza operaciones de escritura y coordina la replicación hacia uno o más servidores secundarios (Esclavos), que manejan las operaciones de lectura.
- **Ventajas**:
    - Fácil de implementar y escalar.
    - Reduce la carga de lectura en el servidor principal.
- **Desventajas**:
    - El maestro es un punto único de fallo (a menos que se use un esclavo como respaldo).

Componentes:

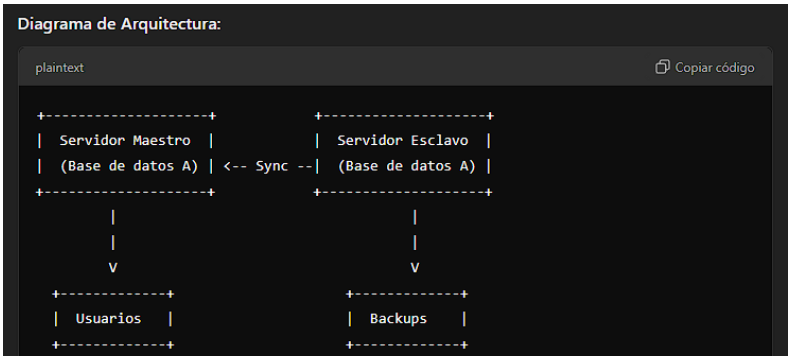
1. **Maestro**:
    - Gestiona todas las operaciones de escritura.

- Genera logs de cambios (INSERT, UPDATE, DELETE) para replicar.

2. **Esclavos**:
   - Reciben los cambios desde el maestro.
   - Atienden consultas de lectura para reducir la carga en el maestro.
3. **Cliente**:
   - Interactúa con el maestro para operaciones de escritura.
   - Puede conectarse a los esclavos para consultas de lectura.

Diagrama de Arquitectura:

plaintext Copiar código



Posibles Alternativas:

Maestro-Maestro

En un modelo Maestro-Maestro, ambos servidores tienen la capacidad de realizar operaciones de lectura y escritura, sincronizándose entre sí para mantener los datos consistentes.

- Ventajas:
  - Tolerancia a fallos robusta: Si uno de los nodos falla, el otro puede asumir su rol sin interrupción.
  - Mejor balanceo de carga: Permite distribuir operaciones de escritura entre ambos nodos.
- Razones para descartarlo:
  - Conflictos de sincronización:
    - Ejemplo: Si dos clientes actualizan el mismo registro en servidores diferentes al mismo tiempo, es necesario implementar estrategias complejas de resolución de conflictos, lo cual incrementa significativamente la complejidad del sistema.
  - Mayor latencia:
    - La constante sincronización entre nodos puede ralentizar el procesamiento de transacciones, especialmente en redes distribuidas geográficamente.

Peer-to-Peer

En este modelo, todos los nodos tienen el mismo rol: pueden realizar operaciones de lectura y escritura. Este esquema es altamente escalable y adecuado para sistemas distribuidos.

- Ventajas:
    - Escalabilidad: Todos los nodos contribuyen a la capacidad de escritura y lectura.
    - Tolerancia a fallos: No existe un punto único de fallo.
- Razones para descartarlo:
    - Complejidad de implementación:
        - Ejemplo: Si un sistema bancario utiliza replicación Peer-to-Peer para registrar transacciones, sería extremadamente difícil garantizar que las actualizaciones se procesen en el mismo? orden en todos los nodos, lo que podría llevar a inconsistencias en los balances de cuentas.
    - Sobrecarga de comunicación:
        - Requiere sincronización constante entre todos los nodos, lo que puede saturar la red y aumentar la latencia en sistemas grandes.

Broadcast Replication

En este modelo, el Maestro envía simultáneamente los datos a múltiples Esclavos, que pueden ser utilizados para consultas de solo lectura.

- Ventajas:
    - Distribución eficiente de datos para sistemas con muchos nodos esclavos.
    - Ideal para sistemas donde los esclavos solo necesitan datos de lectura.
- Razones para descartarlo:
    - Sobrecarga en el Maestro:
        - Ejemplo: En un sistema de monitoreo en tiempo real con cientos de nodos de lectura (como sensores IoT), el Maestro puede colapsar al intentar transmitir datos simultáneamente a todos los nodos.
    - Falta de tolerancia a fallos:
        - Si el Maestro falla, no hay un mecanismo para que los Esclavos se conviertan en Maestros, lo que interrumpe el sistema.

Arquitectura de Relé

En esta arquitectura, un nodo intermedio (el relé) actúa como intermediario entre el Maestro y los Esclavos. El Maestro no replica directamente los cambios a los Esclavos, sino que envía los datos al nodo de relé, el cual se encarga de distribuirlos.

- Ventajas
    1. Reducción de carga en el Maestro:
        - El Maestro se libera de la tarea de replicar datos a múltiples Esclavos, ya que el nodo de relé asume esa responsabilidad.

    - Ejemplo: En un sistema con 10 esclavos, el Maestro solo necesita enviar los cambios al relé una vez, en lugar de replicarlos directamente a los 10 nodos.
2. Escalabilidad:
    - Es más fácil agregar o quitar Esclavos, ya que el relé gestiona las conexiones con ellos, no el Maestro.
3. Centralización de la replicación:
    - El nodo de relé puede realizar transformaciones o filtrados de datos antes de enviarlos a los Esclavos.

- Desventajas
1. Punto único de fallo adicional:
    - Si el nodo de relé falla, todo el sistema de replicación se detiene, aunque el Maestro y los Esclavos sigan funcionando.
2. Mayor latencia:
    - Los datos deben pasar primero por el nodo de relé antes de llegar a los Esclavos, lo que introduce un retraso adicional.
3. Sobrecarga en el relé:
    - Si el nodo de relé no tiene capacidad suficiente, puede convertirse en un cuello de botella, especialmente en sistemas con muchos Esclavos.

Escenarios Donde es Útil

1. Sistemas con muchos Esclavos:
    - Ejemplo: En una aplicación de analítica con 50 nodos esclavos dedicados a consultas de lectura, un nodo de relé puede distribuir los datos de manera más eficiente.
2. Filtrado de datos:
    - Si los Esclavos solo necesitan una parte de los datos (e.g., tablas específicas), el nodo de relé puede filtrar la información, reduciendo el tráfico de red y el almacenamiento.

Razón para Descartarla

Para este diseño en Firebird, la arquitectura de relé se descarta porque:

1. El foco del trabajo está en la simplicidad y consistencia:
    - Añadir un nodo intermedio complica la arquitectura y no ofrece beneficios significativos para un sistema pequeño o mediano.
2. La replicación directa Maestro-Esclavo es suficiente:

- Dado que el número de Esclavos no se espera que sea alto, la carga sobre el Maestro es manejable.

3. Firebird no tiene soporte nativo para un nodo relé:
    - Implementar esta arquitectura requeriría un desarrollo complejo adicional para **crear y gestionar el nodo intermedio.**

Nos terminamos decantando por la arquitectura **Maestro-Esclavo** porque simplifica el diseño y la implementación del sistema y proporciona una vista coherente y sincronizada del sistema, aunque puede ser más difícil de escalar y en términos de seguridad crea un punto único de fallo.

4. Detalles del diseño

4.1 Instalación

1. **Configuración de servidores:**
    - Instalar Firebird 2.5 (o superior) en cada servidor (Maestro y Esclavo).
    - Configurar el acceso mediante protocolos de red seguros, como TCP/IP.
    - Definir roles de Maestro y Esclavo en la arquitectura de replicación.
    - Configurar las tablas de log de replicación (REPLICA_LOG) en el Maestro.
    - Usa el archivo *firebird.conf* para definir:
        1. Ubicación de los logs de replicación.
        2. Configuración de red para permitir conexiones seguras entre los nodos.

2. **Creación de la Tabla de Logs**

En el maestro, crea una tabla de logs que registre las operaciones que deben replicarse:

```
CREATE TABLE REPLICA_LOG (
    ID INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    TABLE_NAME VARCHAR(50),
    OPERATION CHAR(1), -- 'I', 'U', 'D'
    RECORD_ID INTEGER,
    TIMESTAMP TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. **Configuración de Triggers**

Define triggers en las tablas críticas para capturar cambios:

```
CREATE TRIGGER AFTER_INSERT_MY_TABLE
AFTER INSERT ON MY_TABLE
AS
```

```
  BEGIN
    INSERT INTO REPLICA_LOG (TABLE_NAME, OPERATION, RECORD_ID)
    VALUES ('MY_TABLE', 'I', NEW.ID);
  END;
```

4. **Configuración de replicación:**

El diseño propuesto empleará triggers en el Maestro para registrar cambios en REPLICA_LOG y mecanismos manuales para transmitir cambios a los Esclavos.

Protocolo de Replicación

a. Detección de Cambios

- Los triggers registran cambios en la tabla REPLICA_LOG.
- Cada registro incluye:
  - El nombre de la tabla.
  - El tipo de operación (INSERT, UPDATE, DELETE).
  - El ID del registro afectado.
  - La marca de tiempo.

b. Sincronización de Datos

Un procedimiento almacenado periódicamente procesa los logs y aplica los cambios en los esclavos:

```
CREATE PROCEDURE SYNC_TO_SLAVES
AS
BEGIN
  FOR SELECT * FROM REPLICA_LOG INTO :TABLE_NAME, :OPERATION,
:RECORD_ID DO
  BEGIN
    -- Lógica para replicar los cambios en los esclavos
    EXECUTE STATEMENT 'INSERT INTO ' || :TABLE_NAME || ' (...) VALUES
(...)';
  END;
END;
```

c. Protocolo de Commit

a. Two-Phase Commit (2PC)

En este diseño, se opta por el protocolo 2PC para garantizar una consistencia fuerte entre el Maestro y los Esclavos. Esta estrategia asegura que todos los nodos participen de manera coordinada en cada transacción, evitando inconsistencias entre ellos.

Proceso:

1. Preparación:
    - El Maestro envía los cambios a los Esclavos y les solicita confirmación sobre si pueden aplicar las modificaciones.
    - Los Esclavos verifican localmente que pueden procesar los cambios y responden al Maestro con una confirmación positiva o negativa.
2. Confirmación:
    - Si todos los Esclavos responden de forma positiva, el Maestro procede al commit global.
    - Si alguno de los Esclavos responde negativamente o no responde, el Maestro inicia un rollback global para garantizar consistencia.
3. Commit Global:
    - Una vez que todos los nodos confirman, el Maestro finaliza la transacción y replica los cambios definitivos.

Ventajas:

- Garantiza consistencia entre todos los nodos.
- Es ideal para sistemas donde la integridad de los datos es crítica, como en sistemas financieros o de inventario.

Desventajas:

- Introduce latencia, ya que el Maestro debe esperar la confirmación de todos los nodos antes de finalizar la transacción.
- Puede ser más lento en redes con alta latencia o en sistemas con muchos Esclavos.

b. Alternativa Descartada: Commit Asíncrono

El Commit Asíncrono es una estrategia alternativa que prioriza el rendimiento al permitir que el Maestro confirme las transacciones localmente antes de replicarlas a los Esclavos.

Proceso:

1. El Maestro aplica los cambios localmente y confirma la transacción al cliente.
2. Posteriormente, los cambios se replican a los Esclavos en intervalos regulares o cuando se alcanza un umbral de cambios acumulados.

Ventajas:

- Mejora el rendimiento, ya que las operaciones del cliente no están limitadas por la sincronización inmediata con los Esclavos.

Desventajas:

- Introduce una ventana de inconsistencia, ya que los Esclavos pueden no estar sincronizados con el Maestro en todo momento.
- No es adecuado para sistemas que requieren consistencia fuerte.

Por qué se descarta: Para este diseño, la consistencia fuerte es prioritaria, por lo que el Commit Asíncrono no cumple con los requisitos de integridad de datos. Sin embargo, podría ser útil en sistemas donde el rendimiento sea más importante que la sincronización inmediata, como sistemas de analítica o cachés distribuidos.

c. Implementación en el Diseño

En este trabajo, utilizamos 2PC como protocolo de commit debido a su capacidad para garantizar la consistencia global. El proceso queda definido en tres pasos:

1. Preparación: El Maestro envía los cambios a los Esclavos y espera confirmaciones.
2. Confirmación: Los Esclavos validan los cambios y confirman su capacidad para aplicarlos.
3. Commit Global: Una vez que todos los Esclavos confirman, el Maestro finaliza la transacción de manera sincronizada.

Esta implementación asegura que todos los nodos estén completamente sincronizados al finalizar cada transacción, eliminando riesgos de inconsistencia.

d. Grado de Transparencia

- Transparencia para el usuario final:
    - Los usuarios interactúan únicamente con el servidor Maestro para actualizaciones.
    - Las aplicaciones no requieren modificaciones adicionales.
    - Los Esclavos se utilizan internamente para mejorar la tolerancia a fallos y la carga de consultas de solo lectura.
    - Replicación: Los procesos de replicación son invisibles para los usuarios.
    - Fallos: La recuperación se gestiona automáticamente, con mínima intervención.

e. Granularidad

La granularidad se refiere al nivel de datos replicados. Las opciones son:

1. Base de datos completa:
    - Réplica de todas las tablas y datos.

- Ventaja: Simplicidad en la configuración.
- Desventaja: Mayor uso de recursos y ancho de banda.

2. Tabla:
   - Réplica de tablas específicas.
   - Útil para sistemas con alto tráfico de datos no relacionado.
   - Requiere mayor personalización.
3. Tupla (fila):
   - Réplica de filas específicas dentro de tablas.
   - Eficiente para grandes bases de datos con requisitos específicos.
   - Implementación compleja.

Implementación Propuesta:

- Usamos triggers para determinar qué registros (tuplas) deben replicarse.
- Configuramos procedimientos almacenados para replicar solo las tablas críticas.

1. Justificación para replicación completa

Se opta por replicar toda la base de datos en este diseño debido a los siguientes factores:

1. **Simplicidad de configuración y mantenimiento:**
   - Al replicar la base de datos completa, no es necesario definir reglas complejas para seleccionar tablas o registros específicos.
   - Esto reduce el esfuerzo administrativo y facilita el monitoreo del sistema.
2. **Tamaño esperado de las bases de datos:**
   - Si el tamaño total de los datos es moderado, replicar todo tiene un impacto aceptable en el rendimiento y el uso de recursos.
   - **Ejemplo práctico:** Bases de datos empresariales con menos de 50 GB pueden replicarse sin problemas en redes con capacidad adecuada.
3. **Costo de recursos frente a replicación parcial:**
   - La replicación parcial puede parecer más eficiente en términos de tráfico de red o almacenamiento, pero su implementación es más compleja, ya que requiere:
     - Definición de filtros para incluir/excluir tablas o registros.
     - Procedimientos adicionales para asegurar la integridad de los datos en diferentes niveles de granularidad.
   - Para sistemas donde la simplicidad y la integridad global son prioridades, replicación completa es la mejor opción.

2. Escenarios donde replicación de tabla o tupla es más adecuada

En sistemas más complejos o con necesidades específicas, la replicación parcial (a nivel de tabla o tupla) puede ser más eficiente. Ejemplos prácticos:

1. **Replicación de tabla:**
    - **Escenario:** Un sistema multi-tenant donde cada cliente tiene su propia base de datos lógica.
        - En este caso, cada nodo podría replicar solo las tablas relevantes para los clientes que gestiona.
        - **Ventajas:** Minimiza el uso de ancho de banda y recursos al no replicar datos innecesarios.
    - **Desventaja:** Requiere configuraciones más detalladas para evitar inconsistencias.
2. **Replicación de tupla:**
    - **Escenario:** Una aplicación de e-commerce que necesita sincronizar inventarios en tiempo real entre almacenes.
        - Solo se replican las filas que cambian en función de las transacciones (e.g., reducción de stock).
        - **Ventajas:** Ideal para grandes bases de datos distribuidas, ya que reduce el tráfico a lo estrictamente necesario.
        - **Desventaja:** Puede generar complejidad en el manejo de conflictos entre nodos.

3. Consideraciones adicionales para decidir el nivel de granularidad

1. **Complejidad vs. Costo:**
    - **Replicación completa:** Baja complejidad, mayor uso de recursos.
    - **Replicación parcial:** Mayor complejidad en la configuración, pero ahorro significativo en redes saturadas o sistemas con grandes volúmenes de datos.
2. **Requerimientos de consistencia:**
    - **Sistemas financieros o críticos:** Replicación completa asegura consistencia total sin depender de reglas de filtrado.
    - **Sistemas con datos secundarios o no críticos:** Replicación parcial puede reducir costos y aumentar la eficiencia.
3. **Crecimiento futuro:**
    - En sistemas que se espera que crezcan significativamente en tamaño, es recomendable diseñar una solución híbrida, comenzando con replicación completa y migrando a parcial según sea necesario.

Nivel de Granularidad	Ventajas	Desventajas
Base completa	Fácil de implementar, alta integridad	Uso intensivo de recursos
Tabla	Optimiza ancho de banda	Configuración más compleja
Tupla	Eficiente para grandes bases de datos	Manejo complejo de conflictos

f. Fragmentación

La fragmentación mejora el rendimiento distribuyendo datos entre nodos. Tipos principales:

1. **Horizontal:**
    - Filas distribuidas entre nodos.
    - Ejemplo: Clientes con ID 1-1000 en un nodo, y 1001-2000 en otro.
    - Ventaja: Reducción del volumen de datos por nodo, ideal para consultas específicas.
2. **Vertical:**
    - Columnas divididas entre nodos.
    - Ejemplo: Datos personales en un nodo y datos financieros en otro.
    - Útil para tablas con muchas columnas.
3. **Mixta:**
    - Combinación de fragmentación horizontal y vertical.
    - Ejemplo: Clientes de 1-1000 con ciertas columnas en un nodo, y 1001-2000 en otro.
    - Mayor flexibilidad y optimización de consultas.

Implementación propuesta: Fragmentación mixta para dividir tanto filas como columnas, maximizando la eficiencia en consultas específicas y la gestión de tablas grandes.

Escenarios en los que esta técnica resultaría especialmente útil:

Escenario 1: Sistema de Gestión de Clientes y Transacciones Financieras

- **Descripción del Sistema:** Una institución financiera almacena información de clientes y sus transacciones. Los datos están distribuidos geográficamente, y cada sucursal tiene

acceso solo a sus clientes, pero el sistema central necesita ciertas columnas específicas (e.g., nombres, saldos) para informes globales.

- **Fragmentación aplicada:**
    - **Horizontal:** Los datos de los clientes se dividen según su región o sucursal. Por ejemplo:
        - Nodo 1: Clientes de la región norte.
        - Nodo 2: Clientes de la región sur.
    - **Vertical:** En cada nodo, solo se almacenan las columnas críticas localmente (nombre, saldo), mientras que las columnas menos utilizadas (historial detallado) se almacenan en nodos centralizados.
- **Ventajas de la fragmentación mixta:**
    - Reduce la cantidad de datos en cada nodo, optimizando el acceso local.
    - Facilita consultas rápidas sobre datos relevantes para cada región, mientras centraliza información no crítica.

Escenario 2: Plataforma de E-Commerce Global

- **Descripción del Sistema:** Una tienda en línea opera en múltiples países, con productos disponibles localmente pero datos de inventario y ventas globales requeridos para análisis centralizado.
- **Fragmentación aplicada:**
    - **Horizontal:** Los productos se distribuyen según su región de venta:
        - Nodo 1: Productos disponibles en América del Norte.
        - Nodo 2: Productos disponibles en Europa.
    - **Vertical:** Los detalles del producto (precio, stock) se almacenan en los nodos locales, mientras que las imágenes y descripciones (usadas en marketing global) se mantienen en un nodo central.
- **Ventajas de la fragmentación mixta:**
    - Acelera las consultas locales relacionadas con el inventario y las ventas.
    - Reduce el almacenamiento redundante de imágenes en nodos donde no se necesiten.

Escenario 3: Sistema de Salud Distribuido

- **Descripción del Sistema:** Un sistema nacional de salud almacena historiales médicos de pacientes. Los datos se distribuyen entre hospitales, pero algunos indicadores globales son necesarios para análisis de salud pública.

- **Fragmentación aplicada:**
    - **Horizontal:** Los historiales médicos se dividen según los hospitales.
    - **Vertical:** Cada hospital almacena localmente solo información crítica (diagnósticos recientes, prescripciones), mientras que datos históricos o secundarios se centralizan.
- **Ventajas de la fragmentación mixta:**
    - Proporciona un acceso rápido a datos recientes para los hospitales.
    - Centraliza datos históricos, reduciendo la carga en nodos locales.

Comparación con Fragmentación Horizontal o Vertical Exclusivas

- **Fragmentación Horizontal (sola):**
    - Ideal cuando los datos se dividen completamente por criterios geográficos o funcionales.
    - **Limitación:** Los nodos locales pueden almacenar columnas no relevantes, desperdiciando recursos.
- **Fragmentación Vertical (sola):**
    - Adecuada para tablas con muchas columnas no usadas frecuentemente.
    - **Limitación:** Cada nodo necesita almacenar todas las filas, lo que puede generar redundancia.

Por qué optar por fragmentación mixta:
La fragmentación mixta combina las fortalezas de ambas técnicas, maximizando la eficiencia en bases de datos complejas al reducir tanto la redundancia como el tamaño de los datos almacenados en cada nodo. Es particularmente útil en sistemas que necesitan optimización local y global simultáneamente.

5. Recuperación ante fallos

1. **Estrategia:**
    - Si el maestro falla, uno de los esclavos se convierte en maestro usando un proceso de failover.
2. **Pasos de recuperación:**
    - Configurar un script para promover automáticamente un esclavo.
    - Realizar un backup completo del esclavo promovido.
3. **Soporte para tolerancia a fallos:**
    - Implementar monitoreo continuo usando herramientas como Nagios o Zabbix.

El proceso consiste en lo siguiente:

1. **Detección de la Falla del Maestro**

Detectar rápidamente la caída del Maestro es esencial para minimizar el impacto en el sistema. Existen diferentes enfoques:

1. Monitoreo con scripts personalizados:
    - Un script ejecutado periódicamente verifica la disponibilidad del Maestro mediante:
        - Ping de red: Comprueba si el servidor Maestro responde en la red.
        - Pruebas de conexión a la base de datos: Intenta abrir una conexión con el servidor Maestro usando herramientas nativas de Firebird o bibliotecas específicas.
    - Ejemplo de script básico (Linux):

```
#!/bin/bash

if ! isql-fb -user SYSDBA -password masterkey
localhost:/path/to/database.fdb -q; then

    echo "Maestro caído" | mail -s "Alerta: Falla del
Maestro" admin@example.com

    # Aquí se puede activar el proceso de failover

Fi
```

2. Herramientas de monitoreo especializadas:
    - **Nagios o Zabbix:** Configura estas herramientas para monitorear la disponibilidad del Maestro y enviar alertas si detectan que está inactivo.
    - Ventajas:
        - Detección rápida y configurable.
            1. Registro de eventos que facilita auditorías posteriores.
3. Heartbeat (latido de vida):
    - Los Esclavos envían regularmente señales (heartbeats) al Maestro. Si no reciben una respuesta en un tiempo predeterminado, asumen que el Maestro está caído y activan un proceso de failover.

2. **Failover Automático**

En caso de falla del Maestro:

1. **Promoción de un Esclavo como Maestro**:
    - Un script preconfigurado promueve automáticamente un Esclavo para asumir el rol de Maestro.
    - **Pasos básicos:**
        - Detener la replicación en el Esclavo seleccionado.

- Cambiar su configuración para aceptar operaciones de escritura.
- Actualizar las configuraciones de los clientes para apuntar al nuevo Maestro.

2. **Ejemplo de comando para cambiar el rol de un Esclavo a Maestro (Linux):**
    - `gfix -mode read_write /path/to/database.fdb`

3. Sincronización del Maestro Restaurado

Cuando el Maestro original vuelve a estar disponible, es necesario sincronizarlo con los datos generados mientras estuvo inactivo. Esto puede hacerse de dos maneras:

1. **Sincronización manual:**
    - Realiza un backup del nuevo Maestro (anteriormente Esclavo).
    - Restaura el backup en el Maestro original.
    - Reinicia la replicación desde el Maestro restaurado hacia los Esclavos.

Pasos:

```
gbak -b /path/to/new_master.fdb /path/to/backup.fbk
gbak -c /path/to/backup.fbk /path/to/original_master.fdb
```

2. **Sincronización incremental:**

- Durante el periodo de falla, los Esclavos continúan registrando los cambios aplicados localmente.
- Al restaurar el Maestro, un proceso automatizado aplica estos cambios pendientes.
- **Implementación:**
    - Usa triggers y procedimientos almacenados para capturar los datos modificados durante el periodo de inactividad y aplicarlos al Maestro restaurado.

Ejemplo de procedimiento:

```
CREATE PROCEDURE APPLY_PENDING_CHANGES
AS
BEGIN
    FOR SELECT * FROM REPLICA_LOG INTO :TABLE_NAME, :OPERATION, :RECORD_ID DO
    BEGIN
        -- Reaplica los cambios en el Maestro restaurado
        EXECUTE STATEMENT 'INSERT INTO ' || :TABLE_NAME || ' (...) VALUES (...)';
```

```
    END;
END;
```

4. **Herramientas de Soporte**
    1. **Nagios o Zabbix:** Para monitorear la disponibilidad y activar alertas.
    2. **Scripts personalizados:** Automatizan la promoción de Esclavos y la sincronización del Maestro.
    3. **Backups incrementales:** Firebird admite backups diferenciales, que pueden facilitar la recuperación en entornos grandes.

Ventajas del Enfoque

- Minimiza el tiempo de inactividad.
- Proporciona un proceso claro y automatizable para manejar fallos.
- Garantiza la consistencia de los datos tras la restauración del Maestro.

6. Implementación práctica

Métodos Manuales: Implementación Principal

En esta propuesta, los métodos manuales constituyen el núcleo del diseño, alineándose con los objetivos académicos y aprovechando herramientas básicas disponibles en Firebird. Estos métodos incluyen:

1. **Triggers**:
    - Capturan automáticamente las operaciones de INSERT, UPDATE y DELETE en las tablas principales.
    - Registran estos cambios en una tabla de logs (REPLICA_LOG) para su posterior replicación.

Ejemplo:

```
CREATE TRIGGER AFTER_INSERT_MY_TABLE
AFTER INSERT ON MY_TABLE
AS
BEGIN
```

```
  INSERT INTO REPLICA_LOG (TABLE_NAME, OPERATION, RECORD_ID)
  VALUES ('MY_TABLE', 'I', NEW.ID);
END;
```

2. **Tabla de Logs (REPLICA_LOG)**:

- Almacena los cambios pendientes de replicación.
- Incluye información clave como el nombre de la tabla, el tipo de operación, el identificador del registro afectado y la marca de tiempo.

Ejemplo:

```
CREATE TABLE REPLICA_LOG (
    ID INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    TABLE_NAME VARCHAR(50),
    OPERATION CHAR(1), -- 'I', 'U', 'D'
    RECORD_ID INTEGER,
    TIMESTAMP TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. **Stored Procedures**:

- Procesan los logs periódicamente para replicar los cambios en los servidores Esclavos.
- Pueden ejecutarse automáticamente mediante tareas programadas o manualmente en caso de sincronización de emergencia.

Ejemplo:

```
CREATE PROCEDURE SYNC_TO_SLAVES
AS
BEGIN
  FOR SELECT * FROM REPLICA_LOG INTO :TABLE_NAME, :OPERATION,
:RECORD_ID DO
  BEGIN
    -- Lógica para replicar los cambios en los esclavos
    EXECUTE STATEMENT 'INSERT INTO ' || :TABLE_NAME || ' (...) VALUES
(...)';
  END;
END;
```

Ventajas de los Métodos Manuales

- **Transparencia académica:** Permiten comprender y controlar cada paso del proceso de replicación.

- **Adaptabilidad:** Son lo suficientemente flexibles como para ajustar el nivel de granularidad (base de datos completa, tabla o tupla).
- **Sin dependencias externas:** No requieren herramientas adicionales, lo que reduce los costos de implementación.

IBReplicator: Extensión Práctica para Entornos Reales

En escenarios empresariales o de alta demanda, donde el enfoque principal sea la velocidad de implementación y la automatización, herramientas como **IBReplicator** pueden complementar este diseño.

Características destacadas:

1. **Automatización de replicación:** IBReplicator configura la sincronización sin necesidad de desarrollar procedimientos personalizados.
2. **Flexibilidad en granularidad:** Soporta replicación a nivel de base de datos completa, tabla o incluso filas específicas.
3. **Facilidad de configuración:** Interfaces gráficas simplifican el proceso para equipos no especializados.

Limitaciones:

- Es una herramienta comercial, por lo que implica un costo adicional.
- Puede reducir el control sobre los detalles técnicos del proceso.

6. Escenario Práctico: Gestión de Inventario para una Cadena de Supermercados

Descripción del Sistema

Una cadena de supermercados opera en distintas regiones del país, con cada sucursal atendiendo las siguientes necesidades:

1. **Acceso local a los datos**: Cada sucursal necesita consultar el inventario y registrar ventas en tiempo real, incluso si la conexión con el servidor central está temporalmente interrumpida.
2. **Sincronización con un sistema central**: Los datos de ventas y reposición de inventario deben consolidarse en un servidor central para:
    - Gestionar el reabastecimiento de productos.
    - Generar reportes globales de ventas e inventarios.

3. **Optimización de consultas**: Las consultas frecuentes (como ventas por región) deben ser rápidas y no sobrecargar el servidor central.

Rol del Modelo Maestro-Esclavo

El diseño Maestro-Esclavo es ideal para este escenario:

- **Servidor Maestro (Central):**
  - Administra los datos globales de inventario y ventas.
  - Registra los cambios realizados por las sucursales.
  - Sirve como punto central para la consolidación de información.
- **Servidores Esclavos (Locales):**
  - Almacenan copias de los datos relevantes para cada sucursal.
  - Manejan las consultas locales sobre inventarios y ventas.
  - Permiten a las sucursales seguir operando, incluso si la conexión con el Maestro se interrumpe temporalmente.

Ventajas del Diseño Propuesto para este Escenario

1. **Alta Disponibilidad:**
   - Las sucursales pueden seguir operando con los servidores Esclavos locales, minimizando el impacto de una posible caída del servidor central.
2. **Reducción de Carga en el Servidor Maestro:**
   - Las consultas locales sobre inventario y ventas son manejadas por los Esclavos, liberando al Maestro para consolidar datos y generar reportes globales.
3. **Consistencia Garantizada:**
   - El protocolo **Two-Phase Commit (2PC)** asegura que las actualizaciones realizadas en las sucursales se apliquen correctamente en el Maestro, evitando inconsistencias en los datos consolidados.
4. **Flexibilidad en la Granularidad:**
   - En este diseño, se opta por replicación completa para simplicidad. Sin embargo, en un escenario futuro, podría implementarse replicación de

tablas específicas (e.g., inventario o ventas) para optimizar el tráfico de datos.

5. **Tolerancia a Fallos:**
    - Si el Maestro falla, un Esclavo puede promoverse automáticamente como Maestro, asegurando la continuidad del sistema.

Aplicación de Fragmentación Mixta

Para mejorar aún más el rendimiento y la escalabilidad, se propone una estrategia de fragmentación mixta:

- **Fragmentación Horizontal:**
    - Los datos de inventario se dividen según regiones.
    - Ejemplo:
        - Nodo 1: Inventario y ventas de la región norte.
        - Nodo 2: Inventario y ventas de la región sur.
- **Fragmentación Vertical:**
    - En cada nodo, solo se almacenan localmente las columnas críticas (e.g., cantidades disponibles, productos en oferta), mientras que los datos históricos o menos usados (e.g., detalles del proveedor) se mantienen en el servidor Maestro.

Recuperación Ante Fallos

En este escenario, la recuperación ante fallos se maneja de la siguiente manera:

1. **Falla del Maestro:**
    - Un Esclavo local se promueve automáticamente como Maestro, asegurando que las sucursales sigan operando.
2. **Falla de un Esclavo:**
    - Los datos pueden restaurarse desde el Maestro, asegurando que no se pierda información crítica.

Conclusión del Escenario

El sistema de replicación Maestro-Esclavo planteado en este trabajo es una solución robusta y eficiente para la gestión de inventarios de una cadena de supermercados. Asegura:

1. **Operación continua** de las sucursales, incluso en situaciones de fallos.
2. **Optimización de recursos** al delegar las consultas locales a los servidores Esclavos.
3. **Consistencia fuerte** en los datos consolidados mediante el protocolo 2PC.

7. Conclusión

Este diseño de replicación para Firebird 2.5 garantiza:

- Alta disponibilidad mediante el modelo Maestro-Esclavo.
- Tolerancia a fallos con failover automático.
- Escalabilidad con múltiples esclavos para consultas.
- Transparencia total para el usuario final.

~~***Representación Gráfica***~~

~~***1. Arquitectura General del Sistema***~~

- ~~***A (Maestro)**: Realiza operaciones de escritura y administra los logs de replicación.*~~
- ~~***B y C (Esclavos)**: Reciben datos replicados y manejan operaciones de solo lectura.*~~
- ~~***D (Cliente)**: Se conecta al Maestro para escritura y a los Esclavos para consultas de lectura.*~~

~~*Representar el flujo de datos desde el Maestro hacia los Esclavos con flechas unidireccionales.*~~

~~***2. Proceso de Commit Asíncrono***~~

- ~~*Diagrama de dos fases:*~~
  - ~~***Fase 1**: El cliente escribe en el Maestro, que registra los cambios en REPLICA_LOG.*~~
  - ~~***Fase 2**: Los Esclavos sincronizan los cambios en intervalos regulares.*~~

~~***3. Fragmentación Mixta***~~

- ~~*Ejemplo:*~~
  - ~~*Nodo 1: Datos de Clientes (ID 1-1000) con información personal.*~~
  - ~~*Nodo 2: Datos de Clientes (ID 1001-2000) con información financiera.*~~

~~*Usar un esquema que muestre la distribución lógica de filas y columnas.*~~

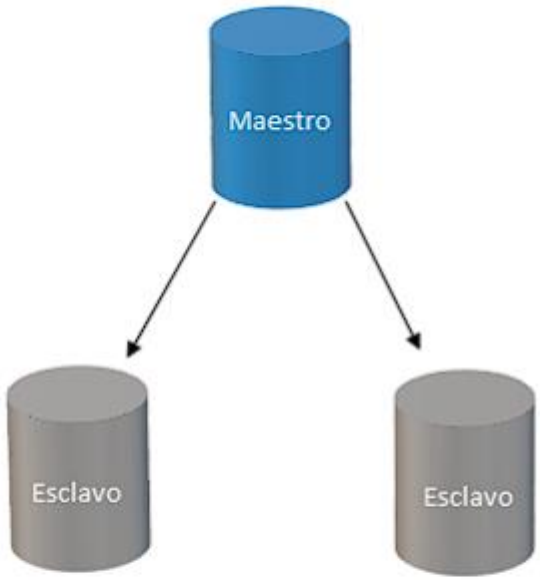


Figura 1- Estructura Maestro-Esclavo

~~7. Representación Gráfica~~

~~Tus gráficos serán fundamentales para clarificar el diseño. Recomendaciones:~~

- ~~**Diagrama de arquitectura**: Simplifica mostrando el flujo de datos entre el Maestro, los Esclavos y el Cliente.~~
- ~~**Proceso de Commit (2PC)**: Un esquema con las fases (preparación y commit) y las interacciones entre Maestro y Esclavos.~~
- ~~**Fragmentación Mixta**: Un gráfico que muestre cómo se dividen los datos horizontal y verticalmente entre nodos.~~

ESTOS SON LOS GRAFICOS QUE HABRIA QUE AGREGAR