UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERA DE LA COMPUTACIÓN

Instrumentación y Evaluación de Algoritmos Distribuidos Eficientes para Medir Inconsistencias entre Réplicas en Bases de Datos Relacionales Distribuidas

TESIS
QUE PARA OBTENER EL GRADO DE
MAESTRA EN INGENIERÍA
(COMPUTACIÓN)

PRESENTA
Claudia Morales Almonte

DIRECTOR DE LA TESIS:
M. C. Javier García García

Ciudad Universitaria, México D.F. Agosto 2008

C'è un cielo sopra e dentro di noi carico di
stelle, di promesse, di futuro ...
ancora tanto da scoprire, da sperimentare
nella sua profonda realtà nella quale
"viviamo, ci muoviamo, ed esistiamo
(Atti 17, 28)"
Adelia Firetti

Agradecimientos

Agradezco a mi familia: a mis padres, a Adelia y M.S.S., gracias a cada persona que - como miembros de una única humanidad - ha participado muy de cerca en este trabajo y en cada paso de mi vida.

Agradezco a mi jurado: Dra. Amparo López Gaona, Dr. Jorge Luis Ortega Arjona, Dr. Sergio Rajsbaum Gorodesky, Dr. Renato Barrera Rivera, por sus valiosas sugerencias .

Agradezco de manera especial a mi tutor M. C. Javier García García por su apoyo, sus consejos y por la revisión atenta de mi trabajo. Gracias por el ejemplo de una vida entregada en cada gesto y con cada persona.

Agradezco a mis profesores de la maestría, por su alegría y por su entusiasmo al compartir con nosotros sus conocimientos.

Agradezco el apoyo recibido por parte del Macroproyecto: Tecnologías para la Universidad de la Información y la Computación.

Agradezco a la UNAM, por brindárnos el espacio para descubrir la riqueza de nuestra cultura y del mundo: un lugar de encuentros. “Por mi raza hablará el espíritu”.

Contenido

Índice de tablas

Índice de imágenes

Capítulo 1

Introducción

La creación de redes y el desarrollo de los sistemas distribuidos dio la posibilidad de compartir recursos de hardware y de software entre computadoras geográficamente dispersas y, además, facilitó el acceso a la información contenida en los sitios de la red. Esta última característica trajo consigo nuevos problemas para la administración y el control de los datos.

En ese contexto surgen las Bases de Datos Distribuidas (BDD) para controlar el acceso a los datos distribuidos y para asegurar su consistencia. El manejador de la Base se encarga de combinar la información de los diferentes sitios para presentarle al usuario una vista única de los datos. Las ventajas de estas Bases de Datos son la agilización de las consultas, la distribución del control, el acceso y la localización rápida de la información, entre otras. Estas características mejoran el desempeño del sistema. Por otro lado, la complejidad de la administración es mayor que en una Base de Datos Centralizada, pues se incluyen problemas para coordinar los sitios, para tratar la concurrencia, la seguridad, el manejo, la distribución de las consultas, etc.

Entre los objetivos de las Bases de Datos Distribuidas están el mejorar la disponibilidad y la recuperación de los datos. La creación de réplicas (copias de los datos) en sitios distintos puede ayudar a lograr este objetivo. Sin embargo el mantener actualizadas todas las copias de las tablas no es una tarea sencilla.

La complejidad que se encuentra al tratar de propagar las actualizaciones de una tabla a sus réplicas es grande. Entre los factores más importantes que influyen en este proceso están:

a) la comunicación entre los sitios, puesto que los datos se vuelven más vulnerables cuando tienen que viajar a través de la red para llegar al usuario final;

b) la concurrencia;

c) la disponibilidad de los sitios (si están activos todo el tiempo o no);

d) la configuración de las réplicas (maestro-esclavo, multimaestro, síncrona o asíncrona), entre otros.

La pérdida de información, el retraso en el envío y la inconsistencia de la información replicada son algunos de los problemas que encontramos en una BDD.

La inconsistencia de los datos se presenta cuando una tabla es modificada y esos cambios no llegan a realizarse en todos los sitios que tienen copias de esos datos (réplicas), o cuando varios sitios se actualizan al mismo tiempo teniendo cada uno información diferente. En una configuración multimaestro, donde todos los sitios pueden realizar modificaciones de los datos, no es posible asegurar la integridad de los mismos en todo momento.

Como veremos más adelante, mantener la consistencia de los datos en las réplicas de una Base de Datos Distribuida se traduce en la confiabilidad de los datos que extraemos de ella. Es importante que el Sistema de Bases de Datos Distribuidas tenga el control de la redundancia y de la propagación correcta de las actualizaciones.

Actualmente existen implementaciones comerciales de Bases de Datos Distribuidas que administran la información del sistema y tratan de solucionar algunos de los problemas mencionados. Además se han desarrollado herramientas de replicación que nos permiten crear y controlar las réplicas de una Base de Datos Distribuida.

En este trabajo implementamos y evaluamos algunos algoritmos para detectar los errores de inconsistencia de una tabla y sus réplicas. A lo largo de los capítulos siguientes explicaremos en que consiste cada algoritmo.

1.1 Objetivos y contribución de este trabajo

Como mencionamos anteriormente, las réplicas de una Base de Datos Distribuida pueden perder su consistencia después de haber sido actualizadas, y la detección y la corrección de estos errores es esencial para asegurar la confiabilidad de los datos. Aunque existen sistemas de replicación que tratan de evitar y/o corregir estos problemas, cuando tenemos una configuración que permite la *actualización en todas partes* (update everywhere), donde todos los sitios pueden modificar los datos en cualquier momento (véase Capítulo 2), es importante contar con herramientas que nos permitan verificar que los cambios que se hicieron a los datos fueron recibidos correctamente por todas las réplicas y medir la inconsistencia de las mismas.

Objetivos

Los objetivos que nos planteamos en este trabajo son los siguientes:

a) implementar algoritmos distribuidos para medir inconsistencias entre réplicas en Bases de Datos Relacionales Distribuidas;

b) optimizar los algoritmos distribuidos mencionados en cuanto a procesamiento y en cuanto a comunicación;

c) comparar estos algoritmos, en cuanto a su tiempo de ejecución, utilizando una Base de Datos Distribuida sintética, homogénea, con tablas replicadas;

d) establecer criterios y definir los rangos en los que es conveniente usar uno u otro algoritmo, con el objeto de poder elegir cuál de ellos es conveniente según las características de la Base de Datos Distribuida y de la red de comunicaciones.

Contribución

Las contribuciones de este trabajo son las siguientes:

1. La implementación de un algoritmo que detecta la inconsistencia entre réplicas de una BDD, adaptando los algoritmos de reconciliación de conjuntos propuesta en [GG08], nos basamos también en [OGGC07] donde se proponen métricas de integridad referencial basadas en el coeficiente de Jaccard, una de ellas para identificar la inconsistencia de réplicas. Dicha implementación se realiza mediante funciones SQL, las funciones en C de [CPI], y Java para integrar todo el cálculo.

   A este algoritmo lo llamamos DPC (Detección con Polinomio Característico), el cual utiliza el siguiente protocolo de Reconciliación de conjuntos [MTZ03]:

   ▷ CPISync (Characteristic Polynomial Interpolation for fast data Synchronization), trata los datos como objetos matemáticos representándolos con un polinomio característico. La detección de las diferencias entre las tablas es un proceso matemático.

2. La implementación de varios algoritmos que detectan la inconsistencia entre réplicas de una BDD, aplicando la operación ANTIJOIN del álgebra relacional de acuerdo a la propuesta de [GG08]. Para esto se diseñaron diferentes expresiones en SQL, que es el lenguaje estándar de manipulación de datos (Standard Query Language), para implementar el ANTIJOIN.

   A este grupo de algoritmos lo llamamos DAJ (Detección con ANTIJOIN):

   ▷ tres de ellos utilizan expresiones SQL para realizar el ANTIJOIN: DAJ_{FOJ} (con el operador FULL OUTER JOIN), DAJ_{EXC} (con el operador EXCEPT) y DAJ_{NE} (NOT EXISTS);

   - ▷ DAJ_{Java} lo hace utilizando el lenguaje Java.

3. La comparación del desempeño de estos algoritmos, estableciendo criterios para elegir alguno de ellos.

4. La definición de una métrica que nos permita saber si hay inconsistencia en las réplicas de la BDD.

Los resultados de este trabajo y otros mecanismos de detección de inconsistencia en los datos pueden ayudar a quien busca conocer el estado de una BDD en cuanto a la consistencia de los datos de las réplicas o en la aplicación de métricas, por ejemplo, las métricas de integridad referencial para Bases de Datos Distribuidas homogéneas propuestas en [OGGC07], o en la corrección de errores en las réplicas de las tablas de una BD.

Además, medir el error en los datos nos indica qué tan confiable es la información almacenada en el sitio y, en consecuencia, el resultado de las consultas que se realizan sobre ella.

1.2 Trabajos relacionados

Como mencionamos anteriormente, en [OGGC07] se proponen métricas de integridad referencial para Bases de Datos Distribuidas y una métrica basada en el coeficiente de Jaccard para medir la inconsistencia entre réplicas. Estas métricas se aplican a un sistema distribuido homogéneo con n sitios, donde existen réplicas.

En [GG08] se propone una adaptación de los algoritmos de reconciliación de conjuntos de [MTZ03] [1] , para medir la inconsistencia entre réplicas de una BDD. La propuesta en [GG08] está basada en expresar las métricas mediante expresiones de conjuntos utilizando los operadores de unión, intersección y diferencia con el objeto de distribuir la carga de trabajo de la evaluación de las métricas entre los servidores donde se encuentra distribuida la Base de Datos. Asimismo, en ese mismo trabajo se propone el uso del operador del álgebra relacional ANTIJOIN para calcular las métricas.

Se han desarrollado varios trabajos que buscan mantener la consistencia de los datos y, especialmente cuando se crean copias de los mismos, se trata de asegurar que esa información se mantenga actualizada (que las réplicas sean exactamente iguales). Un proceso empleado en muchos sistemas de replicación es la sincronización, que busca corregir la inconsistencia en los datos para detectar los errores y realiza operaciones con los datos replicados. En todo este proceso la comunicación y la transferencia de datos son fundamentales.

[1] por una sugerencia inicial del Dr. Sergio Rajsbaum Gorodesky

En [CGM00] se estudia como mantener actualizada una copia local de datos que provienen de una fuente remota y se definen dos métricas para saber la "actualidad" de los datos.

Existe mucha investigación orientada a encontrar nuevas estrategias para transmitir la información. En [GK98] se desarrollan algoritmos algebraicos para la propagación de cambios usando los operadores: SEMIJOIN, ANTI-SEMIJOIN, FULL OUTER JOIN, etc.; estos son importantes en la integración de los datos y en la revisión eficiente de las restricciones de integridad. Además se presenta un conjunto de reglas de propagación básica, ecuaciones para actualizaciones sencillas en una única relación y reglas de propagación de multi-actualizaciones para modificaciones simultáneas en relaciones múltiples.

En [PLH89] se desarrollan técnicas que buscan mejorar el procesamiento de consultas distribuidas, esto es, optimizar el plan de ejecución para atender las consultas, y se basan en dos criterios: las operaciones (que utilizan el SEMIJOIN) y la transmisión de información. El volumen de los datos transferidos puede reducirse substancialmente usando estas técnicas.

Un caso particular de inconsistencia de datos se presenta cuando la información almacenada en un PDA (Personal Digital Assistant) y en una computadora personal es actualizada de forma continua, y también donde es necesario mantener actualizados los datos almacenados en ambos. Se han desarrollado varias técnicas de sincronización para mantener la consistencia de los datos. Un nuevo protocolo para reconciliar los datos de un PDA con una PC es el CPISync explicado en [MTZ01] y [MTZ03], que representa cada conjunto de datos con un polinomio característico, define las diferencias de ambos sitios como una función racional de polinomios y utiliza la interpolación de funciones para encontrar las diferencias entre los datos. Una de las características importantes de esta técnica es que la complejidad de la comunicación depende sólo del número de diferencias entre los dos dispositivos más que de los tamaños de los conjuntos de datos. El procesamiento computacional se distribuye entre los sitios y al final de los cálculos los dos sitios tienen sus datos consistentes.

1.3 Organización del trabajo

La organización del trabajo es la siguiente:

En el **Capítulo 1** se presentan los objetivos y la contribución de este trabajo, así como los trabajos relacionados.

Antes de explicar algunos conceptos relacionados con las Bases de Datos y la replicación, en el **Capítulo 2** se presenta la teoría relacionada con un sistema de comunicaciones, la replicación y la sincronización de datos. Además se explican algunos

modelos de Reconciliación de conjuntos en los que nos basaremos para implementar la reconciliación en tablas.

En el **Capítulo 3** se definen los conceptos relacionados con las Bases de Datos Distribuidas, y las ventajas y desventajas con respecto a las Bases de Datos Centralizadas. Se tratan algunos problemas como la fragmentación y el almacenamiento de las tablas, la creación de réplicas y la inconsistencia de los datos.

En el **Capítulo 4** se explica el diseño y la implementación de la Reconciliación de Tablas basada en dos de los algoritmos presentados en el Capítulo 3 y las ventajas y desventajas de cada uno.

En el **Capítulo 5** se describe el ambiente de desarrollo en el que se realizaron los experimentos, el conjunto de datos, las mediciones y las variantes de los experimentos, y se presentan los resultados obtenidos.

En el **Capítulo 6** se presentan las conclusiones y los trabajos futuros.

Capítulo 2

Conceptos de redes y técnicas de sincronización de datos

Uno de los principales factores que diferencian las Bases de Datos Centralizadas de las Bases de Datos Distribuidas es la distribución de los datos y la comunicación entre los sitios. En este capítulo estudiaremos conceptos relacionados con las redes de comunicación, los sistemas distribuidos y el problema de la complejidad de la comunicación.

Además, tomando en cuenta que muchos de los problemas de los sistemas distribuidos se presentan también en las Bases de Datos Distribuidas, trataremos el caso de la inconsistencia de datos y los algoritmos que se han propuesto para tratar de solucionarla.

2.1 Redes de comunicación

Una red de computadoras está formada por *sitios* (hosts) conectados por una *red de comunicación*, capaces de realizar tareas de forma autónoma y de compartir información, recursos y servicios.

La red de comunicación provee al sistema la capacidad para que un proceso, que se ejecuta en algún sitio, envíe y reciba mensajes de procesos que se están ejecutando en sitios remotos.

Entre los parámetros que deben considerarse en la comunicación se incluyen:

a) el retraso con el cual el mensaje llega a su destino: si hay tráfico en la red, la espera para el envío es grande;

b) el costo de transmisión: hay un costo asociado a cada mensaje, más un costo adicional proporcional a la longitud del mensaje;

c) la confiabilidad de la red: es esencialmente la probabilidad de que el mensaje se entregue correctamente a su destino.

En muchas aplicaciones es necesario que un proceso de un sitio envíe el mismo mensaje a varios sitios; esta operación se llama *broadcast*. Hay redes en las que el costo de hacer un broadcast es el mismo que enviar el mensaje a una sola máquina.

Existen diferentes *tipos de redes*:

LAN (red de área local), MAN (red de área metropolitana), WAN (red de área amplia: los nodos están geográficamente dispersos). Esta clasificación se basa en la extensión de la red (según la distancia física).

La *topología de la red* se refiere a la manera en la que las máquinas están distribuidas (la forma lógica de la red), algunas de ellas son: bus, estrella, anillo, árbol, malla, etc...

Complejidad de la comunicación

A continuación se describe el concepto de complejidad de la comunicación [KN97].

En un sistema, formado por x sitios, debe ejecutarse una tarea que depende de la información distribuida en los diferentes sitios; la meta es realizar la tarea con el menor número de comunicaciones entre los sitios. En este caso no nos interesa el número de pasos del proceso computacional, el tamaño de la memoria u otro recurso utilizado.

La complejidad de la comunicación trata de cuantificar el número de mensajes requeridos para un determinado cómputo distribuido; esto es, cuántas veces tienen que comunicarse los sitios, y cuánta información tienen que transmitirse.

Este problema se plantea en el modelo de Yao [KN97] que supone dos sitios conectados en red, cada uno con una parte fija de la información del sistema. La tarea que debe ejecutarse es una función $\mathcal{F}$, especificada como entrada del modelo. Aunque la solución más inmediata sería enviar toda la información de un sitio a otro y ejecutar ahí la función, la idea es encontrar los caminos más cortos para calcular $\mathcal{F}$ con el menor número de bits transmitidos.

La diferencia entre los algoritmos que planteamos en este trabajo, está precisamente en el número de bits que envían, y en consecuencia el tiempo que emplean para hacerlo.

Este problema abstracto es relevante no sólo en la optimización de redes de computadoras, aparece también en otros contextos como en el estudio de las estructuras de datos, en el diseño de circuitos, entre otros.

2.2 Sistemas distribuidos

Con el desarrollo de las redes locales de alta velocidad, a principios de los 70s empezó también el crecimiento de los sistemas distribuidos. Estos sistemas están formados

por varias entidades que pueden estar en ambientes iguales (homogéneos) o diferentes (heterogéneos), cada una con información parcial y/o redundante del sistema completo (véase Figura 2.1).

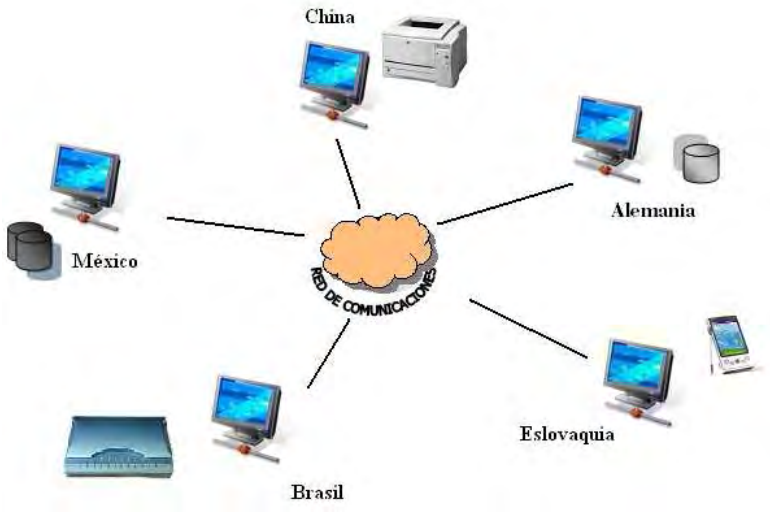

Figura 2.1: Ejemplo de un Sistema Distribuido

Algunas de las características de los sistemas distribuidos son: recursos compartidos, concurrencia, escalabilidad, disponibilidad, tolerancia a fallas y transparencia.

La creación de réplicas facilita la instrumentación de algunas de ellas. Sin embargo no hay que perder de vista que existen criterios que se contraponen, por ejemplo, la disponibilidad y la consistencia de los datos (a mayor número de copias de los datos la disponibilidad es mayor, pero la consistencia es menor).

Entre las ventajas de los sistemas distribuidos están la capacidad de compartir y acceder a la información, la distribución del control, la disponibilidad y la agilización de consultas. Algunas de las desventajas son la complejidad para lograr la coordinación correcta entre los nodos y la complejidad de la comunicación.

La necesidad de acceso a los servicios y recursos de estos sistemas ha propiciado nuevos campos de desarrollo, entre ellos están los Sistemas de Bases de Datos Distribuidas.

Los *Sistemas de Bases de Datos Distribuidas* (SBDD) son un caso particular de los sistemas de cómputo distribuido. Algunas de sus características son la autonomía de cada sitio en sus capacidades de procesamiento y la capacidad de realizar operaciones locales de los datos.

Es importante distinguir la administración de una Base de Datos Distribuida del

procesamiento distribuido. El primero tiene que ver con la administración coordinada de los datos distribuidos en varias computadoras separadas, pero interconectadas. El segundo se basa en un conjunto de programas que pueden hacer llamadas a programas de otros sitios de la red o del mismo sitio [Cod90] y están distribuidos en un sistema de varias computadoras separadas, pero interconectadas.

2.2.1 Replicación de datos

La replicación de datos es una de las tecnologías clave de los sistemas distribuidos [CDK01].

La replicación consiste en crear copias múltiples de los datos llamadas réplicas, en computadoras separadas (clientes o servidores) y nos permite alta disponibilidad, mejora en la ejecución y tolerancia a fallas, como se detalla a continuación.

- ▷ *Alta disponibilidad.* Los usuarios requieren que los servicios sean altamente disponibles, esto es, que a los servicios se acceda con tiempos de respuesta razonables. Algunos escenarios en los que resulta relevante la alta disponibilidad de los datos son aquéllos donde existen constantes fallas en los servidores y aquéllos en los que existen redes particionadas y/o la operación de los sitios constantemente desconectados.

- ▷ *Mejora en la ejecución.* Ocurre cuando los clientes y los servidores almacenan copias de los recursos, para evitar retraso en la carga de los datos del servidor original. La replicación de datos inmutables es trivial e incrementa la ejecución con un costo pequeño para el sistema. Tener réplicas de datos que cambian, como los de la Web, significa una sobrecarga en los protocolos que se encargan de asegurar que los clientes reciban las actualizaciones de los datos.

- ▷ *Tolerancia a fallas.* La alta disponibilidad no significa necesariamente que los datos son correctos, pues pueden no estar actualizados. La tolerancia a fallas de un servicio garantiza el funcionamiento estrictamente correcto a pesar de tener varias fallas.

Un requisito importante cuando los datos son replicados es la *transparencia de la replicación*, esto es, los usuarios no deben estar conscientes de que existen múltiples copias físicas de los datos. Otro requisito de la replicación es la *consistencia*, que tiene que ver con las operaciones realizadas sobre una colección de objetos replicados. En un escenario de esta naturaleza los resultados deben cumplir las especificaciones de correctez para esa colección. Si se realizan operaciones con los datos cuando los sitios están desconectados, puede presentarse inconsistencia, al menos temporalmente.

Existen dos tipos de replicación: la pesimista y la optimista [SS05]. Las técnicas de replicación tradicional tratan de mantener una única réplica consistente bloqueando el acceso a la réplica mientras ésta esté siendo modificada. Estas técnicas son conocidas como *pesimistas*. La replicación *optimista* es un conjunto de réplicas para compartir información eficientemente en una red amplia o en ambientes móviles. La característica que la distingue de la replicación pesimista es que ésta permite el control concurrente.

2.2.2 Modelos de sincronización en réplicas

La clave de un sistema distribuido efectivo que utilice réplicas es mantener las copias de los datos y otros recursos actualizados (*Consistencia en la replicación*) [CDK01].

La *sincronización de datos* es el proceso de establecer la consistencia entre los datos de sitios remotos y mantenerla durante el tiempo. Esto es importante en una gran variedad de aplicaciones: en la sincronización de archivos y de PDA, incluyendo la sincronización de una tabla con su réplica en una Base de Datos Distribuida, etc.

Existe una gran variedad de modelos teóricos sobre este tema. Los modelos se clasifican de acuerdo a cómo son considerados los datos para la sincronización.

La sincronización de datos no ordenados, conocido como *Reconciliación de conjuntos*, trata de encontrar las diferencias entre dos conjuntos.

Dentro del contexto de las Bases de Datos Distribuidas, sean T_1 y T_2 réplicas de una tabla, las diferencias entre ambas réplicas serán:

$$T_1 \oplus T_2 = (T_1 - T_2) \cup (T_2 - T_1)$$

Algunas soluciones al problema de la inconsistencia son las siguientes:

a) *sincronización matemática*. En este caso los datos son tratados como objetos matemáticos y la sincronización se realiza con un proceso matemático;

b) *transferencia completa*. Todos los datos se transfieren a un sitio local para ser comparados;

c) *sincronización con marcas de tiempo*. Consiste en asignar una marca de tiempo a los datos. Para realizar la reconciliación se transfieren a uno de los sitios todos los datos con una marca de tiempo mayor a la sincronización previa.

En este trabajo nos enfocamos a las dos primeras soluciones para detectar los errores de inconsistencia entre réplicas de una BDD, a continuación se explican más a detalle.

2.2.3 Reconciliación usando Sincronización matemática

En [MTZ03] se propone un protocolo de reconciliación de conjuntos basado en la interpolación de polinomios característicos, llamado *CPISync* (Characteristic Polynomial Interpolation for fast data synchronization).

Podemos plantear el problema de la reconciliación en el contexto de las Bases de Datos Distribuidas como sigue:

sean T_1 y T_2 dos réplicas de la tabla T, localizadas en los sitios S_1 y S_2. Determinar las tuplas diferentes entre ellas con el menor número de comunicaciones entre los dos sitios y con el menor número de bits enviados. Supondremos que los datos no están ordenados, como sucede en las tablas en el modelo relacional.

La diferencia entre las tuplas de las réplicas se denota como:

$$I_1 = T_1 - T_2 \text{ y } I_2 = T_2 - T_1$$

Denotaremos como:

I_1 a la tabla con las tuplas de T_1 que no están en T_2;

I_2 a la tabla con las tuplas de T_2 que no están en T_1;

m_1 y m_2 a las cardinalidades de las tablas I_1 e I_2, respectivamente;

m al número de tuplas diferentes entre las réplicas, donde $m = m_1 + m_2$;

$\bar{m}$ a la cota superior de m.

En [MTZ03] los autores proponen una reconciliación de conjuntos. En este trabajo la medición de inconsistencia se realiza considerando los conjuntos formados por los valores de las llaves primarias de las tablas y en términos absolutos será m.

No es la intención de este trabajo dar los detalles técnicos del algoritmo (véase [MTZ03] para detalles), es por eso que a continuación presentamos una visión intuitiva del algoritmo de reconciliación aplicado a tablas.

Ejemplo 2.1

Tomaremos como ejemplo una tabla llamada $DATA$, la cual tiene dos réplicas, $DATA_1$ y $DATA_2$ en sitios distintos, S_1 y S_2, respectivamente. La llave primaria de la tabla es d_pk, que tiene valores enteros positivos de 1 byte (8 bits).
Supongamos que originalmente los valores de las llaves de las réplicas son: $d\_pk = \{1, 2, \ldots, 100\}$ cada una con 100 tuplas.

Suponiendo que las réplicas fueron actualizadas (véase Figura 2.2), en $DATA_1$ se insertaron nuevas tuplas con valores de $d\_pk = 101, 102, 103$ y se eliminó la tupla con

$d\_pk = 100$; en $DATA_2$ se insertaron dos tuplas una con $d\_pk = 201$ y otra con $d\_pk = 202$.

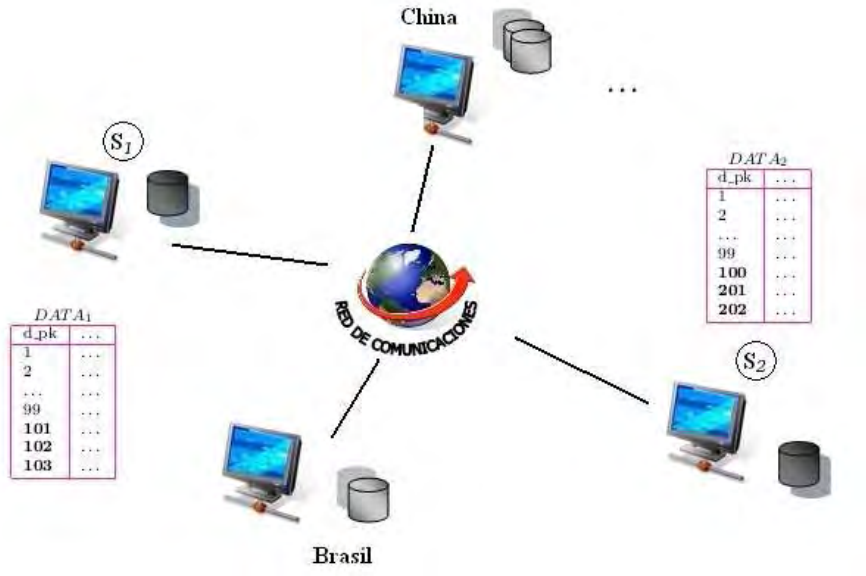

Figura 2.2: Estado de las réplicas después de la actualización

El protocolo de reconciliación que utiliza el algoritmo DPC representa cada conjunto de datos a reconciliar a través de un Polinomio Característico (PC) y, después, realiza la interpolación de la función racional formada por el cociente de los dos polinomios utilizando los coeficientes de los polinomios característicos evaluados en puntos de muestreo. Lo anterior se explicará con un ejemplo a continuación. Utilizaremos álgebra modular en todos los cálculos que realicemos para evitar manejar números grandes. Es importante aclarar que en este protocolo se debe determinar previamente una cota superior del tamaño de m. A esta cota la denotaremos como $\bar{m}$, que será el número máximo de valores diferentes entre las réplicas.

En este ejemplo suponemos que conocemos $\bar{m}$, ya que $DATA_1$ nunca va a tener más de 10 tuplas diferentes con respecto a $DATA_2$ y viceversa ($\bar{m} = 20$).

Para obtener las diferencias procedemos de la siguiente forma:

- ▷ Los sitios previamente acuerdan 20 puntos de muestreo ($evalPoints$), 10 para los posibles errores (diferencias) de cada réplica. Los $evalPoints$ que se elijan no deben de ser raíces de los polinomios característicos, pues en esos valores la evaluación de los polinomios sería 0. Necesitamos definir un campo finito para mapear los valores de d_pk y los $evalPoints$. El valor máximo de d_pk es $2^7 - 1$

(pues de los 8 bits uno representa el signo), esto es, el número mayor que tenemos con 7 bits es el 126.

Para asegurar que los $evalPoints$ sean diferentes a los posibles valores de d_pk, elegimos valores mayores a 126 y menores a $(2^7 - 1) + 20 = 147$ (de acuerdo al valor de $\bar{m}$).

El campo para el mapeo de los datos, F_q, que elegimos es de orden $q = 149$ que es un número primo mayor a 147.

En este ejemplo elegimos como $evalPoints$ los enteros del intervalo $[130, 149]$, que son valores de 8 bits, mapeados a $(-1, -2, \ldots, -20)$.

▷ Con los valores definidos anteriormente podemos iniciar la identificación de las diferencias.

El polinomio característico se define como:

$$PC(x) = (x - e_1)(x - e_2)\ldots(x - e_n)$$

donde $e_1, e_2, \ldots e_n$ son los valores de las llaves primarias de las réplicas, y n es la cardinalidad de la réplica.

El cálculo de PC se realiza de forma concurrente en cada sitio, para los mismos $evalPoints$ (sustituyendo x con cada punto de muestreo).

El polinomio del conjunto de valores de la llave primaria d_pk de la réplica $DATA_1$ para los puntos de muestreo -1 y -2, calculado en S_1, es:

$$PC_1(-1) = (-1-1)(-1-2)\ldots(-1-99)(-1-101)(-1-102)(-1-103)\ mod\ q = 15$$
$$\text{y}$$
$$PC_1(-2) = (-2-1)(-2-2)\ldots(-2-99)(-2-101)(-2-102)(-2-103)\ mod\ q = 129$$
$$\ldots$$

Cuando se requiere calcular las diferencias entonces se procede de la siguiente forma: S_1 envia a S_2 los resultados de PC_1.

El polinomio del conjunto de valores de la llave primaria d_pk de la réplica $DATA_1$ para los puntos de muestreo -1 y -2, calculado en S_2, es:

$$PC_2(-1) = (-1-1)(-1-2)\ldots(-1-100)(-1-201)(-1-202)\ mod\ q = 132$$
$$PC_2(-2) = (-2-1)(-2-2)\ldots(-2-100)(-2-201)(-2-202)\ mod\ q = 73\ \ldots$$

Obsérvese que el polinomio característico debe recalcularse cada vez que se reciba o se elimine una tupla.

▷ El sitio S_2 calcula la división ($mod\ q$) de los polinomios para cada $evalPoint$, realiza la interpolación para obtener las diferencias en los valores de las llaves y también para obtener las diferencias entre las réplicas.

$$\frac{PC_1(x)}{PC_2(x)} = \frac{(x-1)(x-2)\ldots(x-99)(x-101)(x-102)(x-103)}{(x-1)(x-2)\ldots(x-99)(x-100)(x-201)(x-202)}$$

Con esta ecuación queremos remarcar cual es el punto central de este protocolo. El numerador y el denominador son las evaluaciones del polinomio característico en un punto de muestreo dado. Los elementos diferentes, $I_{1/2}$ y $I_{2/1}$, son recuperados aplicando interpolación de funciones racionales a través de la eliminación de Gauss.

Podemos ver que los factores iguales, del numerador y del denominador, se eliminan, $(x-1)(x-2)\ldots(x-99)$, obteniendo una función reducida:

$$\frac{PC_1(x)}{PC_2(x)} = \frac{(x-101)(x-102)(x-103)}{(x-100)(x-201)(x-202)}$$

Los grados del numerador y del denominador de la función racional reducida son m_1 y m_2 (las tuplas diferentes entre ambos conjuntos de datos).

Las tuplas diferentes entre las dos réplicas serán:

$$I_{1/2} = \{101, 102, 103\} \text{ y } I_{2/1} = \{100, 201, 202\}$$

▷ Los resultados de la obtención de diferencias se envían a S_1, para poder después corregir los errores.

Obsérvese que en este protocolo de Reconciliación después de que S_2 encontró las tuplas faltantes a ambas réplicas, éstas son enviadas al sitio S_1 para corregir los errores. Esta tesis no tiene por objetivo corregir los errores de inconsistencia sino detectar los errores para saber el estado de una réplica.

Todas las operaciones se efectúan en álgebra modular (el cálculo de los polinomios, las divisiones, la interpolación, etc.) para evitar resultados con números muy grandes.

Para ejemplificar el funcionamiento de este protocolo nos basamos en el ejemplo de Reconciliación de conjuntos presentado en [GG08].

Complejidad del algoritmo de la obtención de diferencias

Dos características importantes que debemos considerar al aplicar este algoritmo de obtención de diferencias son la complejidad de la comunicación entre los sitios y la complejidad computacional de los algoritmos en cada sitio.

a) En este algoritmo la comunicación entre los sitios es mínima.
S_1 le envía a S_2 el tamaño de la réplica T_1, los resultados del polinomio característico construido con los puntos de muestreo y los valores de la llave primaria de T_1 (para este ejemplo son 20).
La complejidad total de la comunicación es: $O(\bar{m} + 1)$ cadenas enviadas.

b) La complejidad de procesamiento del algoritmo tiene dos componentes:
el costo de evaluar los polinomios característicos
$O(n * \bar{m})$, n es el tamaño de la tabla;
el costo de la interpolación y la factorización es $O(\bar{m}^3)$ operaciones.
La complejidad total es: $O(n * \bar{m} + \bar{m}^3)$ operaciones.

2.2.4 Reconciliación usando Transferencia completa

Otra solución al problema de obtener las diferencias es comparar directamente los conjuntos de datos uno a uno. Esta estrategia, *Transferencia completa*, envía todos los datos de un sitio a otro para hacer una comparación local. De esta forma detecta los valores diferentes entre ambos conjuntos.

Aunque comparar es una operación muy sencilla, el problema de realizar la reconciliación de esta forma involucra el envío de todos los datos de una réplica a los demás sitios, además del tiempo de cálculo. El proceso implica que se envíen n mensajes, donde n es el tamaño de la tabla.

Basándonos en la estrategia de *Transferencia completa* desarrollamos diversos algoritmos que aplican la operación ANTIJOIN del álgebra relacional para detectar las diferencias entre réplicas:

- DAJ_{FOJ} que implementa el ANTIJOIN con el operador FULL OUTER JOIN;
- DAJ_{EXC} que implementa el ANTIJOIN con el operador EXCEPT;
- DAJ_{NE} que implementa el ANTIJOIN con el operador NOT EXISTS;
- DAJ_{Java} que utiliza el lenguaje Java.

Estos algoritmos se basan en la comparación de los datos, en busca de los valores no actualizados en los dos conjuntos (véase Figura 2.3).

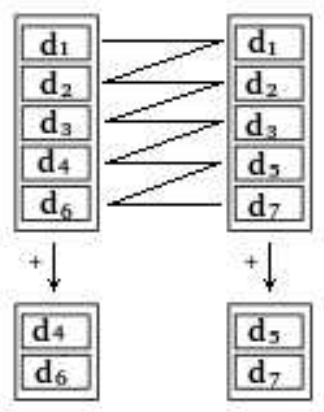

Figura 2.3: Transferencia completa, búsqueda de valores diferentes

Estos algoritmos se explicarán a detalle más adelante.

Capítulo 3

Réplicas en Bases de Datos Distribuidas

En las siguientes secciones se definen conceptos de Bases de Datos Distribuidas. Asimismo, se mencionan sus ventajas y desventajas con respecto a las Bases de Datos Centralizadas; resaltamos la importancia de tener réplicas consistentes de las tablas.

3.1 Bases de Datos Distribuidas

Una *Base de Datos Distribuida* (BDD) es una colección de datos que pertenecen lógicamente al mismo sistema, pero que están dispersos en los diferentes sitios de la red (véase Figura 3.1). Sea $\mathcal{D}$ la BDD global definida como: $\mathcal{D} = \{D_1, D_2, \ldots, D_x\}$, un conjunto de x BD relacionales definidas en x sitios. Cada una de éstas tiene información parcial del sistema y es administrada de forma autónoma.

Sea T una tabla del esquema de la BDD. Denotaremos con T_i la réplica de T en el sitio i, o sea T_i está en el esquema de D_i. Supondremos que el atributo k es la llave primaria de la tabla T. (La notación que utilizamos se basa en [OGG08]).

Cuando se requiere leer o actualizar la información almacenada en la BDD se realizan consultas globales que se traducen a consultas fragmentadas para procesarse en forma distribuidas.

Una BDD satisface, como mínimo, las siguientes cuatro condiciones [Cod90]:

1. los datos están distribuidos en dos o más sitios;
2. los sitios están conectados a una red de computadoras (tipo LAN, MAN, WAN, u otra);

3. en cualquier sitio los usuarios y los programas pueden manejar los datos en su totalidad como si fuera una única BD almacenada en el sitio;

4. todos los datos localizados, que son parte de la BD global, pueden ser manipulados por los usuarios de este sitio de la misma forma como si fuera una BD local aislada del resto de la red.

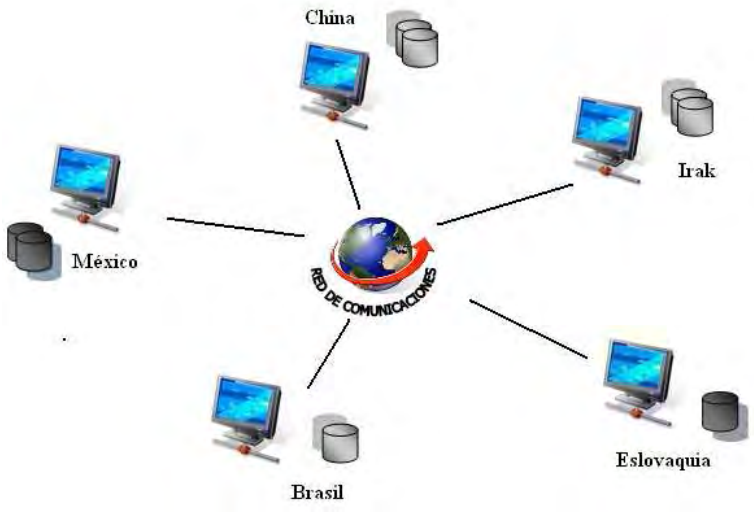

Figura 3.1: Ejemplo de una BDD

Según el tipo de plataforma de los sitios involucrados, el sistema de BDD puede ser *homogéneo* o *heterogéneo*. En el primero todos los sitios tienen el mismo manejador de BD, son conscientes de la existencia de los demás sitios y acuerdan cooperar en el procesamiento de las consultas de los usuarios, aunque la autonomía no es total. En el segundo los manejadores son distintos y puede ser que no todos los sitios tengan información de la existencia del resto y que la cooperación en el procesamiento de las transacciones sea limitada [SKS98]. La complejidad es mayor que en un sistema homogéneo, pues se añaden problemas de traducción entre los modelos de datos de cada sitio.

Un *Sistema Manejador de Bases de Datos Distribuidas* (SMBDD) es un software que permite la gestión de una BDD. Una de las metas del manejador es propiciar la *autonomía local* para que el administrador de una localidad pueda ser independiente del resto del sistema distribuido. Cuando uno de los sitios tiene alguna falla de software o hardware, los otros sitios tienen que estar habilitados para continuar ejecutando sus

tareas, excepto aquellos involucrados con los sitios que están temporalmente fuera de funcionamiento.

En una BDD pueden presentarse las siguientes fallas:

1. fallas en un sitio;
2. pérdida de mensajes;
3. fallas en la conexión con la red;
4. partición de la red (ésta se divide en dos o más subsistemas con conexión escasa), entre otras.

Principios fundamentales

En las BDD podemos considerar como principio fundamental el siguiente:

para el usuario el sistema distribuido debería parecer, exactamente, como si no fuera distribuido.

A continuación mostramos doce reglas que se deben cumplir para lograr este principio [Dat03].

1. *Autonomía local.* Los sitios en un sistema distribuido son autónomos. Esto significa que las operaciones hechas en un sitio son controladas por ese sitio y que no depende de otros sitios para su correcto funcionamiento.
2. *No dependencia de un sitio central.* La regla anterior implica que todos los sitios son tratados como iguales, no hay dependencia de un sitio central o maestro.
3. *Operación continua.* Una ventaja de los sistemas distribuidos es que proveen confianza y disponibilidad.
   - ▷ *Confianza*: indica la probabilidad de que el sistema esté operativo y funcionando en un momento determinado.
   - ▷ *Disponibilidad*: indica la probabilidad de que el sistema esté operativo y esté funcionando contínuamente en un periodo determinado.
4. *Independencia de localización* (Transparencia de localización). Los usuarios del SBDD no necesitan conocer la ubicación física de los datos, pues para ellos es como si los datos estuvieran almacenados en el propio sitio local.

5. *Independencia de fragmentación.* Los usuarios trabajan con tablas globales, no necesitan saber el modo en que se ha fragmentado una determinada tabla.

6. *Independencia de replicación.* Se refiere a que el SMBDD tiene control del número de réplicas que se crean (redundancia). Los usuarios ven cada objeto de datos lógicamente como único y el sistema puede crear réplicas de estos objetos sin que el usuario requiera saber de su existencia (Transparencia de replicación).

7. *Procesamiento de consultas distribuidas.* Se refiere a cómo encontrar estrategias eficientes para satisfacer consultas que involucran varios sitios y en las que se pueden dar varias formas de mover los datos en el sistema.

8. *Administración de transacciones distribuidas.* Hay dos aspectos importantes en cuanto a la administración de transacciones: la recuperación y la concurrencia.

   ▷ *Recuperación*: para asegurar que una transacción es atómica (ejecuta todo o nada) en un ambiente distribuido, el sistema tiene que garantizar que el conjunto de agentes de una transacción envíe mensaje de comprometido (commit) o abortado (roll back) al mismo tiempo. Esto puede lograrse, por ejemplo, con el protocolo *two phase commit* [EN03].

   ▷ *Concurrencia*: debe existir un adecuado control de la concurrencia. Esto se fundamenta en la utilización de diferentes protocolos basados, por ejemplo, en el bloqueo (*locking* [EN03]), como en un sistema no distribuido.

9. *Independencia del hardware.* En un sistema formado por computadoras con plataformas diferentes es necesario poder integrar los datos para presentarle al usuario una única imagen del sistema. Es deseable que todos los sitios puedan utilizar el mismo SMBDD en diferentes plataformas de hardware y además lograr que todas las máquinas colaboren en el sistema distribuido.

10. *Independencia del sistema operativo.* Como en el punto anterior, es deseable también poder correr el SMBDD en diferentes plataformas de sistemas operativos.

11. *Independencia de la red.* Tiene que ver con la capacidad del sistema de soportar diferentes redes de comunicación.

12. *Independencia del SMBD.* Es necesario que las instancias de los SMBDD en sitios diferentes soporten la misma interfaz, aunque no todos los sitios tengan el mismo manejador.

Entre los objetivos más importantes de la arquitectura de una BDD están: separar la fragmentación de los datos de la localización, tener control explícito de redundancia,

lograr la independencia de los SMBDD, alcanzar los diferentes tipos de transparencia y otros de los puntos mencionados anteriormente.

En la Figura 3.2 se muestra una arquitectura de referencia, pues sus niveles son conceptualmente relevantes para entender la organización de una BDD.

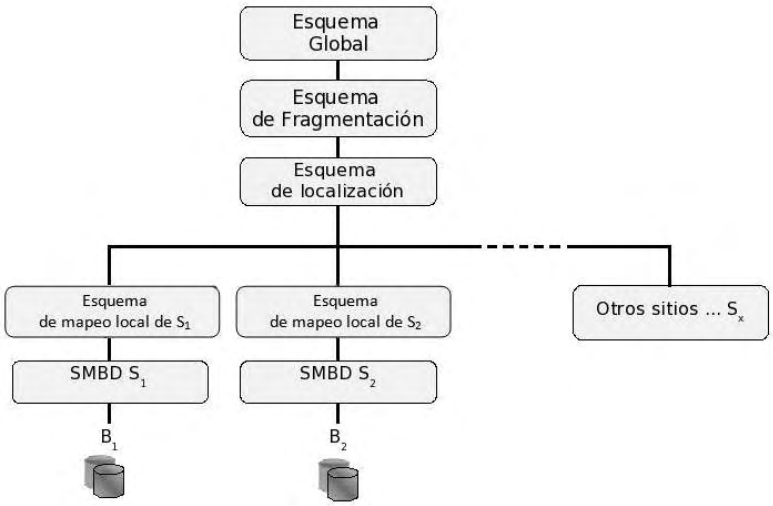

Figura 3.2: Arquitectura de referencia de una BDD

- El *esquema global* define todos los datos que están contenidos en la Base de Datos, como si no fuera distribuida. La existencia de un esquema global distingue el SMBDD de otras familias, como los SMBD (Sistema Manejador de Bases de Datos) federados o multi SMBD, que podrían hacer público sólo parte de su esquema.

- Cada relación global puede ser dividida en varios fragmentos, y el mapeo entre las relaciones globales y los fragmentos está definido en el *esquema de fragmentación.*

- El *esquema de localización y replicación* define el sitio en el cual un fragmento/réplica está localizado. Cada tabla o fragmento puede estar replicado en una o más localidades de la red.

- El *esquema de mapeo local* traduce los fragmentos locales a los objetos que serán manipulados por el SMBD local. Tal mapeo puede ser sencillo, si los SMBD locales son homogéneos, o complejo para sistemas heterogéneos que tienen diferentes modelos de datos.

La creación de una BDD implica una nueva fase, además de las existentes en BD Centralizada: el *Diseño de la distribución*, cuyo objetivo es generar los esquemas locales a partir de los globales. La Figura 3.3 nos muestra todas las fases para crear la BDD y sus salidas.

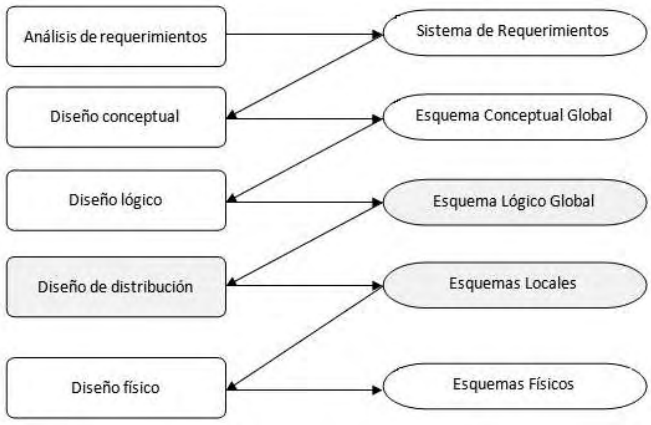

Figura 3.3: Fases en la creación de una BDD

3.1.1 Almacenamiento de las tablas en la BDD

Una tabla T puede ser almacenada de varias formas en el sistema de BDD. Es importante ocultar al usuario los detalles de la distribución. Esto se denomina *transparencia de la red* y se relaciona con la autonomía local. La distribución más simple consiste en asignar todas las tablas de la BDD a los sitios $S_1, S_2, \ldots, S_x$ de la red, sin ninguna transformación. En algunos casos esta aproximación puede ser adecuada, pero en otros tiene que buscarse otras opciones de distribución de los datos.

La cardinalidad o el grado de una misma tabla puede variar en dos sitios diferentes, pues en uno puede existir una fragmentación horizontal (sólo algunas tuplas) o una fragmentación vertical (sólo algunos atributos) de T.

En el diseño de la distribución de una BDD existen dos problemas: la fragmentación y la asignación de fragmentos.

La fragmentación tiene que ver con el problema de partir la tabla. Dígase T: $T(F_1, F_2, \ldots, F_i)$, aplicando la fragmentación horizontal, vertical o mixta. Los fragmentos tienen que cumplir tres condiciones: la completitud, la reconstrucción y la disyunción.

La Figura 3.4 nos muestra un ejemplo de los fragmentos que pueden estar almacenados en varios sitios cuando se hace una consulta de esos datos.

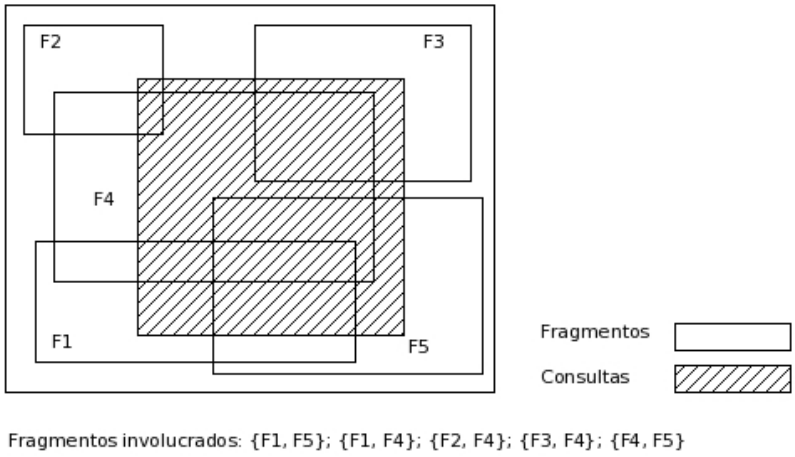

Figura 3.4: Fragmentación de una BDD

El objetivo de fragmentar la BD en tablas más pequeñas y guardar cada fragmento en un sitio diferente es minimizar el número de accesos remotos que realiza la BDD, pues el costo de la comunicación es alto. Además las consultas pueden ser procesadas de forma paralela, reduciendo la transmisión de información.

El problema de la asignación de fragmentos tiene que ver con dónde y cuántas veces almacenar cada fragmento o la tabla completa. Las tablas pueden ser almacenadas en exactamente un sitio (técnica de *Best fit* o no redundante) o pueden crearse réplicas de éstas donde el beneficio de tener una copia de esos datos es más alto que su costo (*All benefical Sites* o redundante). *Additional Replication* es una técnica de replicación que inicia su operación con una réplica (*Best fit*) y va agregando copias del fragmento mientras sea conveniente [CP84]. En el presente trabajo, sin pérdida de generalidad, supondremos que para una tabla existe sólo un fragmento, que puede estar replicado en uno o más sitios.

3.1.2 Ventajas y desventajas de una BDD

La distribución de los datos y de las aplicaciones proporciona ventajas al sistema. Estas ventajas pueden considerarse como los objetivos del SMBDD. A continuación mencionamos algunas de ellas:

- ▷ proporciona autonomía local;
- ▷ mejora el desempeño del sistema;
- ▷ otorga confiabilidad y disponibilidad de los datos;
- ▷ disminuye el costo de comunicación y de mantenimiento;
- ▷ tiene capacidad de expansión.

Las primeras tres ventajas ya se explicaron anteriormente. Las demás, aunque no se han mencionado, se pueden deducir de lo que hemos visto hasta ahora. El costo se refiere a que, teniendo la información distribuida, muchas de las operaciones pueden realizarse de forma local disminuyendo la comunicación entre los sitios y, en consecuencia, el costo de la comunicación. Aquí influye, además, el costo de mantenimiento de los equipos: es más fácil mantener cada computadora del sistema en cada sitio que mantener una sola máquina con el poder de todo el sistema junto. La capacidad de expansión se refiere a que el sistema distribuido soporta el crecimiento incremental con un grado mínimo de impacto en los datos ya existentes.

Sin embargo encontramos también desventajas en una BDD:

- ▷ aumenta la complejidad;
- ▷ requiere una mayor infraestructura del sistema;
- ▷ distribuye el control;
- ▷ disminuye la seguridad.

Para asegurar la transparencia del sistema distribuido, el SMBDD tiene que enfrentarse a nuevos problemas que no se presentaban en BD Centralizadas: la complejidad de algunas operaciones aumenta y la complejidad del sistema requiere tener una infraestructura más extensa (seguridad, comunicación, etc.). Esto significa un costo extra. La distribución del control crea problemas de sincronización y coordinación entre los sitios. Por eso, es importante considerar la dificultad para mantener la integridad de

los datos y los problemas que se presentan al propagar la actualización de las réplicas. El acceso a los datos debe estar protegido para que los datos estén seguros.

Como vimos anteriormente, la distribución de los datos en una BDD, implica considerar aspectos diferentes a los de las BD Centralizadas. La Tabla 3.1 nos indica algunas de estas diferencias.

BD Centralizada	**BD Distribuida**
El control es centralizado. Tiene un solo SMBD.	El control es jerárquico. Tiene un SMBD global y varios SMBD locales.
La organización de los datos es transparente para el usuario. Independencia de Datos.	La localización de los datos es transparente para el usuario. Transparencia en la Distribución.
Existe una sola tabla con los datos. Reducción de redundancia.	Pueden existir réplicas de las tablas. Redundancia de los Datos.
El control del acceso a los datos es mayor. Seguridad alta.	Se agregan problemas de seguridad causados por la red.
Todos los sitios accesan a una sola BD para atender las peticiones.	La disponibilidad de los datos en varios sitios agiliza las consultas.

Tabla 3.1: BD Centralizada vs BD Distribuida

La probabilidad de que la falla en un nodo afecte al sistema distribuido es baja: esto se debe a la autonomía e independencia entre los nodos.

A pesar de las ventajas, el diseño y la administración de las BDD constituyen un gran desafío que incorpora problemas que no estan presentes en las Bases de Datos Centralizadas, por ejemplo, los esquemas de fragmentación y localización de información, el manejo de consultas a sitios distribuidos y los mecanismos de control de concurrencia y confiabilidad en Bases de Datos Distribuidas. Existe la probabilidad de violaciones de seguridad. No se puede asegurar la integridad en presencia de fallas. El aumento de la complejidad se refleja en el costo del software, mayor posibilidad de errores y tiempo extra de procesamiento.

3.1.3 Implementaciones de SMBDD

A continuación mencionamos brevemente algunos de los prototipos e implementaciones de SMBDD relacionales.

Tres de los primeros prototipos más conocidos son:

- *SDD-1* que fue construido en la división de investigación de 'Computer Corporation of America' a finales de los años 70s y principios de los 80s;

- ▷ *R** es una versión distribuida del prototipo System R, construida en el IBM Research a principios de los años 80s;
- ▷ *Distributed Ingres*, una versión distribuida del prototipo *Ingres*, construida en la Universidad de California en Berkeley a principios de los años 80s.

Muchos de los manejadores de BD más conocidos han implementado también manejadores para BDD (con diversos grados de funcionalidad), entre ellos están:

- ▷ *Oracle* (en las versiones 7, 8, 9, 10, 11) [Ora];
- ▷ *SQL Server* (SQL Server 7, 2000, 2005) [SS];
- ▷ *Ingres Star* [IS];
- ▷ *DB2* (DB2 Everyplace) [DB2].

En cuanto a los SMBD de código abierto más utilizados, que cuentan con otras herramientas para el manejo de datos distribuidos, tenemos los siguientes:

- ▷ para MySQL [MyS]: *MySQL Cluster*, *Accessing Distributed Data with the Federated Storage Engine*, *MySQL Distributed Replicated Block Device (DRBD)*;
- ▷ en PostgreSQL [Pos]: herramientas de replicación como Slony-I, PostgreSQL Replicator, PostgreSQL-R, etc.; algunos de estos se mencionan más adelante.

3.2 Replicación de tablas

Algunas veces dos o más sitios necesitan acceder frecuentemente a la misma información de ciertas tablas de la BD. El SMBDD puede mantener copias idénticas de los datos a través de la BDD.

Se dice que una tabla T o un fragmento de la tabla es *replicado* cuando éste se almacena redundantemente en dos o más sitios. La *replicación completa* (full replication) de una tabla se da cuando la tabla es almacenada en todos los sitios. Una *BD completamente redundante* es aquella en la que cada sitio contiene una copia de la BD entera.

Las razones por las cuales se crean réplicas de las tablas en una BDD son:

- ▷ mejorar el rendimiento: distribuir copias de los datos entre los sitios disminuye la transferencia de datos y, en consecuencia, el tráfico de la red; evita los cuellos de botella, pues muchas operaciones se hacen de forma local, y además, mejora el tiempo de respuesta del sistema.

- ▷ aumentar la disponibilidad: los datos se pueden colocar físicamente en el lugar donde se accesan más frecuentemente.
- ▷ facilitar la recuperación: el sistema podría accesar a un servidor alternativo para recuperar información de la BD, en caso de que uno de los sitios fallara o estuviera fuera del alcance de la red.

Entre las desventajas tenemos las siguientes:

- ▷ incrementa el costo de la actualización;
- ▷ incrementa la complejidad del control de la concurrencia.

Para cada actualización de una determinada tabla T hay que propagar la modificación a todos los sitios que tienen una réplica de la tabla. Esto produce una sobrecarga incrementada en el sistema, pues se realizan de forma concurrente. El sistema debe asegurar que todas las réplicas de la tabla T presenten una vista consistente de los datos, en caso contrario pueden producirse resultados erróneos en las consultas.

La *transparencia de replicación* está relacionada con la de localización y se refiere a que, si existen réplicas de objetos de la base de datos, su existencia debe ser controlada por el sistema; la redundancia debe ser mínima para evitar errores de inconsistencia [OV99].

3.2.1 Tipos de replicación

Existen diferentes criterios para la configuración de un conjunto de réplicas, que limitan la posibilidad de modificar las réplicas y el modo en el que se propagan las actualizaciones.

- ▷ Maestro-Esclavo. La Base de Datos permite actualizaciones en el sitio principal (maestro) y las propaga automáticamente a las réplicas en los otros sitios (que son sólo de lectura). También se le conoce como Copia primaria (*Primary copy*).

  En esta configuración se disminuye la complejidad del control de concurrencia mencionado anteriormente.

- ▷ Multimaestro. Las actualizaciones se permiten en cualquier réplica (es también conocida como *Update everywhere*) y se propagan a los demás sitios.

La replicación temprana (*eager replication*) mantiene todas las réplicas exactamente sincronizadas, actualizándolas como parte de una transacción atómica, pero los tiempos de respuesta de la transacción son largos. En la replicación perezosa (*lazy replication*)

las actualizaciones se propagan a los otros sitios de forma asíncrona, después de que la transacción de actualización se ha comprometido, los tiempos de respuesta son más cortos [GHOS96].

En una configuración Multimaestro todos los sitios pueden realizar actualizaciones. La escalabilidad es limitada debido al incremento de conflictos en la administración y en la integridad de las réplicas [SS05]. En particular no es posible asegurar la consistencia de los datos en todo momento.

3.2.2 Consistencia en la replicación

Una de las tareas del SMBDD es mantener la consistencia de los datos. Muchos manejadores de BD comerciales soportan la replicación de los datos con un bajo grado de consistencia (por ejemplo, sin garantizar serialización) [SKS98].

Al crear varias réplicas de una tabla, esperamos que todos los usuarios tengan una vista consistente de los datos, esto es, que los datos de todas las réplicas sean iguales.

Propagación de las actualizaciones

El principal problema de la replicación de datos es que la actualización de un objeto debe ser propagada a todas sus copias.

En una BD multiusuario (como las BDD) los datos pueden modificarse simultáneamente y la posibilidad de inconsistencia aumenta durante la propagación de las actualizaciones.

Ejemplo 3.1

Tenemos una BDD con x sitios. Una de sus tablas T está replicada de la siguiente forma: T_1 en el sitio S_1 y T_2 en el sitio S_2. En un momento determinado se actualizaron las dos réplicas y, por alguna razón, se presentaron problemas durante la propagación de las actualizaciones: las réplicas de T quedaron en un estado inconsistente (en un sistema asíncrono hay inconsistencia mientras no se realicen las modificaciones en todos los sitios involucrados).

Si consideramos el Ejemplo 2.1 (véase Capítulo 2), donde se actualizaron las réplicas de la tabla $DATA$, tenemos que las tuplas con d_pk (llave primaria) afectadas fueron las siguientes: $d\_pk = 100, 101, 102, 103$ en $DATA_1$ y $d\_pk = 201, 202$ en $DATA_2$.

En la siguiente Figura 3.5 podemos ver cual fue el estado final, después de haber insertado y eliminado tuplas de las réplicas.

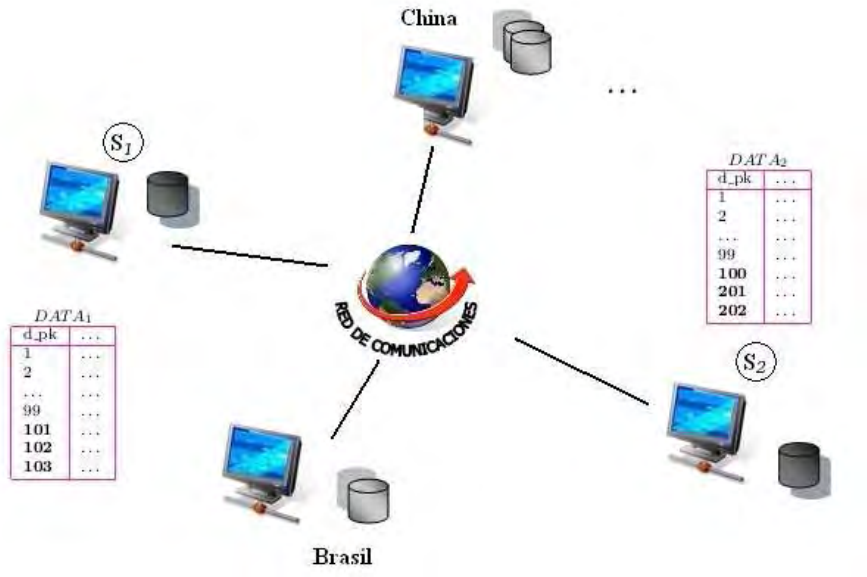

Figura 3.5: Ejemplo de inconsistencia en réplicas

Las tuplas incorrectas de una réplica son aquellas que tienen valores diferentes al conjunto de valores de la otra réplica. En $DATA_1$ las tuplas incorrectas tienen como llave $d\_pk = 101, 102, 103$ y en $DATA_2$ las tuplas $d\_pk = 100, 201, 202$.

Ya que hay tuplas incorrectas en las réplica, decimos que la tabla $DATA$ es inconsistente.

Cuando una BD está en un estado inconsistente proporciona información incorrecta o contradictoria a los usuarios. Estos errores pueden evitarse si se elimina la redundancia o si ésta es controlada por el SMBDD.

3.2.3 Sistemas de replicación en BD

En la actualidad muchos de los manejadores de Bases de Datos Distribuidas más conocidos, como Oracle, SQL Server, DB2, Ingres Star, ya implementan modelos de replicación. También existen herramientas de replicación para las Bases de Datos en general, como: DBReplicator, DBMoto, DBBalance, Daffodil Replicator, etc.[1] . Muchos de ellos soportan los manejadores de Bases de Datos más utilizados.

Una de las tareas más importantes de estos sistemas es la sincronización de datos, que nos ayuda a mantener la consistencia de las réplicas.

[1] Algunos de ellos son de licencia libre y otros comerciales

Para otros manejadores no distribuidos, como PostgreSQL, existen diferentes sistemas para instrumentar la replicación.

En la Tabla 3.2 se mencionan algunos de ellos.

	Síncrono	Asíncrono	Maestro-Esclavo	Multimaestro
Slony-I	No	Sí	Sí	No
PostgreSQL Replicator	No	Sí	Sí	Sí
Postgres–R	Sí	No	Sí	Sí
PG-Cluster	Sí	No	No	Sí

Tabla 3.2: Sistemas de replicación para PostgreSQL

De los programas mencionados anteriormente (todos de licencia libre) *Slony-I* [SI] es una de las herramientas más completas, pues algunos (como PostgreSQL Replicator), tienen versiones no actualizadas y otros están todavía en desarrollo (como PostgreSQL-R).

Capítulo 4

Detección de inconsistencia en réplicas

Hasta ahora hemos visto que la replicación de tablas tiene ventajas y desventajas. Tener copias de los datos aumenta, por un lado, la disponibilidad de la información, pero por el otro la BDD es más vulnerable a los problemas que se presentan al propagar las actualizaciones. Mantener actualizadas las réplicas de una BDD no es una tarea sencilla, sobre todo cuando los datos pueden ser modificados por todos los sitios.

Para detectar los errores de inconsistencia en réplicas nos basamos en la solución a la inconsistencia de datos propuesta en el Capítulo 2: la *Reconciliación de conjuntos*.

En este capítulo se proponen algoritmos para medir la inconsistencia en réplicas y en las siguientes secciones se explica cada uno de ellos, su operación y su implementación, así como el análisis de sus complejidades.

4.1 El problema de la Inconsistencia en réplicas

Replicar los datos en muchos sitios y permitir su modificación donde sea y cuando sea provoca un comportamiento inestable en el sistema de replicación; conforme se incrementa el número de nodos y el tráfico en la red, aumentan también los problemas para actualizar los datos [GHOS96]. La replicación maestro-esclavo reduce este problema, sin embargo, en un sistema multimaestro donde varios sitios necesitan modificar la misma información, es importante que en todos se busque conservar la consistencia.

La mayor o menor consistencia depende también de la configuración del sistema de replicación: si es maestro-esclavo o multimaestro, o si la propagación de las modificaciones es de forma síncrona o asíncrona.

En el Capítulo 2 se presenta un ejemplo (véase Ejemplo 2.1) de inconsistencia en las réplicas de una tabla, donde las actualizaciones hechas en un sitio no se realizaron

en todos los demás.

En la práctica es difícil mantener exactamente los datos con las últimas actualizaciones (freshness). También es difícil medir este grado de "actualidad", ya que necesitamos comparar instantáneamente los datos originales de las copias [CGM00].

Para nuestra BDD decimos que las tuplas de una tabla son correctas si están en todas las réplicas, con todos sus atributos iguales.

Una tupla es incorrecta si se encuentra en una réplica, pero no en todas (identificándola con su llave primaria). Este caso supone la inserción de una tupla que no fue propagada a todas las réplicas, o la eliminación de una tupla que no fue realizada en todas las réplicas. Son las tuplas diferentes entre las réplicas.

También es incorrecta si se encuentra en todas las réplicas (identificándola con su llave primaria), pero no tiene todos los valores de sus atributos idénticos. Este caso se refiere a la actualización de valores en atributos no primos (atributos que no pertenecen a la llave primaria).

Decimos que las réplicas de una tabla son inconsistentes si tienen tuplas incorrectas. Para detectar las tuplas incorrectas podemos considerar dos enfoques:

1. buscar las diferencias de dos réplicas considerando únicamente el atributo llave de las tuplas;

2. buscar las diferencias de dos réplicas considerando las tuplas completas.

En una tabla donde se realizan principalmente inserciones y eliminaciones, debemos esperar que los valores de la llave primaria de sus réplicas sean iguales. Si encontramos diferencias entre estos valores decimos que las réplicas son inconsistentes. En este escenario, la detección de la inconsistencia a través de la llave primaria (primer enfoque) es un buen indicador del estado de las réplicas y de la salud de la BDD.

En el segundo enfoque podemos ver la tupla completa como un valor único, como si estuvieran concatenados todos los atributos, y así encontrar los elementos diferentes entre las réplicas.

En este trabajo nos basamos en el primer enfoque para encontrar las tuplas incorrectas, después de eso aplicamos una métrica para determinar si la réplica es inconsistente.

En [GG08] se presentan métricas de integridad referencial para Bases de Datos Relacionales, fundamentadas en las llaves primarias y foráneas de las tablas, así como en las dependencias funcionales.

4.2 Algoritmos de detección de inconsistencia

En esta sección expondremos los algoritmos que utilizamos para detectar las tuplas incorrectas de las réplicas de una BDD. Éstos se basan en la Reconciliación de conjuntos (para la sincronización de datos) explicada en el Capítulo 2 y buscan detectar, y no solucionar, la inconsistencia de las réplicas de una tabla después de realizar actualizaciones, debida a fallas en los sitios o en la red de comunicaciones.

Implementamos los siguientes algoritmos para la detección de inconsistencias:

- DPC (Detección con Polinomio Característico) basado en CPISync, que trata los datos como objetos matemáticos representándolos con un polinomio característico, realiza la detección de las diferencias entre las tablas con un proceso matemático. Su implementación se hizo con el lenguaje C [CPI], Java y funciones definidas en PostgreSQL.

  Inspirados en la operación ANTIJOIN del álgebra relacional definimos los algoritmos DAJ (Detección con AntiJoin), que tienen que ver con la Transferencia completa de los datos:

- DAJ_{FOJ} que implementa el ANTIJOIN con el operador FULL OUTER JOIN;
- DAJ_{EXC} que implementa el ANTIJOIN con el operador EXCEPT;
- DAJ_{NE} que implementa el ANTIJOIN con el operador NOT EXISTS;
- DAJ_{Java} que utiliza el lenguaje Java para realizar esta operación.

Buscamos que el funcionamiento de los algoritmos fuera lo más parecido posible en cuanto a la forma en que recibían sus entradas y regresaban los resultados. La ejecución de los programas que implementaron estos algoritmos se realizó en las mismas condiciones, para poder hacer una comparación equitativa de los mismos.

Notación

Con el fin de explicar el funcionamiento de los algoritmos definimos una notación para las tablas, sus réplicas y los sitios involucrados en este proceso.

Una tabla T, tiene r réplicas $\{T_1, T_2, \ldots, T_r\}$, almacenadas en sitios distintos.

El sitio que calcula la inconsistencia de una réplica, sin pérdida de generalidad, será S_1 y T_1 la réplica con respecto a la cual se está midiendo la inconsistencia. T_2 representará a todas las demás réplicas de la tabla T (puede substituirse con $T_3, T_4 \ldots, T_r$) y S_2 será el sitio donde se encuentra esa otra réplica ($S_3, S_4 \ldots, S_r$).

Esto es, en S_1 se obtendrá el tamaño de la diferencia simétrica: las tuplas que están en T_1 y no en T_2 más el número de tuplas que están en T_2 y no en T_1.

Las répicas T_1 y T_2 (en sitios distintos) tienen el mismo esquema. Supondremos que el número de tuplas de una réplica será n y llamaremos m al número de tuplas incorrectas, o sea el tamaño de la diferencia simétrica.

Dado que tratamos con réplicas de la misma tabla podemos suponer que la diferencia entre ellas es mínima. Los algoritmos que presentamos pueden aplicarse a BDD con cardinalidades grandes o, por ejemplo, en Data Warehouses, donde las inserciones son muchas y no hay modificaciones. En el presente trabajo suponemos que el número de tuplas incorrectas en una réplica es menor con respecto a su tamaño ($m << n$).

Ya que nos enfocaremos a la detección de inconsistencia basada en el atributo llave (k), para simplificar la notación llamaremos $T[k]$ al conjunto de valores de k, esto es, $\pi_k(T)$.

Las tablas fueron manipuladas con los operadores estándar del álgebra relacional $\sigma, \Pi, \bowtie$, y otros operadores definidos en SQL y, además, por medio de funciones definidas en Java. *DPC* realiza operaciones con los datos en álgebra modular y utiliza Java y C.

A continuación se explica como se realiza la detección de tuplas incorrectas en las réplicas y más adelante se explicará como medimos el error de inconsistencia.

Planteamiento del problema

Dado un par de réplicas T_1 y T_2 de la tabla T, con tuplas no ordenadas, cuya cardinalidad original es n (pues suponemos que al inicio todas las réplicas tienen las mismas tuplas), se debe determinar si las réplicas son inconsistentes si tienen tuplas incorrectas.

Cuando las réplicas son inconsistentes sus cardinalidades cambian, las llamamos n_1 para T_1 y n_2 para T_2.

n_1 y n_2 dependen de las actualizaciones que se realizaron, de las eliminaciones o de las inserciones, y se obtienen como:

$$n_1 = n + m_1 \text{ y } n_2 = n + m_2,$$

respectivamente.

Llamamos m_1 y m_2 al número de tuplas incorrectas en T_1 y T_2, respectivamente; m será el total de tuplas incorrectas y la calculamos como:

$$m = m_1 + m_2$$

Por otro lado, el número de tuplas consistentes de las réplicas, aquellas que están presentes en todas las réplicas, se puede obtener como $n = n_i - m_i$.

Cuando no hay errores de inconsistencia $m_1 = 0$ y $m_2 = 0$.

Para detectar la inconsistencia de una réplica tomamos el conjunto de valores de las llaves primarias $T_1[k]$ y $T_2[k]$ y encontramos las diferencias entre esos valores. Damos por hecho que los valores de k no están repetidos (pues son del atributo llave) y que pueden ser números o caracteres que para el caso del polinomio se mapearán a números enteros en un campo determinado.

4.2.1 Medición de la inconsistencia de las réplicas

La entrada de los algoritmos será $T_1[k]$ y $T_2[k]$ y la salida serán las tuplas incorrectas en T_1, $I_{1/2}$, y en T_2, $I_{2/1}$.

Los algoritmos que implementamos suponen un esquema Multimaestro, aunque pueden aplicarse a Maestro-Esclavo (pues es un subcaso del otro). Dado que todos ellos pueden modificar las réplicas de una tabla, podemos tener tuplas incorrectas en todos los sitios ($m_1 \neq 0$ y $m_2 \neq 0$).

Distribución del control:

el procesamiento se distribuye entre los sitios que tienen las réplicas, S_1 y S_2 (como veremos más adelante en la descripción de los algoritmos). Es importante identificar cuál es cada uno de ellos, pues la mayor parte del procesamiento de los algoritmos la realiza S_2 (detecta las tuplas incorrectas), S_1 calcula el número de errores de inconsistencia. Es por eso que la notación de los sitios, y en consecuencia de los cálculos que realizan, varía de forma dinámica.

Asignar a uno de los sitios el control de la medición de los errores, se debe a que el sistema busca las diferencias de una réplica con respecto a todas las demás, de forma concurrente, y necesita que uno de ellos sea un punto de referencia para los demás sitios (véase la Figura 4.1).

Cualquier sitio puede saber si las réplicas de una tabla son inconsistentes. Esto no depende del esquema de replicación, Maestro-Esclavo o Multimaestro, del sitio que realizó actualizaciones de los datos o de cuándo las hizo.

El número de errores de inconsistencia de una réplica se calcula de la siguiente manera:

$$err_{inconst} = |I| \tag{4.1}$$

donde

$$I = I_{1/2} \cup I_{1/3} \cup \ldots \cup I_{1/r} \cup I_{2/1} \cup I_{3/1} \cup \ldots \cup I_{r/1} \tag{4.2}$$

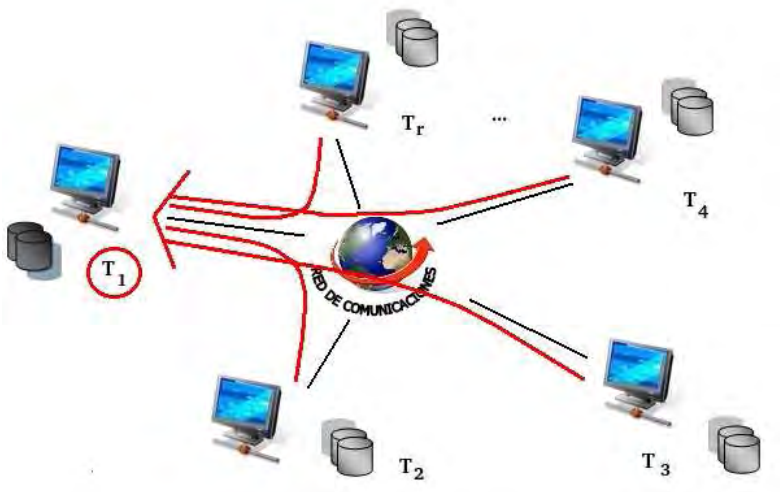

Figura 4.1: Detección de inconsistencia en las réplicas de una tabla

$I_{1/i}$ representa las tuplas de T_1 que no están en T_i, e
$I_{i/1}$ representa las tuplas de T_i que no están en T_1.

En este trabajo representaremos con T_2 a cualquier réplica de T diferente a T_1 (ésta última supondremos que realizará la medición del error de inconsistencia, sin pérdida de generalidad), entonces podemos representar la Ecuación 4.2 como sigue:

$$I = I_{1/2} \cup I_{2/1} \tag{4.3}$$

Esto significa que de acuerdo al error la réplica será:

$$err_{inconst} \begin{cases} > 0 & inconsistente \\ = 0 & consistente \end{cases}$$

Obsérvese que para tuplas idénticas que están repetidas en varias réplicas aparecerán una sola vez en la unión global.

El cálculo de la métrica se realiza fuera de línea, es decir, suponemos que en ese momento todos los servidores detienen su operación para ejecutar los programas que implementan los diferentes algoritmos, de esta forma evitamos que las réplicas sean modificadas mientras se realizan los cálculos.

4.3 Algoritmos de medición

En esta sección se presentan los algoritmos que miden la inconsistencia de las réplicas de una tabla: DPC, DAJ_{FOJ}, DAJ_{EXC}, DAJ_{NE}, DAJ_{Java}. Se explica como fueron implementados, sus diagramas de actividades, diagramas de paquetes y de clases; después se presenta una comparación de sus complejidades.

4.3.1 Algoritmo basado en DPC

Para determinar si las réplicas de una tabla son inconsistentes el algoritmo DPC representa los datos con su polinomio característico y realiza una interpolación para obtener las tuplas diferentes entre un par de réplicas (véase Capítulo 2).

Las réplicas involucradas son T_1 y T_2, localizadas en S_1 y S_2, respectivamente. Todas las operaciones que se realizan utilizan álgebra modular y los datos son representados de acuerdo a un módulo (un número primo, mayor al número más grande del conjunto de datos).

En los experimentos que realizamos los atributos utilizados para la detección y la medición de la inconsistencia fueron numéricos. Nuestros algoritmos no pierden generalidad dado que para DPC los valores serán mapeados a un campo numérico (en la computadora cualquier cadena puede ser representada con 0 y 1). Para DAJ se hace una comparación de cadenas, numéricas o no numéricas.

El procesamiento que realiza este algoritmo se distribuye entre los dos sitios involucrados y, aunque las operaciones de uno son mínimas, se requiere de la comunicación y de la participación de ambos sitios.

El algoritmo y el diagrama de actividades de DPC se muestran a continuación.

1. Procesamiento en el sitio S_1

S_1 mide la inconsistencia y las operaciones que realiza son las siguientes:

- ▷ el algoritmo genera $\bar{m}$ puntos de muestreo ($evalPoints$): para hacerlo existe una función que asegura que los valores obtenidos queden fuera del rango de los valores en $T_1[k]$. Los resultados se almacenan en una tabla.

$$|evalPoints| = \bar{m}$$

  $\bar{m}$ es el número de tuplas diferentes entre las réplicas.
  $|evalPoints|$ es el número de puntos de muestreo. Por cuestiones del cálculo matemático el sistema generará algunos puntos adicionales.

- ▷ S_1 se conecta de forma remota con S_2 y le envía los $evalPoints$.
- ▷ Calcula el polinomio característico (PC_1) de los valores en $T_1[k]$ y hace la evaluación para $\bar{m}$ puntos de muestreo; mientras recorre las tuplas de la tabla obtiene su cardinalidad, n_1, y almacena estos resultados en una tabla.
- ▷ Convierte la tabla en archivo, crea una conexión remota y envía los resultados a todos los S_2 (en forma de archivo).
- ▷ Espera:
  S_1 espera a que el sitio S_2 termine la detección de tuplas incorrectas y a que le regrese los resultados.
- ▷ Si S_1 recibió respuesta de S_2, entonces puede calcular la inconsistencia de las réplicas.

Ya que S_2 representa más de un sitio ($S_3, S_4 \ldots, S_r$), se espera la respuesta de todos estos sitios, para realizar el cálculo del error de inconsistencia, de acuerdo a la Ecuación 4.1.

2. Procesamiento en el sitio S_2

La ejecución del algoritmo en S_2 *es la siguiente:*

- ▷ si el sitio S_2 recibió los $evalPoints$ empieza el cálculo del polinomio característico (PC_2 de $T_2[k]$) de forma concurrente, mientras S_1 calcula su PC_1, S_2 calcula el suyo[1] ; mientras recorre las tuplas de la tabla obtiene su cardinalidad, n_2, y Almacena los resultados en una tabla.
- ▷ Convierte la tabla con el PC_2 en un archivo, pues el programa que realiza la detección lee los polinomios característicos de archivos.
- ▷ Al momento en el que S_2 recibe el PC_1 empieza la detección de tuplas incorrectas de las réplicas. El programa realiza la interpolación y encuentra los valores distintos de las réplicas, almacena los resultados en dos tablas: $I_{1/2}$, las tuplas que están en la réplica T_1 y no están en T_2, e $I_{2/1}$ las tuplas que están en la réplica T_2 y no están en T_1.
- ▷ S_2 envía $I_{1/2}$ e $I_{2/1}$ a S_1 para que éste calcule el número de tuplas incorrectas de una réplica con respecto a las otras.

[1] Obsérvese que la evaluación del polinomio característico en los puntos de muestreo puede hacerse en otro momento, por ejemplo cada vez que se hace una inserción o eliminación

Aquí termina el procesamiento de S_2. Después de identificar las tuplas diferentes S_1 se encargará de realizar el cálculo total de los errores de inconsistencia.

Diagrama de actividades

El diagrama de la Figura 4.2, resume los pasos de este algoritmo.

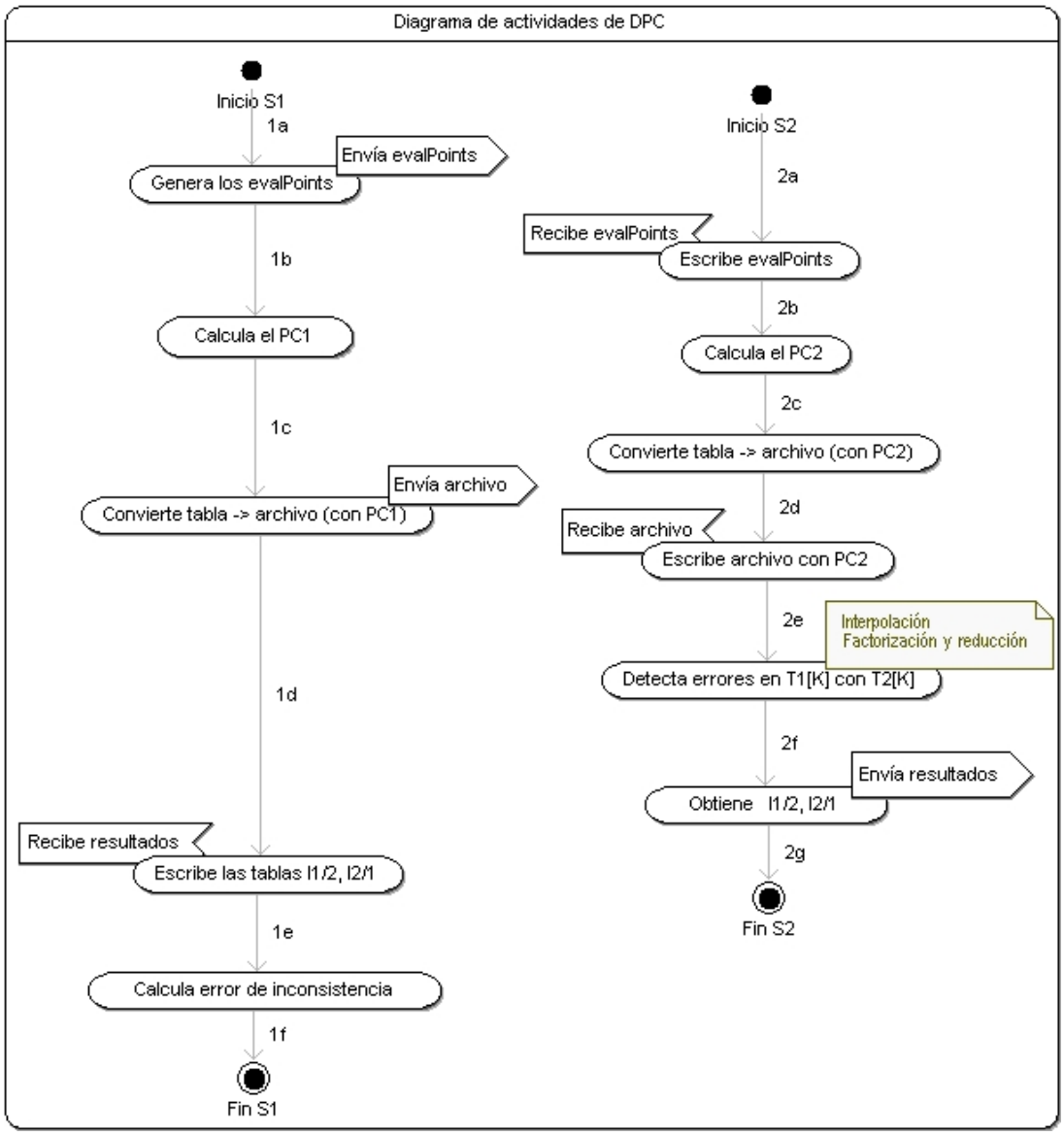

Figura 4.2: Diagrama de actividades de *DPC*

Un caso particular de la reconciliación

Cuando buscamos detectar las diferencias entre las réplicas, considerando que las operaciones que más se realizan en la BDD son la inserción y la eliminación más que la modificación de las tuplas, entonces podemos encontrar una mejora al algoritmo DPC (véase [MTZ01]), al algoritmo optimizado lo llamaremos $DPCop$.

Supongamos que los cálculos de los polinomios PC_1 y PC_2 estén almacenados en BD, de cada sitio. Estos polinomios pueden haber sido calculados con la tabla inicial, cuando aún no había errores de inconsistencia, o bien después de la última actualización.

Tomando como base estos valores, cada vez que haya actualizaciones en las tablas T_1 y T_2, en lugar de recalcular el polinomio característico de todas las tuplas, aunque no hayan sido modificadas, obtenemos sólo el PC de las tuplas afectadas.

- ▷ Si la operación en la tabla fue la eliminación de una tupla d_z, entonces dividimos los valores de PC_{base_1}, PC_{base_2} entre el polinomio de esa tupla, $PC(d_z)$:

$$PC_1 = \frac{PC_{base_1}}{PC(d_z)}$$
y
$$PC_2 = \frac{PC_{base_2}}{PC(d_z)}$$

- ▷ Si, por el contrario, fue la inserción de una tupla d_z, los valores almacenados del polinomio se multiplican por el polinomio de esa tupla, $PC(d_z)$:

$$PC_1 = PC_{base_1} * PC(d_z)$$
y
$$PC_2 = PC_{base_2} * PC(d_z)$$

Los valores de PC_1 y de PC_2 se modifican directamente de la BD.

Después de obtener los nuevos polinomios se continúa con la ejecución normal del algoritmo.

El cálculo de los polinomios de las tuplas actualizadas disminuye el tiempo de procesamiento, sin embargo, el tiempo del cálculo de las raíces, que es grande, se mantiene.

En la parte experimental se presentan los resultados de este algoritmo optimizado.

4.4 Algoritmo basado en DAJ

Los siguientes algoritmos se basan en DAJ y para detectar las tuplas diferentes entre ambos conjuntos utilizamos la operación ANTIJOIN. Los tres siguientes algoritmos

utilizan operadores de SQL para realizar esta operación (DAJ_{FOJ}, DAJ_{EXC}, DAJ_{NE}) y el último realiza esta operación a través de un algoritmo desarrollado en Java (DAJ_{Java}).

El algoritmo realiza una transferencia completa de las tuplas de una de las réplicas para compararlas localmente con las tuplas de otra réplica.

Operaciones en SQL

El proceso de la recuperación de los datos de diferentes sitios en una red es conocido como procesamiento de consultas distribuidas. Para esto se usan diferentes operaciones basadas en el SEMIJOIN.

En las BDD el procesamiento del SEMIJOIN reduce el tamaño-costo del envío de las tablas. Todas las tablas involucradas son transmitidas a un sitio final, donde se ejecutarán todos los JOIN necesarios para resolver la consulta [PLH89] [CL90].

Considerando las dos réplicas, T_1 y T_2, definimos las siguientes operaciones.

- ▷ SEMIJOIN

  La operación SEMIJOIN aplicada a $T_1[k]$ y $T_2[k]$, elimina las tuplas que no están presentes en ambas réplicas y devuelve los valores, sin duplicados, de la réplica izquierda o derecha de un sitio (según la condición que le demos: LEFT SEMIJOIN o RIGHT SEMIJOIN) que existen también en la misma réplica de otro sitio (valores de la intersección) [GK98].

- ▷ ANTIJOIN

  El ANTIJOIN es la operación opuesta al SEMIJOIN. Regresa los valores de $T_1[k]$ que no están en $T_2[k]$ y las de $T_2[k]$ que no están en $T_1[k]$, esto es, los que están fuera de la intersección.
  Existen varias formas de implementar esta operación, en los siguientes algoritmos se explicará cómo se realizó.

4.4.1 Algoritmo basado en DAJ con FOJ

Este algoritmo realiza la operación ANTIJOIN basándose en el operador FULL OUTER JOIN.

- ▷ ANTIJOIN

  El ANTIJOIN es una operación que puede ser implementada con el FULL OUTER JOIN de las dos réplicas, más una condición. En SQL:

SELECT $< T_1[k] >$, $< T_2[k] >$
FROM $< T_1 >$ FULL OUTER JOIN $< T_2 >$
ON $< T_1[k] > = < T_2[k] >$
WHERE $< T_1[k] >$ IS NULL OR $< T_2[k] >$ IS NULL

Si la consulta regresa cero, quiere decir que las dos réplicas son iguales. En caso contrario hay diferencias entre $T_1[k]$ y $T_2[k]$.

El algoritmo y el diagrama de actividades de DAJ_{FOJ} se muestran a continuación.

1. Procesamiento en el sitio S_1

La ejecución del algoritmo en el sitio S_1 *es el siguiente:*

- ▷ obtiene la proyección del atributo llave k y la escribe en una tabla;
  crea esta tabla con los valores ordenados y esto lo hace por cuestiones de cálculo, pues agiliza después la ejecución del algoritmo del ANTIJOIN;
- ▷ crea un archivo con la proyección y lo envía a S_2;
- ▷ si el sitio S_2 terminó los cálculos, le envía los resultados a S_1, donde calcula el número de errores de inconsistencia de la réplica.

2. Procesamiento en el sitio S_2

La ejecución del algoritmo en el sitio S_2 *es la siguiente:*

- ▷ el sitio S_2 calcula la proyección ordenada $T_2[k]$ y la escribe en una tabla;
- ▷ convierte en archivo la proyección $T_2[k]$ y lo envía a S_2;
- ▷ realiza el ANTIJOIN de las dos proyecciones ejecutando el algoritmo de Java;
- ▷ convierte en tabla la proyección $T_1[k]$;
- ▷ realiza el ANTIJOIN de las dos proyecciones utilizando el operador FULL OUTER JOIN de SQL;
- ▷ obtiene las tuplas incorrectas ($I_{1/2}$ y $I_{2/1}$), en ambos sitios y escribe los resultados en S_1.

Diagrama de actividades

El siguiente diagrama de actividades se muestra en la Figura 4.3 y presenta gráficamente los pasos que ejecuta este algoritmo.

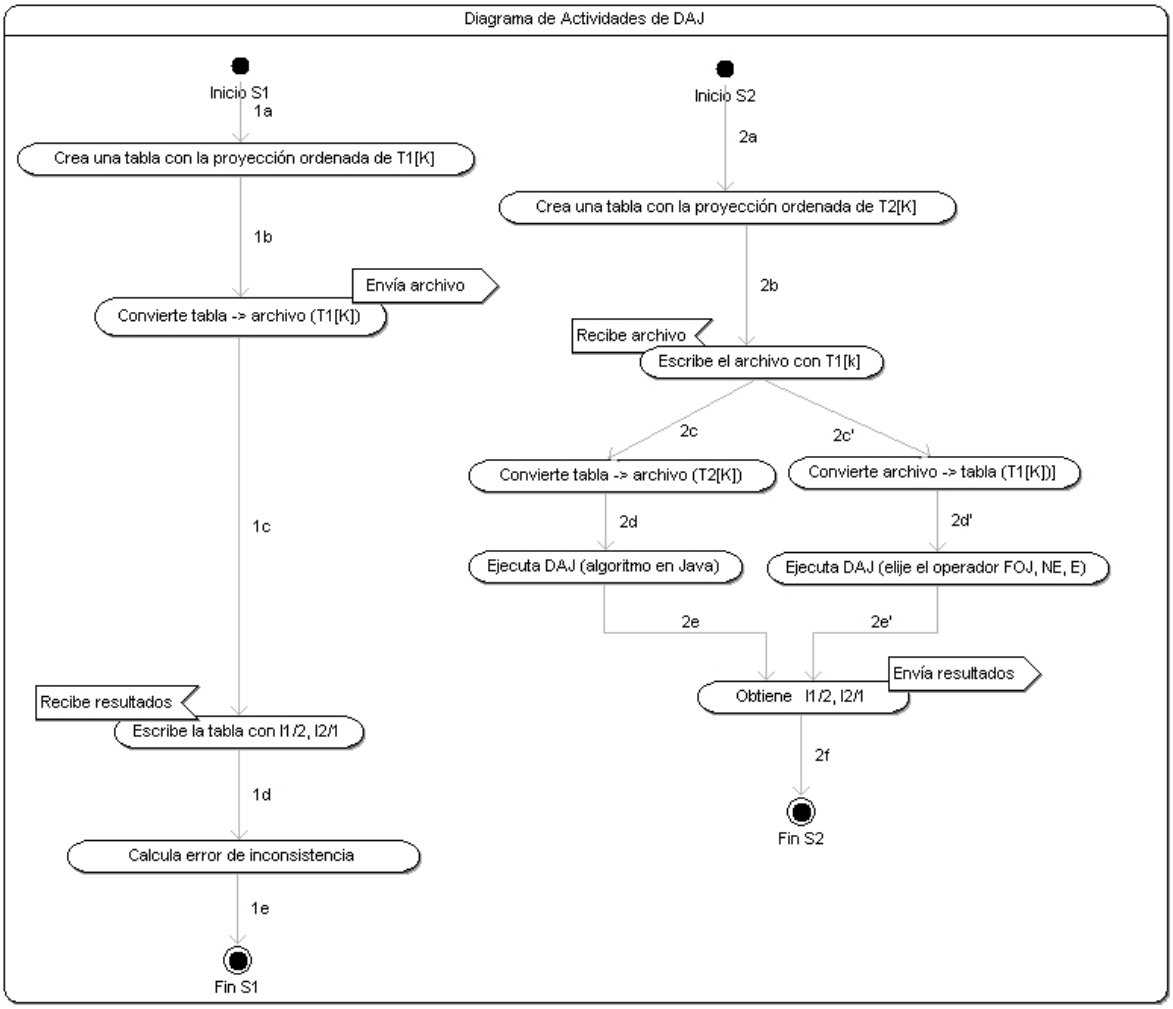

Figura 4.3: Diagrama de actividades del *DAJ*

4.4.2 Algoritmo basado en DAJ con operador NOT EXISTS

Este algoritmo realiza la operación ANTIJOIN basándose en el NOT EXISTS.

▷ ANTIJOIN

El código SQL de estas operaciones es el siguiente:

SELECT $< T_1[k] >$ FROM $< T_1 >$
WHERE NOT EXISTS
(SELECT $< T_2[k] >$ FROM $< T_2 >$
WHERE $< T_1[k] > = < T_2[k] >$);

En este código tenemos una consulta externa y una subconsulta: la primera regresa todos los valores de T_1, la segunda los valores de la intersección de las dos réplicas, $T_1 \cap T_2$. El resultado final despliega los valores que están en $T_1[k]$ y que no están en la intersección, $I_{1/2} = T_1 - (T_1 \cap T_2)$, esto es, los valores de T_1 que no están en T_2.

Esta operación nos regresan sólo la mitad del ANTIJOIN (el LEFT ANTI- SEMI-JOIN). Para encontrar los otros valores (los de T_2 que no están en T_1) realizamos la misma operación intercambiando las réplicas: el resultado del ANTIJOIN será la unión de las dos consultas.

Procesamiento del algoritmo

El procesamiento del algoritmo es muy similar al que se describió para el FULL OUTER JOIN de DAJ_{FOJ}, sólo que el operador que se utilizó para implementar el ANTIJOIN fue el NOT EXISTS (véase el algoritmo y el diagrama de actividades en la Sección 4.4.1).

4.4.3 Algoritmo basado en DAJ con operador EXCEPT

El tercer operador que utilizamos para realizar la operación ANTIJOIN se basa en el EXCEPT.

▷ ANTIJOIN

El código SQL de estas operaciones es el siguiente:

SELECT $< T_1[k] >$ FROM $< T_1 >$
EXCEPT
(SELECT $< T_2[k] >$ FROM $< T_2 >$);

Esta operación nos regresa los valores de $T_1[k]$ que no están en $T_2[k]$, $I_{1/2} = T_1 - T_2$. EXCEPT es un operador estándar de ANSI SQL, que simplifica la tarea de encontrar las diferencias entre dos conjuntos.

Esta operación nos regresan sólo la mitad del ANTIJOIN (el *LEFT ANTI-SEMIJOIN*). Para encontrar los otros valores (los de T_2 que no están en T_1) realizamos la misma operación intercambiando las réplicas: el resultado del ANTIJOIN será la unión de las dos consultas.

Procesamiento del algoritmo

El procesamiento del algoritmo es muy similar al que se describió para el FULL OUTER JOIN de DAJ_{FOJ}, sólo que el operador que se utilizó para implementar el ANTIJOIN fue el EXCEPT (véase el algoritmo y el diagrama de actividades en la Sección 4.4.1).

4.4.4 Algoritmo basado en DAJ con Java

Una alternativa a los operadores descritos es la detección de las tuplas incorrectas a través de un algoritmo desarrollado en Java.

Este programa recorre los valores de las dos réplicas $T_1[k]$ y $T_2[k]$ para encontrar sus diferencias; en la sección de la implementación se describe el programa.

La Figura 4.4 nos muestra la salida de la operación ANTIJOIN, $I_{1/2}$ e $I_{2/1}$.

Procesamiento en el algoritmo

El algoritmo y el diagrama de actividades puede verse en la Sección 4.4.1. Aunque la forma en la que se implementó es distinta, los pasos que realizó para enviar o recibir información del sitio remoto fueron los mismos.

4.5 Análisis de la complejidad de *DPC* y *DAJ*

Después de presentar los algoritmos DPC y DAJ haremos un análisis de la complejidad en la comunicación y en el procesamiento, de cada uno de ellos.

Aunque hay muchos factores que intervienen en la comunicación, tamaño de los datos enviados, ancho de banda, tráfico en la red, etc., en el siguiente análisis consideramos sólo el número de bits transmitidos.

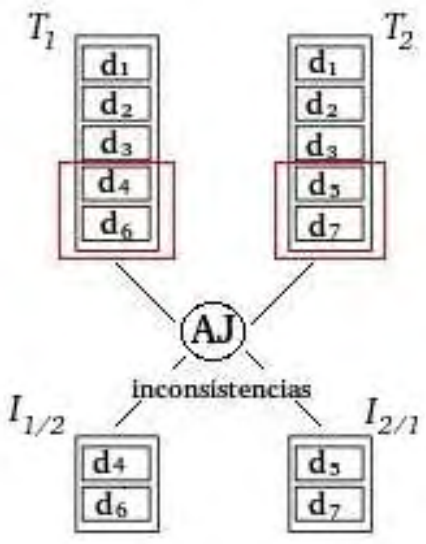

Figura 4.4: Ejemplo de la salida del ANTIJOIN

Algoritmo basado en DPC

La complejidad del algoritmo DPC se presenta a continuación.

- La complejidad en la comunicación se puede obtener a partir de las cadenas transmitidas:
  un conjunto de $\bar{m}$ puntos de muestreo ($evalPoints$), enviados de S_1 a S_2;
  un conjunto de $\bar{m}$ valores del PC_1 evaluado en los $evalPoints$, enviados del sitio S_1 a S_2;
  los resultados de la detección de errores de inconsistencia, $I_{1/2}$ e $I_{2/1}$ ($\bar{m}$ tuplas), enviados de S_2 a S_1.

  La complejidad en la comunicación es del orden:

  $$O(3\bar{m}).$$

- La complejidad de procesamiento depende de las operaciones que cada sitio realiza. Las operaciones de este algoritmo son las siguientes:
  el cálculo de los polinomios PC_1 y PC_2, del orden $O(n * \bar{m})$;
  el cálculo de las raíces, del orden de $O(\bar{m}^3)$ operaciones (véase Capítulo 2);
  el cálculo de la métrica es del orden de $O(\bar{m})$, pues obtiene la cardinalidad de la tabla con los resultados: $I_{1/2}$ e $I_{2/1}$, que son las tuplas incorrectas.

  La complejidad en el procesamiento es del orden:

  $$O(n * \bar{m} + \bar{m}^3 + \bar{m})$$

En la Tabla 4.1 presentamos la complejidad del algoritmo de acuerdo a su distribución.

Complejidad	*DPC*			
	Procesamiento		*Transferencia*	
	S_1	S_2	S_1	S_2
Cálculo de PC	$O(n * \bar{m})$	$O(n * \bar{m})$		
Cálculo de las raíces		$O(\bar{m}^3)$		
Transmisión de PC_1			$O(2\bar{m})$	
Transmisión de resultados				$O(\bar{m})$
Cálculo de la métrica	$O(\bar{m})$			

Tabla 4.1: Complejidad del algoritmo DPC

Dado que el cálculo de los polinomios, PC_1 y PC_2, se hace en forma concurrente consideramos el valor máximo de esos tiempo.

Algoritmos basados en DAJ

A continuación presentamos la complejidad del algoritmos DAJ_{Java}, que fue el que comparamos con el DPC. Los demás algoritmos (DAJ_{FOJ}, DAJ_{EXC} y DAJ_{NE}) cambian sólo en el cálculo del ANTIJOIN.

- La complejidad en la comunicación se puede obtener a partir de las cadenas transmitidas:
  la proyección de la réplica $T_1[k]$ al sitio S_2 (n valores de la llave primaria);
  los resultados de la detección de errores de inconsistencia, $I_{1/2}$ e $I_{2/1}$ ($\bar{m}$ tuplas), enviados de S_2 a S_1.

  La complejidad en la comunicación es del orden:

  $$O(n + \bar{m})$$

- La complejidad de procesamiento depende de las operaciones que cada sitio realiza. Las operaciones de este algoritmo son las siguientes:
  el cálculo de las proyecciones ordenadas de $T_1[k]$ y de $T_2[k]$, que es del orden $O(n) + O(nlgn)$, realizado en cada sitio;
  el cálculo del ANTIJOIN, del orden $O(n + \bar{m})$ comparaciones, realizado en S_2;
  el cálculo de la métrica es del orden de $O(\bar{m})$, pues obtiene la cardinalidad de la tabla con los resultados: $I_{1/2}$ e $I_{2/1}$, que son las tuplas incorrectas.

La complejidad en el procesamiento es:

$$O(n + \bar{2}m + nlgn)$$

En la Tabla 4.2 presentamos la complejidad del algoritmo de acuerdo a su distribución.

Complejidad	*DAJ*			
	Procesamiento		*Transferencia*	
	S_1	S_2	S_1	S_2
Cálculo de la proyección ordenada	$O(n + nlgn)$	$O(n + nlgn)$		
Cálculo del ANTIJOIN		$O(n + \bar{m})$		
Transmisión de la proyección			$O(n + \bar{m})$	
Transmisión de resultados				$O(\bar{m})$
Cálculo de la métrica	$O(\bar{m})$			

Tabla 4.2: Complejidad de los algoritmos DAJ

De forma similar al anterior, dado que las proyecciones se obtienen de forma concurrente, consideramos el valor máximo de esos tiempo.

Se observa que la complejidad en la comunicación de DPC es menor que en DAJ, por el contrario la complejidad en el procesamiento de DPC es mayor que en DAJ.

De estos valores los que podrían verse afectados por el número de sitios involucrados en el cálculo de la métrica son: los tiempos en la transferencia y el cálculo de la métrica.

4.6 Implementación de los algoritmos

En esta sección presentamos la implementación de los algoritmos DPC y DAJ, explicando la programación y la distribución del proceso computacional.

4.6.1 Implementación del algoritmo DPC

La implementación del algoritmo DPC, explicado en la sección anterior (veáse el diagrama de actividades de la Figura 4.2), se describe a través de los diagramas de paquetes y de clases.

Diagramas de paquetes

Los paquetes del algoritmo DPC, para el sitio S_1 y S_2, y sus relaciones se muestran en la Figura 4.5.

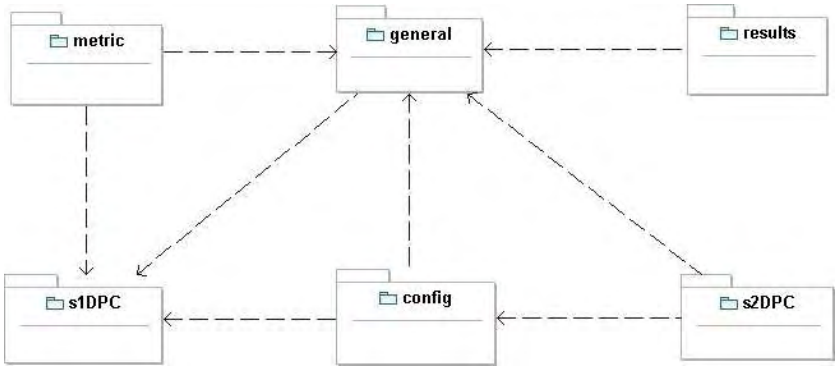

Figura 4.5: Diagrama de paquetes de DPC

Descripción de los paquetes

- ▷ *general*: contiene las clases relacionadas con la conexión y la manipulación de la BDD, la medición de los tiempos (de ejecución y comunicación) de las operaciones realizadas en el sistema y la conexión y el envío de información a un sitio remoto (ConexRemota, ConexLocal, GetTime, ScpFile, AlterTables). Tiene también un script con la configuración general de los experimentos (ScriptExperiments).
- ▷ $config$: contiene las clases con los archivos de configuración del algoritmo, inicializa variables y crea/borra tablas de la BD (ConfigDPC).
- ▷ $s1DPC$: contiene el código del algoritmo DPC en S_1.
- ▷ $s2DPC$: contiene el código del algoritmo DPC en S_2.
- ▷ $results$: contiene las clases que obtienen y procesan los resultados de los experimentos.
- ▷ *metric*: contiene las clases que calculan la métrica de inconsistencia en las réplicas.

Diagramas de clases

En el sitio S_1

El diagrama de clases de la implementación de DPC, en el sitio S_1, se muestra en la Figura 4.6.

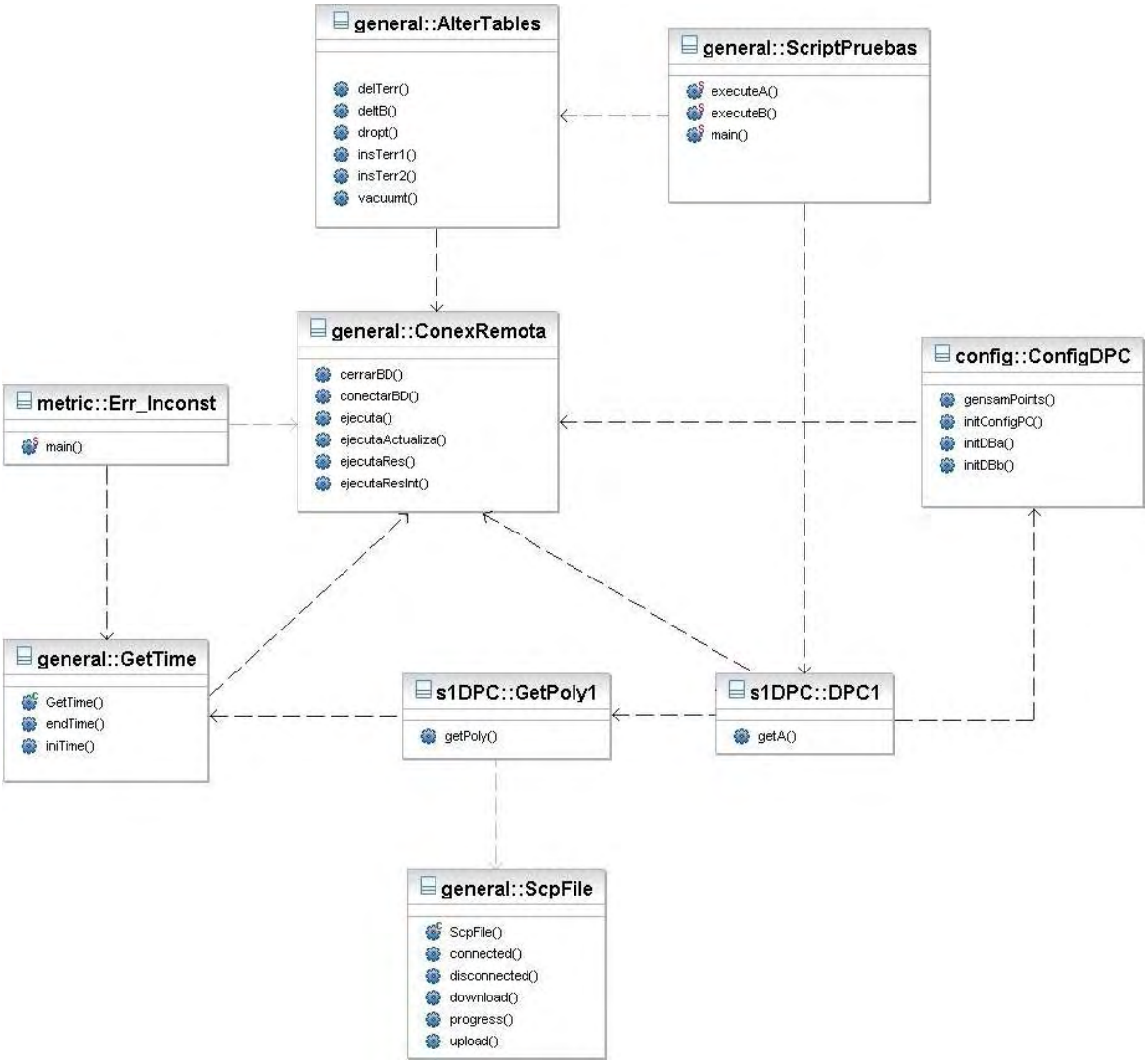

Figura 4.6: Diagrama de clases del algoritmo DPC en el sitio S_1

En el sitio S_2

El diagrama de clases de la implementación de DPC, en el sitio S_2, se muestra en la Figura 4.7.

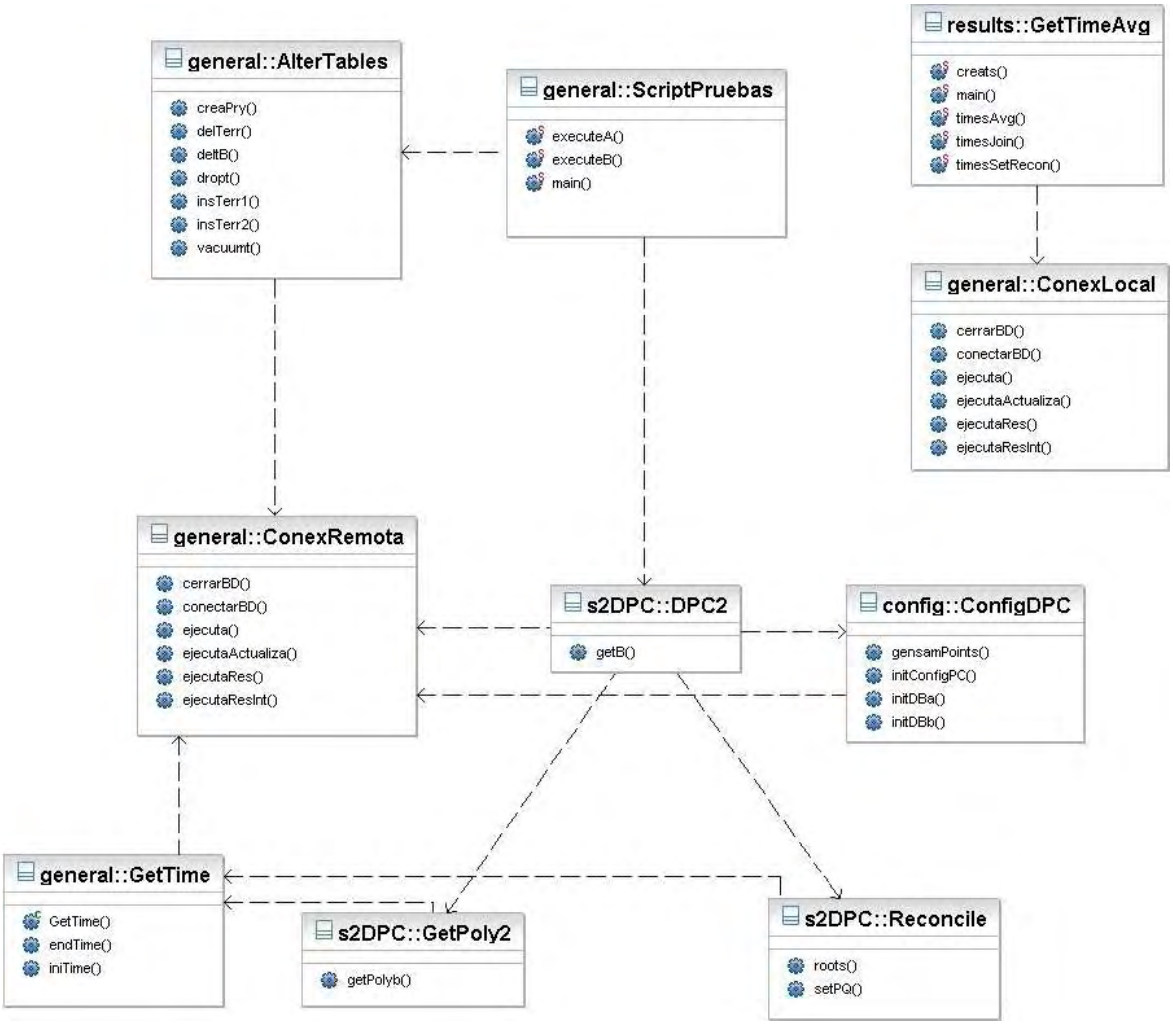

Figura 4.7: Diagrama de clases del algoritmo DPC en el sitio S_2

4.6.2 Implementación de los algoritmos DAJ

La implementación de los algoritmos DAJ_{FOJ}, DAJ_{EXC}, DAJ_{NE} y DAJ_{Java} se presenta en esta sección.

Dado que estos algoritmos siguen la misma metodología (véase Figura 4.3), sólo que realizan la operación ANTIJOIN aplicando diferentes operadores, las clases desarrolladas para cada uno de ellos están en los mismos paquetes.

Diagramas de paquetes

Los paquetes del algoritmo DAJ, para el sitio S_1 y S_2, y sus relaciones se muestran en la Figura 4.8.

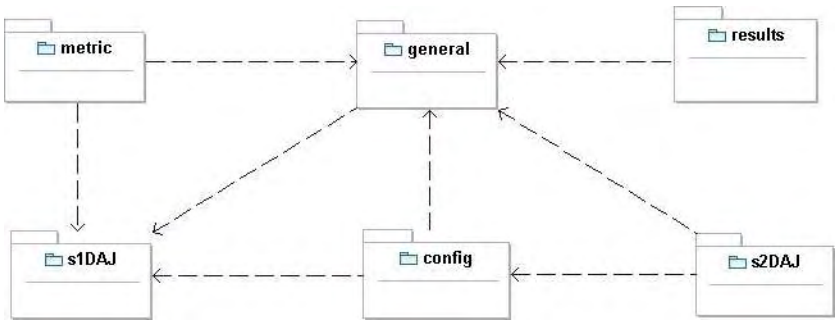

Figura 4.8: Diagrama de paquetes de DAJ

Descripción de los paquetes

- ▷ *general*: contiene las clases relacionadas con la conexión y la manipulación de la BDD, la medición de los tiempos (de ejecución y comunicación) de las operaciones realizadas en el sistema y la conexión y el envío de información a un sitio remoto (ConexRemota, ConexLocal, GetTime, ScpFile, AlterTables). Tiene también un script con la configuración general de los experimentos (ScriptExperiments).
- ▷ $config$: contiene las clases con los archivos de configuración del algoritmo, inicializa variables y crea/borra tablas de la BD (ConfigDAJ);
- ▷ $s1DAJ$: contiene el código del algoritmo DAJ en S_1.
- ▷ $s2DAJ$: contiene el código del algoritmo DAJ en S_2.

- ▷ *results*: contiene las clases que obtienen y procesan los resultados de los experimentos.

- ▷ *metric*: contiene las clases que calculan la métrica de inconsistencia en las réplicas.

La elección de enviar las proyecciones, de un sitio a otro, como archivos y no como tablas se basó en el tiempo de transferencia que tardaba cada uno. Para el primer caso, con una instrucción de PostgreSQL convertimos la tabla en archivo, nos conectamos de forma remota a S_2 y le enviamos, en bloque, todos los datos de la tabla; una vez recibido el archivo lo importamos a una tabla: esta conversión es muy rápida.

Para el segundo caso fue necesario que S_1 estableciera una conexión con la BD del sitio S_2, a través del JDBC, entonces creamos la tabla directamente en el sitio remoto. La desventaja de esta opción es que para llenar la tabla se debe insertar tupla por tupla y es como enviar varias peticiones que deben ser traducidas por el controlador, y por eso el tiempo de transferencia es mayor. Cada vez que enviamos una consulta a la BD pasamos por dos capas: por el API del JDBC (conjunto de clases) donde escribimos las consultas y por el controlador del fabricante que las traduce en peticiones y servicios internos del SMBD.

Diagramas de clases

En el sitio S_1

El diagrama de clases de la implementación de DAJ, en el sitio S_1, se muestra en la Figura 4.9.

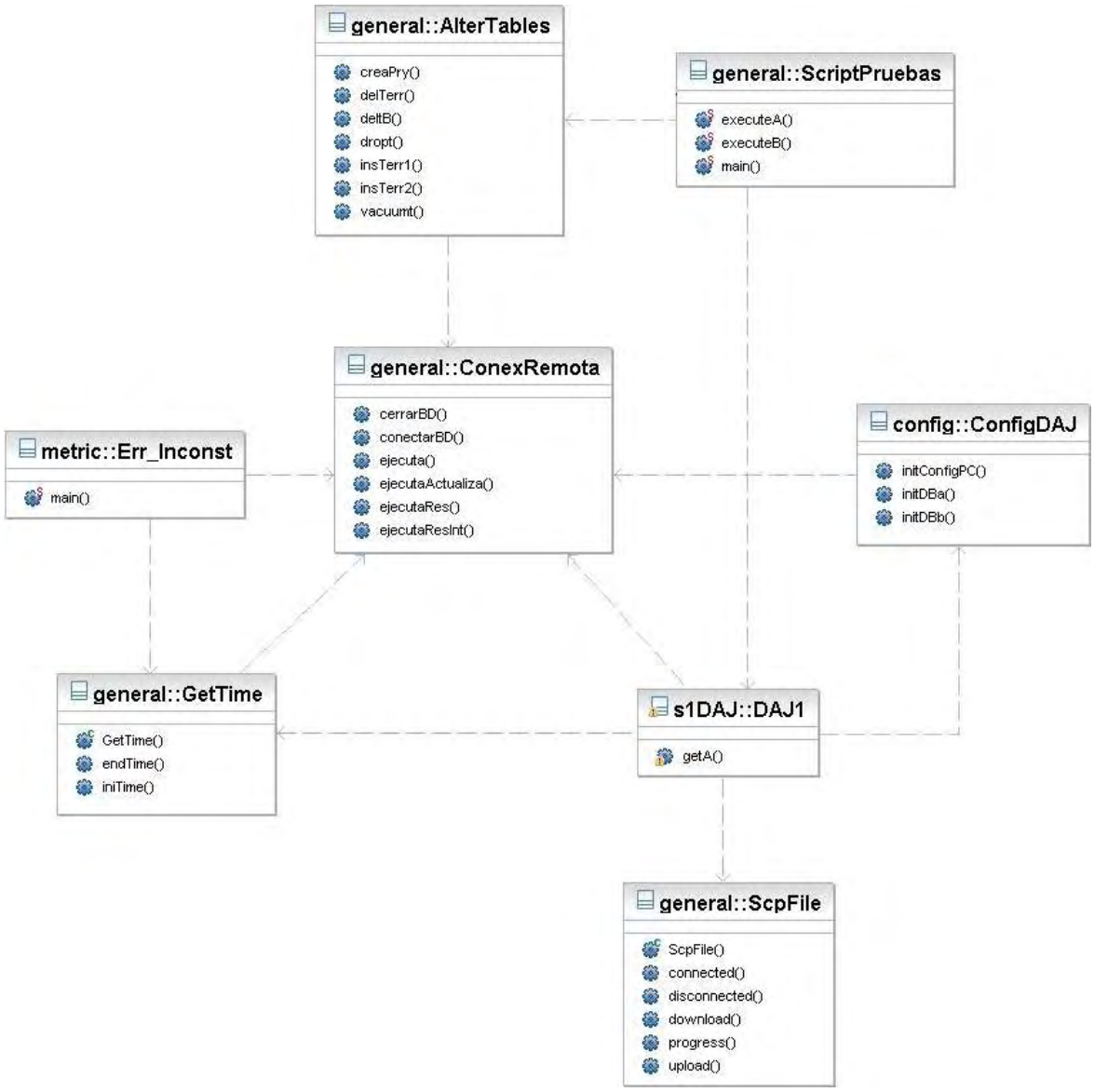

Figura 4.9: Diagrama de clases del algoritmo DAJ en el sitio S_1

En el sitio S_2

El diagrama de clases de la implementación de DAJ, en el sitio S_2, se muestra en la Figura 4.10.

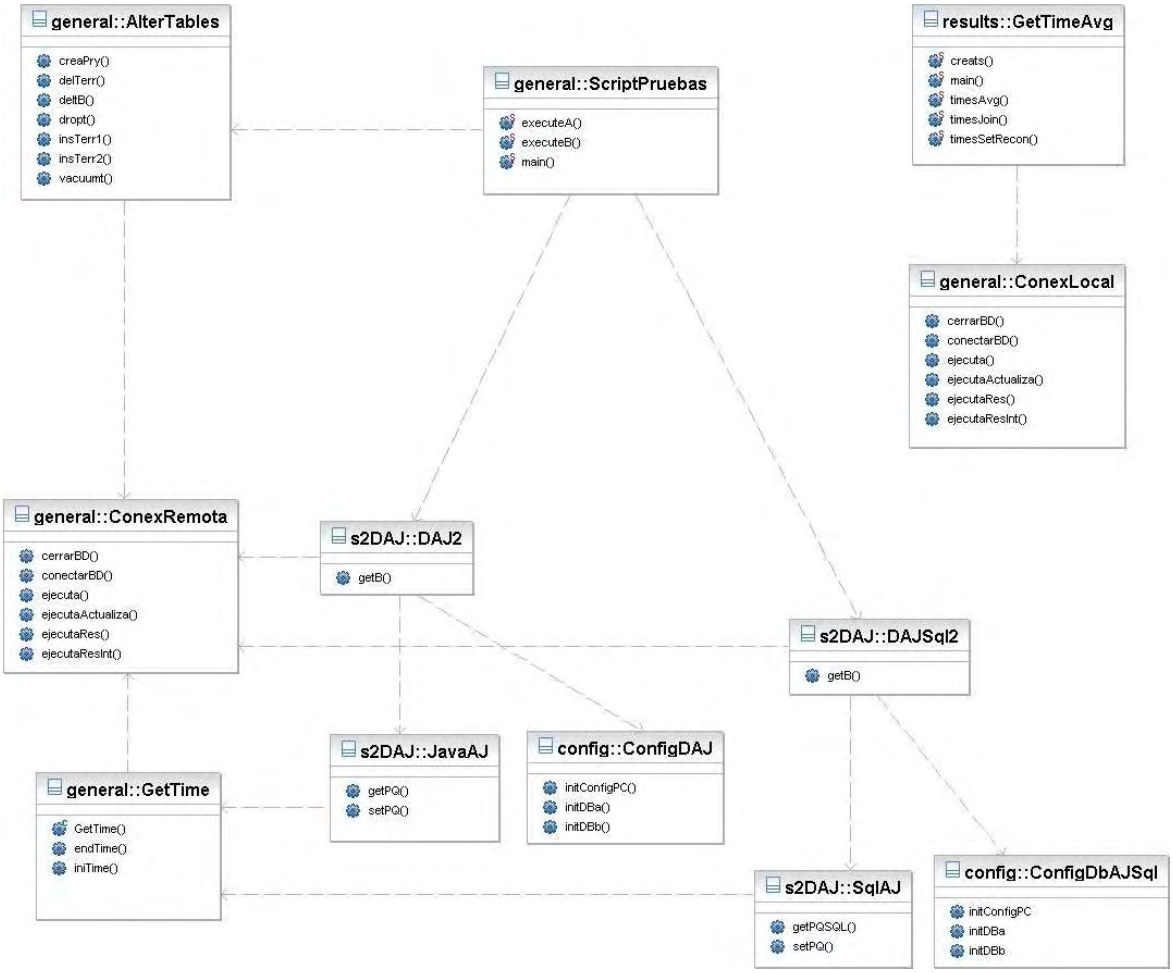

Figura 4.10: Diagrama de clases del algoritmo DAJ en el sitio S_2

Como vimos anteriormente, tenemos varias alternativas para implementar el ANTI-JOIN, con los operadores de SQL (mencionados anteriormente) o con el programa de Java.

El algoritmo programado en Java se basa en el join zig-zag (con dos índices) [GMUW01], que consiste en posicionar dos apuntadores en los primeros elementos de las tablas (en nuestro caso las proyecciones de las réplicas), comparar cada par de valores y recorrer los elementos mientras no encuentre tuplas iguales; los valores diferentes, que son los que nos interesan, se almacenan en un arreglo.

A continuación se muestra este algoritmo.

Programa 1 Algoritmo del ANTIJOIN: código procesado en el sitio S_2

```
Entrada:
        archivo1 - proyección de T1
        archivo2 - proyección de T2
Output:
        I1/2[] - tuplas inconsistentes en T1
        I2/1[] - tuplas inconsistentes en T2

 while (Tupla1!=null & Tupla2!=null)
   if (Tupla1=Tupla2)             // si las tuplas son iguales
       Tupla1.next();             // apunta al siguiente
       Tupla2.next();
   else if (Tupla1<Tupla2)
       I1/2[i1]=Tupla1;        // Tupla1 no está en archivo2
       i1++;
       Tupla1.next();
   else                           // Tupla2 es menor que Tupla1
       while (Tupla2!=null)       // mientras exista un valor en archivoB
         if (Tupla1>Tupla2)
             I2/1[i2]=Tupla2;  // Tupla2 no está en archivo1
             i2++;
             Tupla2.next();
         else
             break;               // si Tupla1<Tupla2 o
                                  // Tupla1=Tupla2 sale del ciclo

           // si Tupla1->null los valores de archivo2 se agregan a I2/1[]
           // si Tupla2->null los valores de archivo1 se agregan a I1/2[]
```

Los experimentos exploratorios nos muestran el desempeño de cada uno de estos algoritmos DAJ.

Capítulo 5

Experimentación y resultados

En este capítulo presentamos los resultados del desempeño, en cuanto al tiempo de procesamiento y de comunicación, de dos de los algoritmos de detección de inconsistencia en réplicas: DPC y DAJ_{Java}. Elegimos sólo el DAJ_{Java} pues fue el más eficiente de los algoritmos de la familia DAJ, de acuerdo a los resultados obtenidos en los experimentos exploratorios.

5.1 Experimentación

En las siguientes secciones describimos el ambiente de experimentación (hardware, software, BDD) y las mediciones realizadas en cada experimento.

5.1.1 Plataforma

Hardware

Los experimentos se realizaron en 4 equipos:

- dos servidores personales con un procesador Intel Core 2 Duo a 1.66 GHz, disco duro de 120 GB a 5400 RPM y 1 GB de memoria RAM, cada uno;
- dos servidores con 2 procesadores Intel Core Duo a 1.66 GHz, disco duro de 160 GB a 15000 RPM y 2 GB de memoria RAM, cada uno.

Además se realizaron en tres tipos de redes diferentes: una red local (LAN) y dos redes metropolitanas (MAN).

- La LAN se instrumentó con un switch 3com a 10/100 Mbps (Auto-speed sensing). Para configurarla conectamos las máquinas al switch y la velocidad de transmisión promedio fue de 10 MB/s.

- Las MAN se instrumentaron a través de los dispositivos que forman la RedUNAM y del modem conectándonos con una red privada de banda ancha.

  Para configurar la primera Red MAN conectamos las máquinas a través de la RedUNAM [1] teniendo una velocidad de transmisión de 700 KB/s en promedio. La segunda configuración de Red MAN [2] , con características diferentes, disminuyó la velocidad de transmisión a 128 KB/s en promedio.

Software

En seguida presentamos el software instalado en los servidores y los programas necesarios para la experimentación.

- El sistema operativo de los equipos, tanto de los servidores personales, como de los otros servidores, fue Linux (Fedora 8).
- El SMBD de la BD en cada equipo fue PostgreSQL 8.2, utilizando Slony como sistema de replicación. Simulamos una BDD homogénea: todos los equipos tienen el mismo manejador y el mismo modelo de datos.
- La versión de Java que utilizamos fue la 6.0 con bibliotecas para la comunicación entre los sitios. Como compilador de C utilizamos el gcc-3.3.6 (compatible con el CPISync) y las bibliotecas gmp-4.1.2 y ntl-5.3.1 (para las operaciones de aritmética modular).
- Los algoritmos distribuidos de DPC y DAJ se implementaron en Java y C, con funciones tomadas del código del CPISync [CPI], otras fueron definidas en PostgreSQL. Los diagramas de clases se presentaron en el capítulo anterior (véase Figura 4.6 y Figura 4.9).

5.1.2 Descripción de la BDD

Simulación de la BDD

El conjunto de datos fue generado con el programa DBGEN de TPC-H [TPC05] con factores de escalación 1 y 2 (n_{fe_1} y n_{fe_2}): este parámetro define la cardinalidad del conjunto de datos, 1 GB y 2 GB respectivamente. Creamos la BD en uno de los sitios y para la experimentación utilizamos tres de las tablas generadas, $CUSTOMER$, $ORDERS$ y $LINEITEM$.

[1] Centro de Monitoreo RedUNAM, http://www.noc.unam.mx/
[2] Conexión con Prodigy infinitum

El diagrama de la Figura 5.1 nos muestra el esquema de la BD generada con TPC-H.

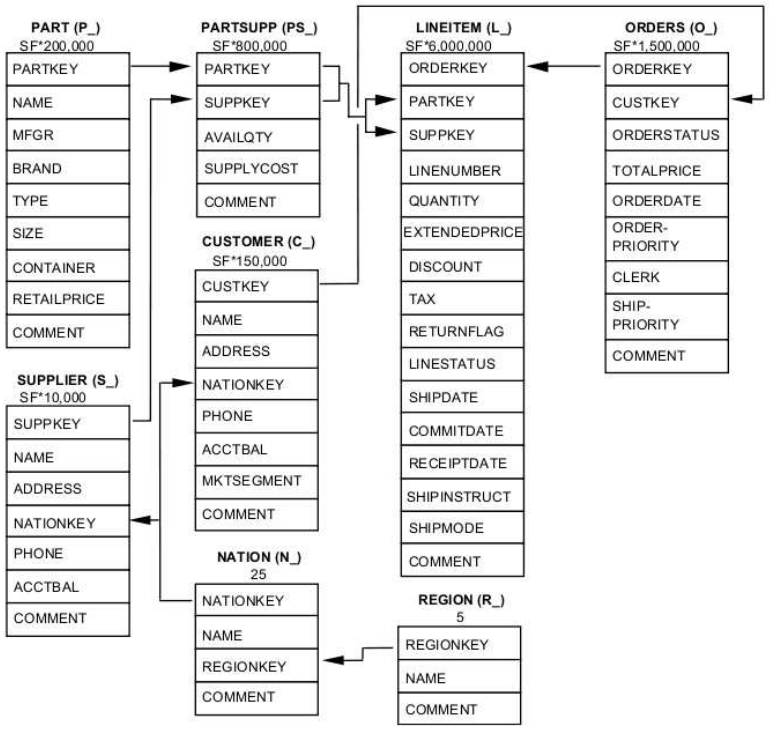

Figura 5.1: Esquema de la BD generada con TPC-H

Con Slony, un software de replicación, creamos copias de esas tablas en todos los demás servidores.

Los experimentos se realizaron con las réplicas de las tablas $CUSTOMER$, $ORDERS$ y $LINEITEM$ (véase Tabla 5.1), para $CUSTOMER$ se crearon dos réplicas, $CUSTOMER_1$ y $CUSTOMER_2$, se hizo lo mismo para $ORDERS$ y $LINEITEM$.

tabla	k	tipo	cardinalidad (n_{fe_1})	cardinalidad (n_{fe_2})	k_{max}	$orden$
$CUSTOMER$	$c\_custkey$	integer	150,000	300,000	300,000	311,111
$ORDERS$	$o\_orderkey$	integer	1,500,000	3,000,000	6,000,000	6,110,011
$LINEITEM$	($l\_orderkey$,	(integer,	6,000,000	12,000,000	(12,000,000,	120,111,611
	$l\_linenumber$)	integer)			7)	

Tabla 5.1: Descripción de la BD

tabla: son las tablas replicadas para la experimentación;

k: atributo llave utilizado para detectar los errores de inconsistencia. Sus valores están en el intervalo: [1, k_{max}];

tipo: es el tipo de dato de k;

cardinalidad (n_{fe_1}): es la cardinalidad de la tabla generada con un factor n_{fe_1};

cardinalidad (n_{fe_2}): es la cardinalidad de la tabla generada con un factor n_{fe_2};

k_{max}: es el valor máximo de k;

orden: es el orden del campo en el que se mapean los valores de k y los $evalPoints$, para el algoritmo DPC.

Condiciones iniciales y generación de errores

En la configuración inicial de los experimentos los servidores se conectan de acuerdo al tipo de red que se está evaluando. Cada sitio tiene réplicas consistentes (con la misma información) de las tablas de la BD

Antes de ejecutar cada experimento los errores son inducidos, se agrega un número de tuplas incorrectas (diferentes a las tuplas existentes) en las réplicas, de acuerdo a los valores de $\bar{m}$ (número de errores). Los valores erróneos de k que insertamos fueron menores a *orden* y mayores a k_{max}, éstos se indican en la Tabla 5.1. Los errores se distribuyeron en ambas réplicas de acuerdo a los valores de m_1 y m_2.

Las estadísticas fueron actualizadas al iniciar cada experimento (la operación VACUUM de PostgreSQL permite hacer esto). Se verificó que las réplicas volvieran a un estado consistente.

Las condiciones de la experimentación fueron restringidas, evitando que factores externos pudieran alterar los resultados obtenidos en las mediciones de tiempo (como la ejecución de otros procesos, el tráfico en la red, etc.). Las condiciones en las que se realizaron los experimentos para los algoritmos fueron las mismas, para que los tiempos obtenidos fueran comparables.

5.1.3 Mediciones

Para comparar el desempeño de los algoritmos DPC y DAJ medimos sus tiempos de procesamiento y de comunicación para compararlos.

El primero se refiere al tiempo en el que cada algoritmo realiza las operaciones de detección y medición de la inconsistencia de las réplicas de una tabla. Para DPC

medimos el tiempo que tarda en calcular el polinomio característico (t_{PC_i}) en cada sitio S_i, el tiempo para encontrar las diferencias entre las réplicas, ó cálculo de raíces (t_{Roots}), y el tiempo en el que calcula la métrica (t_{metric}). Para el algoritmo DAJ, medimos el tiempo que tarda en obtener las proyecciones (t_{Proy_i}) en cada sitio S_i, el tiempo de ejecución del ANTIJOIN (t_{AJ}) y el tiempo en el que calcula la métrica (t_{metric}).

Algunas de las operaciones, como el cálculo de los polinomios o la obtención de las proyecciones, se realizan al mismo tiempo en los sitios remotos con respecto al sitio que obtiene la métrica (sitio central). Por esta razón se considera sólo el tiempo máximo de estas operaciones.

Los tiempos de procesamiento totales para cada algoritmo son:

$$\begin{aligned} t_{DPC_{proc}} &= max(t_{PC_1}, t_{PC_2}) + t_{Roots} + t_{metric}, \\ t_{DAJ_{proc}} &= max(t_{Proy_1}, t_{Proy_2}) + t_{AJ} + t_{metric} \end{aligned} \tag{5.1}$$

El segundo, el tiempo de comunicación, se refiere al tiempo en el que se transfiere el polinomio característico (t_{PC_1}) y la proyección (t_{Proy_1}), de S_1 a S_2, y los resultados de la detección de errores (t_{result}), de S_2 a S_1.

Para los algoritmos DAJ este valor está relacionado con la cardinalidad de las réplicas, mientras que para el DPC depende de la cardinalidad de la cota superior seleccionada de la diferencia simétrica (número de puntos de muestreo).

Denotaremos el tiempo de comunicación del DPC y del DAJ como:

$$\begin{aligned} t_{DPC_{com}} &= t_{PC_1} + t_{result}, \\ t_{DAJ_{com}} &= t_{Proy_1} + t_{result} \end{aligned} \tag{5.2}$$

Los tiempos totales de los dos algoritmos serán:

$$\begin{aligned} t_{DPC_{tot}} &= t_{DPC_{com}} + t_{DPC_{proc}} \\ t_{DAJ_{tot}} &= t_{DAJ_{com}} + t_{DAJ_{proc}} \end{aligned}$$

Recordamos que para realizar estas mediciones consideramos que los sitios están fuera de línea, es decir, no hay otros procesos (externos) que intervengan en la ejecución de nuestros programas.

5.2 Experimentos realizados

5.2.1 Experimentos exploratorios

Los experimentos exploratorios que realizamos nos ayudaron a obtener información general sobre el desempeño de los algoritmos, con estos resultados elegimos los valores del plan de experimentos.

Experimentos exploratorios para *DPC*

Con los primeros experimentos exploratorios identificamos el crecimiento del algoritmo DPC, en cuanto a tiempo, conforme aumenta el valor de $\bar{m}$ (número de errores).

A continuación presentamos los objetivos y las variaciones de los experimentos realizados:

- Identificar la demanda de recursos de este algoritmo.

  Ejecución del algoritmo DPC para diferentes valores de m y medir el tiempo que tardaba el algoritmo para obtener las diferencias entre dos réplicas de una tabla, esto es, el cálculo de raíces (la aplicación del algoritmo de reconciliación).

- Identificar como responden los algoritmos cuando cambia la distribución de los errores en las réplicas.

  Variación en la distribución del error, m_1 y m_2, entre las réplicas T_1 y T_2: 0-100%, 50-50%, etc.

En los algoritmos que evaluamos tratamos, más que la invocación de operaciones remotas, de transferir los datos de un sitio a otro para realizar, después, operaciones de forma local. En DAJ, por ejemplo, se envía el archivo con la proyección al sitio S_2 y ahí se crea la tabla y se ejecuta la operación ANTIJOIN con la tabla local. A pesar de las necesarias transferencias de los datos, buscamos disminuir al mínimo la comunicación entre los sitios realizando cálculos en cada sitio y, al final, regresando sólo los resultados totales del proceso.

Experimentos exploratorios para *DAJ*

En el capítulo anterior presentamos diferentes algoritmos DAJ para medir la inconsistencia en réplicas, éstos detectan las tuplas incorrectas aplicando la operación ANTIJOIN.

Considerando que utilizamos operadores SQL para implementar los algoritmos: DAJ_{FOJ}, DAJ_{EXC} y DAJ_{NE}, en los experimentos exploratorios comparamos los tiempos de respuesta de cada uno de ellos para identificar cuál responde de forma más rápida.

A continuación presentamos algunas de las variantes de los experimentos realizados:

- ANTIJOIN entre réplicas con o sin proyecciones.

  Ejecución de la operación ANTIJOIN aplicada a los atributos llave de dos réplicas completas (existen varios atributos). Otra alternativa fue crear tablas con las proyecciones del atributo llave, y realizar la operación con esas nuevas tablas (con un sólo atributo).

- ANTIJOIN entre réplicas con o sin índices.

  Medición de los tiempos de ejecución del ANTIJOIN cuando las llaves de las proyecciones tienen índices y compararlos con los tiempos cuando no los tienen.

- ANTIJOIN entre réplicas con o sin tuplas ordenadas.

  Medición de los tiempos de ejecución del ANTIJOIN cuando se crean proyecciones ordenadas o no ordenadas.

Además medimos el tiempo de respuesta del programa que desarrollamos en Java, para hacer el ANTIJOIN, y lo comparamos con el desempeño de las implementaciones anteriores.

Los experimentos se realizaron con las réplicas de $CUSTOMER$, $ORDERS$ y $LINEITEM$ (datos generados por TPC-H con factor de escalación 1) cuyas cardinalidades son $150,000$, $1,500,000$ y $6,000,000$, respectivamente, con un error de $\bar{m} = 1,500$ (distribuido equitativamente entre las dos réplicas). Aplicamos los operadores EXCEPT, NOT EXISTS y FULL OUTER JOIN para ejecutar el ANTIJOIN.

Los resultados de estos experimentos exploratorios, fueron de gran utilidad, ya que nos permitieron identificar cuál de los algoritmos de la familia DAJ responde de forma más rápida, además nos ayudaron a definir el plan de experimentos.

5.2.2 Plan de experimentos

El plan de experimentos fue diseñado con el objeto de responder a las siguientes preguntas:

a) ¿Cómo afecta la cardinalidad de las tablas en el tiempo de procesamiento?

b) ¿Cómo cambia el desempeño cuando varía el número de errores?

c) ¿Si los errores se distribuyen entre las réplicas de la tabla o si se encuentran presentes en una sola réplica, cambian los resultados?

d) ¿Cómo varían los resultados cuando se trata de una red tipo LAN o MAN?

Estas preguntas nos ayudaron a identificar algunos de los factores que intervienen en la medición de la inconsistencia de las réplicas. La Tabla 5.2 resume estas variantes.

Los valores de la columna Variante del plan de experimentos se definieron considerando los resultados que obtuvimos en los experimentos exploratorios.

De acuerdo a lo anterior definimos el plan de experimentos (véase Tabla 5.3).

Variación en:	**Variante**
Algoritmo	DPC DAJ_{Java}
Tipo de Red	LAN (switch) MAN (internet)
Recursos	2 servidores personales 2 servidores
Cardinalidad de las tablas (n)	$CUSTOMER$, $ORDERS$, $LINEITEM$ (n_{fe_1} y n_{fe_2})
Num. Errores ($\bar{m}$)	100; 200; 300; 500; 1,000; 1,500; 2,000; 2,500
Distribución del error entre las réplicas (m_1,m_2)	50 % (de $\bar{m}$) en T_1 y 50 % (de $\bar{m}$) en T_2

Tabla 5.2: Variantes para el Plan de experimentos

Plan de experimentos		
Configuración	*Id Experimento*	*Factor de escalación*
Servidores personales red LAN	Experimento 1.1 Experimento 1.2	n_{fe_1} n_{fe_2}
Servidores red LAN	Experimento 1.3 Experimento 1.4	n_{fe_1} n_{fe_2}
Servidores personales red WAN	Experimento 2.1 Experimento 2.2	n_{fe_1} n_{fe_2}
Servidores red MAN	Experimento 2.3 Experimento 2.4	n_{fe_1} n_{fe_2}

Tabla 5.3: Plan de experimentos

Con este plan de experimentos tratamos de resolver las preguntas que nos planteamos antes y esto nos facilitó la comparación del desempeño de los algoritmos DPC y DAJ con las ventajas y desventajas de cada uno.

Todos los tiempos resultantes que se presentan en la siguiente sección son un promedio de las ejecuciones realizadas.

5.3 Resultados

Antes de presentar los resultados obtenidos en el plan de experimentos, presentamos brevemente los resultados de los experimentos exploratorios.

5.3.1 Resultados de los Experimentos exploratorios

A continuación presentamos los resultados de los Experimentos exploratorios para los algoritmos DPC y los de la familia DAJ.

Resultados de los Experimentos exploratorios para DPC

Los resultados de los experimentos exploratorios para DPC son los siguientes:

- ▷ Realizamos experimentos con valores de $\bar{m} = 10; 20; \ldots; 100; 200; \ldots; 1,000; 4,000$ y encontramos que el algoritmo requiere mayores recursos de cómputo disponibles (de memoria y procesador) conforme crece el valor de m (independientemente de la cardinalidad de las réplicas). Para valores de $\bar{m} > 4,300$ los tiempos obtenidos fueron del orden de horas (en algunos casos días), al contrario de los tiempos obtenidos para m pequeñas (de segundos). La complejidad de procesamiento del algoritmo es de $\mathrm{O}(\bar{m}^3)$.

- ▷ Distribuimos el error entre T_1 y T_2 (m_1, m_2): 0-100%, 50-50%, etc.

  De los resultados obtenidos nos dimos cuenta que la diferencia en la distribución de los errores entre una tabla y su réplica no afecta los tiempos de transmisión del polinomio ni de la proyección, pues los valores de $\bar{m}$ son pequeños con respecto a la cardinalidad de la tabla. Por esta razón presentamos sólo los resultados para una distribución $m_1 = 50\%$ de $\bar{m}$ y $m_2 = 50\%$ de $\bar{m}$.

El análisis de los resultados de estos experimentos, nos permitió elegir los valores de $\bar{m}$ para la experimentación. Estos resultados se reflejan en los valores seleccionados en el Plan de Experimentos.

Entre los problemas a los que nos enfrentamos estaba cómo transmitir los datos entre los sitios: ¿a través de tablas o de archivos? Esto afectaba también en la eficiencia de los algoritmos. Realizamos entonces experimentos para medir los tiempos de comunicación. En el caso del envío de PC no se veía mucho la diferencia, pues los tiempos estaban por debajo de 1 segundo; en cambio, en los algoritmos basados en DAJ, los tiempos de envío de las proyecciones aumentaban. Nos dimos cuenta de que la creación de las tablas, de forma remota, hacía más lento el proceso de transferir la información, pues además de pasar por las capas del driver y del manejador de la BD, tenía que realizar una petición remota y, después de crear las tablas, se insertaban las tuplas, lo que provocaba que la respuesta del sistema fuera lenta. El envío de los datos a través de archivos provocó la disminución de los tiempos: no se realizaban peticiones remotas a la BD, sino de forma local. Los datos se enviaban de un solo bloque a un sitio, y ahí se creaba la tabla: de esta forma la ejecución de la operación local fue más rápida .

Resultados de los Experimentos exploratorios para DAJ

Los resultados de los experimentos exploratorios para DAJ son los siguientes:

- ANTIJOIN entre réplicas con o sin proyecciones.

  Se realizaron experimentos para ver cómo influye, en el tiempo de respuesta del ANTIJOIN, la creación de tablas con un solo atributo (k) y la aplicación de la operación en réplicas completas (existen varios atributos). Los resultados obtenidos nos mostraron que no hay una diferencia significativa.

- ANTIJOIN entre réplicas con o sin índices.

  La creación de índices en las réplicas agilizó los resultados de la operación ANTIJOIN. Los tiempos obtenidos en los experimentos en réplicas con índices fueron menores a los tiempos obtenidos sin índices, en algunos casos se ve mejor esta diferencia.

  La Tabla 5.4 nos muestra los tiempos promedio (en segundos) del ANTIJOIN aplicado a las réplicas, con y sin índices, de $LINEITEM$.

	FULL OUTER JOIN	**NOT EXISTS**	**EXCEPT**
Con índice	12.15 seg.	56.33 seg.	104.82 seg.
Sin índice	31.37 seg.	$>$ 5 hrs.	105.80 seg.

Tabla 5.4: Tiempos del ANTIJOIN aplicado a réplicas con y sin índices

  El índice creado para obtener ambas proyecciones fue un índice B-tree.

  En los tiempos obtenidos con el FULL OUTER JOIN, vemos que el uso del índice agiliza el tiempo de respuesta del manejador. Con el operador NOT EXISTS la mejora con el uso de índices fue importante, los tiempos disminuyeron de horas a segundos. En el caso del operador EXCEPT la diferencia en los tiempos fue mínima, pues este operador aunque tenga el índice no lo utiliza porque hace una subconsulta donde selecciona todos los valores del atributo para compararlos contra los de la otra tabla.

  Cabe aclarar que si bien el índice ayudó a disminuir los tiempos de la consulta hay que considerar que influye en el costo de mantenimiento de las tablas, cuando hay inserciones o eliminaciones.

- ANTIJOIN entre réplicas con o sin tuplas ordenadas.

  Los tiempos de ejecución del ANTIJOIN cuando se crean proyecciones ordenadas o no ordenadas, de k, se mantuvieron constantes, la diferencia fue muy pequeña.

Realizamos nuevos experimentos para comparar los algoritmos DAJ que realizan el ANTIJOIN con SQL y con Java. De acuerdo a los resultados anteriores, para los algoritmos que utilizan SQL, elegimos crear tablas con las proyecciones ordenadas de k, creando también índices.

A continuación mostramos las consultas realizadas en PostgreSQL, con los operadores: EXCEPT, NOT EXISTS y FULL OUTER JOIN, para realizar el ANTIJOIN entre las réplicas de $LINEITEM$.

En cada caso medimos los tiempos de ejecución de las consultas y obtuvimos la salida del planificador del SMBD (con el comando EXPLAIN y EXPLAIN ANALYZE de PostgreSQL), que determina el plan de ejecución de la consulta y el costo de su ejecución.

$LINEITEM_1$ y $LINEITEM_2$ son las proyecciones de la concatenación de la llave $(l\_orderkey, l\_linenumber)$ de las réplicas de $LINEITEM$, ili_1 y ili_2 son sus respectivos índices.

a) ANTIJOIN utilizando el operador FULL OUTER JOIN

Consulta

SELECT $< LINEITEM_2[k] >, < LINEITEM_1[k] >$
FROM $< LINEITEM_2 >$ FULL OUTER JOIN $< LINEITEM_1 >$
on $< LINEITEM_2[k] > = < LINEITEM_1[k] >$
WHERE $< LINEITEM_2[k] >$ IS NULL
or $< LINEITEM_1[k] >$ IS NULL

Salida

En el Plan de ejecución de la Figura 5.2 observamos que para el operador FULL OUTER JOIN la creación del índice agiliza la respuesta de esta operación.

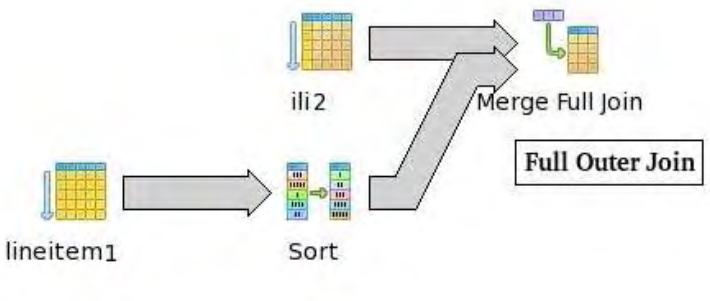

Figura 5.2: Plan de ejecución del operador FULL OUTER JOIN

El costo del plan de ejecución se muestra en la Figura 5.3 observamos que para realizar la consulta se hace uso del índice.

```
"Merge Full Join  (cost=1008242.88..4307838.30 rows=155778021 width=8) (actual time=178996.586..178996.586 rows=0 loops=1)"
"  Merge Cond: (lineitem2. k  = lineitem1. k )"
"  Filter: ((lineitem2. k  IS NULL) OR (lineitem1. k  IS NULL))"
"  ->  Index Scan using ili2 on lineitem2  (cost=0.00..169031.97 rows=6001213 width=4) (actual time=43.559..13396.696 rows=6001213 loops=1)"
"  ->  Sort  (cost=1008242.88..1023245.91 rows=6001213 width=4) (actual time=34554.466..94118.128 rows=30012979 loops=1)"
"        Sort Key: lineitem1. k "
"        ->  Seq Scan on lineitem1  (cost=0.00..86450.13 rows=6001213 width=4) (actual time=20.417..15090.977 rows=6001213 loops=1)"
```

Figura 5.3: Costo del Plan de ejecución del FULL OUTER JOIN

b) ANTIJOIN utilizando el operador NOT EXISTS

Consulta

SELECT $< k >$ FROM $< LINEITEM_1 >$
WHERE NOT EXISTS
    (SELECT $< k >$ FROM $< LINEITEM_2 >$
    WHERE $< LINEITEM_1[k] > = < LINEITEM_2[k] >$)
UNION
SELECT $< k >$ FROM $< LINEITEM_2 >$
WHERE NOT EXISTS
    (SELECT $< k >$ FROM $< LINEITEM_1 >$
    WHERE $< LINEITEM_2[k] > = < LINEITEM_1[k] >$)

Salida

Este operador utiliza los índices creados en k para hacer el recorrido de la tabla en busca de los valores en los que difieren ambas réplicas, la consulta regresa los valores sin repetición.

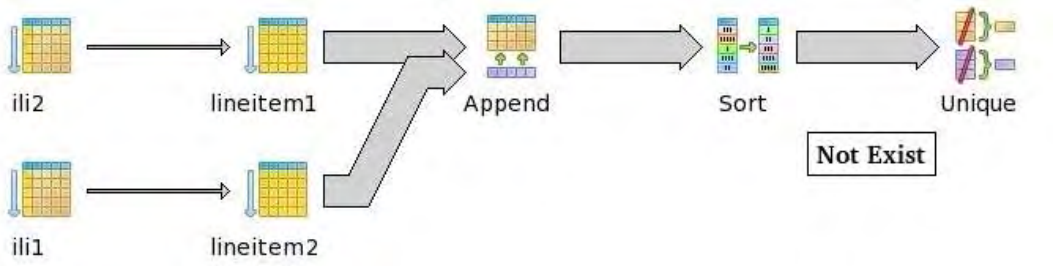

Figura 5.4: Plan de ejecución del operador NOT EXISTS

El costo de cada operación se muestra en la siguiente Figura 5.5.

```
"Unique  (cost=4094227.37..4124233.43 rows=6001212 width=4) (actual time=101378.409..101378.409 rows=0 loops=1)"
" ->  Sort  (cost=4094227.37..4109230.40 rows=6001212 width=4) (actual time=101378.405..101378.405 rows=0 loops=1)"
"     Sort Key: k"
"     ->  Append  (cost=0.00..3172434.75 rows=6001212 width=4) (actual time=101378.395..101378.395 rows=0 loops=1)"
"           ->  Seq Scan on lineitem1  (cost=0.00..1696184.64 rows=3000606 width=4) (actual time=50707.044..50707.044 rows=0 loops=1)"
"                 Filter: (NOT (subplan))"
"                 SubPlan"
"                 ->  Index Scan using ili2 on lineitem2  (cost=0.00..9.39 rows=35 width=4) (actual time=0.006..0.006 rows=1 loops=6001213)"
"                       Index Cond: ($0 = k)"
"           ->  Seq Scan on lineitem2  (cost=0.00..1416237.98 rows=3000606 width=4) (actual time=50671.346..50671.346 rows=0 loops=1)"
"                 Filter: (NOT (subplan))"
"                 SubPlan"
"                 ->  Index Scan using ili1 on lineitem1  (cost=0.00..9.53 rows=43 width=4) (actual time=0.006..0.006 rows=1 loops=6001213)"
"                       Index Cond: ($0 = k)"
```

Figura 5.5: Costo del Plan de ejecución del NOT EXISTS

c) ANTIJOIN utilizando el operador EXCEPT

Consulta

SELECT $< k >$ FROM $< LINEITEM_1 >$
EXCEPT (SELECT $< k >$ FROM $< LINEITEM_2 >$)
UNION
SELECT $< k >$ FROM $< LINEITEM_2 >$
EXCEPT (SELECT $< k >$ FROM $< LINEITEM_1 >$)

Salida

En la siguiente figura se puede observar que en esta implementación del ANTI-JOIN se realiza una subconsulta donde se seleccionan todos los valores del atributo k. En el plan de ejecución observamos que no se utiliza el índice, es por eso que en los resultados del Experimento Exploratorio 2 no hubo diferencia en la consulta, cuando se creó el índice.

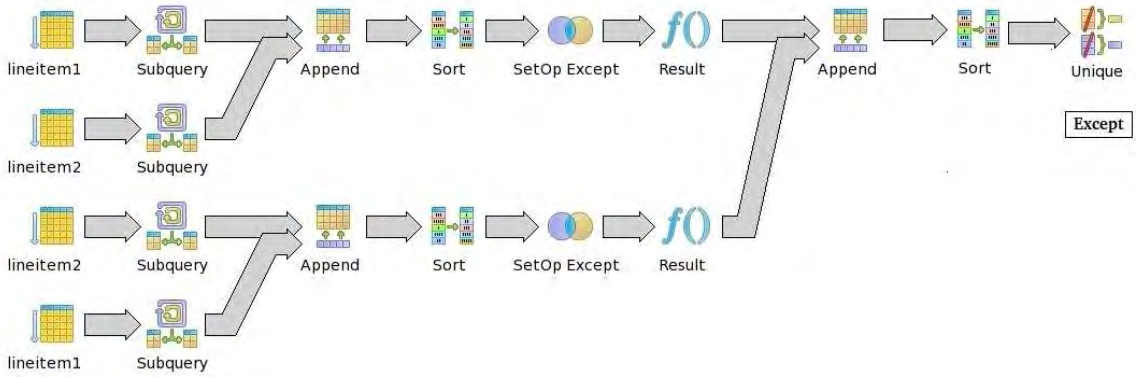

Figura 5.6: Plan de ejecución del operador EXCEPT

El costo de las operaciones del plan de ejecución se muestra en la Figura 5.7.

```
"Unique  (cost=4865896.20..4877898.62 rows=2400485 width=4) (actual time=406974.770..406974.770 rows=0 loops=1)"
" ->Sort  (cost=4865896.20..4871897.41 rows=2400485 width=4) (actual time=406974.764..406974.764 rows=0 loops=1)"
"   Sort Key: k"
"   ->  Append  (cost=2196511.64..4513047.54 rows=2400485 width=4) (actual time=406974.751..406974.751 rows=0 loops=1)"
"     ->  Result  (cost=2196511.64..2256523.77 rows=1200243 width=4) (actual time=202215.542..202215.542 rows=0 loops=1)"
"       ->  SetOp Except  (cost=2196511.64..2256523.77 rows=1200243 width=4) (actual time=202215.538..202215.538 rows=0 loops=1)"
"         ->  Sort  (cost=2196511.64..2226517.71 rows=12002426 width=4) (actual time=153416.320..179026.438 rows=12002426 loops=1)"
"           Sort Key: k"
"           ->  Append  (cost=0.00..292924.52 rows=12002426 width=4) (actual time=10.348..113380.949 rows=12002426 loops=1)"
"             ->  Subquery Scan "*SELECT* 1"  (cost=0.00..146462.26 rows=6001213 width=4) (actual time=10.343..35637.892 rows=6001213 loops=1)"
"               ->  Seq Scan on lineitem1  (cost=0.00..86450.13 rows=6001213 width=4) (actual time=10.334..13476.345 rows=6001213 loops=1)"
"             ->  Subquery Scan "*SELECT* 2"  (cost=0.00..146462.26 rows=6001213 width=4) (actual time=5.199..35748.011 rows=6001213 loops=1)"
"               -> Seq Scan on lineitem2 (cost=0.00..86450.13 rows=6001213 width=4) (actual time=5.193..13825.893 rows=6001213 loops=1)"
"     ->  Result  (cost=2196511.64..2256523.77 rows=1200243 width=4) (actual time=204759.202..204759.202 rows=0 loops=1)"
"       ->  SetOp Except  (cost=2196511.64..2256523.77 rows=1200243 width=4) (actual time=204759.196..204759.196 rows=0 loops=1)"
"         ->  Sort  (cost=2196511.64..2226517.71 rows=12002426 width=4) (actual time=155330.189..181594.771 rows=12002426 loops=1)"
"           Sort Key: k"
"           ->  Append  (cost=0.00..292924.52 rows=12002426 width=4) (actual time=21.280..115295.089 rows=12002426 loops=1)"
"             ->  Subquery Scan "*SELECT* 3"  (cost=0.00..146462.26 rows=6001213 width=4) (actual time=21.276..37530.212 rows=6001213 loops=1)"
"               ->  Seq Scan on lineitem2  (cost=0.00..86450.13 rows=6001213 width=4) (actual time=21.269..15522.526 rows=6001213 loops=1)"
"             -> Subquery Scan "*SELECT* 4" (cost=0.00..146462.26 rows=6001213 width=4) (actual time=11.851..35896.418 rows=6001213 loops=1)"
"               ->  Seq Scan on lineitem1  (cost=0.00..86450.13 rows=6001213 width=4) (actual time=11.845..13975.940 rows=6001213 loops=1)"
```

Figura 5.7: Costo del Plan de ejecución del EXCEPT

DAJ : **Java vs SQL**

La Tabla 5.5 nos muestra el tiempo promedio (en segundos) del ANTIJOIN implementado con los operadores SQL y con el algoritmo en Java (véase Capítulo 4).

Realizamos estos experimentos para las tablas, $CUSTOMER$, $ORDERS$ y $LINEITEM$.

	Tiempos de ejecución		
	$CUSTOMER$	$ORDERS$	$LINEITEM$
JAVA	0.14 seg.	1.63 seg.	8.01 seg.
FULL OUTER JOIN	0.19 seg.	2.51 seg.	11.15 seg.
NOT EXISTS	1.13 seg.	13.69 seg.	56.33 seg.
EXCEPT	1.45 seg.	16.51 seg.	104.82 seg.

Tabla 5.5: Tiempos de ejecución del ANTIJOIN

En la gráfica de la Figura 5.8 se observa mejor la diferencia de los tiempos de ejecución del ANTIJOIN realizado con FULL OUTER JOIN y con Java.

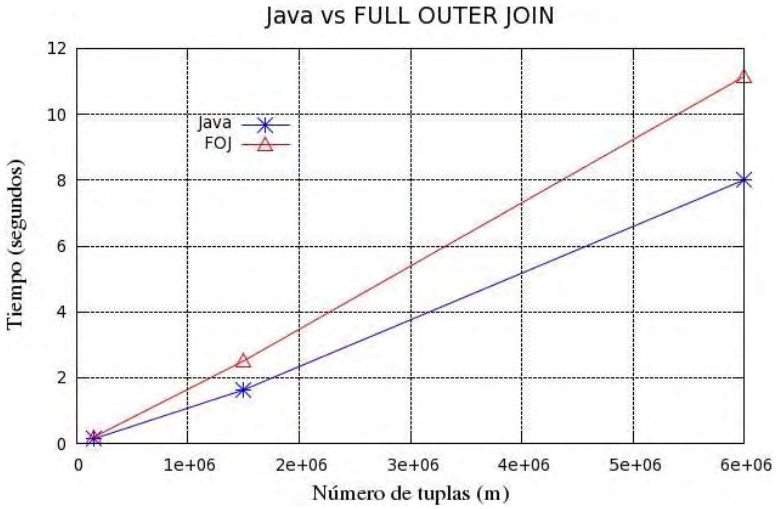

Figura 5.8: Tiempos de Java vs FULL OUTER JOIN

De acuerdo con los resultados observamos que el operador FULL OUTER JOIN es la operación SQL menos costosa, aunque tiene una diferencia significativa con la operación hecha en Java. Esto se debe a que el algoritmo en Java lee directamente los valores de la llave de dos archivos y los va comparando, sin embargo la operación hecha en SQL requiere pasar por el manejador, esto implica pasar por un mayor número de capas intermedias.

Es por eso que consideramos que la forma más eficiente para realizar el ANTIJOIN es utilizar el algoritmo en Java.

La diferencia entre los algoritmos DAJ es la forma en la que realizan el ANTIJOIN, pues el proceso anterior a la detección de las diferencias y el cálculo de la métrica son los mismos.

De los resultados anteriores podemos decir que el algoritmo con el menor tiempo de ejecución es el DAJ_{java}.

5.3.2 Resultados del Plan de experimentos, LAN

En esta sección presentamos los resultados de la evaluación de dos de los algoritmos implementados: el DPC y el DAJ_{Java}, se eligió éste último de acuerdo a los resultados obtenidos en los Experimentos Exploratorios, pues tuvo el menor tiempo de respuesta.

Como mencionamos anteriormente, es importante definir una cota superior de error, para el algoritmo DPC, este valor corresponderá a los valores de $\bar{m}$ indicados en cada Experimento.

T_1 y T_2 representan en los experimentos a las réplicas de las tablas: $CUSTOMER$, $ORDERS$ y $LINEITEM$.

Siguiendo el plan de experimentos descrito en la Tabla 5.3, a continuación presentamos los experimentos realizados y los resultados obtenidos (tiempos de procesamiento y comunicación). Al final presentaremos los tiempos totales de cada algoritmo para comparar su desempeño.

Los primeros se refieren a los experimentos realizados con una red tipo LAN y los siguientes con las dos redes tipo MAN.

Experimento 1.1

Objetivo

Medir los tiempos de procesamiento y de comunicación en los que cada algoritmo, DPC y DAJ_{Java}, detecta los errores de inconsistencia. Observar el desempeño de los algoritmos cuando se tienen las réplicas con las cardinalidades n_{fe_1} (pequeñas), almacenadas en los servidores personales.

Configuración

- ▷ Red: LAN
- ▷ Equipo: 2 servidores personales
- ▷ Cardinalidad de las tablas con n_{fe_1}: 150,000; 1,500,000 y 6,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)
- ▷ Número de errores ($\bar{m}$): 100; 200; 300; 500; 1,000; 1,500; 2,000; 2,500
- ▷ Distribución del error entre T_1 y T_2 (m_1, m_2): 50-50%

Resultados

a) Tiempos de comunicación

La gráfica de la Figura 5.9 nos muestra los tiempos de comunicación de los algoritmos DPC y DAJ_{Java}.

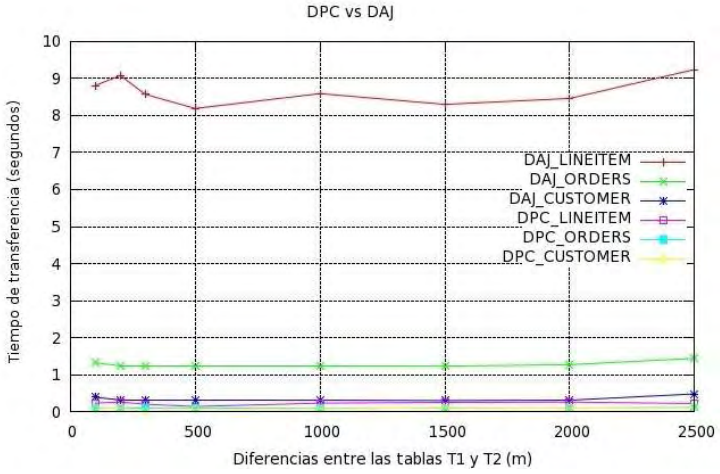

Figura 5.9: Tiempos de comunicación, Experimento 1.1

b) Tiempos de procesamiento

Los tiempos de procesamiento del Experimento 1.1 se muestran en la gráfica de la Figura 5.10.

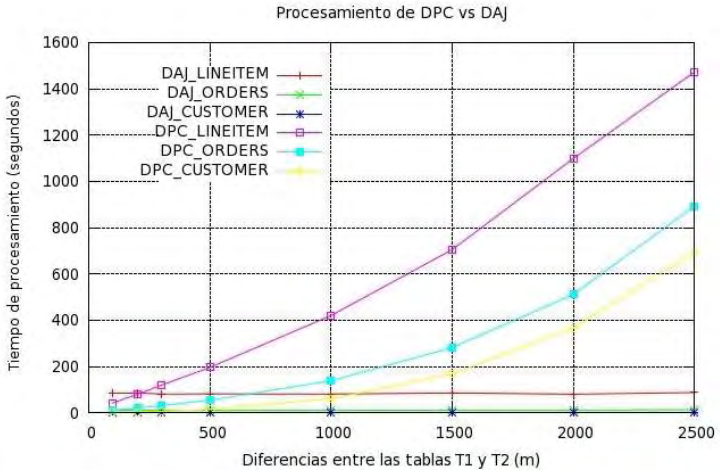

Figura 5.10: Tiempos de procesamiento, Experimento 1.1

Para ver mejor la diferencia entre los dos algoritmos presentamos la gráfica con escala de tiempo logarítmica en la Figura 5.11.

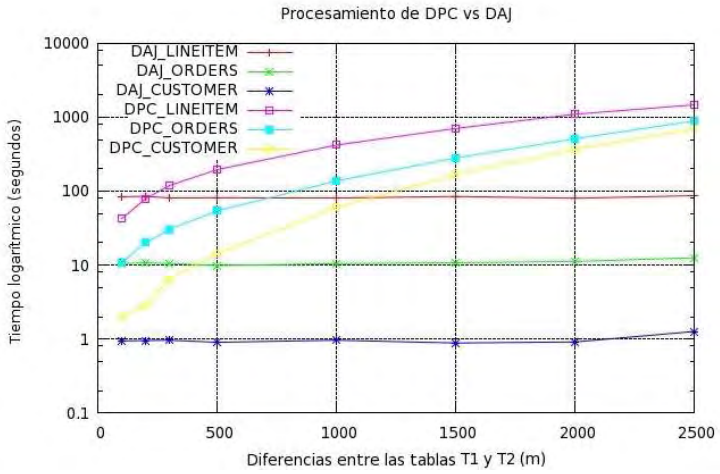

Figura 5.11: Tiempos de procesamiento (escala logarítmica) Experimento 1.1

El tiempo que tardó DAJ_{Java} en encontrar las tuplas incorrectas de las réplicas de $CUSTOMER$ fue casi cero y es por eso que no se ve la línea en la gráfica.

Conclusiones del Experimento 1.1

En la Figura 5.9 observamos que para DPC los tiempos de comunicación, con respecto al número de errores ($\bar{m}$), varían muy poco: de 0.05 a 0.2 segundos; sucede lo mismo

para las réplicas de las tres tablas ($CUSTOMER, ORDERS, LINEITEM$). Esto se debe a que este algoritmo envía, en cualquiera de los casos, $\bar{m}$ valores con el PC_1 (de acuerdo al número de puntos de muestreo que se generaron). Es por eso que el tiempo de comunicación depende sólo del número de diferencias entre las dos tablas. Por otro lado, la transferencia de la proyección, en DAJ_{Java}, crece conforme aumenta la cardinalidad de la tabla.

- ▷ *Ejemplo de DPC:* si los errores de inconsistencia de las réplicas de la tabla $CUSTOMER$ son $2,500$ tuplas ($\bar{m} = 2,500$), el algoritmo que detecta la inconsistencia de las réplicas de $CUSTOMER$ debe generar $2,500$ puntos de muestreo, más algunos puntos extra, para calcular el PC_1 evaluando la función en esos puntos. Si la tabla es $LINEITEM$ con $2,500$ errores, el proceso es el mismo y el número de datos transmitidos es el mismo.

- ▷ *Ejemplo de DAJ:* si los errores de inconsistencia de las réplicas de $CUSTOMER$ son $2,500$ tuplas ($\bar{m} = 2,500$), S_1 crea la proyección de la tabla ($152,500$ tuplas) y la envía a S_2. Si la tabla es $LINEITEM$ con 2,500 errores, S_1 crea la proyección de la tabla ($6,002,500$ tuplas) y la envía a S_2. El número de datos transmitidos cambia conforme a la cardinalidad de T_1, en este caso como $m << n$ no afecta el número de errores.

El tiempo de procesamiento de DPC, crece de la misma forma que el cálculo de raíces ($O(\bar{m}^3)$), mientras más grande sea el número de diferencias entre las dos tablas ($\bar{m}$), mayor va a ser el tiempo requerido para encontrar los datos inconsistentes. Por el contrario, en DAJ_{Java} los valores pequeños de error ($\bar{m} << n$) no afectan los tiempos del cálculo de la proyección y del ANTIJOIN, y se mantienen casi constantes en cada tabla.

Experimento 1.2

Objetivo

Medir los tiempos de procesamiento y de comunicación en los que cada algoritmo, DPC y DAJ_{Java}, detecta los errores de inconsistencia. Observar el desempeño de los algoritmos cuando se tienen las réplicas con las cardinalidades n_{fe_2} (grandes), almacenadas en los servidores personales.

Configuración

La configuración del experimento es la misma que la anterior y la única variante fue:

- ▷ cardinalidad de las tablas con n_{fe_2}: 300,000; 3,000,000 y 12,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)

Resultados

a) Tiempos de comunicación

Incrementando la cardinalidad de las tablas obtuvimos los resultados presentados en la gráfica de la Figura 5.12.

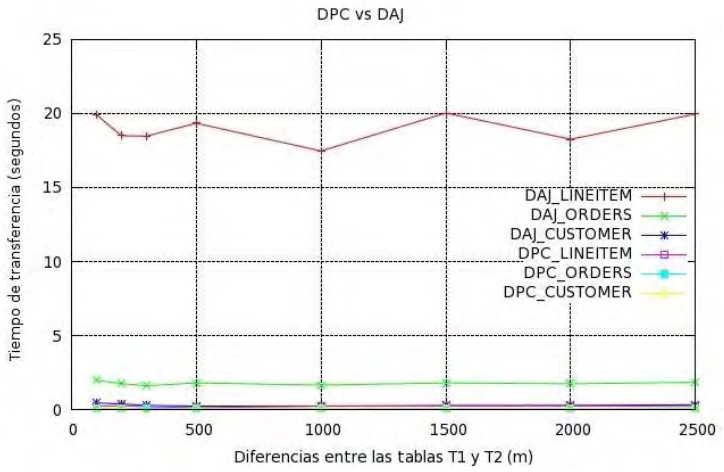

Figura 5.12: Tiempos de comunicación, Experimento 1.2

b) Tiempos de procesamiento

Estos resultados se pueden observar en la gráfica de la Figura 5.14.

Conclusiones del Experimento 1.2

En las gráficas anteriores observamos que con el aumento de la cardinalidad de las réplicas de las tablas ($CUSTOMER$, $ORDERS$, $LINEITEM$), de n_{fe_1} a n_{fe_2}, incremento también los tiempos en la transferencia de las proyecciones, para DAJ_{Java}, sin embargo para el algoritmo DPC los tiempos de comunicación se mantuvieron constantes, ya que enviába el resultado de los polinomios para los mismo puntos que el Experimento 1.1.

Los tiempos de procesamiento de DAJ_{Java} se incrementaron conforme al cálculo del ANTIJOIN de las proyecciones de las réplicas. En DPC el tiempo para encontrar las

diferencias se mantuvo constante, sólo aumentó el tiempo para calcular el polinomio característico de las proyecciones.

Comparación del Experimento 1.1 con el Experimento 1.2

Las siguientes gráficas nos muestran el comportamiento de los dos algoritmos para los servidores personales conectados a una red tipo LAN, variando la cardinalidad de las réplicas. Presentamos el comportamiento de los tiempos de comunicación en la Figura 5.13 y los de procesamiento en la Figura 5.14 conforme al crecimiento de n.

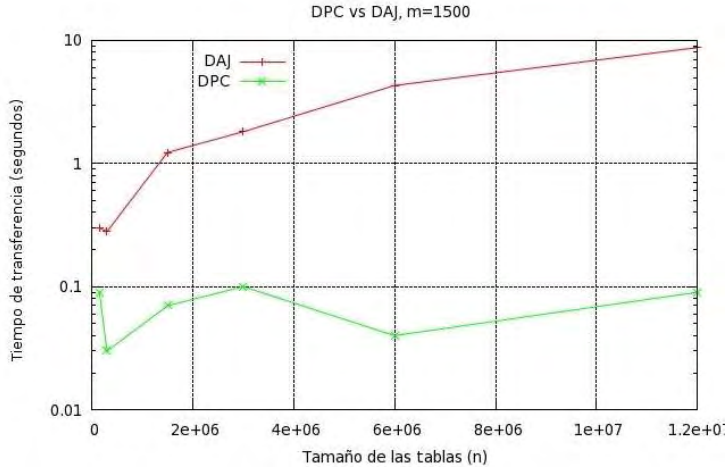

Figura 5.13: Tiempos de comunicación, Experimento 1.1 y Experimento 1.2

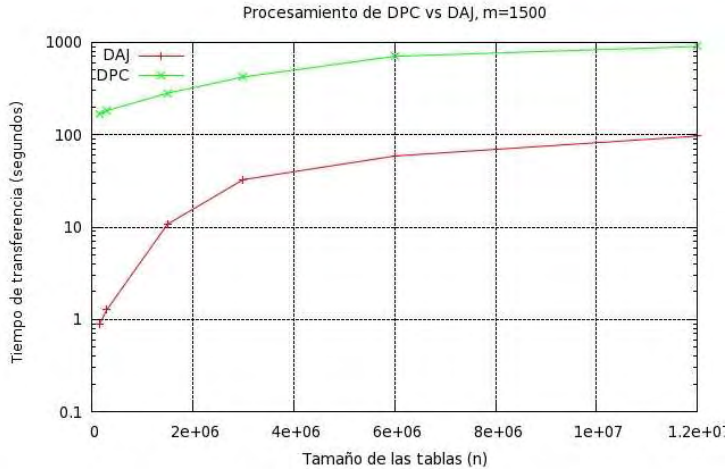

Figura 5.14: Tiempos de procesamiento, Experimento 1.1 y Experimento 1.2

Experimento 1.3

Objetivo

Medir los tiempos de procesamiento y de comunicación en los que cada algoritmo, DPC y DAJ_{Java}, detecta los errores de inconsistencia. Observar el desempeño de los algoritmos cuando se tienen las réplicas con las cardinalidades n_{fe_1} (pequeñas), almacenadas en los servidores con mayores recursos..

Configuración

- ▷ Red: LAN
- ▷ Equipo: 2 servidores
- ▷ Cardinalidad de las tablas con n_{fe_1}: 150,000; 1,500,000 y 6,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)
- ▷ Número de errores ($\bar{m}$): 100; 200; 300; 500; 1,000; 1,500; 2,000; 2,500
- ▷ Distribución del error entre T_1 y T_2 (m_1, m_2): 50-50%

Resultados

a) Tiempos de comunicación

En este caso no presentaremos las gráficas con los tiempos resultantes, ya que son prácticamente los mismos con respecto a los tiempos del Experimento 1.1.

b)Tiempos de procesamiento

Las variaciones en el procesamiento, conforme crece la cardinalidad de las réplicas, se muestra en la Figura 5.15

Conclusiones del Experimento 1.3

Los tiempos de comunicación de los algoritmos no cambiaron con respecto a los tiempos del Experimento 1.1 (véase la gráfica de la Figura 5.9). La diferencia que pudimos observar en este experimento fue la disminución, aunque pequeña, de los tiempos de procesamiento, los cálculos se realizaron más rápido.

Es importante considerar, al utilizar los algoritmos de medición de inconsistencia, la importancia de los recursos computacionales del procesador (para las operaciones que realiza el algoritmo DPC) y de la memoria (indispensable para el en DAJ_{Java}).

Experimento 1.4

Objetivo

Medir los tiempos de procesamiento y de comunicación en los que cada algoritmo, DPC y DAJ_{Java}, detecta los errores de inconsistencia. Observar el desempeño de los algoritmos cuando se tienen las réplicas con las cardinalidades n_{fe_2} (grandes), almacenadas en los servidores con mayores recursos.

Configuración

La configuración de este experimento tuvo una variante con respecto al Experimento 1.3.

- ▷ Cardinalidad de las tablas con n_{fe_2}: 300,000; 3,000,000 y 12,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente).

Resultados

a) Tiempos de comunicación

Dado que la configuración de la red se mantuvo igual (red LAN con las mismas características: velocidad, etc.), los resultados de la comunicación no cambiaron con respecto a los tiempos del Experimento 1.2: las gráficas son prácticamente las mismas, pues el número de cadenas transmitidas fue el mismo para cada caso (véase Figura 5.12).

b) Tiempos de procesamiento

Las variaciones en el procesamiento, conforme crece la cardinalidad de las réplicas, se muestra en la Figura 5.15.

Conclusiones del Experimento 1.4

De las gráficas observamos que, entre los factores que implican el tiempo de respuesta de los algoritmos, están la cardinalidad de las réplicas, pues de esto depende el tiempo en el que se transmiten los datos (para el DAJ_{Java}) y el número de errores de inconsistencia, pues el tiempo de procesamiento crece conforme aumenta este valor (para el DPC). En este experimento observamos la diferencia en los tiempos de comunicación, con respecto al experimento anterior, y la ventaja de tener mayores recursos computacionales (usando los servidores), sobre todo se vió una mejora en el DAJ_{Java} que tuvo problemas de sobrecarga de memoria en los servidores personales (en el Experimento 1.2).

Procesamiento en los Experimentos 1.3 y 1.4

La gráfica de la Figura 5.15 muestra los tiempos de acuerdo a la cardinalidad de las tablas (para n_{fe_1} y n_{fe_2}).

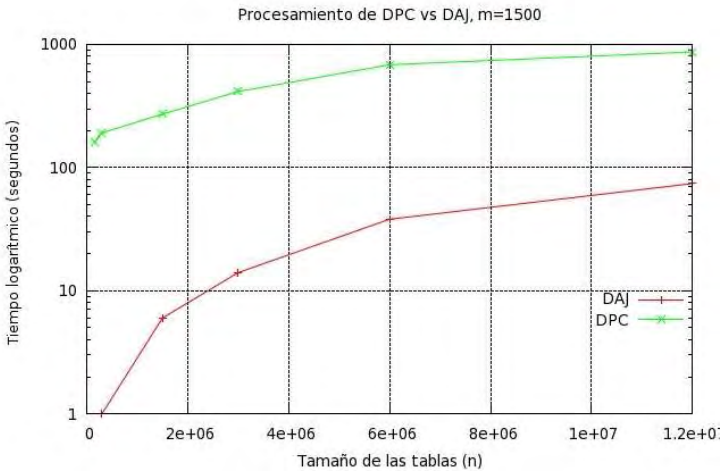

Figura 5.15: Tiempos de procesamiento, Experimento 1.3 y 1.4

DPC optimizado: Un caso particular en la detección de inconsistencias

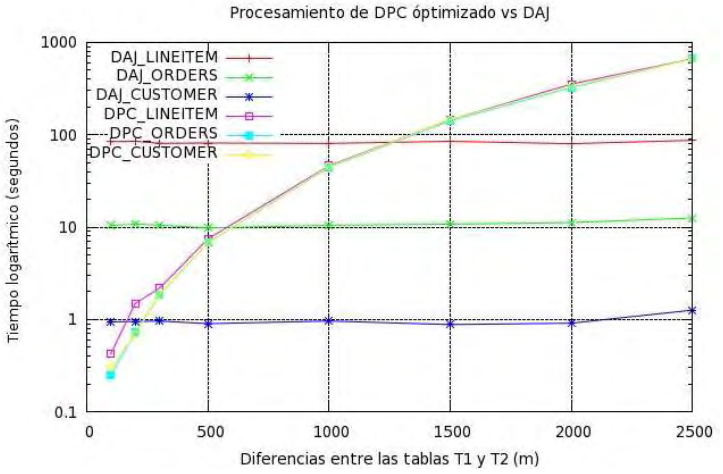

Figura 5.16: Tiempos de procesamiento para *DPC* optimizado

Como explicamos en el capítulo anterior, cuando propusimos la detección basada en *DPC* por medio del atributo llave, tenemos un caso especial en donde se realizan

sobre todo inserciones y eliminaciones de tuplas en las tablas de la BDD. Para este caso guardamos una tabla con los polinomios de las tablas originales (PC_{base}), cuando aún no se agregaba error (para un número grande de $evalPoints$); de esta forma, cuando se ejecutaba el algoritmo, calculaba sólo el PC de las tuplas que se agregaban (se modificaron los valores anteriores de la BD). Tener una base del cálculo de los polinomios disminuyó el tiempo de procesamiento en DPC, al algoritmo con esta modificación lo llamamos $DPCop$, pues se optimizó su respuesta.

En la Figura 5.16 se muestra la gráfica de los tiempos de procesamiento del Experimento 1.1 optimizado con base en PC_{base} en escala logarítmica.

5.3.3 Resultados del Plan de experimentos, MAN

Los resultados de la configuración anterior mostraron claramente como el tiempo de comunicación es proporcional al número de cadenas enviadas y a la velocidad de la red. Nótese que las características de la red fueron ideales (sin tráfico, concurrencia, etc.). Sin embargo, en la realidad, la comunicación no puede ser controlada igualando estas condiciones ideales, pues en una red MAN hay muchos otros factores que influyen en el envío de la información, la cual no llega directamente como lo simulamos en el experimento anterior.

Por esta razón, con el objeto de simular condiciones más realistas con respecto a la comunicación, en esta segunda configuración utilizamos una red MAN y conectamos todos los equipos directamente a Internet. Realizamos los mismos experimentos para dos tipos de red: la primera era una conexión rápida, con una velocidad de 700 KB/s, y la segunda conexión lenta con una velocidad de 100 KB/s, en promedio.

Experimento 2.1 y Experimento 2.2

Objetivo

Medir los tiempos de procesamiento y de comunicación en los que cada algoritmo, DPC y DAJ_{Java}, detecta los errores de inconsistencia. Observar el desempeño de los algoritmos cuando se tienen las réplicas con las cardinalidades n_{fe_1} y n_{fe_2} para el Experimento 2.1 y Experimento 2.2, respectivamente, almacenadas en los servidores personales con menores recursos conectados a dos redes MAN; comparar las diferencias de los tiempos resultantes con los obtenidos en una red LAN.

Configuración

La diferencia de estos experimentos fue la cardinalidad de las réplicas.

- ▷ Red: MAN

- ▷ Equipo: 2 servidores personales
- ▷ Cardinalidad de las tablas con n_{fe_1}: 150,000; 1,500,000 y 6,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)
- ▷ Número de errores ($\bar{m}$): 100; 200; 300; 500; 1,000; 1,500; 2,000; 2,500
- ▷ Distribución del error entre T_1 y T_2 (m_1, m_2): 50-50%

El Experimento 2.2 cambió del 2.1 en los valores de n:

- ▷ cardinalidad de las tablas con n_{fe_2}: 300,000; 3,000,000 y 12,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)

Resultados

a) Tiempos de comunicación

La mayor diferencia que se observó en el cambio de tipo de red fue la del tiempo de comunicación. La gráfica con los tiempos de comunicación con respecto a la cardinalidad, de los dos algoritmos, para la red lenta y rápida, se presenta a continuación en la Figura 5.17.

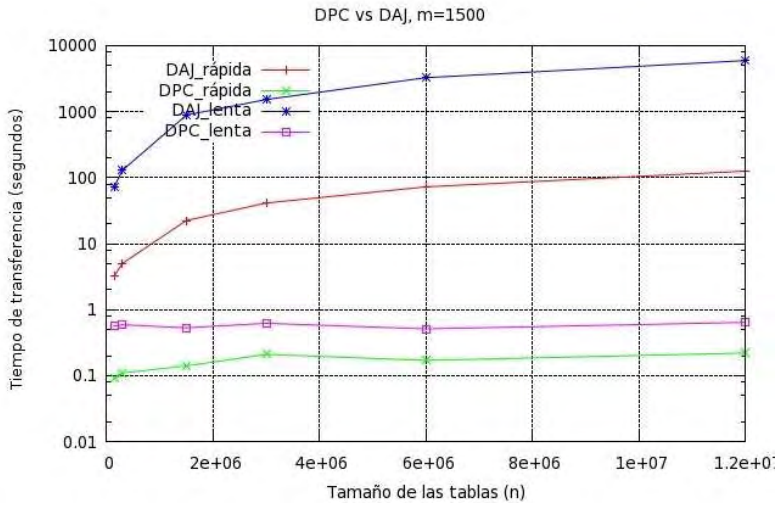

Figura 5.17: Tiempos de comunicación, Experimento 2.1 y Experimento 2.2

b) Tiempos de procesamiento

Los tiempos de procesamiento del Experimento 2.1 y del Experimento 2.2 no se alteraron, pues cambiaron las características de la comunicación, pero el equipo (servidores personales y servidores) fue el mismo: la única modificación fueron los tiempos de comunicación (véase Figura 5.14).

Conclusiones del Experimento 2.1 y del Experimento 2.2

La principal diferencia en los resultados de estos dos experimentos, con respecto a los realizados con la red LAN, fueron los tiempos de comunicación. Los tiempos de procesamiento pueden observarse en los experimentos anteriores realizados con los servidores personales.

Los tiempos de comunicación aumentaron, pues hubo factores que no dependieron del ambiente de experimentación, como el tráfico en la red, etc. De las gráficas anteriores, con los Tiempos de comunicación, observamos que las velocidades de transferencia cambiaron mucho, esto incrementó el tiempo total de respuesta del DAJ_{Java}. Los tiempos del DAJ_{Java} en la red lenta fueron más grandes que los de la red rápida.

Experimento 2.3 y Experimento 2.4

Objetivo

Medir los tiempos de procesamiento y de comunicación en los que cada algoritmo, DPC y DAJ_{Java}, detecta los errores de inconsistencia. Observar el desempeño de los algoritmos cuando se tienen las réplicas con las cardinalidades n_{fe_1} y n_{fe_2} para el Experimento 2.3 y Experimento 2.4, respectivamente, almacenadas en los servidores con mayores recursos conectados a dos redes MAN; comparar las diferencias de los tiempos resultantes con los obtenidos en una red LAN.

Configuración

La diferencia de estos experimentos fue la cardinalidad de las réplicas.

- ▷ Red: MAN
- ▷ Equipo: 2 servidores
- ▷ Cardinalidad de las tablas con n_{fe_1}: 150,000; 1,500,000 y 6,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)
- ▷ Número de errores ($\bar{m}$): 100; 200; 300; 500; 1,000; 1,500; 2,000; 2,500

- ▷ Distribución del error entre T_1 y T_2 (m_1, m_2): 50-50%

El Experimento 2.3 cambió del 2.4 en los valores de n:

- ▷ Cardinalidad de las tablas con n_{fe_2}: 300,000; 3,000,000 y 12,000,000 ($CUSTOMER$, $ORDERS$, $LINEITEM$, respectivamente)

Resultados

a) Tiempos de comunicación

Dado que la configuración de la red era la misma que el Experimento 2.1 y el Experimento 2.2, los resultados de la comunicación cambiaron muy poco. Podemos referirnos a la gráfica de la Figura 5.17.

b) Tiempos de procesamiento

De forma similar a las mediciones anteriores, el tiempo de procesamiento de los algoritmos para los servidores fue prácticamente el mismo que se presenta en la gráfica de la Figura 5.15.

Conclusiones del Experimento

Los tiempos del DAJ_{Java} en la red rápida son siempre menores a los de DPC en la red lenta. Observamos que DAJ_{Java} en la red lenta es más rápido que el DPC, por un intervalo, pero cuando las transmisiones de las proyecciones se vuelven más grandes, la técnica de DPC empieza a mostrar su ventaja por el numero de conexiones.

Si la cardinalidad de las tablas sigue creciendo, entonces los problemas que se pueden presentar al transmitir esa información tienen que considerarse. Es necesario evaluar estos algoritmos y elegir el que nos convenga según las características de nuestra red y la cantidad de datos.

5.3.4 Resultados con el tiempo total de los algoritmos

Las gráficas de los experimentos anteriores presentaron de forma separada el tiempo de procesamiento y de comunicación de cada uno de los algoritmos, a continuación mostraremos los resultados de los tiempos totales de DAJ_{Java} y para DPC mostraremos los tiempos del algoritmo optimizado $DPCop$.

$$t_{DPC_{tot}} = t_{DPC_{com}} + t_{DPC_{proc}}$$
$$t_{DAJ_{tot}} = t_{DAJ_{com}} + t_{DAJ_{proc}}$$

1)Tiempos totales, servidores personales - LAN

Los tiempos totales de los algoritmos que detectan la inconsistencia en las réplicas, ejecutados en los servidores personales, conectados a una red LAN, se muestran en la gráfica de la figura 5.18.

Para identificar los intervalos en los que un algoritmo es mejor que el otro, en cuanto a sus tiempos totales, realizamos la resta de los tiempos.

En la siguiente tabla se muestran los tiempos de comunicación y de procesamiento de los algoritmos, DPC y DAJ_{Java}, aplicados a las réplicas de la tabla $LINEITEM$.

	DPC			**DAJ**		
m	**t_PC**	**t_Transf**	**t_Roots**	**t_Proy**	**t_Trans**	**t_AntJoin**
100	41.9784	0.2348	1.0426	74.9245	8.8137	9.0339
200	**77.8177**	**0.2562**	**1.4887**	**75.2149**	**9.0756**	**9.8710**
300	117.0445	0.2040	2.1714	72.9523	8.5697	7.7198
500	188.2118	0.1376	7.4468	73.2804	8.1934	7.9395
1,000	**373.9067**	**0.2255**	**46.1862**	**71.9857**	**8.5936**	**8.5300**
1,500	559.6718	0.2528	144.7942	71.3286	7.7966	13.4372
2,000	745.4933	0.2568	352.9210	70.4059	8.4568	9.5412
2,500	810.3766	0.2228	661.8975	74.0474	9.2372	12.7512

Tabla 5.6: Tiempos de comunicación y procesamiento de DPC y DAJ_{Java}

Presentamos los resultados del algoritmo DPC en su versión optimizada.

	Suma			**Diferencia**	
m	DPC	DPC_{Op}	DAJ_{Java}	DPC_{Op}-DAJ_{Java}	$DPC - DAJ_{Java}$
100	43.2558	1.2774	92.7721	-91.4946	-49.5162
200	**79.5626**	**1.7449**	**94.1614**	**-92.4165**	**-14.5988**
300	119.4200	2.3754	89.419	-86.8665	30.1781
500	195.7962	7.5844	89.4133	-81.8289	106.3829
1,000	**420.3184**	**46.4116**	**89.1093**	**-42.6976**	**331.2091**
1,500	704.7188	145.0470	92.5623	52.4846	612.1565
2,000	1,098.6710	353.1777	88.4039	264.7738	1,010.2671
2,500	1,472.4969	662.1203	96.0359	566.0844	1,376.4610

Tabla 5.7: Suma y diferencia de los tiempos de DPC y DAJ_{Java}

En esta tabla podemos identificar el punto en el que un algoritmo es mejor que otro. Vemos que hasta $m = 200$ la diferencia $DPC - DAJ$ es negativa esto significa que hasta ese punto nos conviene reconciliar las tablas con el algoritmo de DPC, para $m = 300$, en cambio nos conviene DAJ_{Java}. Si consideramos el caso especial para la obtención de las diferencias por DPC, visto en el Capítulo 4, nos conviene usar DPC hasta $m = 1,000$, según la diferencia $DPC - DAJ$, que para DPC no suma el tiempo de cálculo de todo el polinomio característico.

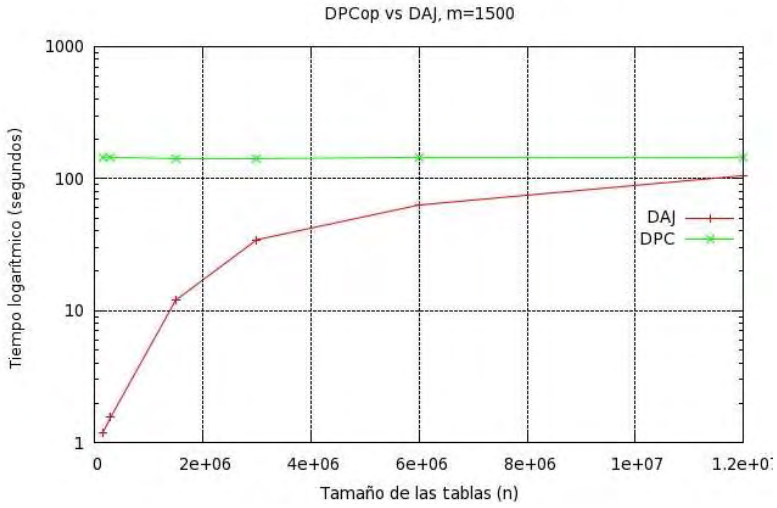

Figura 5.18: Tiempo total para: red LAN, servidores personales

2)Tiempos totales, servidores - LAN

La gráfica de la Figura 5.19 nos muestra los tiempos totales de los algoritmos, para Experimento 1.3 y Experimento 1.4.

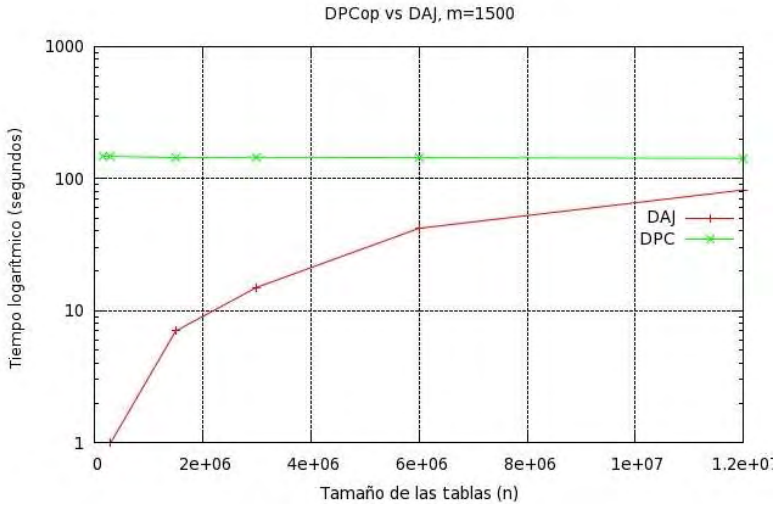

Figura 5.19: Tiempo total para: red LAN, servidores

3)Tiempos totales, servidores personales - MAN

La gráfica de la Figura 5.20 nos muestra los tiempos totales de los algoritmos, para Experimento 2.1 y Experimento 2.2.

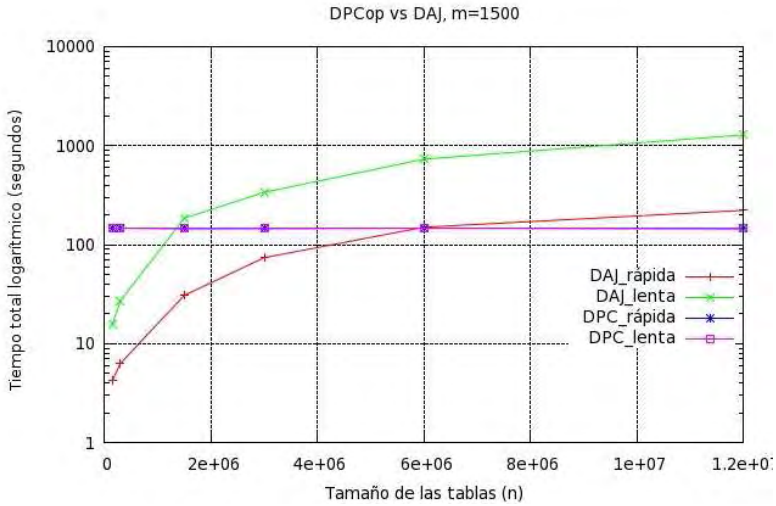

Figura 5.20: Tiempo total para: red MAN, servidores personales

Las siguientes gráficas nos muestran los tiempos totales de DPC, sin optimización, en escala logarítmica y en escala lineal, Figura 5.21 .

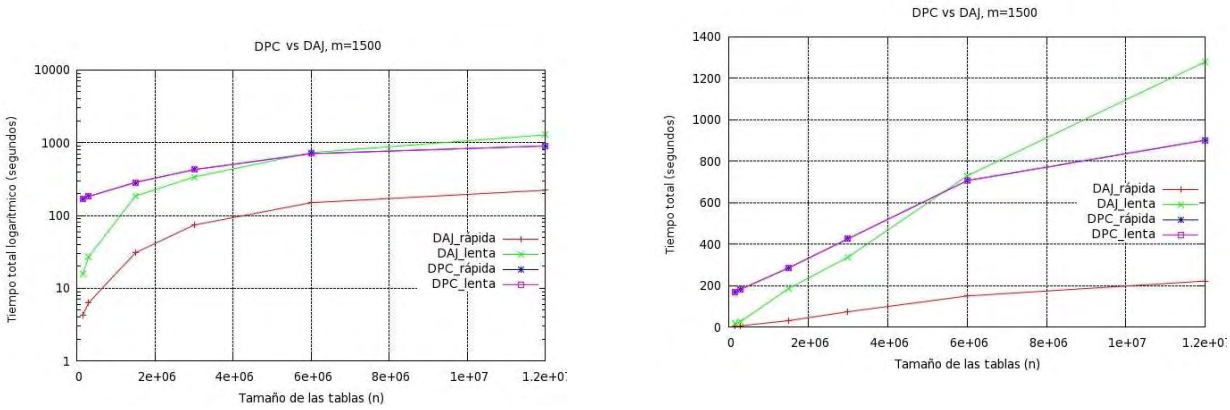

Figura 5.21: Tiempo total para: red MAN, servidores

4)Tiempos totales, servidores - MAN

La gráfica de la Figura 5.22 nos muestra el tiempo total de los algoritmos ejecutados en los servidores, conectados a través de las redes MAN, Experimento 2.3 y Experimento 2.4.

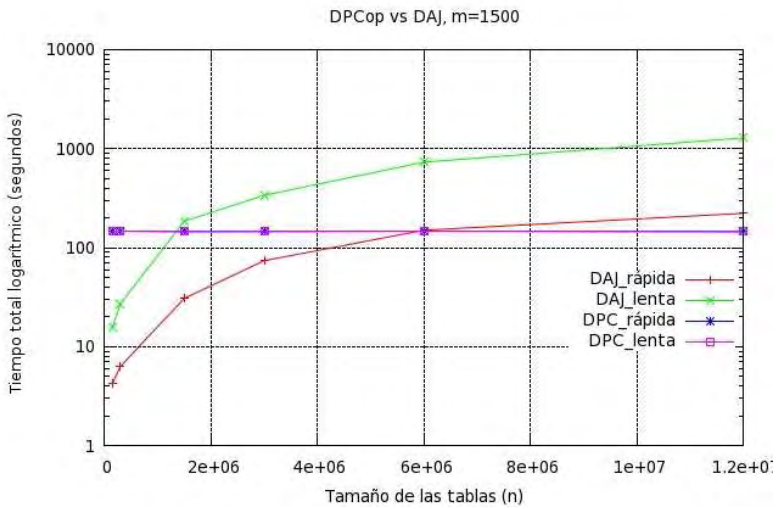

Figura 5.22: Tiempo total para: red MAN, servidores (DPC optimizado)

Capítulo 6

Conclusiones y trabajos futuros

6.1 Conclusiones

De acuerdo a los objetivos que nos planteamos al inicio de este trabajo, implementamos diversos algoritmos para medir la inconsistencia en las réplicas de una tabla, y los evaluamos de acuerdo a su desempeño en términos de su tiempo de ejecución.

Los algoritmos para detectar y medir los errores de inconsistencia de las réplicas de una BDD: DPC, DAJ_{Java}, DAJ_{FOJ}, DAJ_{EXC} y DAJ_{NE}, fueron implementados utilizando diferentes operadores.

- ▷ Obtuvimos que de los tres operadores: FULL OUTER JOIN, EXCEPT, NOT EXISTS, utilizados por los algoritmos DAJ para realizar la operación ANTIJOIN, el operador FULL OUTER JOIN fue el que ejecutó la consulta con los tiempos más cortos.

- ▷ En los mismos tres algoritmos (DAJ_{FOJ}, DAJ_{EXC} y DAJ_{NE}) observamos que los tiempos de respuesta mejoraban cuando creábamos índices en las proyecciones de las réplicas.

- ▷ Ya que la operación ANTIJOIN realizada en Java (DAJ_{Java}) respondía a las consultas de forma más rápida que el DAJ_{FOJ} elegimos el algoritmo DAJ_{Java} para compararlo con el DPC (véase Capitulo 5, resultados de los Experimentos Exploratorios).

- ▷ Aunque decidimos no utilizar el algoritmo DAJ_{FOJ} para compararlo con la detección de errores basada en polinomios (DPC), sin embargo, no descartamos los algoritmos que aplicaron los operadores SQL, pues la ventaja de realizar un ANTIJOIN en un lenguaje de consultas estándar es que hace más portables los algoritmos implementados.

▷ De los resultados obtenidos en el Plan de Experimentos (véase Capitulo 5, resultados del Plan de Experimentos), realizado para los algoritmos DPC y DAJ_{Java} (al que llamaremos sólamente DAJ), observamos que el tiempo total de los algoritmos ($t_{DPC_{tot}}$ y $t_{DAJ_{tot}}$) depende en diferentes proporciones de los tiempos de procesamiento y de comunicación (t_{proc} y del t_{com}).

En las siguientes Tablas mostraremos una comparación de los tiempos totales de los algoritmos, indicando los valores para los cuales el DPC responde de forma más rápida que el DAJ. Los valores de la columna $DPCop$, son los tiempos resultantes del algoritmo DPC optimizado, que parte de un PC_{base} para calcular los polinomios de la réplica.

Para una configuración en la que los servidores están conectados a una red LAN, observamos que para un número de errores dado ($\bar{m}$), el algoritmo DPC era más rápido que el DAJ.

En la Tabla 6.1 observamos que para las réplicas de las tablas con cardinalidades n, conviene utilizar DPC para valores de $\bar{m}$ menores o iguales al indicado en la tabla, esto de acuerdo al número de errores que generamos en los experimentos.

	LAN	
n	$DPCop$	DPC
150000	200	nunca
300000	500	100
1500000	500	100
3000000	500	100
6000000	1000	100
12000000	1000	200

Tabla 6.1: Resultados para la red LAN

Para las mismos experimentos realizados en una red MAN rápida y una MAN lenta, observamos que los valores de $\bar{m}$ aumentaron, esto significa que el intervalo de valores en los que el algoritmo DPC es conveniente, aumentó.

En los experimentos realizados con las redes MAN, los tiempos de respuesta del sistema aumentaron (pues intervinieron diferentes factores en la comunicación), se observó la conveniencia de usar algoritmos que tengan una mínima comunicación entre los sitios.

Para DPC los tiempos de comunicación, con respecto al número de errores ($\bar{m}$), varían muy poco. Esto se debe a que este algoritmo envía, para cualquiera de las réplicas ($CUSTOMER$, $ORDERS$, $LINEITEM$, con n_{fe_1} o n_{fe_2}), $\bar{m}$ valores

	MAN rápida		**MAN lenta**	
n	$DPCop$	DPC	$DPCop$	DPC
150000	300	100	1000	500
300000	300	200	1500	500
1500000	500	300	1500	1000
3000000	500	300	2000	1000
6000000	1500	300	siempre	1500
12000000	2000	300	siempre	1500

Tabla 6.2: Resultados para la red MAN

con el PC_1 (de acuerdo al número de puntos de muestreo que se generaron). Es por eso que el tiempo de comunicación depende sólo del número de diferencias entre las dos tablas. Por el contrario, la transferencia de la proyección, en DAJ, crece conforme aumenta la cardinalidad de la tabla.

El tiempo de procesamiento de DPC, crece de la misma forma que el cálculo de raíces ($O(\bar{m}^3)$), mientras más grande sea el número de diferencias entre las dos tablas, mayor va a ser el tiempo requerido para encontrar los errores de inconsistencia. Por el contrario, en DAJ los valores pequeños de error ($\bar{m} << n$) no afectan los tiempos del cálculo de la proyección y del ANTIJOIN, y se mantienen casi constantes en cada tabla.

La desventaja del algoritmo DPC es que se debe tener a priori una cota superior del número de diferencias.

Buscar la eficiencia de los algoritmos no fue una tarea sencilla, durante la implementación nos enfrentamos con diversos cuestionamientos, entre los que podemos citar los siguientes: ¿cómo transmitir los datos del polinomio y de las proyecciones, a través de archivos o tablas? ¿Qué operador SQL utilizar para realizar el ANTIJOIN?, etc. Para responder a estas preguntas realizamos experimentos exploratorios tratando de identificar la mejor solución para cada alternativa.

Tratando de optimizar los algoritmos y buscando que la evaluación de estos fuera lo más equitativa posible, convertimos en archivos las proyecciones de las réplicas para los algoritmos DAJ, ya que en DPC los polinomios se almacenan en archivos.

Los experimentos que así realizamos nos permitieron constatar que los tiempos totales de todos los algoritmos disminuyeron.

Hallamos que la obtención del polinomio característico en DPC podía optimizarse si se creaba una función Java en lugar de utilizar el SMBD PostgreSQL. Efectivamente,

después de realizar algunos experimentos sencillos vimos que implementando la misma función en Java, el tiempo se redujo (en ciertos casos cambió de horas a segundos).

También decidimos distribuir el procesamiento computacional de los algoritmos entre los sitios del sistema de la BD; de esta forma se disminuyó la carga de trabajo de cada sitio y la ejecución concurrente de algunos cálculos ayudó también a aprovechar los recursos computacionales.

Uno de los inconvenientes de la medición de inconsistencia usando los algoritmos basados en DAJ es que uno de los sitios requiere la información completa del atributo llave de la réplica de otro sitio para detectar los errores de inconsistencia. Por lo tanto deben comunicarse constantemente y, entonces, pueden intervenir factores como retraso, pérdida de información y otros problemas relacionados con la red, además de la complejidad en la comunicación.

Desde el punto de vista de la comunicación es recomendable el algoritmo DPC, pues el uso de la red es mínimo. Sin embargo, la complejidad del procesamiento es grande debido a que involucra operaciones con matrices en aritmética modular y esto hace lento el proceso de la detección de las tuplas incorrectas. DPC es recomendable cuando el número de diferencias entre las tablas es mínimo.

Podemos concluir que los algoritmos basados en la transferencia completa es conveniente cuando se cuenta con una red de comunicaciones segura y de gran velocidad (vimos cual fue la diferencia con una red lenta), además de recursos computacionales adecuados (sobre todo en memoria).

Aunque hay factores que no se tomaron en cuenta la modificación del ambiente donde se ejecutan los algoritmos alteraría los tiempos resultantes. En los experimentos presentados las condiciones para los dos algoritmos fueron las mismas.

6.2 Trabajos futuros

Los resultados de este trabajo sugieren que los algoritmos implementados se pueden aplicar, de acuerdo a los criterios definidos, para obtener métricas de integridad referencial de una BDD a partir de una BD relacional, como para el proyecto Refined.

Con respecto a los algoritmos implementados, se realizaron experimentos con redes muy pequeñas y sería interesante repetirlos en una BDD real, donde se vean más claramente los problemas que se presentan en la red (como concurrencia, tráfico, etc.), o en una BDD heterogénea con diferentes manejadores en los sitios.

Además podríamos realizar el ANTIJOIN con otros lenguajes de programación, para probar su desempeño.

Otro trabajo futuro interesante sería tratar de mejorar el algoritmo CPISync distribuye el proceso computacional entre los procesadores de los sitios que tienen las réplicas, y cada sitio realiza todos los cálculos de la detección de inconsistencia.

Mientras llevábamos a cabo los experimentos surgieron nuevas ideas para agilizar este proceso, por ejemplo, ya que las condiciones de una tabla y de sus réplicas deberían de estar en un estado consistente, al inicio o después de una reconciliación, podemos aplicar el algoritmo optimizado del DPC, donde tenemos como base los cálculos del PC que se modificarán cada vez que se realice una actualización. Podríamos disminuir los tiempos de respuesta de este algoritmo si distribuimos los cálculos computacionales como en un procesamiento paralelo.

Tratar de mejorar estos algoritmos, o buscar nuevas soluciones al problema de la detección de inconsistencia en réplicas de una BDD, nos abre a nuevas áreas de conocimiento, pues no sólo involucra conceptos de BD, sino también de matemáticas.

Es necesario buscar cómo explotar al máximo las ventajas y los recursos que nos proporciona un sistema distribuido, para mejorar el desempeño de los algoritmos que presentamos en este trabajo.

Apendice A

A.1 Slony

Slony-I A es un sistema de replicación de tipo maestro-esclavo que tiene la capacidad de replicar grandes bases de datos con un número limitado de sitios esclavos. Slony-I nació con la idea de crear un sistema de replicación que no fuese vinculado a las versiones específicas de PostgreSQL y que puediera ser inicializado y detenido en una base de datos existente, sin necesidad de un ciclo dump/reload. No es un sistema de gestión de red y no tiene ninguna funcionalidad dentro del mismo para detectar un sitio con fallas. Funciona a base de Stored Procedures y Triggers.

Los componentes de Slony-I son:

- ▷ Slon. Es un demonio escrito en C que implementa el servicio de Replicación, éste debe ejecutarse en cada sitio del sistema.
- ▷ Slonik. Es un lenguaje interpretado que sirve para administrar y definir las características de la replicación, de las tablas y de los sitios involucrados.

La Figura A.1 nos muestra la arquitectura de Slony (un maestro y varios esclavos)

Entre las ventajas tenemos que:

es de código libre (Open Source), funciona en gran cantidad de versiones de PostgreSQL, es fácil de instalar, está bien documentando, tiene una actividad alta en la comunidad Open Source, soporta la actualización y modificación del esquema en tiempo de ejecución.

Por otro lado tenemos las siguientes desventajas:

soporta pocos nodos, no implementa arquitectura multimaestro, no detecta errores ni es capaz de recuperarse de ellos, requiere una red de comunicaciones altamente confiable y con tasa de transferencia alta.

Una de las desventajas de esta herramienta es que sólo funciona para un sistema con configuración maestro-esclavo, pero actualmente se está desarrollando la versión Slony-II que soportará actualizaciones en todos los sitios de la red (multimaestro).

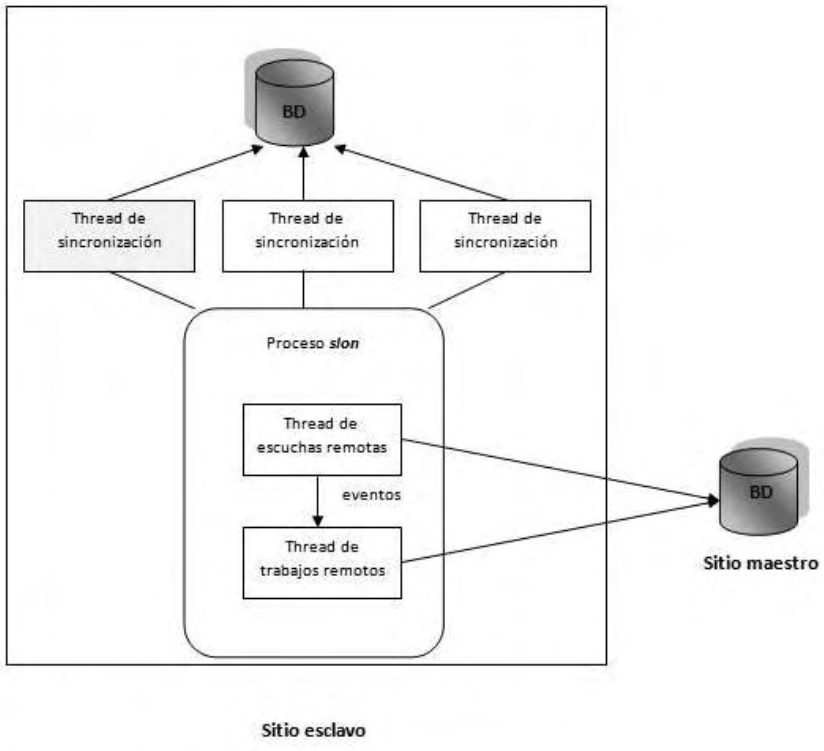

Figura A.1: Componentes de Slony

La complejidad de propagar las actualizaciones y mantener la consistencia de los datos es uno de los problemas más grandes de la replicación multimaestro. Los problemas que se presentan en las herramientas de replicación y la gran variedad de sistemas y de versiones nos indican que las dificultades que causa la replicación siguen buscando respuestas eficientes que mejoren el funcionamiento del sistema de BDD.

Mencionamos esta herramienta porque nos servirá en la parte experimental de este trabajo.

Bibliografía

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, 3th edition, 2001.

[CGM00] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. *SIGMOD Rec.*, 29(2):117–128, 2000.

[CL90] J. Chen and V. Li. Domain-specific semijoin: a new operation for distributed query processing. *Inf. Sci.*, 52(2):165–183, 1990.

[Cod90] E.F. Codd. *The Relational Model for Database Management-Version 2*. Addison-Wesley, 1st edition, 1990.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases, Principles and Systems*. McGraw Hill, 1st edition, 1984.

[CPI] CPISync. *Boston University. Laboratory of Networking and Information Systems*.
http://nislab.bu.edu/nislab/projects/stringrecon/download.htm.

[Dat03] C.J. Date. *An Introduction to Database Systems*. Addison Wesley, 8th edition, 2003.

[DB2] DB2. http://www-306.ibm.com/software/data/db2imstools/.

[EN03] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 4th edition, 2003.

[GG08] J. García-García. *Measurement and Repair of Referential Integrity Errors*. PhD thesis, Posgrado en Ciencia e ingeniería de la computación, UNAM, Por presentar, 2008.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[GK98] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3):22–27, 1998.

[GMUW01] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.

[IS] Ingres-Star. http://www.ingres.com/.

[KN97] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[MTZ01] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. page 232, 2001.

[MTZ03] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *Information Theory, IEEE Transactions on*, 49(9):2213–2218, Sept. 2003.

[MyS] MySQL. http://www.mysql.com/.

[OGG08] C. Ordonez and J. García-García. Referential integrity quality metrics. In *Decision Support Systems Journal*, volume 44, 2008.

[OGGC07] C. Ordonez, J. García-García, and Zhibo Chen. Measuring referential integrity in distributed databases. In *CIMS '07: Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*, page 232, 2007.

[Ora] Oracle. http://www.oracle.com/database/index.html.

[OV99] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[PLH89] W. Perrizo, J. Lin, and W. Hoffman. Algorithms for distributed query processing in broadcast local area networks. *IEEE Trans. on Knowl. and Data Eng.*, 1(2):215–225, 1989.

[Pos] PostgreSQL. http://www.postgresql.org/.

[SI] Slony-I. *Enterprise level replication system*. http://slony.info/.

[SKS98] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. Mcgraw-Hill College, 3rd edition, 1998.

[SS] SQL-Server. http://www.microsoft.com/latam/sql/default.aspx.

[SS05] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), 2005.

[TPC05] TPC. *TPC-H Benchmark*. Transaction Processing Performance Council, http://www.tpc.org/tpch, 2005.