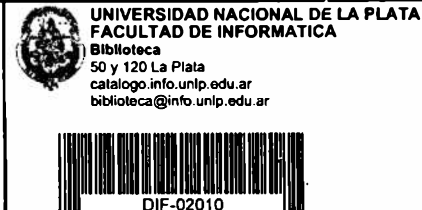
TRABAJO DE GRADO

PROCESAMIENTO DE TRANSACCIONES SOBRE UNA BASE DE DATOS

Directores: De Giusti, Armando
Bertone, Rodolfo

Alumnas: Asad Elias, María
Horak, Andrea

TES
98/5
DIF-02010
SALA



ÍNDICE

1 - Presentación del Problema

2 - Algunas definiciones:

2.1 - Base de Datos

2.2 - Seguridad e Integridad de las Bases de Datos

2.3 - Administración de transacciones

2.3.1 - Estados de una transacción

2.3.2 - Clasificación de fallos

2.3.3 - Recuperación ante fallos

2.4 - Procesamiento distribuido

2.5 - Recuperación y Concurrencia

3 - Alternativas de solución ante fallos

3.1 - Método de recuperación basado en la bitácora

3.1.1 - Gestión de registros intermedios (buffering)

3.2 - Método de recuperación basado en la doble paginación

4 - Paradox Engine: Motor de bases de datos

4.1 - Introducción

4.2 - Sistema de Bases de Datos de Pascal

4.3 - Ventajas del Sistema de Bases de Datos del Paradox

4.4 - El problema del procesamiento de transacciones

5 - Btrieve: Sistema de administración de registros

6 - Alternativa elegida para la recuperación del sistema

7 - CONSIST: Unidad de administración de bases de datos
del Paradox - Engine

7.1 - Presentación de la Unidad
7.2 - Funcionamiento de la Unidad
7.3 - Documentación de la Unidad

8 - Casos de Prueba
8.1 - Programa de prueba
8.2 - Casos no comparativos
8.3 - Casos comparativos
8.4 - Análisis de la función "Bitácora"

9 - Conclusiones

10 - Bibliografía

Apéndice A: diagrama del método de recuperación basado en bitácora

Apéndice B: diagrama del método de recuperación basado en la doble paginación

Apéndice C: mensajes de error de la Unidad.

1 - PRESENTACIÓN DEL PROBLEMA

El problema que dio origen al tema de investigación está relacionado con el manejador de Bases de Datos del Paradox.

Con dicho manejador se incorpora a un lenguaje de programación convencional, en este caso Pascal, todo el manejo de Bases de Datos disponibles en el entorno de Paradox.

El principal inconveniente observado por los desarrolladores radica en que, a diferencia de otros manejadores o motores de bases de datos, Paradox - Engine no presenta la posibilidad de manejo de transacciones; y por lo tanto, no tiene previstas soluciones para problemas de seguridad que pueden surgir del uso cotidiano de las tablas que componen la base.

Se plantea entonces, en este trabajo el estudio de las posibles operaciones de manejo de bases de datos, que puedan llevar a la pérdida de consistencia en la información almacenada en ella, al utilizar el Paradox - Engine; como ser operaciones de actualización y borrado de datos que luego de la ejecución mantengan a la base de datos en un estado consistente, como así también la recuperación del sistema que involucra a la base de datos, luego de un fallo, que restaure a la misma a un estado consistente previo a la caída. Para desarrollar operaciones de este tipo es necesario rediseñar las funciones ya existentes para el manejo de bases de datos incorporando mecanismos que controlen y mantengan la atomicidad de las transacciones ejecutadas, de acuerdo a lo expresado en los párrafos siguientes.

Se analizarán a continuación algunas cuestiones generales concernientes al funcionamiento de sistemas de bases de datos:

Un sistema de computadoras está sujeto a fallos que se producen debido a diferentes causas tales como la interrupción del suministro eléctrico, errores de software, etc. En cada uno de estos casos se pierde información referente al sistema de base de datos.

Una parte integral de un sistema es un esquema de recuperación que es responsable de la eliminación de fallos y de la restauración de la base de datos a un estado consistente que existía antes de que ocurriera el fallo.

A menudo, varias operaciones de la base de datos forman una UNIDAD LÓGICA DE TRABAJO: TRANSACCIÓN. Cada transacción es o debe ser atómica, esto es, se realizan todas las operaciones que componen la transacción o no se realiza ninguna, lo cual es fundamental para mantener la consistencia de la base de datos. Una base de datos inconsistente por cualquier motivo (mala recuperación del sistema ante un fallo o una actualización incorrecta de los datos de la base de datos) conllevará a la producción de resultados que no reflejen la realidad.

Una cuestión importante en el procesamiento de transacciones sobre bases de datos es la conservación de la atomicidad a pesar de la posibilidad de fallos que pudieran ocurrir al sistema.

Entonces si, por algún motivo un fallo sobreviniere, se debe garantizar la consistencia de las bases de datos anulando todas las transacciones que no terminaron al momento del fallo y guardando los cambios a las bases de datos de las transacciones que sí finalizaron.

Para ejemplificar lo aquí descripto, se suponen dos situaciones:
PRIMERA: Se realiza una operación sobre dos tablas de la base de datos. En la primera se agrega un registro que contiene, entre otros valores, un dato numérico que representa una cantidad la cual debe ser actualizada en la segunda tabla. Si se produce un fallo en el sistema luego de haber incorporado el registro en la primera tabla y antes de haber actualizado la segunda, la consistencia de la base, en particular de las tablas involucradas, se ha violado, por lo tanto no se debería permitir la incorporación del registro ya que la operación no había finalizado con éxito al momento del fallo.

SEGUNDA: Se realiza una operación de transferencia de fondos entre dos clientes de un banco y, paralelamente una actualización de los fondos de uno de estos dos clientes. Ambas operaciones se realizan sobre la misma tabla. Llamemos al primer cliente A y al segundo B; y a las operaciones O1 y O2.

	A	B
tiempo T1	500	100
tiempo T2	O1 lee los fondos de ambos clientes	
tiempo T3	O2 lee los fondos de ambos clientes	
tiempo T4	O1 transfiere 50 de A a B	
tiempo T5	O2 incrementa la cantidad de B en 20	
tiempo T6	O1 actualiza los fondos:	
	450	150
tiempo T7	O2 actualiza los fondos de B	
	-----	120

Resultado: el estado de la cuenta del cliente B no refleja la realidad. La base de datos quedó en estado inconsistente, ya que B tiene en su cuenta 170 y no 120. El error se produjo porque se permitió que O2 actualizara un dato sin tener en cuenta la actualización realizada por O1. Ambas operaciones terminaron con éxito, sin embargo no se conservó la consistencia de la base de datos.

2 - ALGUNAS DEFINICIONES

A continuación se darán algunas definiciones que ayudarán a entender mejor el problema planteado y las alternativas de solución propuestas en este informe:

2.1 - BASE DE DATOS

Una base de datos es un conjunto de datos persistentes utilizados por sistemas de aplicaciones de una empresa determinada.

Una empresa podría ser una solo persona o una corporación o entidad con cierto grado de complejidad.

Se entiende por datos persistentes a los datos almacenados en una base de datos. Con esto sugerimos que la información de una base de datos difiere de otros tipos de datos, como ser los datos de entrada, los datos de salida, las colas de trabajo, los resultados intermedios.

Los datos de entrada son los que ingresan por primera vez al sistema y pueden o no modificar a o convertirse en datos persistentes.

Los datos de salida son mensajes o resultados que emanan del sistema. Esta información podría derivarse de los datos persistentes.

2.2 - SEGURIDAD E INTEGRIDAD DE LAS BASES DE DATOS

Los términos Seguridad e Integridad se escuchan muy a menudo en los contextos de bases de datos:

- Seguridad implica asegurar que los usuarios están autorizados para llevar a cabo lo que tratan de hacer.
- Integridad implica asegurar que lo que tratan de hacer es correcto.

El problema de la seguridad tiene varios aspectos, entre ellos:

- aspectos legales, sociales y éticos;
- cuestiones de política interna;
- problemas de operación: asignación de contraseñas;
- materias de relevancia específica para el sistema mismo de base de datos;

De éstos, el único tema atendible es el último, aunque desde el punto de vista de este tema tampoco compete, ya que tiene que ver con la asignación de derechos de acceso o autorizaciones sobre diferentes objetos de información; a los usuarios del sistema.

En cuanto a la integridad, se refiere a la corrección de la información contenida en la base de datos. La mayoría de los sistemas actuales realizan casi toda la verificación de la integridad de los objetos a través de código de procedimientos escrito por los usuarios. Sería preferible poder especificar restricciones de integridad desde el sistema mismo y así, liberar a los programadores de esta tarea; a la vez de garantizarla para todos los sistemas creados.

En cuanto al tema que preocupa a esta investigación se entiende que garantizar la seguridad e integridad de las bases de datos, más allá de una aplicación específica, se refiere a salvaguardar a la base de datos de cualquier fallo que pueda ocurrir; lo cual está íntimamente ligado al concepto de CONSISTENCIA: Una base de datos consistente es un base de datos que conserva la integridad de sus datos a pesar de los fallos que pudieran ocurrir.

2.3 - ADMINISTRACIÓN DE TRANSACCIONES

Una transacción es una unidad lógica de trabajo, que no es por fuerza una sola operación sobre la base de datos, es más bien una secuencia de operaciones mediante la cual un estado consistente de la base de datos se transforma en otro estado consistente. Se debe mantener la atomicidad de las transacciones. No se

puede permitir que se realicen sólo algunas operaciones de la secuencia.

La situación ideal sería tener la garantía absoluta de que no se producirán fallas en la ejecución de la secuencia de operaciones, pero la posibilidad de falla siempre existe.

En el procesamiento de transacciones se debe ofrecer lo más cercano a esa garantía esperada, esto es: si todas las operaciones de la secuencia se realizan entonces la transacción se considera realizada. Si alguna de las operaciones de la secuencia sufre una falla, entonces se anula completamente la transacción.

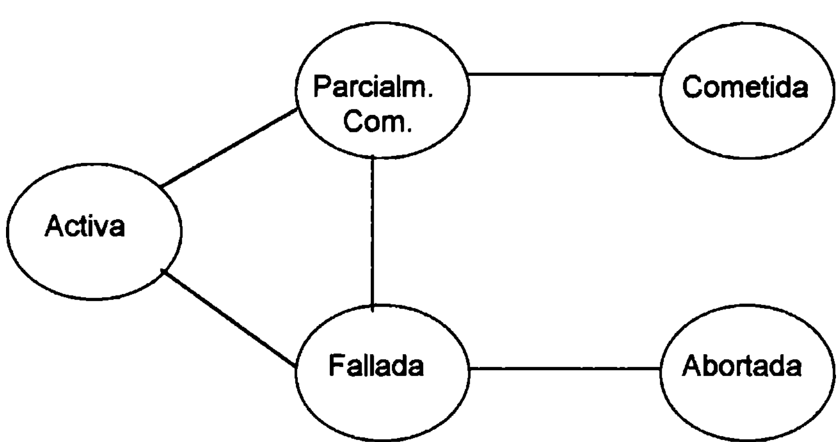
De esta manera, una transacción se lleva a cabo en su totalidad o se anula en su totalidad. Así, una secuencia de operaciones, la cual en esencia no es atómica, aparenta serlo desde el punto de vista externo.

2.3.1 - ESTADOS DE UNA TRANSACCIÓN

En este punto se definen con precisión todos los estados posibles de una transacción a lo largo de su ejecución:

Activo: el estado inicial.
Parcialmente cometido: después de que se haya ejecutado la última sentencia.
Fallado: después de descubrir que la ejecución normal no puede proceder.
Abortado: después de que la transacción haya retrocedido y se haya restaurado la base de datos al estado en que estaba antes de empezar la transacción.
Cometido: después de la terminación "con éxito".

Para entender los diferentes estados se realizó un diagrama que muestra la evolución de una transacción:



2.3.2 - CLASIFICACIÓN DE FALLOS

Se analizan a continuación cuatro tipos de fallos:

- *Errores lógicos:* la transacción no puede continuar su ejecución normal debido a alguna condición interna. Por ejemplo: overflow.
- *Errores del sistema:* el sistema ha entrado en un estado no deseable por lo cual la transacción, por el momento, no puede continuar su ejecución normal. Por ejemplo: bloqueo de transacciones.
- *Caída del sistema:* el hardware funciona mal, causando la pérdida del almacenamiento volátil. El contenido del almacenamiento no volátil permanece intacto.
- *Fallo de disco:* un bloque del disco pierde su contenido debido, por ejemplo a la rotura de la cabeza. Este fallo daña al almacenamiento no volátil.

En los tres primeros casos de trata de fallos que afectan a la o las transacciones que se están ejecutando, o al sistema pero no al almacenamiento físico.
Es decir, luego de un fallo de alguno de estos tipos, la base de datos permanece intacta.

El último caso de fallo, en cambio, afecta a la base de datos porque se trata de una falla en un medio físico (disco).

Ahora se agruparán los tipos de fallas en dos categorías:

- Fallas del Sistema o Caídas Suaves
- Fallas en los Medios de Almacenamiento o Caídas Duras

Las fallas del sistema afectan a todas las transacciones que se están realizando, se pierde el contenido de la memoria principal, pero no dañan físicamente a la base de datos.

Las fallas de los medios de almacenamiento causan daños a la base de datos, ya que se produce un daño físico de la misma, como así también de otras partes del sistema y afectan al menos a las transacciones que están utilizando la porción de la base afectada.

2.3.3 - RECUPERACIÓN ANTE FALLOS

Caídas Duras: La recuperación del sistema implica restaurar la base de datos a partir de una copia de respaldo. Es necesario que el usuario mantenga dichas copias actualizadas para poder recuperar en forma exitosa al sistema y correr sólo con el percance de perder las últimas actualizaciones. Sin una buena copia de respaldo, la recuperación del sistema no será tan exitosa como es esperado, pero esa situación depende de los usuarios.

Se puede considerar el hecho de recuperar al sistema de una caída de este tipo a partir de una copia de seguridad y de la bitácora. El resultado sería aceptable en cuanto a la buena recuperación si se considera una copia de respaldo lo suficientemente actualizada y, si la bitácora no resultó dañada por el fallo, actualizar a la base de datos restaurada con las transacciones terminadas exitosamente al momento del fallo, que residen an la bitácora.

Esta forma de recuperación asegura obtener una base de datos más actualizada que si sólo se restaura a la misma desde una copia de respaldo, pero el éxito depende de cuán actualizada sea la copia utilizada.

Caídas Suaves: los métodos de recuperación del sistema ante caídas de este tipo son el tema de estudio de esta investigación, para mantener la consistencia de las bases de datos. En el punto 3 se evaluarán las distintas alternativas, las cuales suponen recuperar las transacciones cometidas al momento del fallo guardando los cambios a la base de datos y descartando aquellas que no hayan terminado.

2.4 - PROCESAMIENTO DISTRIBUIDO

El término "procesamiento distribuido" significa que varias máquinas pueden conectarse entre sí para formar una red de comunicaciones, de manera que una sola tarea de procesamiento de datos pueda abarcar varias máquinas de la red.

Un apoyo completo para bases de datos distribuidas implica que una sola aplicación deberá ser capaz de trabajar en forma "transparente" con datos dispersos en varias bases de datos diferentes, administradas por usuarios distintos, ejecutadas en máquinas diferentes y conectadas entre sí mediante varias redes de comunicación distintas.

El término "transparente" significa aquí que la aplicación trabajaría, desde el punto de vista lógico, como si un solo programa, en una sola máquina administrara todos los datos.

Un sistema que funcione en forma distribuida debe garantizar la seguridad e integridad de las bases de datos implicadas, mediante una buena administración de transacciones concurrentes.

2.5 - RECUPERACIÓN Y CONCURRENCIA

Los problemas de recuperación y concurrencia están íntimamente ligados al procesamiento de transacciones.

En el punto 2.3 se analizó el concepto de "transacción", el término "administración de transacciones" y la recuperación del sistema ante fallos, ya sean caídas suaves (fallas lógicas) o caídas duras (fallas físicas).

Ahora se analizarán tres problemas de concurrencia:

La mayor parte de los manejadores de bases de datos son sistemas para múltiples usuarios; es decir son sistemas que permiten a dos o más transacciones tener acceso a la misma base de datos al mismo tiempo. Estos sistemas necesitan un mecanismo de control de concurrencia a fin de asegurar que ninguna transacción concurrente interfiera con las operaciones de las demás.

En esencia son tres las situaciones en las cuales una transacción, aunque correcta en sí, puede producir un resultado incorrecto debido a la interferencia de alguna otra transacción, si no existe un mecanismo de control adecuado:

El problema de la modificación perdida: La situación es que actúan dos transacciones A y B sobre una base de datos:

- la transacción A lee un registro R en el momento t1;
- la transacción B lee el mismo registro R en el momento t2;
- la transacción A actualiza R, de acuerdo a lo observado en el momento t1, en el momento t3;
- la transacción B actualiza R, de acuerdo a lo observado en el momento t2, en el momento t4.

En el momento t4 se pierde la actualización del registro R realizada por la transacción A, porque B graba su registro modificado encima del registro modificado por A.

El problema de la dependencia no comprometida: Este problema se presenta cuando se permite a una transacción leer o modificar un registro que ha sido actualizado por otra transacción, y esta última todavía no lo ha comprometido; pues puede pasar que nunca se comprometa, y en su lugar se anule, en cuyo caso la primera transacción habrá visto algunos datos que nunca existieron.

El problema del análisis inconsistente: Se supone como en el primer caso, dos transacciones A y B trabajando sobre un mismo registro R:

- en el momento t1 A lee R;
- en el momento t2 A comienza el proceso de actualización sobre R;
- en el momento t3 B lee R;
- en el momento t4 B actualiza R, de acuerdo a la lectura efectuada en el momento t3, cuyo dato coincide con la lectura efectuada por A en el momento t1.

Es claro que el resultado producido por A al finalizar el proceso de actualización es incorrecto ya que A no tomó en cuenta la actualización efectuada por B.

Se dice que A ha observado un estado inconsistente de la base de datos y por lo tanto ha realizado un análisis inconsistente. Este caso es distinto del anterior ya que A no depende de una modificación no comprometida, pues B compromete todas sus acciones antes que A finalice las suyas.

Una solución a plantear sobre los problemas que acarrea el acceso concurrente es el BLOQUEO: cuando una transacción requiere la seguridad de algún objeto en el cual está interesada, éste no cambiará de alguna manera no predecible sin que ella lo advierta, es decir, adquiere un bloqueo sobre ese objeto. El efecto del bloqueo es "impedir el acceso de otras transacciones" al objeto en cuestión y en particular, evitar que lo modifiquen.

Se analizarán ahora los tipos de bloqueo y las actividades críticas. Existen dos tipos de bloqueo:

- Bloqueo exclusivo: una transacción que obtiene bloqueo exclusivo sobre un objeto impide que cualquier otra transacción acceda al objeto bloqueado, hasta que ella lo libere.
- Bloqueo compartido: una transacción que obtiene bloqueo compartido sobre un objeto impide que cualquier otra transacción obtenga bloqueo exclusivo sobre el mismo objeto; y permite que cualquier otra transacción obtenga bloqueo compartido sobre él, hasta que ella lo libere.

En esencia, son cuatro las actividades que una transacción puede realizar sobre un objeto, en particular sobre un registro de una base de datos:

- Lectura
- Actualización
- borrado
- Escritura

De las cuatro actividades enunciadas se entienden como críticas las tres últimas, las cuales pueden producir, sin una buena administración, un estado inconsistente de la base de datos durante la ejecución de transacciones concurrentes.

- Una transacción no puede actualizar un registro de la base de datos si no tiene asignado un bloqueo exclusivo sobre ese registro. Si no fuera así otra transacción podría actualizar el mismo registro produciendo un resultado inconsistente.
- Una transacción no puede borrar un registro de la base de datos sin no tiene asignado un bloqueo exclusivo sobre ese registro. Si no fuera así otra transacción podría querer acceder a un registro que en realidad no existe.
- Una transacción necesita bloqueo exclusivo sobre un registro que desea escribir a la base de datos para evitar que otra transacción pueda leer un registro sobre el cual otra transacción no cometió su trabajo.

- En cuanto a la lectura de un registro de la base de datos, es posible que dos o más transacciones obtengan bloqueo compartido sobre el registro en cuestión y de esta manera puedan efectuar la lectura del mismo.

3 - ALTERNATIVAS DE SOLUCIÓN ANTE FALLOS

Se analizará ahora la recuperación del sistema ante caídas suaves, a partir de la bitácora, y una alternativa a este método: la doble paginación.

3.1 - MÉTODO DE RECUPERACIÓN BASADO EN LA BITÁCORA

La bitácora es la estructura más ampliamente utilizada para grabar las modificaciones realizadas a la base de datos.

Hasta el momento no se analizó qué significa el término "bitácora". Ahora se dará una definición:

BITÁCORA:
La bitácora es un lugar en la cual se almacenan paso a paso, todas las operaciones conflictivas que realizan las transacciones que se ejecutan durante la corrida de una aplicación.
Son operaciones conflictivas aquellas que alteran de alguna forma a la base de datos, por ejemplo BORRAR, UNIR, MODIFICAR, INSERTAR la información contenida en ella.

La bitácora sirve para garantizar la atomicidad de las transacciones: todas las operaciones que realiza una transacción se almacenan en la bitácora. Cuando la transacción termina exitosamente se usan los registros de bitácora para actualizar la base de datos con la seguridad de que se garantiza la consistencia de la misma, dada la terminación con éxito de la transacción.

La bitácora también sirve para recuperar al sistema ante un fallo, actualizando la base de datos con los registros de bitácora de las transacciones terminadas no comprometidas, al momento del fallo.

La bitácora debe estar almacenada en almacenamiento no volátil, para que no sea afectada por una caída suave y pueda usarse en la recuperación del sistema. La bitácora crece a medida que se incorporan registros en ella, pero para vaciarla se debe usar un método de limpieza externo al procesamiento de transacciones.

La bitácora es única para un sistema de administración de transacciones sobre una base de datos, independientemente de si se trabaja en sistemas mono o multiusuario.

Se sabe que una transacción debe ser atómica para garantizar la consistencia de la base de datos. Asimismo una transacción puede involucrar más de una operación sobre la base de datos y para garantizar la atomicidad de la transacción se deben realizar todas las modificaciones sobre la base de datos o no realizar ninguna.

Para lograr el objetivo de atomicidad primero se deben guardar en almacenamiento estable, todas las modificaciones producidas por una transacción a la base de datos y cuando ésta sea cometida, modificar la base misma.

Los campos que puede contener la bitácora son:

- Nombre de la transacción
- Nombre del dato
- Valor antiguo del dato
- Valor nuevo del dato
- Identificación del usuario (en sistemas múltiple usuario)
- Identificación de la base de datos (en acceso a múltiples bases de datos)

Los registros que se escriben en la bitácora son:

- <Ti, start>, que indica que la transacción Ti comenzó
- <Ti, xj, V1, V2>, que indica que la transacción Ti modificó el dato xj que tenía el valor V1 y ahora tiene el valor V2
- <Ti, commit>, que indica que la transacción Ti terminó exitosamente.

Siempre que una transacción modifique un dato de la base de datos, es fundamental que se cree el registro de bitácora para esa modificación antes que se modifique la base de datos.

Una vez que exista el registro de bitácora podemos sacar la modificación de la base de datos si esto es deseable. Para que los registros de bitácora sean útiles en la recuperación de fallos del sistema y del disco, el registro debe residir en memoria estable.

La modificación de la base de datos puede hacerse en forma diferida o inmediata:

Modificación diferida de la base de datos: esta técnica garantiza la atomicidad de la transacción grabando todas las modificaciones de la base de datos en la bitácora, pero aplazando la escritura de los datos hasta que la transacción se comete parcialmente. Cuando la transacción está parcialmente cometida, la información en la bitácora asociada a la transacción se usa en la ejecución de las escrituras diferidas.

Si el sistema se cae antes de que la transacción termine su ejecución, o si la transacción aborta, se ignora la información de la bitácora.

Cuando la transacción Ti se comete parcialmente, los registros asociados que están en la bitácora se usan en la ejecución de las escrituras diferidas.

Puesto que puede ocurrir un fallo mientras se lleva a cabo esta actualización, se debe asegurar que todos los registros de bitácora se escriban en almacenamiento estable antes que comiencen las actualizaciones. Una vez que se ha realizado esto, puede tener lugar la actualización real y la transacción entra en estado cometido.

Utilizando la bitácora, el sistema puede enfrentarse a cualquier fallo que resulte en la pérdida de información en memoria volátil.

El esquema de recuperación usa el procedimiento REDO(Ti) que asigna los nuevos valores a todos los datos que actualiza la transacción Ti, cuyos datos y nuevos valores pueden encontrarse en la bitácora.

La operación REDO(Ti) debe ser IDEMPONENTE, es decir que ejecutarla varias veces debe producir el mismo resultado que ejecutarla una vez.

Después de ocurrir un fallo, el sistema de recuperación consulta la bitácora para determinar qué transacciones necesita "volver a hacer".

Una transacción Ti necesita volver a ejecutarse si y sólo si existen en la bitácora tanto el registro <Ti, start> como el registro <Ti, commit>. Entonces, si el sistema se cae después de que la transacción terminó su ejecución, la información en la bitácora se usa en la restauración del sistema de base de datos a un estado consistente previo.

Modificación inmediata de la base de datos: esta técnica permite que las modificaciones se graben a la base de datos cuando la transacción se encuentra en estado activo: se las llama "modificaciones no cometidas". En caso de que ocurra un fallo en el sistema se usa el campo correspondiente al valor antiguo del dato, que se encuentra en la bitácora, para recuperar el sistema.

Antes de que una transacción comience se escribe en la bitácora el registro <Ti,start>. Cada vez que Ti realiza una modificación, ésta se vuelca a la base de datos y se escribe en al bitácora el registro correspondiente y cuando Ti está parcialmente cometida, el registro <Ti,commit> se escribirá en la bitácora.

Puesto que la información de la bitácora se usa para restaurar la base de datos luego de una caída del sistema, es necesario escribir el registro de bitácora correspondiente en almacenamiento estable antes de realizar la modificación a la base de datos.

Empleando la bitácora el sistema puede enfrentarse a cualquier fallo que no resulte en la pérdida de información en memoria no volátil.

El esquema de recuperación usa dos procedimientos:

- UNDO(Ti) que restaura todos los datos que la transacción Ti utiliza al valor que tenían anteriormente.
- REDO(Ti) que asigna los nuevos valores a todos los datos que utiliza Ti.

Los valores referidos pueden encontrarse en la bitácora. Después de ocurrir un fallo el sistema de recuperación consulta la bitácora para determinar qué transacciones hay que "deshacer" y cuales "volver a hacer":

- Ti deber deshacerse si existe en la bitácora el registro <Ti, start> pero no existe el registro <Ti, commit>
- Ti debe volver a hacerse si existen en la bitácora ambos registros, <Ti, start> y <Ti, commit>

3.1.1 - GESTIÓN DE REGISTROS INTERMEDIOS (BUFFERING)

Se analizan a continuación algunos detalles que son esenciales para la implementación de un sistema de recuperación que garantice la consistencia de la base de datos y requiera un tiempo extra mínimo en su ejecución:

Almacenamiento temporal de registros de bitácora: Cada registro de bitácora se graba en memoria en el momento en que se crea. Esto impone un gasto extra en la ejecución del sistema por algunas razones, por ejemplo, la grabación en memoria estable se realiza en bloques. Un registro de bitácora es mucho más pequeño que un bloque, entonces, conviene grabar varios registros de bitácora hasta completar un bloque y luego realizar la grabación a memoria estable. De esta forma se reduce el costo de grabación en nivel físico.

Pero esto implica que un registro de bitácora permanece en memoria un tiempo hasta que es grabado en almacenamiento estable. Es decir que si se produce una caída del sistema se pierden los registros de bitácora que se encuentran en memoria. Por lo tanto hay que imponer algunos requisitos para garantizar la atomicidad de las transacciones:

- La transacción Ti está en estado cometida si el registro <Ti, commit> se ha grabado en almacenamiento estable.
- Antes de grabar el registro <Ti, commit> en almacenamiento estable, se deben grabar todos los registros correspondientes a la transacción Ti.
- Antes de grabar en la base de datos un bloque que se encuentra en memoria principal, deben haberse grabado en almacenamiento estable todos los registros de bitácora que involucren a los datos del bloque.

Almacenamiento temporal de la base de datos: La base de datos se almacena en disco, y los bloques de la base de datos que se necesitan se traen a memoria principal para trabajar. Puede resultar necesario sobreescribir un bloque que ha sido modificado cuando se necesite traer otro bloque a memoria.

Las reglas para grabar los registros de bitácora limitan la libertad del sistema para trabajar con bloques de datos. Si la entrada del bloque de datos B hace que el bloque de datos A sea elegido para grabar, todos los registros de bitácora del bloque de datos A deben grabarse en memoria estable antes que se grabe el bloque A.

Puntos de verificación: Cuando ocurre un fallo en el sistema es necesario consultar la bitácora para realizar la recuperación. La consulta de la bitácora tiene dos inconvenientes:

- El proceso de búsqueda consume tiempo.
- La mayor parte de las transacciones que necesitan volver a hacerse, ya escribieron sus actualizaciones a la base de datos. Estamos ante el problema de escrituras duplicadas que si bien no hacen peligrar la consistencia de la base de datos, consumen tiempo.

Estos tipos de gastos pueden reducirse usando el concepto de "puntos de verificación": durante la ejecución el sistema mantiene la bitácora utilizando alguna de las dos técnicas de modificación vistas. Además el sistema hace, periódicamente, puntos de verificación que implican la ejecución de las siguientes tareas:

- grabar en almacenamiento estable los registros de bitácora que residen en memoria principal
- grabar en disco todos los bloques modificados de los registros intermedios
- grabar un registro de bitácora <CHECKPOINT> en almacenamiento estable.

Entonces, cuando se recorre la bitácora se hace desde el último punto de verificación y no desde el principio.

3.2 - MÉTODO DE RECUPERACIÓN BASADO EN LA DOBLE PAGINACIÓN

Una alternativa a la técnica de recuperación basada en la bitácora es la técnica de doble paginación, que, en ciertas circunstancias ofrece menos accesos a disco que los métodos de bitácora pero tiene algunas desventajas.

La base de datos se divide en bloques de longitud fija denominados PAGINAS. No es necesario grabar estas páginas en disco siguiendo un orden específico. Para localizar una página, se usa una tabla de paginación que contiene una entrada por cada página de la base de datos, en la cual se almacena un puntero a cada página.

La idea es mantener dos tablas de paginación: la tabla de paginación actual y la tabla de paginación doble.

Al inicio de la transacción, ambas tablas son idénticas. La tabla de paginación doble no se modifica. La tabla de paginación actual se actualiza cuando se modifica un registro de la base de datos. Todas las operaciones del tipo INPUT y OUTPUT utilizan la tabla de paginación actual.

Ahora se verá como se modifica el registro X que reside en la página i-ésima:

- Si la página i-ésima no se encuentra en memoria, hacemos INPUT(X)
- Modificamos la tabla de paginación actual:

a) Se localiza una página libre en disco

b) Se saca esa página de la lista de páginas libres

c) Se modifica la tabla de paginación actual de forma que la entrada i-ésima apunte a la página nueva hallada en a)

- Se asigna el nuevo valor al registro X de la página i-ésima, en la página del buffer.

Cuando la transacción termina, la tabla de paginación actual se escribe en almacenamiento no volátil y se convierte en la tabla de paginación doble; ahora puede empezar la próxima transacción.

La tabla de paginación doble es la que se utiliza para la recuperación ante una caída, por eso es importante que resida en almacenamiento estable. La tabla de paginación actual puede residir sólo en almacenamiento volátil ya que no es usada para recuperar al sistema ante una caída.

Cuando el sistema se cae, simplemente se localiza la tabla de paginación doble y se la copia a memoria principal, a su vez, se vuelcan a la base de datos los registros modificados que residen en esa tabla.

Esta forma de recuperación es mucho más sencilla que la del método basado en la bitácora y no necesita ejecutar ningún algoritmo de recuperación que consume tiempo. Sin embargo, la doble paginación presenta algunas desventajas:

Fragmentación de los datos: para lograr una transferencia de datos más rápida es conveniente mantener cercanas en disco las páginas que contengan datos interrelacionados. Esto no es posible con la técnica de doble paginación que hace que las páginas que contienen a los registros de la base de datos cambien de lugar cuando se actualizan. Con lo cual habría que recurrir a esquemas de gestión de almacenamiento físico más complejos y lentos, para optimizar la performance del sistema.

Recolección de basura: cada vez que se comete una transacción, las páginas que contienen los datos modificados por ella, se tornan inaccesibles ya que se inicializó una página nueva con los valores actuales y se cambió el puntero en la tabla correspondiente, a la nueva página. La página antigua no forma parte de las páginas libres ni contiene información útil: es basura. Periódicamente es necesario localizar éstas páginas y añadirlas a la lista de páginas libres. Este proceso llamado de "recolección de basura", implica tiempo extra y complejidad en el sistema.

Además, se ve como desventaja que la técnica es más difícil de adaptar, en contraposición con los métodos basados en bitácora, a los sistemas que permiten la ejecución de transacciones concurrentes, en los cuales se necesita algún tipo de bitácora auxiliar para recuperar al sistema ante caídas, aún cuando se utilice la doble paginación.

4 - PARADOX ENGINE: MOTOR DE BASES DE DATOS

4.1 - INTRODUCCION

El Engine o motor de bases de datos del Paradox está compuesto por conceptos que involucran a:

- Lenguaje de programación Orientado a Objetos: ya sea C++ o Pascal, de acuerdo al ambiente elegido para trabajar.
- Sistema de manejo de bases de datos relacionales del Paradox
- Funcionalidades de C++ o Pascal del Paradox Engine: este es el Sistema de Bases de Datos o Database Framework que provee interfaces de programas de aplicación Orientadas a Objetos (APIs) de C++ o Pascal, al Paradox Engine.

La terminología Orientada a Objetos está en desarrollo y todavía subsisten algunas inconsistencias. Los de lenguajes de programación que incorporan la filosofía Orientada a Objetos han desarrollado conceptos de esta última en diferentes direcciones. A continuación se da una lista de los conceptos utilizados por C++ y Pascal:

C++	Pascal
Clase	Tipo Objeto (o simplemente Objeto)
Objeto = Instancia de una clase	Objeto = instancia de un Objeto
Dato miembro	Campo
Función miembro	Método
Clase base	Antecesor u objeto base
Clase derivada = subclase	Objeto descendiente o derivado
Clase abstracta	Objeto abstracto

Para trabajar con Paradox Engine es necesario contar con:

- Librerías de Pascal o C++ (dependiendo de la elección realizada) del Paradox Engine
- Sistema de Bases de Datos de C++ o Pascal

El sistema de bases de datos ofrece importantes ventajas, propias del paradigma orientado a objetos, como ser:

- Abstracción de datos vía encapsulamiento
- Polimorfismo vía funciones virtuales
- Extensibilidad vía herencia y clases derivadas
- Reusabilidad vía todo lo anterior

El sistema de bases de datos no convierte a las bases de datos relacionales del Paradox a bases de datos orientadas a objetos. Las mismas son y permanecen siendo relacionales, pero accesibles en nuevas formas a través del sistema de bases de datos.

El mercado de bases de datos ha cambiado dramáticamente en los últimos años. Los ambientes basados en PC, ya sean compartidos o no, son cada vez más sofisticados; los usuarios pueden utilizar los numerosos beneficios que proveen los métodos de la programación orientada a objetos en el diseño y mantenimiento de aplicaciones de bases de datos.

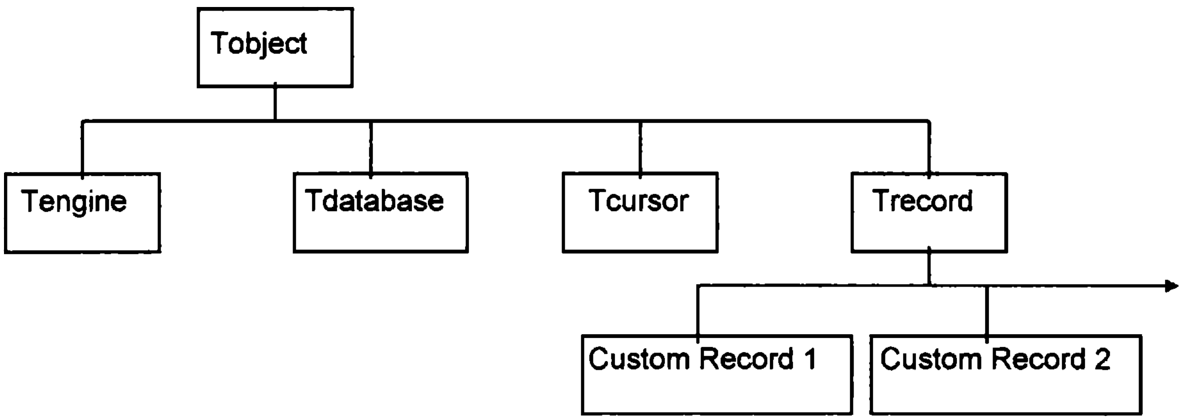
4.2 - SISTEMA DE BASES DE DATOS DE PASCAL

Para desarrollar lo concerniente al tema de estudio propuesto, se elegió el ambiente de programación Orientado a Objetos de Pascal, por lo que de ahora en más se analizarán las cuestiones concernientes al motor de bases de datos que tienen que ver con la elección realizada.

El sistema de Bases de Datos de Pascal es una capa de objetos sobre el Paradox Engine. Provee nuevas y opcionales interfaces de programación de aplicaciones Orientadas a Objetos al Paradox Engine. Opcional significa que se pueden desarrollar

aplicaciones Engine sin usar las llamadas API al sistema de Bases de Datos. Las aplicaciones que usan las librerías del Sistema de Bases de Datos son Orientadas a Objetos y aquellas que no las usan son Procedurales o no Orientadas a Objetos.

Pascal define una estructura jerárquica de objetos que se describe a continuación:



A continuación se realiza una descripción de cada objeto. Antes cabe aclarar que cada objeto que se desprende del objeto base Tobject, contiene sus constructores y destructores que son métodos que permiten inicializar y disponer cada Tipo Objeto.

Tobjetc

Es el objeto base. Los otros objetos estándares del sistema de bases de datos están definidos en la unidad OOPXENG, del Paradox Engine. Los programas que usan el sistema de bases de datos de Pascal necesitan la siguiente sentencia:

uses oopxeng;

para disponer de los beneficios que brinda esta unidad. Esta unidad también contiene a la unidad PXOOPXMSG, que contiene los mensajes de error del Paradox Engine.

Tengine

Objeto derivado de Tobject. Encapsula la información específica del motor. Sólo un objeto Tengine, llamado comunmente Engine, se necesita para una aplicación. Todas las aplicaciones de un sistema de bases de datos deben comenzar creando y abriendo un objeto Tengine en uno de estos tres modos, para poder controlar la secuencia de operaciones:

PXLOCAL: para sistemas monousuario

PXNET: para sistemas multiusuario

PXWIN: operaciones concurrentes bajo entorno Windows.

Los parámetros ambientales de Tengine están almacenados en el registro Tenv, que contiene el tipo Engine, tamaño de buffers y varios límites tales como cantidad máxima de tablas abiertas.

Además de las relaciones comunmente incluídas en la Programación Orientada a Objetos, entre objetos a través de la herencia, el sistema de bases de datos insiste sobre las relaciones de pertenencia establecidas entre objetos. Estas relaciones se definen vía campos, generalmente punteros privados a objetos, entre el objeto pertenecido y el objeto propietario. Entonces, un objeto Tengine toma conocimiento de los objetos "base de datos" que le pertenecen a través de Tengine.engobj; y cada objeto "base de datos" toma conocimiento de los objetos "cursor" que le pertenecen, a través de Tdatabase.dbobj. El acceso a los objetos pertenecidos se limita a la ejecución de los métodos de los objetos propietarios. Una situación similar existe entre registros y cursores.

Tdatabase

Objeto derivado de Tobject. Encapsula los conceptos de una base de datos como una colección de tablas relacionadas y accesos estructurados. Provee funciones sobre manejo de base de datos que operan sobre un conjunto de objetos persistentes tales como tablas, índices, etc. Los métodos de Tdatabase tratan a las tablas como una entidad y las acceden por nombre.

Para crear y abrir una base de datos, primero se debe crear y abrir el Engine y luego invocar el contructor de Tdatabase.

Tcursor

Objeto derivado de Tobject. Encapsula las estrategias específicas de acceso a una tabla. Sus métodos permiten navegar a través de los registros de una tabla ordenada (indexada) o no. Se puede abrir más de un cursor sobre la misma tabla en cualquier momento.

Los objetos Tcursor representan la noción de cursor de una tabla: una conexión activa a una tabla Paradox que admite la navegación y manipulación de registros. Se puede definir a un cursor como un manejador activo de tabla que conoce como localizar, acceder, agregar y borrar registros.

Trecord

Objeto derivado de Tobject. Provee los métodos para leer y escribir los campos de un registro. Encapsula las propiedades de un registro genérico, esto es un registro para el cual su estructura de campos y tabla asociada no se conoce hasta que corre la aplicación del sistema de base de datos.

Un objeto Trecord siempre se crea en el contexto de un objeto Tcursor existente. El constructor de Trecord crea un registro genérico para el cursor dado. El cursor "conoce" su tabla y la estructura de sus campos. El registro "conoce" a qué cursor pertenece a través del campo Trecord.curH.

Custom Records

Los registros de trabajo o Custom son objetos de una clase definida por el usuario, derivada de Trecord en Pascal, que ofrece mayor flexibilidad en el acceso a registros.

Los registros definidos para una tabla Paradox son registros genéricos que contienen un conjunto de números de campo, nombres y tipos establecidos a través de una interacción con Paradox o a través de las llamadas al Engine.

Con el sistema de bases de datos se pueden mapear algunos o todos los campos de una tabla dada, a variables que representan a esos campos. La clase de registros de trabajo permiten a una aplicación tener su propia vista de una tabla. Sólo se incorporan a esta clase los campos que el usuario necesita y, a través de éstos se pueden modificar los valores de la tabla definida en Trecord. Con esta clase se pueden definir registros específicos para una aplicación y las correspondientes rutinas de validación, a través de los pre y post procesos que provee Trecord, que se invocan automáticamente antes y después de una operación de insertar, actualizar y agregar

registros. Entonces, se pueden reescribir estos métodos de validación en una clase derivada de Trecord, de registro de trabajo, para realizar cualquier acción tal como una validación antes de insertar un registro; o también realizar una operación tal como multiplicar el campo precio por el campo cantidad y obtener un campo "virtual" que no está representado en la tabla original.

Estos registros de trabajo permiten acceder a los datos de un campo como variables simples de Pascal en este caso, sin hacer referencia al número de campo original, definido en el correspondiente registro genérico.

La clase de registros de trabajo debe redefinir ciertos métodos de Trecord para manejar los campos de los registros de trabajo correctamente.

Para realizar esta tarea se provee de la utilidad GENERATE que permite generar una clase de registros de trabajo como hija de Trecord, a partir de una tabla de registros genéricos existente, con un nombre definido por el usuario (o puede tener el mismo nombre de la tabla referida) y con los campos que el usuario necesita para realizar sus actividades.

4.3 - VENTAJAS DEL SISTEMA DE BASES DE DATOS PARADOX

Algunas de las ventajas del sistema de base de datos Paradox son:

- La organización de objetos y funcionalidades en una estructura jerárquica de clases provee un sistema conceptual más claro para analistas y programadores.

- Los mecanismos de control de acceso para datos y código, disponibles en los temas de ayuda de Pascal en nuestro caso, provee herramientas para construir aplicaciones más seguras, en cuanto a la validación de tipos y pertenencias.

- La existencia de métodos constructores y destructores en los objetos derivados de Tobject facilita la programación asegurando una correcta inicialización y disposición de objetos.

- Todos los métodos en una clase estándard son virtuales, permitiendo de esta forma clases derivadas para explotar polimorfismo y tipeo de datos en tiempo de corrida.

- Provee control de acceso concurrente a través de la definición del objeto Tengine. Es responsabilidad del usuario implementar el bloqueo de registros o tablas, asegurando de esta forma el uso exclusivo de los mismos.

- En general, las funciones del sistema de bases de datos necesitan menos argumentos que sus pares procedurales; por ejemplo: la llamada a un objeto "cursor" conoce los manejadores de tabla y de registro, a través de alguno de sus campos, entonces cada método para el objeto "cursor" necesita dos parámetros menos que los que necesitaría cada función procedural para el manejo de cursores.

- Las clases de registros custom o de trabajo, derivadas de Trecord en Pascal, ofrecen diferentes vistas de las tablas existentes; permiten la validación y consolidación de registros a través de pre y post procesos que permiten realizar controles sobre los datos durante operaciones tales como PUT y GET. La utilidad GENERATE automatiza el hecho de crear objetos del tipo registro de trabajo.

4.4 - EL PROBLEMA DEL PROCESAMIENTO DE TRANSACCIONES

Paradox Engine, a través de la jerarquía de clases definida y de sus métodos, habilita al usuario a trabajar con tablas ofreciendo tipeo de datos, control de concurrencia, relaciones de pertenencia entre objetos, manejo de registros de trabajo como simples variables del lenguaje seleccionado, sea C++ o Pascal, pero no asegura la consistencia de bases de datos al no proveer de ningún medio que englobe una transacción (secuencia de operaciones) como una unidad indivisible de trabajo.

Permite la realización de operaciones individuales sobre tablas aún cuando aquellas conformen una sola transacción:

- Permite, para cada método de actualización, la opción de salvar los resultados producidos, inmediatamente después de finalizada la operación, a través del seteo de la variable SAVEEVERYCHANGE a True. De esta no se considera el hecho de que una operación pueda formar parte de una unidad indivisible. Si se produjera un fallo en el sistema durante la ejecución de un conjunto de operaciones que debieran realizarse en su totalidad, sólo se habrían guardado los cambios producidos por las operaciones realizadas hasta que ocurrió el fallo.

- A través de la función FORCEWRITE se fuerza el salvado de los buffers asociados a la base de datos a la cual se le aplica este método. En caso de producirse una caída durante el proceso de salvado quedarían actualizados sólo algunos cambios efectuados a la base de datos y no todos.

- El método CLOSE de Tengine cierra el motor y todas las bases de datos abiertas. Los cambios producidos son salvados a disco. En caso de producirse una caída durante el proceso de salvado quedarían actualizados sólo algunos cambios efectuados a la base de datos y no todos. Además se perderían los cambios ya realizados que se encuentran almacenados en memoria.

De esta manera es factible pensar que se pueden actualizar a la base de datos, algunas operaciones de la secuencia que debiera ser indivisible, y otras no. Esta situación no es deseable en un sistema que quiere garantizar la consistencia de sus tablas, a través de la ejecución total de una secuencia de operaciones o de la no realización de ninguna de ellas.

Se desarrolló la Unidad Consist de procesamiento de transacciones que soluciona la pérdida de consistencia por fallos del sistema, almacenando todos los cambios producidos en la bitácora, evitando además la pérdida de los cambios producidos por las transacciones completadas que se encuentran en almacenamiento no volátil. Para solucionar el problema de la seguridad de las bases de datos el usuario sólo debe invocar el uso de la Unidad al inicio de su aplicación.

5 - BTRIEVE: SISTEMA DE ADMINISTRACIÓN DE REGISTROS

Btrieve es un producto que, considerando solamente el manejo consistente de bases de datos, ofrece administración de las mismas para programas de aplicación, liberando al programador de tener en cuenta cuestiones atenientes a la seguridad e integridad de bases de datos, para mantener la consistencia de las mismas.

Ofrece interfaces con varios lenguajes como ser:

- Basic
- Pascal
- Turbo Pascal
- Cobol
- C

Entre otras características de importancia, ofrece:

- Acceso a registros a través de claves múltiples
- Acceso relacional entre archivos
- Mantenimiento automático de claves
- Soporte para claves duplicadas, modificables, segmentadas y nulas
- Archivos particionados
- Controles de integridad
- Tamaño irrestricto de archivos
- Completo conjunto de utilidades para mantenimiento y creación de archivos
- Acceso multi-usuario

Para brindar una solución al problema de la consistencia de bases de datos, utiliza la técnica de doble paginación ya descripta.

Estructura física de los archivos: Por cada archivo contiene:

- Una página header: es la primera página. Guarda las caracteísticas del archivo.
- Una página índice: Guarda las claves de acceso s las páginas de datos.
- Páginas de datos: son los registros del archivo.

Cada pagina es de 512 bytes. El control de cada archivo se hace a través de un FCR (registro de control de archivos). Usa B-TREE para mantener índices. Esta estructura tiene las siguientes ventajas:

- No requiere mantenimiento periódico
- Tiene acceso rápido
- Provee almacenamiento eficiente.

Provee expansión dinámica de páginas para incorporar nuevos registros, según se necesiten. Los registros pueden ser:

- De longitud fija: Si no contienen datos ocupan lugar vacío
- De longitud variable

Utiliza técnicas de recuperación de páginas basura (páginas que no son "libres" y tampoco contienen datos útiles por ejemplo un dato que se ha borrado).

Utilización del disco:

Cuando se crea un archivo el usuario debe determinar cuál será el espacio físico de páginas y el espacio lógico de registros.

Incorpora algoritmos opcionales para manejar páginas índices y mantenerlas ordenadas. Estos algoritmos consumen mayor acceso a disco.

Presenta opciones para que el usuario estime la cantidad de páginas y la longitud de cada página, ya sean páginas índices, páginas de datos o páginas totales del archivo.

Procesos de integridad:

Protege al sistema ante caídas con la técnica opcional de PRE-IMAGING, para operaciones individuales de insertar, actualizar y borrar)

Posee procesamiento de transacciones con la técnica opcional de PRE-IMAGING con conjunto de operaciones.

Si estas técnicas están activadas, provee del algoritmo RECOVER para recuperar al sistema ante fallos.

Descripción de la técnica de PRE-IMAGING:

Para cada archivo abierto existe u archivo temporario que posee extensión PRE, donde se guardan los cambios producidos al archivo principal.

Los cambios se guardan en buffer. Cuando se cambia una página, se escribe una copia de esa página en el buffer de pre-image. Cuando finaliza la operación, se escribe primero el buffer de pre-image en la página correspondiente, luego se escribe el archivo de pre-image y por último se escriben las actualizaciones al archivo de datos en disco.

Si se llenaran los buffers antes de finalizar las operaciones, se escriben los cambios para liberar espacio. Al finalizar las operaciones se vuelven a grabar los cambios.

Procesamiento de transacciones:

Pre-image provee consistencia para archivos individuales. Btrieve usa un archivo de control de transacciones para tener la cuenta de los archivos involucrados en una transacción. Este archivo se crea al inicio del sistema (cuando se crea el Record Manager).

Cuando se reinicia el sistema luego de una caída (se reinicia el Record Manager), el usuario debe especificar la misma locación del archivo de control de transacciones, para asegurar una recuperación exitosa.

Btrieve usa el archivo de pre-image para implementar integridad de transacciones. La única diferencia entre operación normal y procesamiento de transacciones es que el archivo de pre-image continúa creciendo mientras dura la transacción, no sólo durante una operación. Entonces, el archivo de pre-image requiere gran cantidad de espacio de disco cuando se usan transacciones en un procedimiento.

Performance:

Se mejora el rendimiento del sistema deshabilitando los procedimientos de recuperación automática de datos (técnicas de pre-imaging y procesamiento de transacciones) (SIC Btrieve – Manual de Referencia Cap.2 "Performance Optimizations").

Acceso acelerado: opción que permite Btrieve de no grabar ningún cambio hasta que no se llenen los buffers.

Acceso restringido a archivos:

Provee restricciones de acceso para garantizar la seguridad, permitiendo asignar nombres de usuarios.

Control de concurrencia:

Provee tres métodos diferentes para control de concurrencia:

1. Control de transacciones: bloqueo de archivos mientras dura la transacción.

2. Método pasivo: no bloquea archivos, pero si una estación modifica un dato (o lo borra) cuando otra lo modificó, Btrieve devuelve un estado de conflicto, por lo que la última estación que modificó el dato, debe rehacer su operación desde el momento anterior a la modificación. Este método permite acceso concurrente más rápido.

3. Bloqueo de registros: una estación bloquea un registro de un archivo hasta que:
   a) se actualiza o borra el registro (la estación puede leer o modificar otros registros sin liberar ese bloqueo)
   b) la estación bloquea otro registro del archivo
   c) la estación libera el bloqueo
   d) la estación cierra el archivo
   e) la estación resetea el sistema y cierra todos los archivos abiertos.

Btrieve admite los siguientes tipos de bloqueos:

1. Bloqueo de espera: la transacción espera que se libere el bloqueo que le impide acceder al archivo o registro.

2. Bloqueo de no espera: si el archivo o registro al que desea acceder una transacción está bloqueado, esta no espera, retorna a la aplicación con estado de ocupado (busy).

6 - ALTERNATIVA ELEGIDA PARA LA RECUPERACIÓN DEL SISTEMA

Por las desventajas que presenta la técnica de "doble paginación" descripta:

- fragmentación de los datos
- algoritmos de recolección de basura que insumen tiempo de ejecución
- necesidad de una bitácora auxiliar para la recuperación de un sistema que contemple acceso multi-usuario.

Ante algunas desventajas que presenta Btrieve:

- necesidad por parte del usuario de establecer el tamaño de páginas de datos, páginas totales del archivo, páginas de índice
- necesidad de determinar espacio físico de páginas y espacio lógico de registros cuando se crea un archivo
- la técnica utilizada para el procesamiento de transacciones insume gran cantidad de espacio de disco
- requiere un algoritmo de recuperación de páginas basura que insume tiempo

Teniendo en cuenta que se permitirá la ejecución de transacciones concurrentes, es que se eligió la técnica basada en "bitácora":

- con modificación diferida de la base de datos,
- con puntuales excepciones de modificación inmediata de la base de datos, como se analizará en el punto 6.1;

En lo concerniente a la Unidad misma:

- el usuario no debe preocuparse por la creación de bases transitoria, ya que la unidad realiza esta tarea internamente, cuando el usuario involucra en su aplicación a una base de datos
- el usuario no debe determinar tamaño de páginas, ni de registros, ni totales del archivo creado.
- el usuario sólo debe diseñar sus bases de datos y utilizar la unidad desarrollada, en sus aplicaciones, si desea garantizar la consistencia de las mismas.

- ante una caída del sistema y luego de su reinicio, la unidad misma contempla mecanismos de recuperación que se ejecutarán sin ninguna previsión que deba realizar el usuario antes de correr su aplicación y durante el uso de la Unidad.

Asumiendo las siguientes desventajas:

- el tiempo que consume el algoritmo de recuperación ante caídas y
- esta técnica insume mayor cantidad de accesos a disco que la de doble paginación.

Se cree que es conveniente correr con estas desventajas y no con las que presenta la otra técnica que, a entender y según los análisis realizados, son mayores.

Además teniendo en cuenta que Btrieve, como se mencionó anteriormente, utiliza el método de doble paginación para garantizar la consistencia de las bases de datos, es que se decidió implementar un nuevo producto utilizando una nueva técnica.

7 - CONSIST: UNIDAD DE ADMINISTRACION DE BASES DE DATOS DEL PARADOX - ENGINE

7.1 - PRESENTACION DE LA UNIDAD

Para incorporar al Paradox - Engine el manejo de transacciones en sistemas mono o multi-usuario, se desarrolló una Unidad que, asociada a las ya existentes permiten programar aplicaciones asegurando unicidad de las transacciones y, por lo tanto, consistencia de las bases de datos involucradas.

La unidad fue diseñada de acuerdo a las clases existentes en el ambiente, modificando algunos métodos e incorporando nuevos.

Las modificaciones realizadas a los métodos existentes tienen que ver con cuestiones específicas de la unidad, a saber:

- Como ya se ha visto, una transacción debe conservar su atomicidad, por lo que debe asegurarse que se graben en la base de datos todos los cambios efectuados por ella o que no se grabe ninguno.

- Para garantizar la consistencia de las bases de datos, se usa una tabla transitoria asociada a cada base abierta, que tiene vigencia mientras esté en uso su base definitiva asociada, por lo tanto se modificaron los métodos de apertura, cierre, creación, borrado, renombrado y vaciado de bases de datos, teniendo en cuenta las nuevas tablas asociadas incorporadas.

- Los registros que usa una transacción no son actualizados a la base de datos, hasta tanto no terminen las aplicaciones o programas que involucran una o más transacciones, por lo cual hubo que modificar los métodos de actualización teniendo en cuenta el almacenamiento en las tablas transitorias y en la bitácora.

Los métodos incorporados tienen que ver con el manejo de la bitácora:

- Se desarrolló una nueva clase TCurBitac, en la cual se agruparon los métodos de clase para el manejo de bitácora, para solucionar el problema de la recuperación ante fallos del sistema o caídas suaves y la administración de transacciones

- Para el problema específico de la administración de transacciones, se implentó una bitácora en la cual se almacena todo el trabajo realizado por las aplicaciones activas, por lo que hubo que incorporar todos los métodos concernientes al manejo de la bitácora, tales como:
    - Obtener registros de la base de datos ya sea para sistemas mono o multi-usuario.
    - Guardar la información relacionada con una transacción ejecutada, para sistemas mono o multi-usuario.
    - Verificar la existencia de registros en tablas definitivas y transitorias.
    - Activar y desactivar la bitácora.
    - Limpiar la bitácora.

- Para el problema específico de recuperación ante fallos del sistema, se implementaron métodos de clase que permiten, una vez recuperado el sistema, restaurar las bases de datos a un estado consistente previo. Estos métodos se relacionan con revisar la bitácora a partir del último punto de verificación, descartar las transacciones no cometidas y guardar los cambios efectuados por las transacciones que terminaron exitosamente.

- Para el problema de la recuperación ante fallos en los medios de almacenamiento se provee de métodos que verifican el estado de la bitácora y, si ésta no fue afectada por el fallo, habilitan la recuperación del sistema a partir de una copia de respaldo y la restauración de las últimas actualizaciones almacenadas en bitácora, de las transacciones terminadas exitosamente al momento de la falla.

- Cabe aclarar que para hacer uso de la Unidad desarrollada para el procesamiento de transacciones hay que aumentar el número de buffers y manejadores de tablas provistas por defecto por el sistema de bases de datos.

7.2 - FUNCIONAMIENTO DE LA UNIDAD

La nueva Unidad desarrollada se compone de varias clases que se describirán a continuación. Cabe aclarar que ésta es una unidad más del conjunto de unidades que provee el Paradox Engine y su uso es opcional.

Análisis de las clases que componen la Unidad:
Existe una nueva clase llamada TBaseDatos que hereda las propiedades y métodos de TDataBase (clase ya existente en el Paradox Engine). Esta nueva clase involucra todos los métodos de manejo de tablas. Por cada tabla que se activa, la Unidad crea una tabla asociada que contendrá los registros modificados durante la ejecución de transacciones. Las tablas transitorias tienen vigencia mientras existe la tabla principal, desde que se crea o abre dicha tabla hasta que se borra o destruye la misma.

Entonces, cada vez que se inicializa una tabla, se hace lo mismo con su tabla transitoria asociada. Lo mismo sucede cuando se cierra una tabla; también se cierra su tabla transitoria asociada.

Si el usuario desea abrir una tabla existente, la Unidad, a través de los métodos implementados, abre la tabla asociada. Si falla la apertura de alguna de ellas, la Unidad no permite que ninguna quede abierta; simplemente no realiza la operación sobre ninguna de ellas.
Si existe una tabla que no contenga una tabla transitoria asociada (por ejemplo una tabla creada sin usar la Unidad, o una tabla importada desde otro Administrador), la Unidad contempla un método que permite la creación de la misma, a partir de la estructura de la tabla principal.

Cuando el usuario crea una tabla, la Unidad, a través de sus métodos, crea ambas: la tabla principal y la tabla transitoria, según la estructura definida.

El único recaudo que debe hacer el usuario al crear una tabla es no colocar como primer carácter del nombre de la misma al símbolo numeral (#), ya que la Unidad asignará como nombre a la tabla transitoria asociada, el mismo nombre que el de la tabla principal, reemplazando el primer carácter por el símbolo numeral (#).

Si el usuario desea cambiar el nombre de una tabla, la Unidad también cambia el nombre de su tabla transitoria asociada, siguiendo la regla que el primer carácter del nombre de la tabla transitoria es el símbolo (#).

Para borrar una tabla, la Unidad verifica si la misma está en uso y si quedan actualizaciones pendientes en bitácora. Si esto sucede, no se cierra la tabla y en su lugar se emite un mensaje de error. Al cerrar una tabla también se cierra su tabla transitoria asociada.

Para garantizar la atomicidad de las transacciones no se permiten escrituras forzadas a disco. Sí se habilita un nuevo método de escritura forzada que tiene por objeto salvar las transacciones pendientes de la tabla asociada al manejador.

Para armar registros de bitácora se desarrolló una nueva clase llamada Treg, que hereda los métodos y propiedades definidos en la clase Trecord del Paradox Engine. Aquí además se implementaron métodos para verificar igualdad entre registros con la misma estructura y testear el salvado de registros actualizados.

El manejo de punteros a las tablas, tanto principales como tablas transitorias, se hace a través de los métodos implementados en la nueva clase Tmanejcursor, que hereda las propiedades de la clase Tcursor del Paradox Engine. Aquí se implementaron métodos que permiten inicializar, abrir, anular y cerrar cursores a una tabla, listar los registros de una tabla, insertar un registro en la posición del cursor, añadir un registro al final de una tabla, borrar o modificar el registro al que apunta un cursor, encontrar un registro igual a un registro dado como parámetro, posicionando el cursor en el registro hallado.

Esta clase además implementa tres métodos para garantizar la atomicidad de las transacciones ejecutadas. Estos son INICIO y ÚLTIMO o ÚLTIMO INMEDIATO. Todas las operaciones que se realicen entre INICIO y ÚLTIMO o ÚLTIMO INMEDIATO se considerarán una unidad indivisible, ésto es, se realizan todas o no se realiza ninguna. El funcionamiento y uso de estos métodos se verá a continuación, en el análisis del funcionamiento de la bitácora para el procesamiento de transacciones.

Análisis del funcionamiento de la bitácora para el procesamiento de transacciones:
A continuación se detalla el comportamiento de la bitácora para el procesamiento de transacciones.

Los conceptos de administración de transacciones, a través de la bitácora, sus propiedades y métodos fueron implementados en la clase TCurBitac.

Para preservar la consistencia de las bases de datos a través del procesamiento de transacciones se utiliza una tabla denominada bitácora, la misma tiene por objetivo mantener las operaciones que conforman una transacción hasta que las mismas sean salvadas definitivamente en las tablas originales.

A través de los registros almacenados en la bitácora se puede establecer si una transacción ha finalizado o si hubo un fallo durante el procesamiento de la misma. Además prevee la posibilidad de que se produzcan caídas del sistema durante la etapa de almacenamiento definitivo.

La bitácora tiene por objetivo guardar la secuencia de operaciones que constituyen una transacción y la marca de inicio y finalización de la misma.

Los registros de la bitácora tienen dos fines: indicar el comienzo y fin de una transacción y mantener la información de cada operación involucrada. Para lograr esto, cada registro tiene la siguiente estructura:

- TABLA
- NRO DE TRANSACCIÓN
- NREG
- OPERACION
- USUARIO
- SALVADO

TABLA: nombre de la tabla a la cual pertenece el registro. Si es un registro de inicio o de fin, guarda este dato (inicio o fin) aunque es un dato redundante.

NRO DE TRANSACCIÓN: es el número de la transacción que se está efectuando. Esto sólo se contabiliza en un sistema monousuario.

NREG: número de registro (posición) en la tabla transitoria.

OPERACION: nombre de la operación que se realiza. Puede ser INICIAL, FINAL, UNIR, BORRAR, etc.

USUARIO: nombre del usuario en la red. Permite identificar a qué transacción pertenece la operación que se realiza.

SALVADO: marca que indica si el registro ha sido salvado definitivamente o no. Su fin es preveer una caída durante el proceso de almacenamiento definitivo de una transacción finalizada exitosamente.

Proceso de trabajo con bitácora:
Toda transacción, compuesta de una o más operaciones, se encuentra acotada por dos registros, uno de inicio y uno de fin. Estos registros los coloca el sistema a través de las funciones INICIAL y ÚLTIMO o ÚLTIMO INMEDIATO que el usuario utiliza para englobar un conjunto de operaciones en un grupo indivisible.

Al encontrar la función INICIAL, guarda en bitácora un registro indicando el tipo de operación, nombre del usuario. A continuación incorpora el registro con las

operaciones siguientes, por ejemplo UNIR, INSERTAR, BORRAR, MODIFICAR, etc. Por último y al llegar a la función ÚLTIMO o ÚLTIMO INMEDIATO, agrega un registro en bitácora indicando final y nombre del usuario.

ÚLTIMO difiere de ÚLTIMO INMEDIATO en que el primero efectiviza el salvado de la transacción al terminar de trabajar el sistema, en cambio ÚLTIMO INMEDIATO lo hace en forma automática al llegar a él.

Las funciones de UNIR e INSERTAR actúan en forma semejante: guardan en la tabla transitoria el registro a ser unido o insertado y en la bitácora la posición de dicho registro en la tabla mencionada.

La función BORRAR guarda en la tabla transitoria el registro a ser borrado y en la bitácora su posición. Al efectuarse el salvado definitivo, localiza el registro en la tabla principal y lo borra. Es responsabilidad del usuario verificar la existencia del registro que se desea borrar.

La función MODIFICAR implica el almacenamiento de dos registros: uno con los datos viejos (operación modifVie) y otro con los datos nuevos (operación modifNue). En bitácora habrá dos registros, cada uno de ellos referenciando a cada operación. La verificación de la existencia del registro a modificar está a cargo del usuario.

Almacenamiento definitivo:
Para guardar los cambios realizados por una o más transacciones a la tabla, el sistema localiza en la bitácora el registro INICIAL de un usuario y verifica en la misma la existencia de las tablas involucradas, los registros en las tablas transitorias y el registro de FIN. Para realizar ésto recorre la bitácora buscando los registros del mismo usuario, testeando para cada uno de éstos la existencia de la tabla principal y de la tabla transitoria y que los registros referenciados en el campo NREG de la bitácora se encuentren en la tabla transitoria. De finalizar con éxito, se inicia el proceso de salvado definitivo, recorriendo nuevamente la bitácora, buscando los registros de un usuario particular y ejecutando la operación indicada en el mismo. Una vez que se concretó el salvado de cada operación, marca cada una de ellas como efectivizada. El

objetivo de esta marca es preveer una caída durante el proceso, que llevaría a duplicar ciertas operaciones una vez reiniciado el sistema. De esta forma y con el uso de la Unidad, al iniciarse nuevamente, el sistema testea si las transacciones residentes en bitácora han finalizado exitosamente y si las operaciones involucradas en cada transacción han sido salvadas. De no ser así, solamente almacenará en forma definitiva las operaciones marcadas como NO SALVADAS.

Una vez que el sistema ha salvado todas las transacciones existentes en la bitácora, pone una marca de VERIFICADO. De esta forma, en el próximo proceso de almacenado sólo considera las transacciones posteriores a dicha marca, evitando de esta forma, recorrer toda la bitácora nuevamente.

Una forma de mantener copias de respaldo actualizadas para poder recuperar al sistema de una falla en los medios de almacenamiento es realizar la misma luego de que termine de correr la aplicación.

Como el contenido de la bitácora no se borra a menos que se ejecute un procedimiento externo para tal fin, con una copia de respaldo realizada con el criterio mencionado y con los registros de bitácora, desde el penúltimo punto de verificación hasta el momento del fallo, se puede recuperar al sistema exitosamente con todas las actualizaciones realizadas residentes en bitácora.

Una vez reparado el fallo, los pasos a seguir serán:

- restaurar el sistema y la base de datos a partir de una copia de respaldo
- verificar el estado de la bitácora y de las tablas transitorias asociadas

Si no fueron dañadas por la falla:

- iniciar el proceso de recuperación basado en bitácora, de la base de datos restaurada, a partir del penúltimo punto de verificación.

Análisis de algunos procedimientos:

IniSis y FinSis: inicia y finaliza el sistema.

Inicio: activa la bitácora y testea en la misma la existencia de transacciones completas no efectivizadas en las tablas originales. De ser así las efectiviza y coloca un marca de SALVADO.

Fin: guarda las operaciones completas no efectivizadas en las tablas correspondientes y coloca una marca de SALVADO.

Limpieza: es un procedimiento externo a IniSis y FinSis que vacía la bitácora y las transacciones referenciadas en ella. Es un procedimiento adicional que está disponible para que el usuario lo ejecute cuando considere necesario.

Transformación de tablas: adapta las tablas no creadas a través de la operación específica que provee la Unidad, para que puedan ser usadas por el sistema. Cabe aclarar que el método de creación de tablas de la Unidad agrega una tabla transitoria, según lo explicado anteriormente. Si una tabla fue creada mediante otro sistema entonces no tiene una tabla transitoria asociada, la cual es necesaria para operar con la Unidad.

7.3 - DOCUMENTACION DE LA UNIDAD

Para entender las operaciones contempladas en la Unidad desarrollada, se exponen las clases y métodos diseñados:

TBaseDatos = Object(TDatabase)
private
dbt:PDatabase;
La clase TBASEDATOS hereda de TDATABASE. Permite manejar las tablas principales y las tablas asociadas, necesarias para guardar transitoriamente los

registros, hasta que sean definitivamente almacenados.
dbt es el puntero a la tabla transitoria, cuyo nombre se forma con el nombre original de la tabla, reemplazando la inicial por #. Por este motivo no deben crearse tablas cuya inicial sea el carácter numeral (#).
Los métodos siguientes reemplazan a los métodos predefinidos de la clase TDataBase.

constructor Init(eng: Pengine);
Constructor; inicializa la tabla principal y la tabla transitoria asociada.

destructor Done; virtual;
Destructor; si la tabla principal está abierta, la cierra. Lo mismo sucede con la tabla transitoria.

function abrir: Retcode; virtual;
Abre la base de datos y su tabla transitoria asociada. En caso de que falle la apertura de la última, cierra ambas.

function cerrar: Retcode; virtual;
Cierra la base de datos y la tabla transitoria asociada.

function crearTabla(tableName:String; desc:PTableDesc): Retcode; virtual;
Crea una tabla a partir de su descriptor y la tabla asociada. En caso de fallar la creación de alguna de ellas, anula ambas.

function vaciarTabla(Name: String): Retcode; virtual;
Borra el contenido de una tabla y de su familia asociada. Borra el contenido de la tabla transitoria asociada. Las tablas no deben estar en uso.

function renombrarTabla(oldName, newName: String): Retcode; virtual;
Renombra una tabla y su familia. Renombra la tabla transitoria. La tabla a ser renombrada no debe estar en uso, ni tener registros pendientes en la bitácora, de ser así, resultará error.

Antes de renombrar se testea que no se repitan los nombres.

function borrarTabla(Name: String): Retcode; virtual;

Borra una tabla y su familia. Borra la tabla transitoria. La tabla a ser borrada no debe estar en uso, ni tener registros pendientes en bitácora, de ser así, resultará error.

function forzarEscritura(eng:Pengine): Retcode; virtual;

Permite salvar las transacciones pendientes de la tabla asociada al manejador.

function forceWrite: Retcode; virtual;

Anula la función que fuerza la escritura en disco. Se reemplaza por otra función que permite en cualquier momento grabar en forma definitiva el contenido de la bitácora.Se deja el nombre original para que en caso de ser invocada, se evite su acción.

TReg=Object(TRecord)

Incorpora nuevos métodos a la clase Trecord.

function ArmaReg(tableName: String; op: String; n:recordNumber; eng: Pengine): Retcode; virtual;

Arma el registro que será incorporado en la bitácora.

function Iguales(rec: Preg): boolean; virtual;

Indica si el contenido de ambos registros es igual. Da por cierto que ambos registros tienen la misma estructura.

TManejCursor=Object(TCursor)

Tmanejcursor encapsula los conceptos de manejo de los registros en las bases de datos Paradox. Hereda las características de Tcursor y redefine aquellos métodos necesarios para preservar la consistencia de las bases.

constructor Init;

Constructor. Crea un objeto TManejCursor sin abrir una tabla Paradox. Crea un manejador para la tabla transitoria.

constructor InitAndOpen(db:PBaseDatos; tableName:String; indexID: FieldNumber);

Constructor. Hace un objeto TManejCursor y abre la tabla Paradox especificada.

destructor Done: virtual;

Destructor. Destruye el cursor si está abierto.

function open(db:PBaseDatos;tableName:String;indexID:FieldNumber): Retcode; virtual;

Abre un cursor sobre una tabla especificada.

function close: Retcode; virtual;

Cierra el cursor si está abierto.

function Inicio(eng: Pengine): Retcode; virtual;

Crea el primer registro de un grupo indivisible.

function Ultimo(eng: Pengine): Retcode; virtual;

Crea el último registro de un grupo indivisible.

function UltimoInmediato(eng: Pengine): Retcode; virtual;

Crea el último registro de un grupo indivisible e inicia el proceso de salvado definitivo. Se utiliza cuando hay registros bloqueados que pueden ser necesitados por otras transacciones.

function unirRec(rec: Precord; eng: Pengine): Retcode; virtual;

Une un registro al final de la tabla especificada. El registro puede ser del tipo Custom (registro de trabajo) o Generic (registro genérico).

function insertarRec(rec: PRecord;eng:PEngine): Retcode; virtual;

Inserta un registro en la posición actual del cursor. El registro puede ser ser del tipo Custom (registro de trabajo) o Generic (registro genérico)

function borrarRec(eng:PEngine): Retcode; virtual;

Borra el registro en la posición actual del cursor.

function modificarRec(rec: PRecord;eng:PEngine): Retcode; virtual;

Modifica el registro en posición actual del cursor. El registro puede ser ser del tipo Custom (registro de trabajo) o Generic (registro genérico)

function EncuentraIguales(reg:PReg):Retcode; virtual;

Posiciona el cursor en un registro igual al dado como parametro,devolviendo PXSUCCESS o fin de tabla en caso de no haber registros iguales

**function listarTable(tableName: String;var cant:RecordNumber): Retcode;
virtual;**

Lista una tabla.

function guardarec(rec:PRecord): Retcode;virtual;

Guarda un objeto registro

TCurBitac=Object(TCursor)

TCurBitac es una nueva clase que encapsula los conceptos de administración de transacciones sobre las tablas del Paradox, a través de la bitácora.

function ObtenerRec(i:integer;Bitareg:PReg): Retcode;virtual;

Permite obtener el siguiente registro de una transacción realizada en sistema monousuario.

function ObtenerRecNet(us:string;Bitareg:PReg): Retcode;virtual;

Permite obtener el siguiente registro de una transacción realizada en sistema multiusuario.

**function operar(nom: string; rec:PReg; op:string; us:string; nt:integer;
cur:PManejCur;eng:PEngine): Retcode;virtual;**

Realiza el guardado definitivo del registro con manejador Rec.

**function GuardarTrans(nt: integer; pos: RecordNumber;eng:PEngine):
Retcode;virtual;**

Guarda transitoriamente una transacción en bitácora y en las tablas transitorias. Se utiliza en sistemas monousuario.

**function GuardarTransNet(us:string; pos: RecordNumber;eng:PEngine):
Retcode;virtual;**

Guarda transitoriamente una transacción en bitácora y en las tablas transitorias. Se utiliza en sistemas multiusuario.

**function verifica(i:integer; pos:RecordNumber;eng:Pengine):
boolean;virtual;**

Verifica la existencia de los registros y las bases principales y finales en sistemas monousuario.

**function verificaNet(us:string; pos:RecordNumber;eng:Pengine):
boolean;virtual;**

Verifica la existencia de los registros y las bases principales y finales en sistemas multiusuario.

function salvar(eng:PEngine): Retcode;virtual;

Actualiza las tablas definitivas con los registros de las tablas transitorias, utilizando la informacion de la bitacora. Cuando finaliza, coloca una marca. Se utiliza en sistemas monousuario.

function salvarNet(eng:PEngine): Retcode;virtual;

Actualiza las tablas definitivas con los registros de las tablas transitorias, utilizando la informacion de la bitacora. Cuando finaliza, coloca una marca. Se utiliza en sistemas multiusuario.

function verificaSalvado(nt:integer; pos:RecordNumber;eng:Pengine): boolean;virtual;

Verifica al iniciarse el sistema, que las transacciones existentes en la bitácora, posteriores a la última marca de salvado, se encuentren guardadas definitivamente. Este método prevee posibles interrupciones del sistema durante el proceso de salvado, situación que puede generar que algunos registros de una transacción hayan sido grabados definitivamente y otros no. Se utiliza en sistemas monousuario.

function verificaSalvadoNet(us:string; pos:RecordNumber;eng:Pengine): boolean;virtual;

Verifica al iniciarse el sistema, que las transacciones existentes en la bitácora, posteriores a la última marca de salvado, se encuentren guardadas definitivamente. Este método prevee posibles interrupciones del sistema durante el proceso de salvado, situación que puede generar que algunos registros de una transacción hayan sido grabados definitivamente y otros no. Se utiliza en sistemas multiusuario.

function GuardarTransIni(nt:integer; pos: RecordNumber;eng:PEngine): Retcode;virtual;

Guarda transitoriamente una transacción en la bitácora y en las tablas transitorias asociadas, al inicio del sistema. Se utiliza en sistemas monousuario.

function GuardarTransNetIni(us:string; pos:RecordNumber; eng:PEngine): Retcode;virtual;

Guarda transitoriamente una transacción en la bitácora y en las tablas transitorias asociadas, al inicio del sistema. Se utiliza en sistemas multiusuario.

function revisar(eng:PEngine): Retcode;virtual;

Se invoca al inicio del sistema y descarta aquellos registros de la bitacora posteriores a la marca de salvado, presumiendo que son operaciones incompletas.

function marcarFin(eng:PEngine): Retcode; virtual;

Los siguientes procedimientos activan y desactivan la bitacora:

function IniSis(eng:PEngine): Retcode;

Inicia el sistema activando la bitácora.

function FinSis(eng:Pengine): Retcode;

Cierra el sistema iniciando el salvado de las transacciones guardadas en bitácora y desactivando la misma.

function ModificarTabla(db:PBaseDatos; nom: String): Retcode;

Modifica la base de PARADOX no creadas a través de la función específica de la unidad, poder ser usada por la misma. Esta función debe usarse una sóla vez.
Una vez creada debe abrirse la base. Los mensajes de error corresponden a los de createTable y PXERR_TRANSEXISTE, indicando que ya existe una transitoria con ese nombre.

function Limpiar(eng:Pengine): Retcode;

Limpia de la bitácora los registros de las transacciones ya salvadas. Vacía las bases transitorias de las bases involucradas en operaciones almacenadas en la bitácora. No vacía aquellas bases transitorias no involucradas. Debe ejecutarse antes de IniSis o después de finSis.

function Recuperar(eng:Pengine): Retcode;

Verifica si la bitácora y las tablas transitorias fueron afectadas por un fallo en los medios de almacenamiento. Si no fue así, invoca a la función Salvar. Debe usarse antes de IniSis y después de FinSis. Previamente el usuario debe restaurar el sistema y las tablas principales desde una copia de respaldo.

8 - CASOS DE PRUEBA

El programa de prueba de la unidad debe contener actividades críticas que sometan a la base de datos a un análisis de consistencia. Para probar el funcionamiento de la unidad se implementaron en esta aplicación operaciones tales como agregar un registro a una o más tablas, modificar uno o varios registros en una tabla y agregar registros en otra, borrar registros de una tabla y modificar registros en otra. El resultado esperado coincidió con el resultado obtenido en cuanto a que las actualizaciones a la base de datos fueron correctas (se agregaron los registros que correspondía añadir, con sus valores de campos tal como se especificó, se borraron sólo los registros que se indicaron y se modificaron los registros involucrados en esta operación, reemplazando en éstos los datos especificados para modificar; todas las operaciones se realizaron en tiempos de ejecución aceptables). La aplicación sólo contempla la ejecución de las funciones implementadas en la unidad, ya que sólo interesa comprobar el funcionamiento de las mismas, no se hizo énfasis en el diseño óptimo del sistema ya que esa es una cuestión del programador de la aplicación y no aporta demasiado a la prueba del funcionamiento de los métodos de la unidad.

Además, se provocó una caída en el sistema y se observó el comportamiento los métodos de recuperación implementados, basados en "bitácora". La recuperación fue exitosa ya que, una vez reiniciado el sistema se actualizó la base de datos sólo con los resultados de las transacciones que habían terminado exitosamente (almacenadas en la bitácora) entre el último punto de verificación (también almacenado en la bitácora) y el momento en que se produjo la caída.

8.1 - PROGRAMA DE PRUEBA

Presentación de la aplicación:

Para probar el funcionamiento de la unidad se desarrolló una aplicación, utilizando lenguaje Pascal 7.0 Orientado a Objetos, bajo entorno Windows, que permite acceso concurrente a través de diferentes usuarios. Las tablas respetan la estructura del Paradox - Engine.

Funcionamiento de la aplicación:

La actividad que se describe en este programa de prueba tiene que ver con la ejecución de obras en la vía pública (pavimentos, reparación de calzadas y veredas, ejecución de obras hidráulicas) para lo cual se considera:

- Una tabla de CÓMPUTO de obra en la cual figuran, bajo un mismo código de obra, los ítems a ejecutar para realizar una tarea predeterminada con cantidades estimadas de acuerdo al volumen del trabajo requerido.

- Una tabla de MEDICIÓN de obra, en la cual figura, bajo un mismo código de certificado (número bajo el cual se agrupan los trabajos realizados a pagar), la cantidad ejecutada de los ítems, en un periodo de tiempo dado, de las obras presupuestadas y almacenadas en la base de datos descripta en el punto anterior.

Las tareas que se realizan son:

- Almacenar, una vez presupuestadas, las obras con sus respectivos ítems en la tabla del COMPUTO de obra. Una obra se identifica con un único código de obra.

- Almacenar, una vez ejecutados, bajo un único código de certificado, la cantidad ejecutada de los ítems correspondientes a las obras realizadas, en un periodo de tiempo dado.

- Actualizar, en la tabla del COMPUTO de obra, la cantidad "ejecución_acumulada", de los ítems ejecutados.

Nota: los ítems se identifican con un único código de ítem.

A continuación se describe la estructura de las tablas:

COMPUTO

NRO_ORDEN: código de obra
ITEM: código de ítem
CANT_COMP: estimado del ítem según un presupuesto efectuado para esa obra
EJEC_ACUM: acumulado del ítem según lo ejecutado, almacenado en uno o varios certificados.

MEDICION

NRO_CERT: código de certificado
NRO_ORDEN: código de obra
ITEM: código de ítem
EJECUCION: ejecutado del ítem para un certificado.

Las actividades que se implementaron en el programa, utilizando los métodos desarrollados en la unidad CONSI, son:

a) ALTA DE COMPUTO
b) ALTA DE MEDICION
c) BAJA DE MEDICION

a) Involucra la tabla COMPUTO, en la cual se da de alta a los registros de acuerdo a los datos ingresados por el usuario.

b) Involucra ambas tablas. En COMPUTO verifica la existencia del ítem a incorporar como ejecutado de una obra y actualiza el campo "EJEC_ACUM". En la tabla MEDICION se ejecuta el alta de los registros, de acuerdo a los datos ingresados por el usuario.

c) Involucra ambas tablas. En COMPUTO verifica la existencia del ítem a descartar como ejecutado de una obra y actualiza el campo "EJEC_ACUM". En la tabla

MEDICION se ejecuta la baja de los registros, de acuerdo a los datos ingresados por el usuario.

Resultado obtenido:

Luego de ejecutar el programa de prueba, bajo el entorno Windows, desde varias aplicaciones, y de someter al sistema a una caída suave se observó:

- Los métodos de la Unidad funcionan correctamente, de acuerdo a lo detallado en el punto 6.2 : "Análisis del funcionamiento de la bitácora para el procesamiento de transacciones". Al observar el estado de la bitácora y de las tablas transitorias se comprobó que los resultados de las operaciones fueron almacenados en ellas y una vez que finalizó la ejecución del sistema, se actualizó la base de datos con los métodos de actualización que recorren la bitácora, que, para cada usuario y cada operación, localizan el registro correspondiente en la tabla transitoria y actualizan el registro en la tabla principal.

- El control de acceso multi-usuario fue satisfactorio. Al observar el estado de la bitácora se comprobó que se almacenaron en ella los registros actualizados (ya sea por operaciones de unión, borrado o modificación) con el nombre identificatorio correcto especificado por cada usuario, lo que permite una correcta actualización cuando se guardan los cambios en forma definitiva.

- La actualización de las tablas se realizó correctamente, garantizando la consistencia de la base de datos.

- La recuperación ante fallos del sistema fue satisfactoria, de acuerdo a lo esperado como resultados de los métodos de recuperación implementados en la Unidad. Al someter al sistema a una caída suave y comprobar luego de ésta el estado de la bitácora, al reiniciar se actualizaron sólo las transacciones comprometidas, de acuerdo a lo observado previamente.

8.2 - CASOS NO COMPARATIVOS

Se incluyen en este apartado diferentes pruebas realizadas a la Unidad, sin ser necesariamente parte de una aplicación particular, que reflejan el comportamiento de la misma en cuanto a tiempos de ejecución.

En todos los casos, los tiempos medidos se refieren al recorrido de la bitácora y, si es posible, al almacenamiento en tabla principal.

Los tiempos de ejecución fueron medidos en centécimas de segundo y las pruebas corrieron en una computadora del tipo Pentium 133 con 32 MB de memoria RAM.

Luego de realizar varias pruebas se observó que el tiempo de ejecución no varía si se trata de una transacción que involucra a una o más tablas. Se registraron los siguientes tiempos:

Tiempo de ejecución de una transacción (con una operación) que aborta su ejecución:

OPERACIÓN	TIEMPO DE VERIFICACION DE BITÁCORA / cent.segundos
UNIR	5 - 6
MODIFICAR	5 - 6
BORRAR	5 - 6

Tiempo de ejecución de una transacción (con una operación) que termina bien:

OPERACIÓN	TIEMPO DE VERIFICACION DE BITÁCORA / cent.segundos
UNIR	11
MODIFICAR	16
BORRAR	11

Tiempo de ejecución de una transacción (con dos operaciones) que terminan bien y actúa sobre una tabla:

SECUENCIA DE OPERACIONES	TIEMPO DE VERIFICACIÓN DE BITACORA / cent.segundos
UNIR/UNIR	17
UNIR/MODIFICAR	22
UNIR/BORRAR	17
MODIFICAR/UNIR	22
MODIFICAR/MODIFICAR	22
MODIFICAR/BORRAR	22
BORRAR/UNIR	16
BORRAR/MODIFICAR	16
BORRAR/BORRAR	17

Tiempo de ejecución de n transacciones, en modo monousuario, que actúan sobre varias tablas. Algunas transacciones terminan bien y otras abortan su ejecución.

SECUENCIA DE OPERACIONES	TIEMPO DE VERIFICACION (c.s.)
ALEATORIA	44

Tiempo de ejecución de n transacciones, en modo multiusuario, que actúan sobre varias tablas. Algunas transacciones terminan bien y otras abortan su ejecución.

SECUENCIA DE OPERACIONES	TIEMPO DE VERIFICACION (c.s.)
ALEATORIA	71

8.3 - CASOS COMPARATIVOS

Se incluyen en este apartado diferentes pruebas realizadas a la Unidad, ejecutando una o más transacciones, en sistemas mono o multiusuario, considerando terminaciones exitosas o no exitosas de las transacciones; que reflejan el comportamiento de la misma en cuanto a tiempos de ejecución, comparando resultados obtenidos como se detalla en los cuadros.

En todos los casos se contemplaron los tiempos de recorrido de la bitácora y su posterior almacenamiento en las tablas principales (cuando esto fuera posible). En el caso de la comparación sin el uso de la Unidad se contempló bajadas a disco para el almacenamiento de datos.

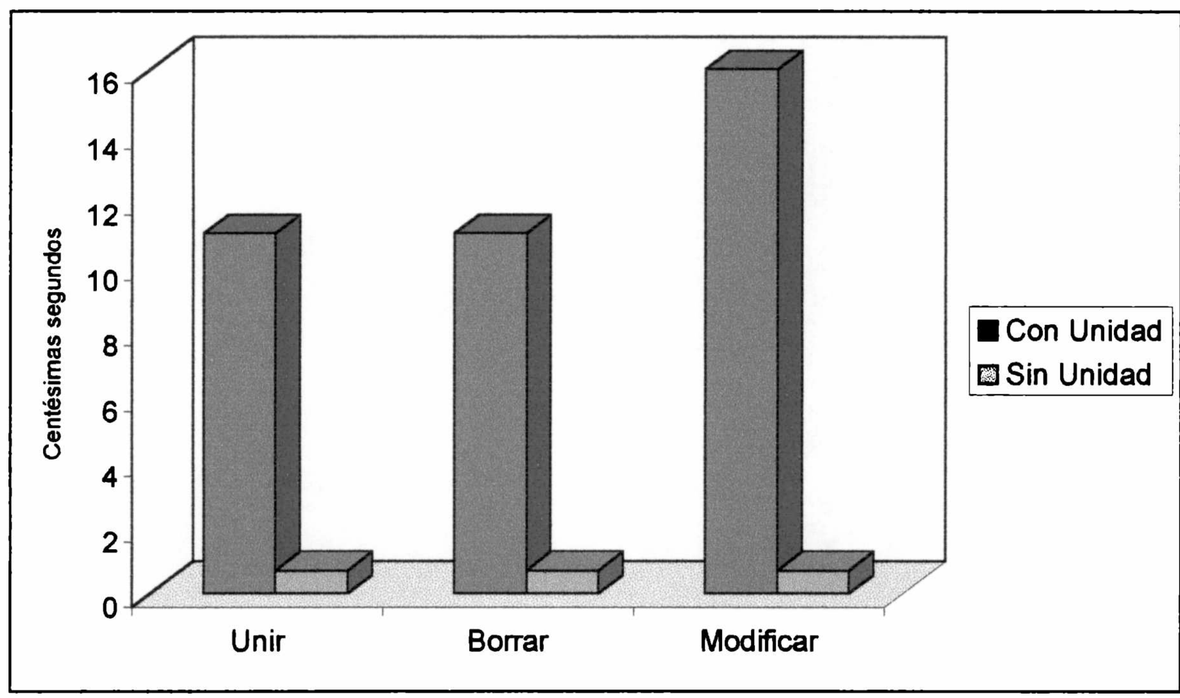
Los tiempos de ejecución fueron medidos en centécimas de segundo y las pruebas corrieron en una computadora del tipo Pentium 133 con 32 MB de memoria RAM.

En estos casos no se observaron tiempos diferentes cuando se almacenan en la bitácora los registros que involucran a una transacción todos juntos o intercalados con los registros que involucran a otras transacciones, tampoco se observaron diferencias de tiempos considerables si las transacciones actúan sobre una o más tablas.

Cabe aclarar que no es posible medir los tiempos de ejecución de operaciones cuando éstas abortan, si no se usó la Unidad desarrollada. Los casos comparables se reducen, entonces, a mediciones de tiempos de ejecución de transacciones con terminación exitosa, con y sin el uso de la Unidad.

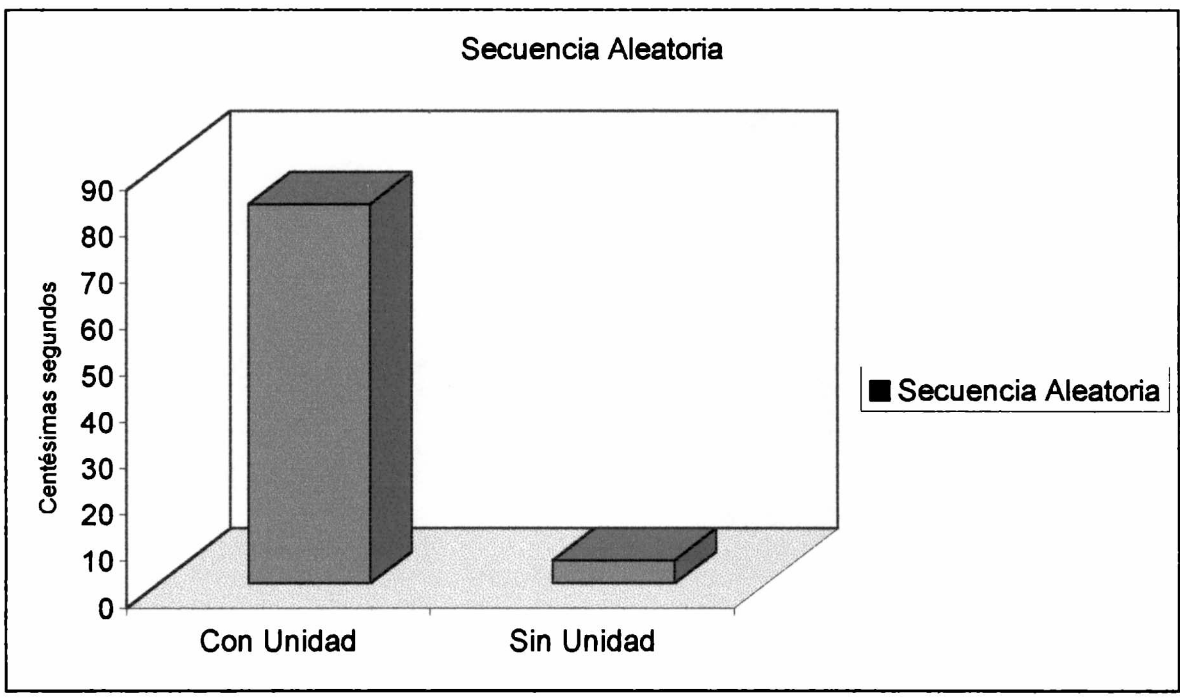
Comparación entre tiempos de ejecución de una transacción:

OPERACIÓN	TIEMPO DE VERIFICACION CON UNIDAD (c.s.)	TIEMPO DE VERIFICACION SIN UNIDAD (c.s.)
UNIR	11	0.7
BORRAR	11	0.7
MODIFICAR	16	0.7



Comparación de tiempos de ejecución de n transacciones en modo monousuario: la secuencia de operaciones seguida es la misma para ambos casos y se trata de la combinación aleatoria de UNIR, BORRAR y MODIFICAR.

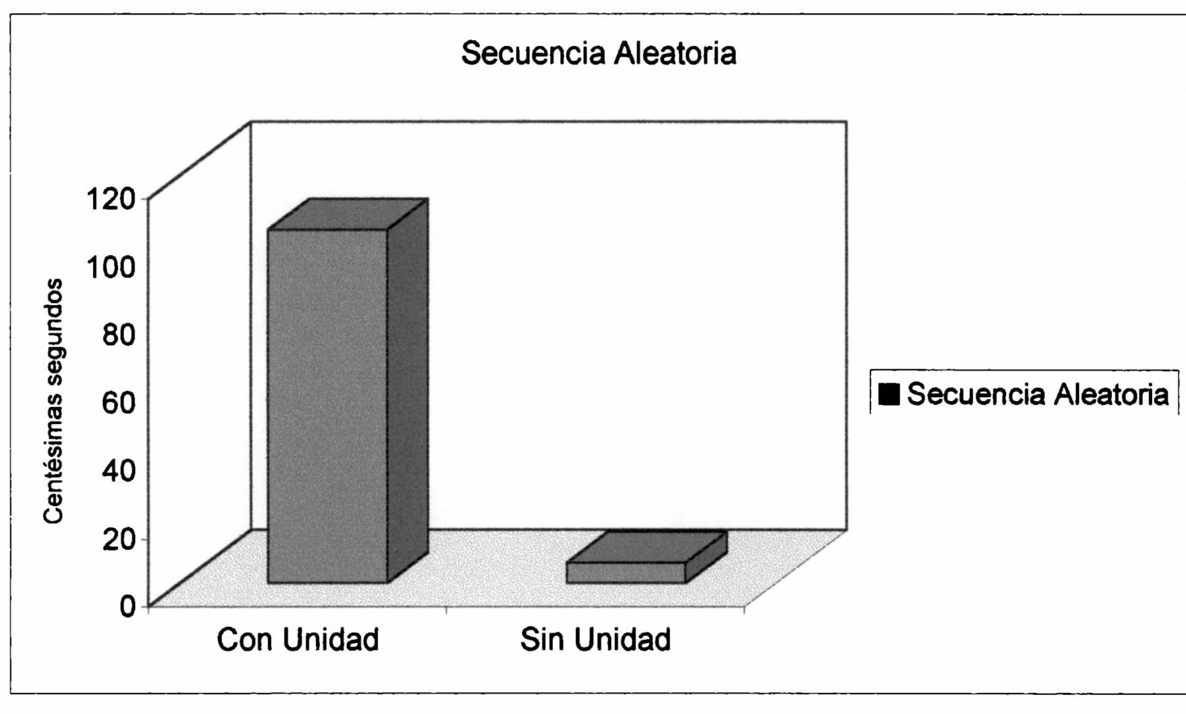
SECUENCIA DE TRANSACCIONES	VERIFICACION (c.s.) CON UNIDAD	VERIFICACION (c.s.) SIN UNIDAD
ALEATORIA	82	5



Luego de comparar los tiempos que demanda la ejecución de la secuencia aleatoria de n operaciones, con y sin el uso de la unidad, y para distintos valores de n, se concluye que se mantiene la relación expresada en el gráfico.

Comparación de tiempos de ejecución de n transacciones en modo multiusuario: la secuencia de operaciones seguida es la misma para ambos casos y se trata de la combinación aleatoria de UNIR, BORRAR y MODIFICAR.

SECUENCIA DE TRANSACCIONES	VERIFICACION (c.s.) CON UNIDAD	VERIFICACION (c.s.) SIN UNIDAD
ALEATORIA	104	6



Luego de comparar los tiempos que demanda la ejecución de la secuencia aleatoria de n operaciones, con y sin el uso de la unidad, y para distintos valores de n, se concluye que se mantiene la relación expresada en el gráfico.

8.4 - ANÁLISIS DE LA FUNCION "BITÁCORA"

Para determinar qué forma toma el procesamiento de transacciones con el uso de la Unidad se observó el comportamiento de la recuperación basada en bitácora, teniendo en cuenta dos parámetros: cantidad de accesos a recursos y tiempo de ejecución.

Clasificación de acuerdo a la cantidad de accesos a recursos:
Se analizará a través del siguiente cuadro, la cantidad de accesos a los diferentes recursos, que se realizan al ejecutar cada operación conflictiva:

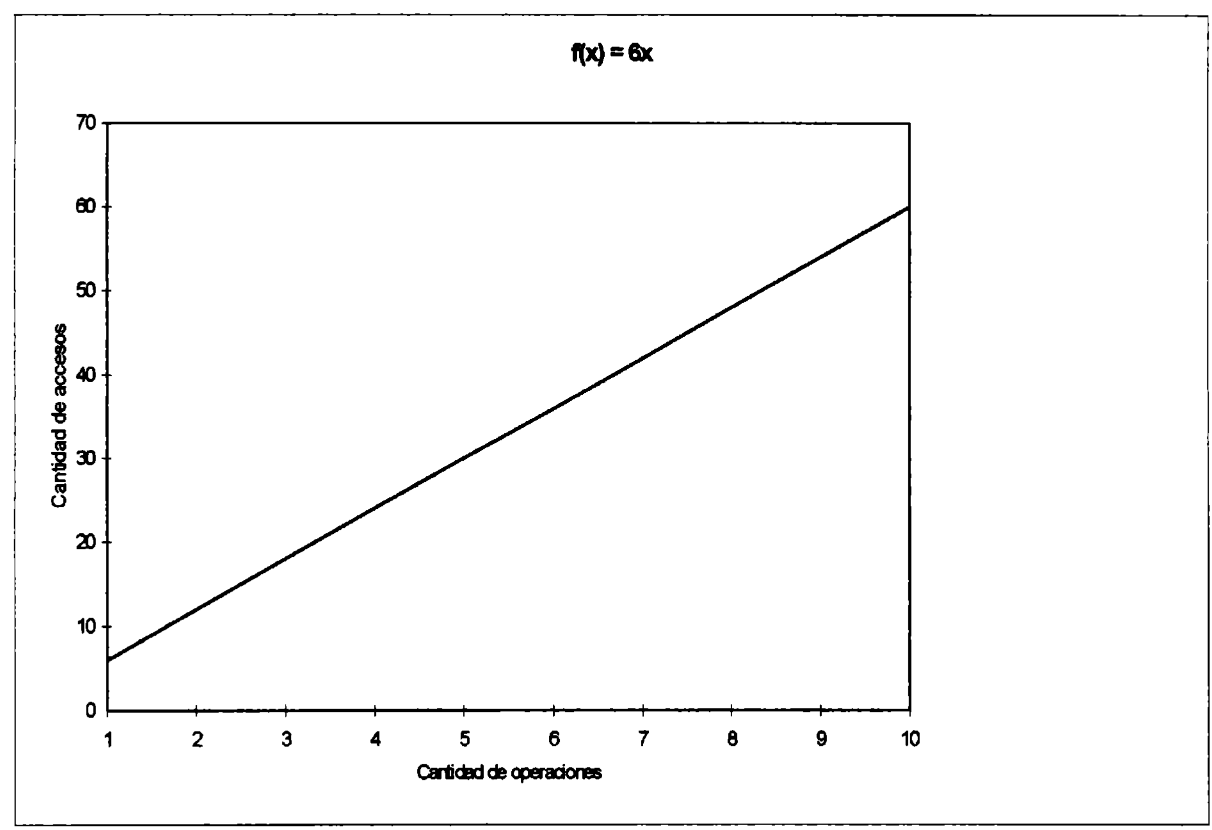
OPERACION	CANTIDAD DE ACCESOS		
	BITÁCORA	**TABLA TRANSITORIA**	**TABLA PRINCIPAL**
UNIR	OP.INICIAL OP.UNIR OP.FINAL	OP.UNIR	ACTUALIZACIÓN
BORRAR	OP.INICIAL OP.BORRAR OP.FINAL	OP.BORRAR	ACTUALIZACIÓN
MODIFICAR	OP.INICIAL OP.MODIFVIE OP.MODIFNUE OP.FINAL	OP.MODIFVIE OP.MODIFNUE	ACTUALIZACIÓN

Del cuadro se observa que si se ejecutara una secuencia de transacciones que sólo involucre operaciones de BORRAR y UNIR, por cada una de éstas se registran tres accesos a bitácora, un acceso a la tabla transitoria y un acceso a la tabla principal. De la misma forma, si se considera un conjunto de transacciones que realizan sólo operaciones de MODIFICAR, la cantidad de accesos en este caso son, por cada transacción, cuatro a la bitácora, dos a la tabla transitoria y uno a la tabla principal.

Teniendo en cuenta estos resultados y promediando la cantidad de accesos (la secuencia de operaciones que realiza una transacción se considera aleatoria), se obtiene el siguiente cuadro:

Operación	**Bitácora**	**Tabla Transitoria**	**Tabla Definitiva**	**Total de accesos**	**Promedio de accesos**
1	3 - 4	1 - 2	1	5 - 7	6
2	6 - 8	2 - 4	2	10 - 14	12
3	9 - 12	3 - 6	3	15 - 21	18
4	12 - 16	4 - 8	4	20 - 28	24
5	15 - 20	5 - 10	5	25 - 35	30
6	18 - 24	6 - 12	6	30 - 42	36
7	21 - 28	7 - 14	7	35 - 49	42
8	24 - 32	8 - 16	8	40 - 56	48
9	27 - 36	9 - 18	9	45 - 63	54
10	30 - 40	10 - 20	10	50 - 70	60

Por lo tanto la cantidad total de accesos que se producen al recorrer la bitácora y almacenar los cambios producidos a la tabla principal se considera una función lineal de la forma $f(x) = 6x$.


f(x) = 6x
70
60
50
40
30
20
10
0
Cantidad de accesos
1
2
3
4
5
6
7
8
9
10
Cantidad de operaciones

Clasificación de acuerdo al tiempo que demanda la ejecución de operaciones:

De los resultados obtenidos al medir los tiempos de ejecución (tiempo de verificación de bitácora) de las operaciones y considerando tiempos promedios de ejecución de secuencias aleatorias de operaciones, que se ejecutaron en ambos modos: monousuario o multiusuario, se resume lo siguiente:

CANTIDAD DE OPERACIONES	TIEMPO PROMEDIO DE VERIFICACIÓN (c.s.)
1	13.5
2	19.5
4	26.5
6	38.5
7	46
8	53.5
9	63
10	82
11	117.5
12	159.5
13	182
14	209
15	242.5
16	283.5
17	299.5
18	305.5

La función, además registra los siguientes porcentajes de crecimiento:

CANTIDAD DE OPERACIONES	TIEMPO PROMEDIO DE VERIFICACIÓN (c.s.)	PORCENTAJE DE CRECIMIENTO
1	13.5	
2	19.5	44
4	26.5	36
6	38.5	45
7	46	20
8	53.5	15
9	63	18
10	82	29
11	117.5	43
12	159.5	26
13	182	14
14	209	15
15	242.5	16
16	283.5	17
17	299.5	5
18	305.5	2

De lo expuesto se deduce que es una función creciente ya que a medida que se ejecuten más operaciones, la bitácora crecerá y el tiempo que demande recorrerla será mayor.

Graficando los resultados del cuadro anterior y analizando la forma que describen, se ve que su dibujo se asemeja al de la función $f(x) = x^2$, considerando una variación C, donde C es una variable que se analiza a continuación:

CUADRO DE COMPARACIÓN

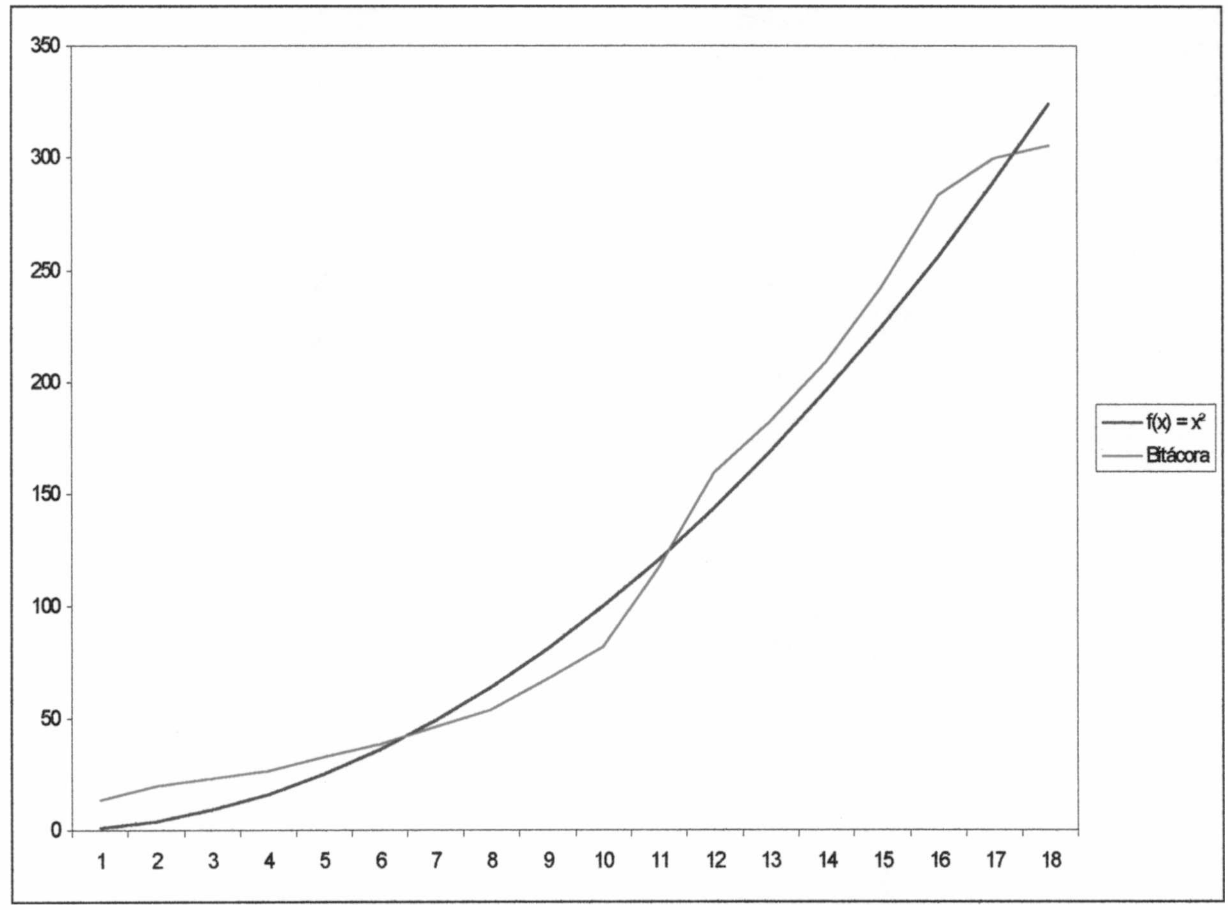
Cantidad de Operaciones	**Tiempo promedio de verif. (c.s.)**	**$f(x) = x^2$**	**C**
1	13.5	1	12.5
2	19.5	4	15.5
4	26.5	16	10.5
6	38.5	36	2.5
7	46	49	-3
8	53.5	64	-10.5
9	63	81	-18
10	82	100	-18
11	117.5	121	-3.5
12	159.5	144	15.5
13	182	169	13
14	209	196	13
15	242.5	225	17.5
16	283.5	256	27.5
17	299.5	289	10.5
18	305.5	324	-18.5

De la observación de las tres tablas anteriores se concluye que existe una oscilación entre la función que representa los tiempos de ejecución y la función $f(x) = x^2$. Los puntos de intersección coinciden con los momentos en que se produce el llenado de los buffers (de acuerdo a la definición realizada en el ambiente sobre la cantidad de Record Buffers).

Si se analizan los porcentajes de crecimiento se observa que la función presenta picos que coinciden también con los momentos en los que se llenan los buffers.

Si se analiza la variable C, se ve que esta cambia de signo en los mismos puntos.

A continuación se muestran los gráficos de la función ***f(x) = x²*** y de la función que describe la recuperación basada en bitácora, para la cantidad de operaciones expresadas.



9 - CONCLUSIONES

De lo analizado en los capítulos precedentes se concluye que un buen sistema de procesamiento de transacciones es aquel que no sólo garantiza la consistencia de la base de datos durante la ejecución normal de una aplicación, sino que además provee mecanismos de recuperación ante posibles fallas, para mantener la seguridad e integridad de la misma.

Cabe aclarar que las aplicaciones pueden correr en ambientes mono o multiusuario, por lo que el sistema debe ser capaz de administrar transacciones concurrentes.

Tanto para la administración de transacciones como para la recuperación del sistema ante caídas se elegió el método basado en la bitácora con modificación diferida y puntuales excepciones de modificación inmediata de la base de datos. Estas excepciones se contemplan para posibles operaciones que puedan desarrollarse para una aplicación particular, que por su importancia o prioridad necesitan salvar los cambios producidos en forma inmediata. La elección se debió a que, entre otros aspectos, la principal desventajas que presenta el método basado en la bitácora es la gran cantidad de accesos a disco que se producen. Igualmente es preferible correr con este inconveniente y no con las desventajas que presenta el método de recuperación basado en la doble paginación, según se describe en el Capítulo 6.

También se analizó Btrieve que utiliza el método basado en la doble paginación para la recuperación del sistema ante fallos, como se describe en el Capítulo 5. Este sistema, al igual que la unidad desarrollada, consume gran cantidad de tiempo si se habilitan los métodos de procesamiento de transacciones provistos, por lo que admite deshabilitarlos para optimizar los tiempos de ejecución.

Para desarrollar la Unidad de procesamiento de transacciones se estudió, en el Capítulo 4, el motor de bases de datos del Paradox, sus ventajas, propias del paradigma Orientado a Objetos y otras propias del motor; y sus desventajas.

A través de los métodos existentes en la Unidad desarrollada se da solución a los problemas que presenta el Paradox Engine, deshabilitando escrituras forzadas a disco de operaciones individuales, habilitando escrituras de este tipo de todas las operaciones que componen una transacción y almacenando todos los resultados producidos por las transacciones en la bitácora y tablas transitorias (creadas por los métodos de la Unidad, asociadas a cada tabla principal existente), que se encuentran alocadas en almacenamiento no volátil.

Para la recuperación del sistema ante caídas suaves se implementaron en la Unidad, métodos que verifican, al inicio de una aplicación, el estado de la bitácora. Si se produjo un fallo, entonces se restauran las tablas con los resultados guardados en la bitácora, de las transacciones terminadas no salvadas definitivamente, al momento del fallo.

Las cuestiones atinentes a la recuperación del sistema ante un fallo en los medios de almacenamiento o caídas duras se refieren a restaurar el sistema y la base de datos a partir de una copia de seguridad y actualizar, mediante la ejecución de un método externo provisto en la Unidad, la base restaurada con los datos de la bitácora si ésta no fue afectada por el fallo.

Para comprobar el funcionamiento de la unidad se creó un programa de prueba que somete a las tablas a la ejecución de transacciones que realizan actividades críticas tales como actualización y borrado de registros.

Se sometió al programa de prueba a un fallo y se observó que la recuperación fue satisfactoria en cuanto a que al restaurar el sistema y, previa observación de las tablas y de la bitácora, se almacenaron en las tablas definitivas sólo los resultados de las transacciones terminadas no salvadas al momento de la caída.

Analizando luego de su ejecución y recuperación ante un fallo los resultados obtenidos se comprobó, tal cual lo esperado, que las tablas conservaron su consistencia.

Además se sometió a la Unidad a diferentes pruebas para comprobar tiempos de ejecución y cantidad de accesos. Con lo cual se observó que la cantidad de accesos que se realizan es una función de la forma $f(x) = 6x$, y que el tiempo que demanda la verificación de la bitácora y la actualización de las tablas es una función similar a la función $f(x) = x^2$ con una variación C. En la gráfica de ambas funciones se pueden observar puntos de intersección. En estos puntos la variable C cambia de signo y esto coincide, como ya se analizó en el apartado 8.4, con los momentos en que se produce el llenado de los buffers, según la especificación realizada en el ambiente. El sistema no es demasiado sensible, en cuanto a tiempos de ejecución, si se ejecuta en modo mono o multiusuario. Cabe aclarar que la administración de recursos en sistemas mono o multiusuario la realiza el ambiente.

De lo analizado en el apartado 8.3 surge que la diferencia de tiempo de ejecutar un conjunto de operaciones con y sin el uso de la unidad es muy significativa, dado que al usar la Unidad se realizan verificaciones y accesos a recursos que no se hacen cuando ésta no se utiliza, si bien se consideraron accesos a disco en las pruebas realizadas sin el uso de la unidad.

Se observó también en el apartado 8.2 que el sistema registra diferencias considerables de tiempo si se encuentran, durante el recorrido de la bitácora, operaciones que han abortado su ejecución. En esta situación el tiempo de verificación de la bitácora es menor. Esto se debe a que, si alguna transacción abortó su ejecución, se verifica su estado en la bitácora pero no se produce almacenamiento en las tablas.

La performace de una aplicación puede mejorarse desabilitando el uso de la Unidad, pero se corren los riesgos propios de no contar con el procesamiento de transacciones tales como llevar a la base de datos a un estado inconsistente luego de la ejecución de operaciones y no poder restaurar la misma a un estado consistente esperado luego de un fallo.

De esta forma se incorporó al Paradox-Engine para su versión de Pascal, una Unidad que asociada con las ya existentes, permite el manejo de transacciones y la recuperación ante caídas suaves y duras y, por lo tanto, brinda soluciones para los problemas de seguridad e integridad que puedan surgir en el uso cotidiano de las tablas, ya sea a través de la ejecución normal de transacciones o de posibles fallos que puedan afectar al sistema o a los medios de almacenamiento.

10 - BIBLIOGRAFÍA

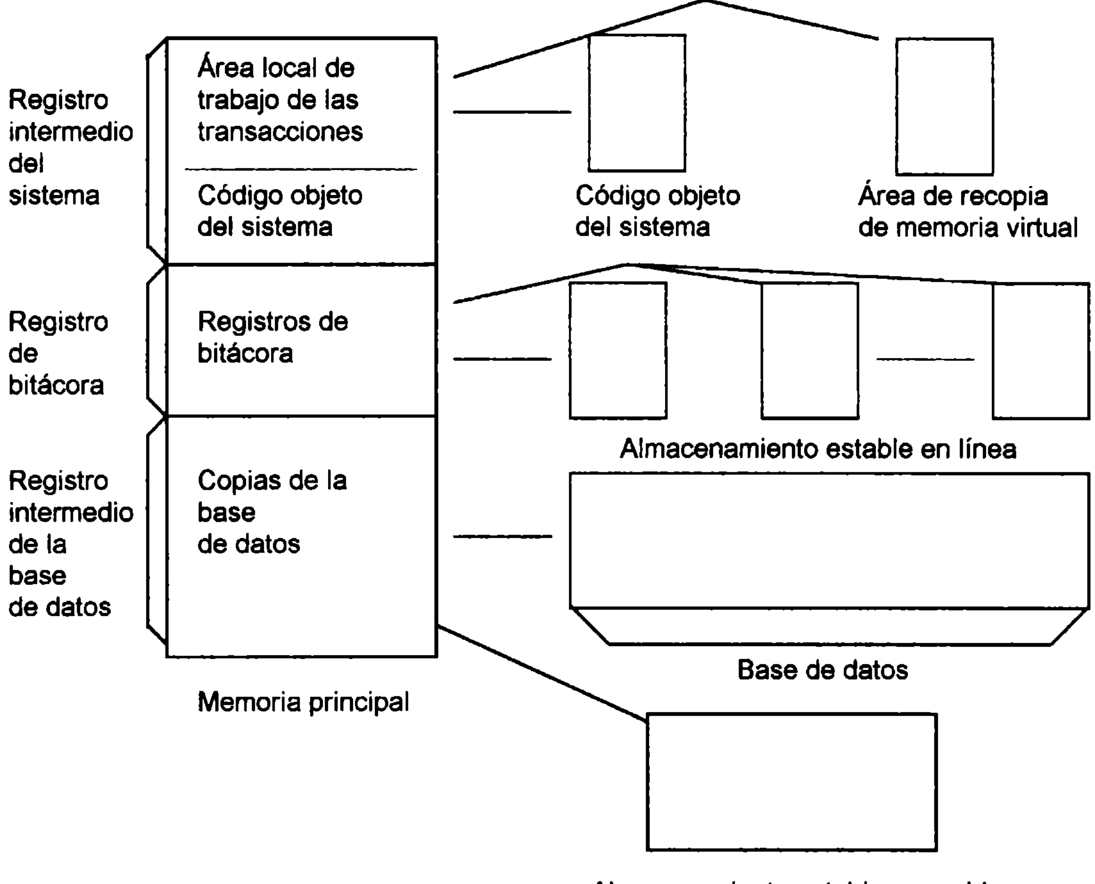
- Paradox Versión 3.5 - Guía del Usuario. 1992
- Btrieve Versión 4 - Manual de Referencia - 1992
- "Introducción a los Sistemas de Bases de Datos" - C.J. Date – Addison Wesley 1990
- "Fundamentos de Bases de Datos" - Korth y Silberschatz – Mc Graw Hill - 1992
- Pascal 7.0 - Borland - Manual de Referencia - 1993
- Paradox Engine - Borland - Manual de Referencia para su versión sobre Pascal. 1993
- "A First Couse in DataBase Systems" - Ullman, Windom – Prentice Hall - 1997
- "Diseño conceptual de Bases de Datos" – Batini, Ceri, Navathe – Addison Wesley1994
- "Applied Sofware Measurement" – Capers Jones – McGraw Hill 1996

Páginas Web Consultadas

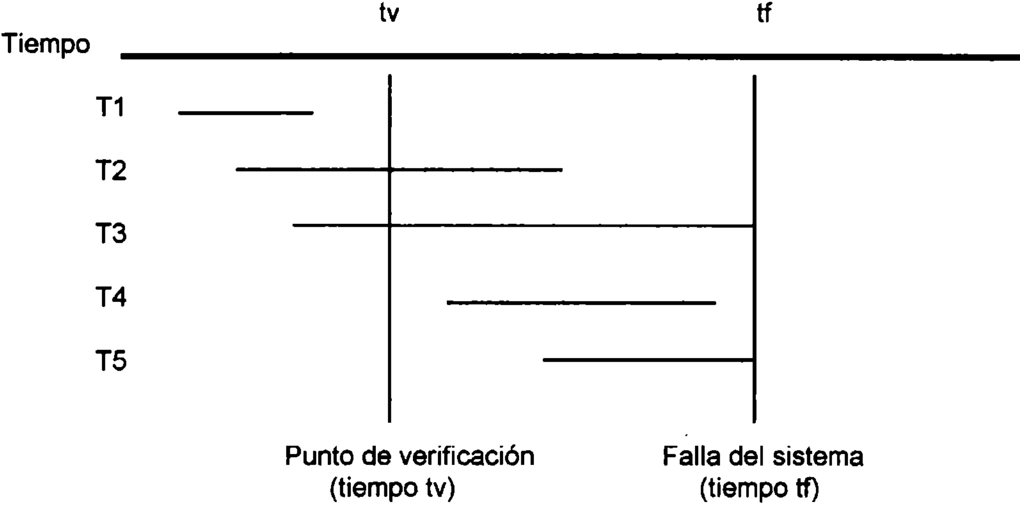
- http:// www.cs.byu.edu/courses/cs501r.2/homepage.html
- http:// www.cs.umb.edu/class/cmsc424-0201-f96/
- http:// www.unik.no/~nettdb
- http:// www.btrieve.com
- http:// www.borland.com
- http:// www.cs.runit.edu.au/researchprograms/researchgroups/dbgroup/index.html

APÉNDICE A

DIAGRAMA DEL MÉTODO DE RECUPERACIÓN BASADO EN BITÁCORA



MODELO DE ALMACENAMIENTO



CINCO CATEGORÍAS DE TRANSACCIONES

El esquema muestra la ejecución de cinco transacciones al momento tf en el que se produce un fallo en el sistema.

Cuando se recupera al sistema de este fallo, se recorre la bitácora desde el último punto de verificación (momento tv) y se analiza:

* Las transacciones t2 y t4 deben rehacerse, ya que, según el diagrama, existen en la bitácora tanto el registro <Ti,start> como el registro <Ti,commit>.

* Las transacciones t3 y t5 no deben rehacerse, ya que para ambas, existe el registro <Ti,start> pero no existe el registro <Ti,commit>.

* El resultado de la transacción t1 ya se volcó el la base de datos cuando se realizó el último punto de verificación, ya que en ese momento existían en la bitácora los registros <T1,start> y <T1,commit>.

APÉNDICE B

DIAGRAMA DEL MÉTODO DE RECUPERACIÓN BASADO EN LA DOBLE PAGINACIÓN

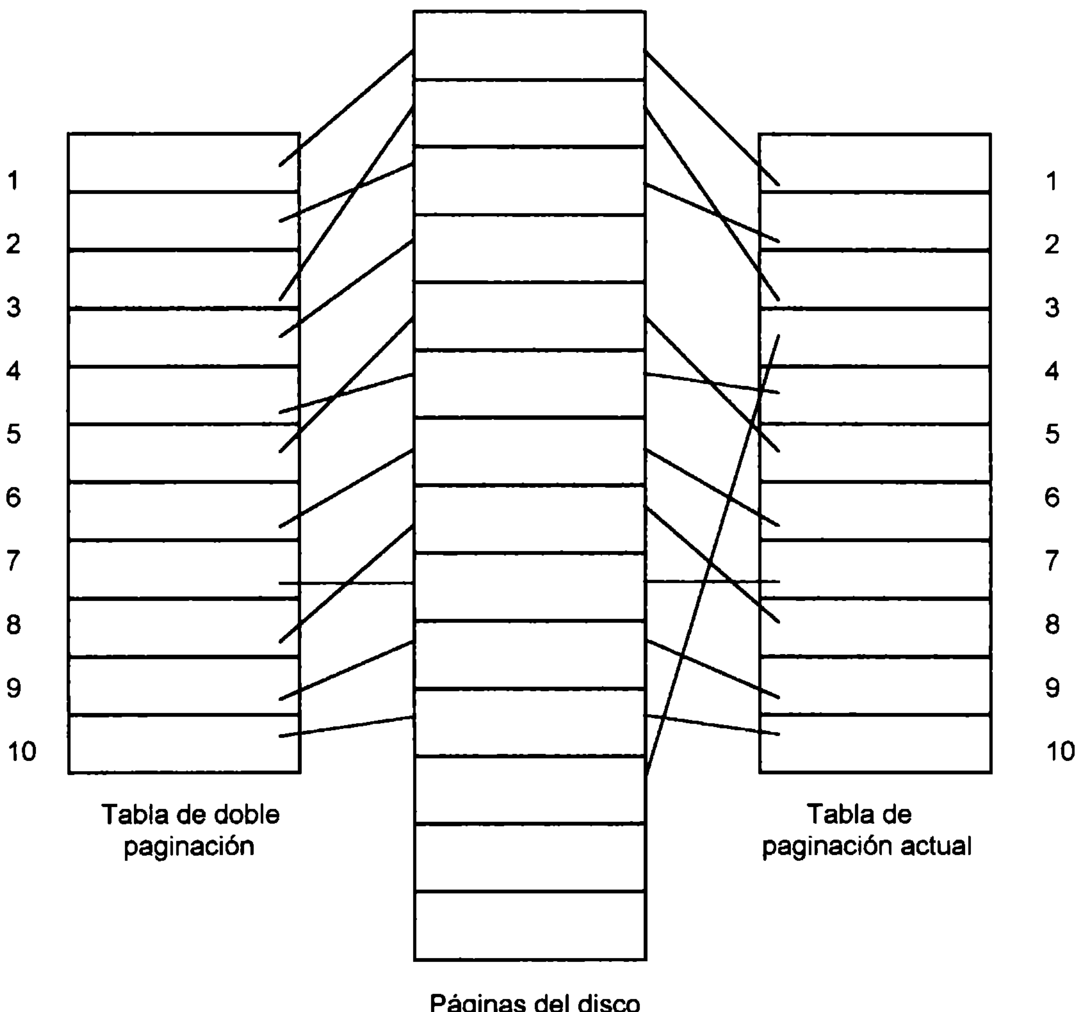


TABLA DE PAGINACIÓN DOBLE Y ACTUAL

APÉNDICE C

MENSAJES DE ERROR DE LA UNIDAD

ERROR	CODIGO	MENSAJE
PXERR_INVENGINETYPE	400	Tipo Engine inválido
PXERR_ENGINEOPEN	401	Engine ya abierto
PXERR_ENGINENOTOPEN	402	Engine no abierto
PXERR_DBALREADYOPEN	403	Base de datos ya abierta
PXERR_DBNOTOPEN	404	Base de datos no abierta
PXERR_CURSORALREADYOPEN	405	Cursor ya abierto
PXERR_CURSORNOTOPEN	406	Cursor no abierto
PXERR_RECALREADYATT	407	Registro ya asociado a cursor
PXERR_RECNOTATT	408	No hay registro asociado a cursor
PXERR_INVFIELDTYPE	409	Tipo de campo inválido
PXERR_INVCURRRECORD	410	Registro actual inválido
PXERR_TABLESDIFFER	411	Tablas diferentes
PXERR_INVKEYCOUNT	412	Cuenta de clave inválida
PXERR_NOKEYMAP	413	No existe mapeo de claves
PXERR_NOCLEARNULL	414	No puede ejecutar clearNull sobre un registro genérico
PXERR_DATACONV	415	Falló la conversión de un valor de campo
PXERR_BLOBNOTOPEN	416	BLOB no abierto
PXERR_TRANSNOTOPEN	417	Tabla transitoria no se puede abrir
PXERR_TRANSNOTCLOSE	418	Tabla transitoria no se puede cerrar
PXERR_TRANSEXISTE	419	Existe una tabla con el nombre de la transitoria
PXERR_YAEXISTE	420	Existe una tabla con dicho nombre
PXERR_OPERACIONANULADA	421	Operación anulada
PXERR_CURSORTRANS	422	No abre el cursor de transitoria
PXERR_GUARDARDATOSTEMPORARIOS	423	No puede guardar datos en bitacora o tablas transitorias

ERROR	CODIGO	MENSAJE
GUARDADOINCOMPLETO	424	No se completó el guardado definitivo
OPERACIONNOEXISTENTE	425	No existe la operación
PXERR_BITACORA	426	Error en bitácora
BITACORANOABIERTA	427	Bitácora no abierta
PXERR_NOSALVADEF	428	No se puede realizar el salvado definitivo
PXERR_BITACORAACTIVA	429	Bitácora ya activa
MEDIOSAFECTADOS	430	La bitácora o las tablas asociadas fueron afectadas por un fallo en los medios de almacenamiento.