Seguridad de la información

Temas de la clase: recuperación de caídas en procesos interactivos y en procesos batch. Punto de reenganche. Control de secuencia de procesos. Optimización de código.

Seguridad de la información

Una de las características que hacen óptima a una aplicación, es su capacidad de reponerse ante una caída inesperada.

Nos referimos a la seguridad de la información en el sentido de cuán segura está la misma dentro de nuestro sistema (no hablamos de confidencialidad, ni de permisos). Hablamos de tener la seguridad de que quedó bien grabada, íntegra y completa. Especialmente en casos en que se están actualizando datos en más de una tabla o archivo (una inserción compleja).

Si no tenemos previsiones para estos casos ni modos de detección de estas situaciones, puede ocurrir que se pierda mucho tiempo y también datos.

Los procesos de recuperación que ejecuta una aplicación ante una caída difieren si se trata de un proceso interactivo o de un proceso batch.

En procesos interactivos

Si bien los gestores de bases de datos disponen de previsiones para estas situaciones y tienen un manejo transaccional que permite deshacer una transacción, es interesante comprender cuál es la situación ante una caída inesperada.

En un proceso interactivo, que está corriendo en un entorno multiusuario, puede haber muchos usuarios que están trabajando. Entre esos usuarios hay algunos que están modificando registros y otros solamente están consultando

Muchos datos pueden estar actualizándose simultáneamente y algunos de ellos surgen de transacciones inmediatamente anteriores. Si no hubiera un control de que las mismas hayan terminado correctamente, podría haber datos calculados en base a otros que son inconsistentes y por lo tanto nuevos datos también erróneos.

Ante una caída, se deberá verificar cuáles transacciones terminaron correctamente y cuáles no y entonces ver cuáles habría que “deshacer”.

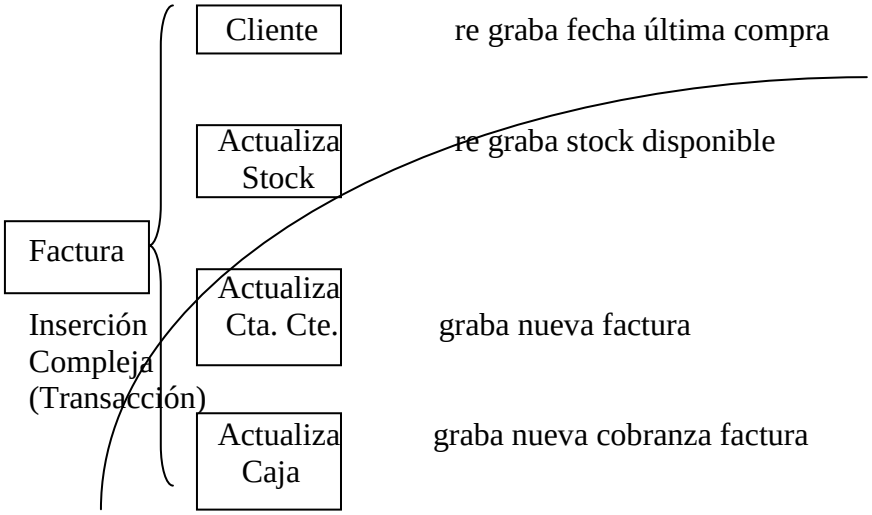
Dada la característica de "interactivos" de estos procesos, no es posible reproducir la secuencia de ingreso de datos que fue ocurriendo (en un proceso batch seguramente si, ya que los nuevos datos seguramente provienen de alguna tabla o archivo que contiene "novedades").

Por eso los gestores de bases de datos guardan un archivo de bitácora (conocido como archivo de Log) en donde se van guardando datos de cada transacción para permitir su posterior "vuelta para atrás".

El siguiente ejemplo muestra por qué es necesario resguardar los datos de cada transacción:

Caso: proceso interactivo de ingreso de una factura

Ingreso la factura y se actualiza stock, se actualiza cuenta corriente y se actualiza caja



En este proceso, cada vez que se carga una factura y se oprime el botón "aceptar", se hacen estas cuatro actualizaciones y además se imprime la factura.

En la tabla de clientes regraba la fecha de última compra, en la tabla de stock regraba el stock disponible, en la tabla de cuenta corriente agrega un registro nuevo por la factura (graba nueva factura) y en la tabla de movimientos de caja graba la nueva cobranza de factura.

Si alguno de estos pasos no se cumpliera cuando se termina de grabar la factura, podría pasar que si consultamos por una opción del sistema nos mostraría una situación y si lo hacemos por otra opción (otra consulta) nos mostraría otra.

Podría pasar por ejemplo, que si vamos a una pantalla que se llamara "Consultas de Clientes", apareciera *Cliente: Juan Pérez, fecha de última compra: hoy*. Y si la transacción se grabó parcialmente, cuando entramos por otra pantalla que se llama "Resumen de cuenta del cliente", para este mismo cliente, *diría que su última factura es de hace un mes*, (en la tabla se iba a guardar la factura de hoy pero no llegó a grabarse).

Entonces si miramos por una pantalla me dice que el cliente vino hoy y si entramos por otra consulta del mismo sistema me dice que hoy ese cliente no vino.

Debe quedar claro que aquí se produjo una incoherencia en los datos del sistema.

Tabla de Clientes

Saldo	Fecha Ultima Compra
1.000	19 de Mayo

Resumen de Cuenta

Saldo	Fecha
900	10 de Abril

No llegó a actualizarse

A este conjunto de actualizaciones se lo conoce como transacción o inserción compleja.

Si se grabó solo parte de esta transacción no sirve. Y cuando se cae el sistema no sabemos cuáles transacciones quedaron completas y cuáles no. Entonces, si el manejador de base de datos no gestionara transacciones, sería nuestro problema detectar las incompletas y ver qué hacemos con ellas.

La existencia del Log hace que sea posible efectuar un “roll back” (deshacer) de una transacción.

No profundizaremos en cómo se graban los archivos de Log ya que todo este proceso es automático y transparente para nosotros porque lo hace el gestor de bases de datos.

En procesos batch

Una situación que puede ocurrir y a veces ocurre en la vida real es que mientras se estaba ejecutando un proceso muy largo que hacía una actualización masiva de datos, el mismo se interrumpió.

La causa de esto podría ser un corte de luz, un error en el programa en tiempo de ejecución, etc.

Dado que no había ninguna previsión, lamentablemente habrá que empezar todo de vuelta. Pero a veces esto no es tan fácil o simplemente es imposible.

Vamos a distinguir algunos casos posibles. Veremos qué pasa en esos casos, qué problemas podemos tener y cómo deberíamos prever su resolución.

Las situaciones que describiremos, podrían ocurrir tanto con archivos como con bases de datos.

Por ejemplo en un proceso de ingreso de Cobranzas de un instituto como Ort, podríamos tener una inserción compleja que involucra lo siguiente: en una tabla regraba un campo que indica

si está moroso o no (tabla ´alumnos´) y en otra tabla agrega un registro con los datos del recibo (tabla ´caja´).

Entonces si esa inserción compleja quedara por la mitad, nos queda información inconsistente, porque podría haber quedado grabado en la tabla de movimientos de caja del día, que se hizo una cobranza con el recibo tal a un alumno determinado, y no quedar grabada en la tabla de alumnos la actualización de la deuda de ese alumno, ¿se entiende? Debíamos actualizar dos tablas pero si una de esas dos no quedó actualizada, va a pasar que si consultamos el listado de caja dice que se cobró a este alumno y si pedimos el resumen de la cuenta corriente de ese alumno, muestra que sigue debiendo, y eso es una incoherencia que no podemos permitir.

El problema es: no quedó la información o no quedó completa que es más o menos lo mismo. Si quedó incompleta no sirve, y si no quedó tampoco sirve obviamente.

Otro ejemplo: tenemos un programa que actualiza saldos de cuentas de Clientes de un banco a partir de un archivo de movimientos del día.

Podríamos tener estas tres alternativas, pero no son las únicas posibles:

- Que Clientes fuera una tabla y que un proceso vaya actualizando los saldos a partir de una recorrida del archivo de novedades.

- Que Clientes fuera un archivo secuencial en el que se pudiera regrabar y que un proceso similar al del punto anterior vaya actualizando los saldos.

- Que Clientes fuera un archivo secuencial pero que el proceso de actualización genere un nuevo archivo (como un apareo). Se lee un registro del archivo de Novedades donde viene un movimiento cuyo importe hay que sumar o restar al saldo original. Entonces se va generando un nuevo archivo con los datos del archivo original más las modificaciones del saldo de cada registro.

¿En estos tres casos qué pasaría si el proceso se interrumpe? En alguno de ellos pasaría lo mismo, que es que queda desactualizado. ¿Pero qué tenemos que hacer en cada caso?

Tendríamos soluciones distintas:

Si dijimos que era una tabla, en Novedad leemos un registro de un cliente y regrabamos el saldo, leemos otro registro y regrabamos el saldo. Se cortó el proceso en la mitad del archivo, ¿qué podemos hacer?

Podrían sugerirnos guardar la fecha o la hora en el momento en que se hizo la operación. ¿Qué operación? Nos dirían: la actualización. Preguntaríamos: ¿cuál actualización?

¡Estábamos actualizando miles de registros!

El problema es que no sabemos hasta cuál registro actualizó y entonces corremos el riesgo por ejemplo, de duplicar un saldo.

También nos podrían proponer guardar en cada registro una marca de actualizado o no. No es conveniente, entre otras cosas porque después de este proceso habría que blanquear la marca para una futura nueva actualización.

O también nos hablarían de un contador. ¿Y dónde lo guardamos? ¿Y cuándo lo guardamos? ¿Y sirve para contar qué?

Nos responderían, bueno lo guardan en un archivo, cada vez que graban y cuenta la cantidad de grabaciones que van haciendo.

¿Cuál sería para el caso planteado la única opción 100% segura? Hacerlo todo de nuevo.

Para poder hacerlo de vuelta se necesita tener una copia actualizada del archivo, recuperar los datos de ésta y volver a procesar.

En algunos casos existe la posibilidad de recuperar las copias de seguridad y de empezar a hacer todo de vuelta. Esto siempre y cuando sea todo recuperable, es decir, que la copia está actualizada como para repetir todo el proceso y que no sea la situación de que esta tabla la estaba compartiendo con otro proceso que estaba ejecutando otro usuario.

El problema es que esta empresa podría ser un banco, y el proceso tardar cuatro horas en correr, y que debe estar listo antes de que abra el banco y que solo falten dos horas para que abra el banco. El tiempo es un factor importante a tener en cuenta.

En el caso del archivo secuencial regrabable pasa lo mismo que en el caso anterior, no sabemos cuáles registros quedaron actualizados y cuáles no
.
Supongamos que tenemos la copia de seguridad bien actualizada, pero no tenemos tiempo de recuperarla.

Queremos saber a cuál registro llegamos y actualizar de ahí en más porque no hay tiempo y sería crítico que nos equivocáramos, que creyéramos que actualizamos uno y no lo actualizamos o viceversa.

En las dos primeras situaciones corremos este riesgo.

Si empezamos de vuelta el proceso de la primera o de la segunda alternativa sin recuperar la copia de seguridad vamos a volver a actualizar algún registro que ya actualizamos.

En el caso de la tercera alternativa no pasa lo mismo porque genera un archivo nuevo. Si se cortó el proceso, cuando empiece de vuelta va a volver a generar un archivo nuevo, empieza todo de cero, no hay nada que se duplique, porque está generando un archivo nuevo. Pero podríamos volver a tener el problema de que no tenemos tiempo de hacer el proceso completo otra vez.

Sigamos tomando como ejemplo nuestro instituto, y en él debe haber un proceso que mensualmente emite las facturas que les van a enviar a ustedes por correo.

Ese proceso seguramente hace una recorrida de una tabla de Alumnos. Encuentra un alumno, por ejemplo, Sandra Pérez. Entonces de ahí irá a otra tabla donde dice qué conceptos se le facturan a Sandra Pérez (por ej. hay que cobrarle cuota, seguridad, etc.), con eso calculará el importe de la factura, grabará la factura y después pasará al alumno que sigue.

El tiempo de ejecución es largo, porque hay un montón de alumnos.

¿Qué pasaría si se nos corta la ejecución por la mitad, o por el principio o por el final cuando ya falta poco?

Este proceso actualiza la tabla de Alumnos con una marca de "facturado" y agrega un nuevo registro en la tabla de Facturas.

En definitiva lo que necesitamos es ver cómo implementar lo que se llama un reenganche o un re arranque desde el punto más cercano al corte que ocurrió para perder el menor tiempo posible preservando la información segura.

Tener en cuenta esta problemática, tiene sentido solo para procesos que sean críticos. Pueden ser críticos para la integridad de los datos o también podrían ser críticos por el tiempo que toma su ejecución.

Por ejemplo, tenemos un proceso que es simplemente un programa que corre todas las noches listando el resumen de cuenta del último mes de todos los clientes de esta sucursal del banco. Imprime toda la noche y está calculado que tarda 8 horas en imprimir.

Si se cortó la ejecución cuando faltaban 10 minutos de listado, obviamente no vamos a empezar de vuelta. Si se cortó cuando faltaban 7 horas y 55 minutos de listado, podríamos empezar de vuelta, porque perdimos solo 5 minutos.

Si bien estamos hablando de un listado, que no actualiza nada, es igualmente crítico ya que es un listado necesario y el tiempo que dura su ejecución es importante.

En todos los ejemplos mencionados se ve claramente que necesitamos retomar el proceso desde donde se interrumpió. ¿Qué podríamos hacer para retomar lo más cercano posible a donde se cortó?

Tenemos que poder ver en algún lugar, por donde iba más o menos el proceso cuando se cortó. No me propongan ir a la impresora y fijarse hasta dónde se imprimió. Si yo le digo a mi jefe que me encargó este programa: "hay que ir a fijarse al papel cuál es el último que imprimió" el me contestará "usted vaya a fijarse a tesorería el papel con su liquidación final".

Necesitaríamos guardar ese dato en algún lado. La idea es que un programa debería detectar, sin intervención del operador, que hubo una caída anterior. Esto podría ocurrir cuando se vuelve a ejecutar el programa (una vez que se re inició la máquina).

Y una vez que detectó la caída, lanzaría automáticamente el proceso de reenganche.

En otras palabras, un programa seguro, siempre al comenzar se fijaría si su última ejecución terminó bien. Si no fuera así ejecutaría sus tareas de reenganche o re arranque antes de continuar.

Nosotros apuntamos a una solución automática, que no dependa de la intervención de nadie. Entonces la idea de guardar "por donde iba el listado" es lo más conveniente.

A este dato que vamos a guardar lo llamaremos **Punto de Reenganche.**

Este punto de re arranque tiene que guardar algo que nos diga: "debemos retomar desde más o menos por tal registro". Ahora, la pregunta es ¿cuándo usar esto del punto de reenganche?

¿Y cada cuánto usarlo? ¿Cuándo implementarían ustedes este control de reenganche?

¿En qué situación, mientras están desarrollando un programa ustedes dirían, le vamos a poner control para que no pase esto, y cuándo no?

Implementar este mecanismo es costoso. Decidir su uso depende un poco de la duración del proceso, no tanto de las fallas, ya que se supone que no puedo saber a ciencia cierta la posibilidad de fallas. Depende de la duración y de lo crítico que sea este proceso.

Si se trata de un listado que si no se emite hoy, se emite mañana y esta información mañana no la necesita nadie con urgencia, no tiene sentido implementar una previsión de caídas (ya hablamos de esto más arriba). Entonces tendremos que ver cuán oportuno será en cada caso implementar estas previsiones.

Al hablar de implementar procesos de reenganche la primera pregunta era: cuándo lo hacemos y cuándo no. Está contestada con lo que dijimos recién.

La segunda es, ya que decidimos hacerlo, ¿cada cuánto lo hacemos? Cada cuánto lo hacemos quiere decir, ¿cada cuántos registros grabo el punto de reenganche? O ¿cuándo, ante qué situación?

En el resumen de cuenta de recién, se calcula un total por cada cliente. En un listado de un resumen de cuenta, cuando terminan todos los movimientos de una cuenta se muestra un subtotal. Y al final del listado podría mostrarse un total general.

Un buen momento para guardar un punto de re arranque es cuando cambia de cliente, es decir, en el corte por cliente. Cuando se produce un corte de control la información está más consistente (tenemos un subtotal calculado, terminamos de listar un cliente). En ese momento tenemos el total del cliente que terminó y si tenemos que empezar de vuelta, ya sabemos que ese cliente terminó. O bien guardamos el que terminó, o guardamos el que iba a empezar, a criterio del programador.

¿Dónde guardaríamos este punto de re arranque? En un archivo. ¿Qué guardaríamos en este archivo? Los datos que se necesitan para reenganchar. En el caso de ejemplo, guardaría el último cliente y el total general acumulado hasta el momento.

Quizás ustedes se preguntarán si estos casos son probables. Les aseguro que si lo son.

Para nosotros como programadores, la probabilidad debería ser “siempre puede ocurrir”.

Aunque haya un 1 % de probabilidad de que esto pase debemos preverlo porque si no lo hacemos el día que pase tendremos un gran problema.

En una empresa grande debe ser difícil que se corte la energía que alimenta las máquinas, pero yo no creo que por eso no hagan copias de seguridad (backup) de los datos o no tengan un montón de medidas de seguridad para con la información. No pueden dejar de tomar recaudos con respecto a la información porque eso puede significar mucho dinero.

Recuerden que un corte de energía sería solo uno de los problemas que podría causar la caída de un sistema.

Aún en un listado (que no actualiza ningún dato) puede ser crítica la situación en el sentido de que se complica empezar de vuelta porque tarda mucho, y entonces nos interesa el tiempo.

Ejemplo de un archivo de punto de reenganche:

Archivo de reenganche

Podría ser un archivo de texto.

Clave de corte total
Cliente anterior acumulado

Este archivo que guarda los puntos de reenganche podría tener un solo registro que vamos re grabando, re grabando y re grabando…O también podría ser un archivo en el que se fueran agregando registros y quedara como una historia de todos los puntos que fue habiendo, que fueron produciéndose.

Tenemos que pensar que cada vez que vamos a grabar el punto de reenganche, estamos haciendo un acceso más al disco. Este archivo podría ser de organización secuencial pero no sería conveniente que fuera una tabla de una base de datos.

Si fuese una tabla podría ocurrir que ante una caída el dato no quedara grabado y luego no fuese accesible. Además es más rápido guardar datos en el archivo de texto.

En todos los casos, sería conveniente que el archivo de punto de reenganche se cerrara luego de cada grabación. De ese modo prevenimos que el buffer no hubiera bajado al momento de una caída del sistema.

Ya dijimos que grabar el punto de re arranque es costoso. Si no fuera así, lo mejor y más seguro sería grabar uno por cada registro que se procesa.

Dependerá de lo crítico que sea cada caso.

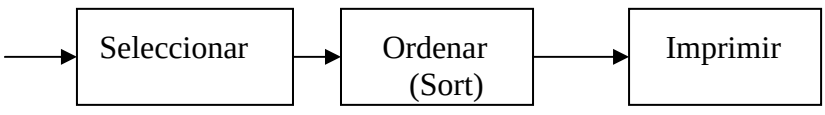
En el ejemplo que mencionamos anteriormente (de la facturación de Ort), estaría bien, grabar el punto de re arranque cuando terminó una factura, grabar la que terminó o grabar cuál es el número de la próxima, la que empieza. Eso sería mucho más cómodo para recuperar, porque aquí si guardo el punto de reenganche mientras se procesan los detalles, tendría que guardarme los subtotales, y todos los cálculos que estoy haciendo. Es más cómodo grabar una factura terminada o cuál factura va a empezar, lo que significa lo mismo.

Cuánto más consistente esté la información al momento de guardar el punto de reenganche, menor será la cantidad de datos que deberíamos guardar.

La diferencia con el caso de los resúmenes de cuenta es que en ese caso todos los registros son iguales en la misma tabla, y en el caso de la facturación hay cabeceras y detalles en distintas tablas. En el primer ejemplo podríamos guardar el punto de reenganche cuando hay un corte por fecha o por cliente o en la mitad de un cliente, pero en el ejemplo actual no sería conveniente guardarlo en la mitad de un alumno.

Lotes de procesos

Veamos otro caso. Supongamos que tenemos que hacer un listado de todas las personas del mundo que se llaman María ordenado por apellido. Entonces disponemos de un archivo con toda la gente del mundo. Tendríamos que tomar ese archivo, seleccionaríamos los datos, después ordenaríamos y después imprimiríamos.



Todo esto en realidad podría haber sido un solo programa. El mismo seleccionaría, ordenaría y luego imprimiría. Pero que sea así separado nos da alguna ventaja. ¿Qué ventaja nos da que sean tres programas en vez de uno?

Esta separación de tareas hace más fácil cualquier modificación de las mismas y también sirve para tener posibilidad de reenganchar el proceso ante una caída de alguna de las tareas.

Si quisiera cambiar el criterio de búsqueda, solo cambiaría el “Seleccionar”. Si lo quisiera ordenado por edad en vez de por apellido, cambiaría el “Ordenar”. Y si quisiera que en vez de imprimir lo guardara en un archivo, cambiaría el “Imprimir”. O sea que es más ventajoso tenerlo así separado si pienso que puede haber modificaciones.

Ahora pregunto: ¿en cuál de estos tres programas implementarían punto de reenganche? ¿En cuáles preverían que pudiera haber un reenganche?

El archivo, como era de todo el mundo tiene millones de registros.

Por supuesto que no podemos tener un solo punto de reenganche para los tres programas.

¿Pondrían puntos de reenganche en el proceso “Seleccionar”? ¿Y cada cuánto lo pondrían?

En este caso, el archivo de entrada es todo el padrón del planeta, son millones y millones de registros. Obviamente pondríamos puntos de reenganche para ese proceso y para hacerlo nos plantearíamos el tamaño de este archivo.

Si decimos que estamos buscando un nombre en la población de todo el mundo, va a ser gigantesco; si decimos que estamos buscando ese mismo nombre en nuestra libreta de direcciones va a ser mucho más chico. Sería conveniente grabar un punto de reenganche cada cierta cantidad de registros.

En el proceso “Ordenar” no podríamos guardar puntos de reenganche porque se trata de un programa que ejecuta un algoritmo de ordenamiento (sort) y no tenemos forma de retomarlo desde algún punto. Entre otras razones tenemos el hecho de que probablemente (lo más seguro) es un programa de terceras partes, a cuyo código no tenemos acceso y otra razón es el modo de funcionamiento de los programas que ordenan, que implica el manejo de muchos archivos temporarios simultáneamente.

¿Y en el proceso “Imprimir”? Podemos hablar de imprimir o de generar un archivo que para este ejemplo sería lo mismo. Iríamos guardando un punto de reenganche, cada cierta cantidad de registros. En esta ocasión no hay corte de control, vamos a listar todo el archivo.

Ahora si ya estamos de acuerdo con respecto a los puntos de reenganche, continuemos analizando nuestro ejemplo. Nuestra tarea batch era hacer estos tres procesos. Entonces, supongamos que se cortó el proceso en algún lado y nosotros queremos que la recuperación sea automática.

Si se cortó la ejecución mientras estaba corriendo el tercer proceso, es muy probable que si no nos fijamos, el proceso de re arranque va a empezar de vuelta por el primer proceso de nuestra tarea batch, cuando este ya había finalizado exitosamente (y encima de todo era muy largo!)

¿Cómo haríamos para que no empiece desde el principio nuevamente?

Cada uno de estos tres programas se tendría que fijar de algún modo si el anterior terminó bien, es decir, tener alguna manera de distinguir si el anterior a él pudo terminar su ejecución correctamente.

Además tendría que ver si él mismo terminó bien. Y en ese caso, el programa tendría que ejecutar su propio reenganche.

Estamos hablando de ver si tiene que ejecutarse o no. Ya hemos explicado cómo reenganchar adentro de cada uno de los programas. Ahora estamos viendo **cómo se daría cuenta cuál tiene que correr y cuál no** dentro de una secuencia de procesos.

Entonces tendría que haber algo por encima de los tres programas. Algo que indique que el programa anterior terminó correctamente. ¿Qué podríamos hacer como para que un programa sepa que el anterior terminó bien? Evidentemente **hay que poner alguna marca** (que obviamente no va a poder estar en la memoria principal).

Podría ser:

- Una marca en un registro de un archivo
- Que la marca sea la existencia de un archivo (que la misma existencia de un archivo indique que este proceso se corrió)
- Que la marca sea el uso de un archivo en exclusividad, por ejemplo.

Una forma de hacer esto sería, que este Sort del ejemplo se ejecute si encuentra el archivo que dejó el programa Seleccionador, el cuál cuando empieza lo crea desde cero. Entonces si está el archivo y está completo se ejecuta el Sort y sino no. Pero cuidado, porque el archivo podría existir pero estar trunco.

Lo que tendría que hacer cada programa antes de correr es fijarse si el anterior corrió y finalizó bien.

O sea, tiene que fijarse si él mismo terminó mal o no, y también tiene que fijarse si el programa anterior terminó mal o bien.

Si implementáramos este control de secuencia en un archivo, ¿qué le podríamos poner a ese registro del archivo para saber si el anterior terminó y si éste (él mismo) terminó?

¿Qué les parece esto? Cada vez que uno de estos programas se corre, le grabo en este archivo su fecha y hora de corrida. Cuando termina le grabo en qué fecha y hora se terminó de ejecutar. Si no termina quedará la fecha y hora que tenía de la corrida anterior.

El proceso Selección se ejecutó a las 10hs, a las 11hs se ejecutó Ordenar y a las 12hs se ejecutó Imprimir y esta secuencia se ejecuta varias veces en el día.

¿Cómo nos damos cuenta en este ejemplo de cuál sería el programa próximo que hay que correr? El que tiene la menor hora.

Y si terminó mal va a tener la fecha de antes de haberse corrido mal. Y va a haber que correrlo porque terminó mal

Para que esto funcionara tendríamos que tener acceso a este archivo por hora menor, o sea, por ejemplo, poner fecha y hora como clave secundaria, si fuera un archivo indexado. Buscaría la clave secundaria menor y ahí encuentra el programa que hace más tiempo que no se corre.

Lo que interesa es que ustedes tengan en claro que si implementan un reenganche en un proceso batch, tiene que controlar la secuencia de ejecución de las tareas del mismo, porque si no es así, no sabe qué procesos tienen que ejecutarse y cuáles no.

Al margen del reenganche interno de cada uno, lo que tiene que controlar es que se cumpla la secuencia.

Queremos que de algún modo el “reenganchador” sepa qué proceso debe correr.

Cuando el proceso batch empieza de vuelta, este proceso batch que era un conjunto de tareas (en este caso tres), debe saber cuál tiene que ejecutarse y cuál no.

Con todo lo hablado tenemos en claro las previsiones que deberíamos implementar para tener reenganche en un proceso batch.

Optimización de código

Ahora que tenemos criterios para prevenir las caídas de nuestros sistemas, podemos detenernos a examinar el código de nuestros programas.

Cada vez que vamos a escribir un programa, diseñamos los pasos para que el mismo funcione correctamente.

Pero ocurre que aunque logremos un funcionamiento correcto, no siempre elegimos la lógica más conveniente. En general, no nos damos cuenta de esto hasta que llegue alguna queja de los usuarios, referente a demoras en la ejecución o a un uso demasiado intensivo de recursos (por ejemplo, que los demás programas de la red se vean afectados por la ejecución del nuestro).

Más adelante, hablaremos detalladamente sobre las técnicas para probar programas. Durante las pruebas pueden aparecer problemas como estos, pero en general ocurren en el entorno de producción pasado un tiempo del momento de implementación.

Son muchas las cosas que podemos revisar en nuestro código, pero entre ellas, las más importantes son los accesos a datos y la lógica de resolución.

De los accesos a los datos ya hemos hablado cuando nos referimos a las consultas a las bases de datos, y cuando tratamos el diseño en capas.

Pero no hemos hablado acerca de la lógica. Podría parecernos una obviedad, ya que la lógica ya la habíamos pensado cuando escribimos el programa. Si bien esto es cierto, podríamos ver que la lógica elegida para la resolución no resultó óptima y debería ser mejorada.

A esta altura de la carrera que estamos cursando, ya entendemos de variables, de ciclos, de funciones y de subrutinas, pero debemos ver si en cada caso su uso es apropiado.

Debemos comprender, que una cierta lógica, por ejemplo una función o una consulta de datos, puede funcionar muy bien en una llamada aislada, pero no en un ciclo de llamadas de muchísimas iteraciones.

A continuación veremos un caso de estudio muy interesante, que servirá de ejemplo para las cuestiones que estamos tratando.

Caso de estudio: “el cuello de botella del servidor de correo”

En nuestra organización disponemos de una máquina que conecta nuestra red lan a internet. Los mensajes de correo electrónico llegan a esta máquina (decenas de miles de mails por día para aproximadamente 3000 personas) y luego son transferidos a la red interna.

Pero entre esos miles de mensajes diarios, había muchos que eran indeseados (spam). Entonces se probaron programas de filtro de spam (buscadores de coincidencias, “string matching tools”), pero como ninguno fue muy satisfactorio se decidió crear uno propio.

Luego de varias pruebas el servicio se instaló pero inmediatamente surgió un problema.

La máquina que distribuye el correo, que está muy ocupada, se vio sobrecargada debido a que el programa de filtro de spam tomaba más tiempo que el necesario para todo el otro procesamiento de cada mensaje, causando que las colas de mensajes se llenaran y la distribución de los mismos se retrasara por horas.

El filtro anti spam funcionaba asi:

Cada mensaje entrante se trataba como un string y un verificador de coincidencias contextual examinaba el string para ver si contenía frases tales como "haga millones en su tiempo libre" y otras. Estas frases consideradas “de spam” estaban guardadas en un archivo de “patrones de spam”
.
Los mensajes tienden a ser recurrentes y por lo tanto esta técnica resultó efectiva. Y si un mensaje de spam no era capturado, se agregaba la frase a la lista de frases de spam para capturarlo la próxima vez.

El código original era muy simple: se fijaba si cada mensaje contenía alguna de las frases de spam (patrones)

```
  int issapm(char *mesg)
{
  int i;
  for (i=0; i<npat; i++)
    if (strstr(mesg,pat[i] !=null) {
       printf ("spam : coincide para ", pat[i] );
       return 1;
    }
  return 0;
}
```

¿Cómo podría hacerse más rápido? El string debe buscarse y la función strstr de la biblioteca del lenguaje C es la mejor forma de buscar.

La implementación de strstr es algo como esto:

```
 char *strstr(const char *s1, const char *s2)
{
  int n;
  n = strlen(s2);
  for (;;) {
    s1=strchr(s1, s2[0]);
    if (s1==null)
       return null;
    if (strcmp(s1, s2, n) == 0)
      return (char *) s1;
      s1++;
    }
}
```

Utiliza strchr para encontrar la próxima ocurrencia del primer carácter del patrón y luego llama a strcmp para ver si el resto del string coincide con el resto del patrón. Salta sobre la mayor parte del mensaje buscando el primer carácter del patrón y luego hace una recorrida rápida para chequear el resto.

¿Por qué esto debería correr lento?

Analicen el caso y propongan las razones.