UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

TECNICAS DE OPTIMIZACIÓN PARA LA CONSULTA DE INFORMACIÓN EN UN SISTEMA CONTABLE

INFORME DE ACTIVIDADES PROFESIONALES

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

P R E S E N T A:

Esaú Ugalde Vargas

ASESORA DE INFORME:

M.I. María del Socorro Guevara Rodríguez

Ciudad Universitaria, Cd. Mx., 2017

Agradecimientos

A mi madre y hermano Esther y Óscar, que sin su apoyo, confianza y amor no habría podido culminar mis estudios.

A mi directora de informe, María del Socorro Guevara, por su apoyo y entusiasmo en todo momento en el que escribía este trabajo.

A mi sinodal y profesor, Eduardo Espinosa, que con su conocimiento, paciencia y humildad me mostró una faceta más interesante del saber.

Contenido

Índice de figuras

Introducción

En el presente trabajo se mostrará la aplicación de conceptos y técnicas que aparecen en los temarios de materias pertenecientes al mapa curricular 2010 de la carrera de Ingeniería en Computación Facultad de Ingeniería (UNAM), con el fin de optimizar los módulos (se utilizará la palabra módulo para referirse a alguna parte del sistema con una tarea específica) de un sistema contable al que llamaré a partir de ahora Sistema ~~para el~~de Registro de Gastos y Percepciones ~~(SRGP)~~. Se describirá cómo se aplicaron tales conceptos para mejorar los métodos de consulta de la información almacenada en el ~~SGRP~~SRGP a causa de que su volumen se incrementó más allá de los previsto y la infraestructura no estaba diseñada inicialmente para consultar (pero sí para almacenar) una gran cantidad de datos. Los conceptos que puse en práctica están relacionados estrecha con distintas asignaturas del mapa curricular pero muy en especial con las materias de Algoritmos y Estructuras de Datos, Ingeniería de Software, Bases de Datos, entre otras.

En el presente trabajo explicar~~é~~emos la manera en que apliqué los conocimientos aprendidos en las asignaturas mencionadas para lograr un buen comportamiento y fácil mantenimiento de algunos módulos del SRGP. Serán frecuentes los comparativos entre un antes y un después de someter a un análisis riguroso situaciones en las que hasta un cierto momento no se había aplicado un proceso óptimo de diseño y desarrollo o no se había previsto que su manejo requería mejores técnicas de tratamiento de los datos.

Cabe destacar que, dada la generalidad de las técnicas desarrolladas, se podrá notar que no solo aplican para el SRGP, sino que también aplican para otro tipo de sistemas que requieran explotación intensiva de información y que tengan menos recursos que un sistema diseñado con dicho fin.

Objetivo

Mi objetivo a cumplir fue desarrollar soluciones reutilizables para el SRGP, que pu~~d~~edan solventar su necesidad de consultar grandes cantidades de información en poco tiempo y sin agotar los recursos de los equipos de cómputo (servidores) utilizados para su ejecución.

Las técnicas deb~~í~~ía~~e~~n de ser reutilizables y ajustables en el desarrollo del SRGP, ya que se este se ~~debe~~ actualizaría constantemente ena ~~los~~ distintos momentos incluso posteriores ~~a este reporte de actividades~~al presente trabajo y posiblemente durante algunos años más~~,~~ dadas sus dimensiones.~~—~~ Es muy probable que aún ahora (2017) ~~surgieran~~ surjan nuevos módulos del sistema ~~-~~que requ~~ji~~ri~~i~~eran la extracción de información para presentarla en forma de reportes.

Capítulo 1 La organización

El Instituto para la Organización de Elecciones (IOE)

Se trata de un organismo público encargado de organizar las elecciones a distintos niveles.

Misión

Organizar procesos electorales que sean democráticos y fidedignos, para hacer que se obedezcan los derechos electorales de los ciudadanos.

Visión

Ser un organismo a través del cual se puedan efectuar los procesos electorales de manera confiable, democrática e imparcial en la que toda la información se encuentre al alcance de la ciudadanía.

El puesto desempeñado

En la organización laboré como Profesional en Desarrollo de Software desde el 1 de marzo de 2016 hasta el 15 de marzo de 2017 en el SRGP. Los objetivos del puesto son analizar, diseñar y construir los sistemas de información, que en mi caso fue el ~~SGRP~~SRGP, con base en una arquitectura establecida previamente, tomando en cuenta las necesidades de los usuarios y las normas de la organización; con la finalidad de desarrollar sistemas de calidad, siguiendo los patrones de diseño y estándares de desarrollo de software establecidos.

A continuación en la Figura 1.1 se muestra el organigrama del área en la que me desempeñé. Laboré en el Departamento de Desarrollo de Sistemas

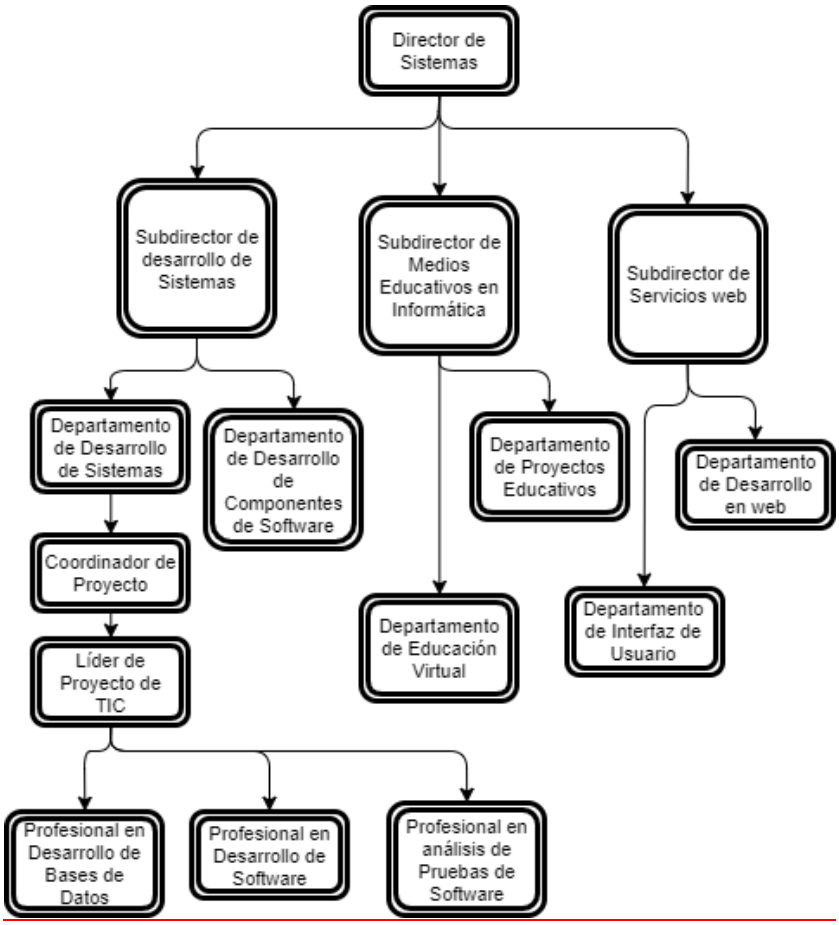


FIGURA 1.1 ORGANIGRAMA DEL ÁREA EN LA QUE LABORÉ. FUENTE: ELABORACIÓN PROPIA

El desarrollo del SRGP se realizaba al mismo tiempo ~~en el~~ que se definían distintas reglas de negocio para agregar módulos que formarían parte de este en el futuro o modificar los módulos ya existentes.

La metodología de desarrollo puesta en práctica es conocida como *Scrum*. *Scrum* es un marco de trabajo enfocado al desarrollo incremental y mantenimiento de productos complejos. Este marco de trabajo es adecuado para este tipo de

Con formato: Conservar con el siguiente

Con formato: Fuente: Cursiva

Con formato: Epígrafe, Centrado Sangría: Primera línea: 0 cm, Interlineado: sencillo

Con formato: Fuente: Cursiva

Con formato: Fuente: Cursiva

desarrollo de software ya que permite la creación de productos de gran calidad cuyos requerimientos cambien[1].

Es importante destacar que cuando comencé a laborar en el IOE como desarrollador del SRGP la gran mayoría del sistema ya estaba implementado (pero no terminado debido a que podrían cambiar los requerimientos establecidos o agregar nuevos), por lo tanto, no tuve la necesidad de analizar requerimientos para un módulo nuevo (pero sí analizar requerimientos nuevos para un módulo existente). A pesar de ello, el presente trabajo se enfocará más en la optimización de partes del sistema ya existentes cuyo rendimiento era pobre.

Hay que resaltar que, dado lo anterior, el esquema de base de datos ya había sido diseñado e implementado. Por otro lado, no figuraba en los objetivos de mi puesto la implementación, afinación, modelado y otro tipo de tareas concernientes a las bases de datos (no formaba parte de los objetivos, pero no prohibía mi participación en actividades que, por supuesto, no involucraran cambios realizados directamente por mí en el catálogo del sistema). Dichas tareas eran asignadas a administradores de bases de datos.

[1] Schwaber, K. & Sutherland, J. (2013, julio) "*La Guía de Scrum*" Recuperado de http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-es.pdf

Capítulo 2 Antecedentes

2.1 El Sistema de Registro de Gastos y Percepciones (SRGP)

ElLa organización IOE requería la creación de un sistema web con el que los sujetos obligados pudieran registrar de una manera homogénea los gastos que realizan, así como los ingresos que reciben. Y que de estos tipos de gastos e ingresos se generaran reportes en los que se vieran reflejados los registros que efectuaron. Por lo que la organización necesitaba garantizar que el dinero proveído a los sujetos obligados no fuera mal gastado, y estos presentaran informes para dar a conocer el origen y destino de sus recursos. De este modo fue concebida la idea de crear el SRGP.

Como se mencionó anteriormente, la organización a grandes rasgos solicitaba la implementación de un sistema web en el que se registraran operaciones de ingresos y de gastos de los sujetos obligados de todo el país; y que de alguna manera estaas se vieran reflejadaos por medio de ciertos formatos estandarizados en reportes e informes.

Durante mi estancia en la organización me he encargado del desarrollo, mantenimiento y actualización de distintos módulos del SRGP que involucran la generación de reportes, que reflejaran la información existente registrada en la base de datos. Asimismo, tuve como tarea dar mantenimiento y actualizar otros módulos del SRGP que me fueron asignados en los que se requería la captura de otro tipo de registros, sin embargo, estos temas no se abordaráón en este trabajo ya que la actualización y mantenimiento la implementación de dichos módulos no requería optimización alguna.

A grandes rasgos, el SRGP debe de seres una aplicación web en la que los usuarios pueden registrar sus operaciones de ingresos y egresos. No solo con el nivel de detalle descrito, sino que también tengan tienen la oportunidad de especificar claramente los orígenes y destinos del dinero que registrarean. Por ejemplo; podría existir alguna clase de aportante que pudiera donar dinero a un sujeto obligadola institución, a este aportante se le asociaría un identificador,

Con formato: Fuente: Sin Cursiv

entonces un usuario debería registrar esa cantidad en alguna categoría disponible en el sistema, por ejemplo, *Aportantes* e indicar el identificador asociado del aportante. Pero en otro módulo del sistema alguien tendría que registrar antes a dicho aportante para que el usuario pudiera usarlo como origen del dinero registrado.

Esto es solo una pequeña porción del sistema ya que existían otras en las que los orígenes y destinos del dinero fueran diferentes, por ejemplo, se podría especificar que el dinero proviene de transferencias que se realizan entre entidades del mismo sujeto obligado. Imagine que existe el Partido para México y que tiene distintas sedes en cada estado del país, pero que tiene una sede principal en la Ciudad de México. Ahora suponga que alguna organización otorga presupuesto al Partido para México (PM)~~,~~ para realizar gastos de campañas. Dicha organización otorga dinero a cada sede de este partido, pero da más dinero a la cede de la Ciudad de México. Esta última puede transferir dinero a la sede de Guanajuato o cualquier otro estado. Entonces el sistema podría almacenar este tipo de movimientos en alguna categoría llamada, por ejemplo, *Transferencias*.

Más aún, imagine que el PM de algún estado realiza una clase de evento como un convivio por su campaña. El sistema ofrecía la oportunidad de almacenar este evento y de relacionar posteriores gastos con este evento. ~~Por~~ Como ejemplo, suponga que el PM compró 30 000 gorras para regalar en el evento y gracias a ello tuvo egresos por $6,000,00. Algún usuario (capturista) que perteneciera al PM registraría este egreso y podría relacionarlo con el evento.

A pesar de que el SRGP era de grandes dimensiones y tenía muchos módulos, estuve más enfocado a los módulos especializados en la recuperación de la información almacenada en el sistema. Esta información era extraída en forma de reportes como se indica en secciones posteriores.

La extracción de datos es sencilla cuando existe poca información. Sin embargo, conforme esta se va ~~creciendo~~ acumulando deben buscarse técnicas

más eficientes tanto en espacio como en tiempo. Un requerimiento muy importante es que un usuario no espere ~~veinte minutos~~un largo tiempo para obtener algún tipo de reporte, y tampoco es aceptable que la técnica de generación de dicho reporte consuma todos los recursos del servidor, ya que no solamente existe un usuario.

Es importante indicar el tipo de servidores utilizados. Hay dos clases de servidores sobre los que se ejecuta aún (2017) el SRGP: de aplicación y de bases de datos. Los equipos para ejecutar la aplicación usan el sistema operativo Red Hat con el servidor JBoss y el manejador de base de datos Oracle. Las versiones y configuraciones son tareas correspondientes a otra área, misma que proporciona estos servidores. Como al inicio del proyecto (esto es antes de que yo comenzara a laborar en él) no se pudo prever la tasa de crecimiento del sistema, el personal del área mencionada solamente otorgó 10 servidores, aunque estos eran virtuales. A pesar del crecimiento del sistema, por asuntos de índole administrativa, no era posible modificar las características de hardware de los equipos así como proporcionar más servidores físicos.

Con formato: Sangría: Primera línea: 0 cm

Capítulo 3 El problema del Reporte de Suma de Registros

3.1 El Reporte de Suma de Registros

En las etapas iniciales de desarrollo del sistema el volumen existente de información no era muy grande, por lo que las técnicas para la generación adecuada de formatos que mostraran en forma deseada la información eran simples.

Las reglas que se habían definido para la implementación de los módulos existentes podían modificarse para agregar nuevas funcionalidades o cambiar el flujo propio del módulo.

~~Sentadas~~ Propuestas las bases anteriores, entre todos los módulos que formaron parte del sistema existían algunos dedicados de forma exclusiva a la generación de reportes con base en la información guardada en la base de datos. En las siguientes páginas se describirá el problema encontrado en la técnica de generación de cada reporte y la manera en la que contribuí para solucionar o reducirlo al mínimo aceptable (esto último debido a que en algunos casos no era posible desaparecer el problema debido al tiempo con el que se contaba o el hecho de que la tecnología utilizada no proveía los recursos suficientes).

El primer reporte, que~~, al que~~ ~~denominaremos~~ se denominará Reporte de Suma de Registros~~, necesitaba la presentación de la forma en que se indicará más adelante~~ debía presentar en un formato ya establecido la información cuya naturaleza se explica a continuación. Suponga que al usuario final del sistema se le presenta una pantalla en la que él podía registrar sus gastos e ingresos en *n* categorías, y cada una de ~~ellas~~ esas categorías está formada de subcategorías~~,~~ al mismo tiempo estas podrían o no tener subcategorías~~, y~~ así en lo sucesivo, como se presenta en la ~~siguiente f~~Figura 3.1~~.~~

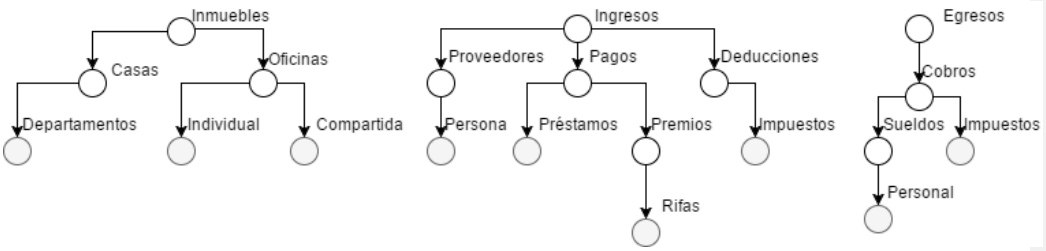


FIGURA 3.1~~3.1~~ ORGANIZACIÓN DE CATEGORÍAS PARA CLASIFICAR REGISTROS. FUENTE: ELABORACIÓN PROPIA

La ~~f~~Figura 3.1 muestra una forma simplificada de la estructura de árbol que existía en el sistema, la que en realidad era muchas veces más grande, también las categorías mostradas son distintas de las que se utilizaban en realidad en el sistema. A pesar de lo anterior, la ~~f~~Figura 3.1 es suficiente para ilustrar la estructura básica de~~,~~ la organización de las categorías presentadas en el Reporte de Suma de Registros y presentar el problema que surgió en con el crecimiento del volumen de información.

Suponga que al usuario solo se le presentan las subcategorías inferiores mostradas en gris para poder introducir registros en el sistema. Así, el usuario tiene que introducir los gastos o percepciones que tuvieron lugar a lo largo de un periodo dado de tiempo en la categoría correspondiente. El Reporte de Suma de Registros necesitaba presentar la suma de los registros introducidos en forma categórica, es decir, mostrar las cantidades en las categorías ~~inferiores~~inferiores, así como la suma acumulada en las categorías superiores, que sería igual al sumatorio de las categorías inferiores o *hijas*. Un ejemplo de la estructura básica del Reporte de Suma de Registros se muestra en la Figura 3.2~~seguida~~.

Categoría	*Monto*
Inmuebles	$1,500,000.00
Casas	$300,000.00
Departamentos	$300,000.00
Oficinas	$1,200,000.00
Individual	$200,000.00
Compartida	$1,000,000.00
Ingresos	$1,260,000.00
Proveedores	$0.00
Persona	$0.00
Pagos	$60,000.00
Préstamos	$50,000.00
Premios	$10,000.00
Rifas	$10,000.00
Deducciones	$1,200,000.00
Impuestos	$1,200,000.00
Egresos	$4,000,000.00
Cobros	$4,000,000.00
Sueldos	$1,000,000.00
Personal	$1,000,000.00
Impuestos	$3,000,000.00
Total	$6,760,000.00

FIGURA 3.2~~3.2~~ EJEMPLO SIMPLIFICADO DEL REPORTE DE SUMA DE REGISTROS. FUENTE: ELABORACIÓN PROPIA

Al igual que con la ~~f~~Figura 3.1 que muestra las categorías en árbol, la estructura del reporte en la F ~~f~~igura 3.2 fue simplificada para evitar mostrar los detalles de visualización y estructura del sistema real, pero también es suficiente para ~~dar~~ expresar ~~la idea d~~el problema y para ~~dar una explicación~~explicar ~~de~~ las metodologías anteriores y posteriores para la generación del Reporte de Suma de Registros. Note que el monto mostrado en la categoría padre es igual a la suma de los montos de sus hijos. Y el padre del padre muestra un monto igual a la suma de sus hijos que pueden ser padres o categorías base.

3.1.1 Generación anterior del reporte

Comentario [MDSGR1]: Puedes poner 3.1.1

Con formato: Título 3, Esquema numerado + Nivel: 3 + Estilo de numeración: 1, 2, 3, … + Iniciar en 1 + Alineación: Derecha + Alineación: 1.2 cm + Sangría: 1.3 cm

~~Primeramente~~Primeramente, hay que señalar que tanto el árbol mostrado en la ~~f~~Figura 3.1 como los montos registrados se guardaron en una base de datos utilizada por el sistema. El propio usuario podía registrar muchos movimientos en una categoría; por ejemplo, registrar diez movimientos en la categoría Rifas, y

cada uno de ellos tendría un monto asociado. Así pues, para determinar la suma de la categoría era necesario considerar el monto de cada movimiento.

Categoría	Tipo de movimiento	Concepto del movimiento	Fecha de registro	Monto	Año
Préstamos	Diario	Préstamo a deudor Juan	04/12/2016	$50,000	2016
Préstamos	Diario	Préstamo mensual	05/05/2016	$200,000	2015
Préstamos	Ajuste	Préstamo trimestral	02/10/2016	$3,000,000	2016
Impuestos	Diario	Pago de agua potable	10/10/2016	$10,000	2016
Impuestos	Ajuste	Pago de luz eléctrica	05/01/2017	$30,000	2016

FIGURA 3.3~~3.3~~ EJEMPLO DE LOS MOVIMIENTOS QUE PODÍA REGISTRAR EL USUARIO. FUENTE: ELABORACIÓN PROPIA

El método que se describirá en seguida fue desarrollado antes de que me fuera asignado el módulo del sistema para generar el Reporte de Suma de Registros. P~~P~~ara obtener el reporte se ~~aprovech~~aprovechaba~~ó~~~~aba~~ el motor de base de datos cuyo lenguaje de consulta es SQL[2], y con este se ~~hacía~~ realizaba la suma de los montos. Se utilizaba una consulta para que el DBMS[3] obtuviera los montos de las categorías, ya fueran base oya ~~fueran categorías padre~~categorías padres. Cuando el número de registros era pequeño esta funcionaba de forma eficiente; con aproximadamente 10 000 registros, no tardaba más de unos segundos. Con el paso de los meses, un problema con la forma de realizar la consulta comenzó a hacerse patente, ya que el volumen de información crecía desmedidamente. La consulta funcionaba de una forma similar a la siguiente: suponga que se requiere determinar el total de la categoría *Ingresos*, para ello es necesario calcular los montos de sus tres categorías hija~~:~~ *Proveedores*, *Pagos* y *Deducciones*~~,~~ y a su vez para calcular estas últimas es necesario conocer los montos de sus categorías hija. La forma en que funcionaba la consulta era calculando el total de cada categoría *siempre*, por ejemplo: *Ingresos* = *Proveedores* + *Pagos* + *Deducciones* = *Persona* + *Préstamos* + *Premios* + *Impuestos* = *Persona* + *Préstamos* + *Rifas* + *Impuestos*. Este cálculo toda vez se

[2] Del inglés Structured Query Language (Lenguaje de Consulta Estructurado)
[3] Del inglés Data Base Management System

Con formato: Español (México)

repetía, es decir para calcular *Ingresos* se sumaba todo su árbol inferior, y para calcular uno de sus subárboles *A*, se sumaba todo el subárbol hijo de *A*. Y si el árbol tenía *N* nodos, el cálculo de suma de subárboles se repetía *N* veces.

Para ilustrar el problema, suponga que el árbol es *balanceado*, es decir, que cada categoría tiene el mismo número de hijos, por decir, tres, como se muestra en la ~~figura~~ Figura 3.4.

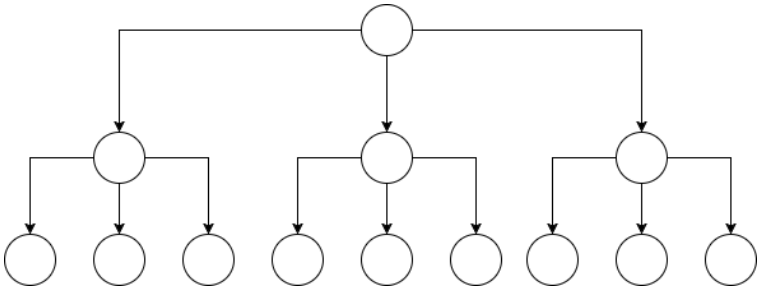


FIGURA 3.4~~3.4~~ ÁRBOL BALANCEADO DE CATEGORÍAS. FUENTE: ELABORACIÓN PROPIA

La forma anterior en la que se obtenía el total de una categoría era sumando siempre sus subárboles, eso con pocos registros ~~en~~ el sistema tiene un buen rendimiento, pero este decrece cuando aumenta el número de categorías a ser sumadas. Por ejemplo, suponga que un usuario solamente registró $ 300 en la categoría *Individual* pero el Reporte de Suma de Registros tiene que mostrar el total de todos los niveles, a saber, el total en *Oficinas* e *Inmuebles* que son el padre y el padre del padre de *Individual* respectivamente, pero como el usuario solo registró $ 300, en las tres categorías se verá reflejado un total de $ 300. El problema radica ~~en que~~que, para efectuar este cálculo, el sistema mandaba a efectuar con el identificador de la categoría la suma, para este ejemplo, la base de datos buscaba por el identificador de *Inmuebles*, y sumaba su árbol inferior; luego buscaba por el identificador de *Oficinas* para después sumar su árbol inferior. Por último, buscaba ~~por~~ el identificador de *Individual* y obtenía su total (este no tiene árbol inferior). Esto introduce una gran cantidad de redundancia a nivel cálculo. De esta forma, se realiza la suma de una misma categoría con una profundidad *m*

(suponga que para la parte superior del árbol la profundidad es cero) *m + 1* veces~~:~~; para *Inmuebles*, una vez, para *Oficinas*, dos y para *Individual*, tres veces. Por fortuna, la jerarquía de las categorías en el sistema no tenía muchos niveles, solamente *m = 4* (es decir, 5 niveles); sin embargo, había *k* categorías raíz, no solo una.

Cabe resaltar que el usuario podía registrar *t* movimientos en una categoría base, como se comentó anteriormente, ~~dijo más arriba~~ por ejemplo~~:~~, registrar diez movimientos en *Rifas*. Para generalizar el problema, aquí se utilizará *t*. La complejidad computacional de la suma de una lista lineal de objetos es *O(n)*, es decir, proporcional al número de objetos (aquí la palabra objeto no se utiliza con la misma connotación usada en el contexto de la programación orientada a objetos). Suponga que cada movimiento registrado en la base de datos es un objeto, y que un usuario registró *t* movimientos en cada categoría base, es decir, *t* objetos, y que cada objeto toma un tiempo *q* para ser sumado. Por ende, para sumar *t* movimientos, el tiempo total era *t * q*, solo para una categoría base. Para las categorías que no son base, el obtener su total era un tiempo *s * q*, ya que cada categoría solo tenía *s* objetos como hijos.

La siguiente expresión da el total del tiempo que utilizaba esta técnica para el caso en el que cada categoría tuviera *s* categorías hija.

$$q_{total} = t \cdot q \cdot (m+1) \cdot s^m + s \cdot q \cdot \sum_{i=0}^{m-1} (i+1) \cdot s^i$$

ECUACIÓN 1

Comentario [MDSGR2]: Puedes llamarla como ecuación 1, o algorit 1

Con formato: Conservar con el siguiente

Con formato: Centrado

Se trata de una expresión simplificada ya que cada categoría en realidad tendría *s* o menos hijos y en cada categoría base el número de movimientos no sería siempre el mismo. Sin embargo, sigue siendo una buena aproximación de cómo funcionaba en realidad. ~~Además~~Además, *q* es una unidad de tiempo genérica, puede ser un segundo, un milisegundo o algún otro intervalo. Para el

ejemplo siguiente, supondremos los siguientes valores para cada una de las variables: *t = 1000, q = 10 [ns], m = 4* y *s = 4*. Sustituyendo los valores en la expresión se llega a un total de 12 801 081 nanosegundos, o aproximadamente 12.8 milisegundos. Relativamente poco tiempo. Hay que señalar que no se toman en cuenta factores que harían crecer la complejidad del problema, como los asuntos de la jerarquía de memoria de la máquina, puesto que los tiempos de acceso de una máquina varían de acuerdo con el tipo de memoria que se esté utilizando: los registros del CPU tienen tiempos de acceso de nanosegundos; la caché, que es pequeña, pero muy veloz; la memoria principal que es de mediana velocidad y la memoria secundaria, de baja velocidad de respuesta[4]. Además, existen optimizaciones que hacen algunos DBMS sobre el plan de ejecución de una consulta; por ejemplo, hay manejadores de bases de datos que utilizan estadísticas sobre las tablas e índices sobre los que se hicieron referencia en las consultas[5].

Comentario [MDSGR3]: Que es?

En la ~~expresión~~ Ecuación 1~~mostrada~~ el término que tiene mayor peso es el de la izquierda del lado derecho de la igualdad, $t \cdot q \cdot (m+1) \cdot s^m$; este término se incrementa en función del número de movimientos. ~~Así se está tomando en cuenta solo el caso ideal en el que las operaciones se realizaran en un tiempo constante de cálculo~~Aquí se está suponiendo que se están realizando las operaciones en un tiempo constante de cálculo.

[4] Tanenbaum, A. (2009: 175) *Sistemas Operativos Modernos* México: Pearson, Prentice Hall.
[5] Nanda, A. (2009, marzo) "Líneas de Base y Planes más Favorables". Extraído de http://www.oracle.com/technetwork/es/articles/sql/o29spm-098177-esa.html

Capítulo 4 Méetodología para resolver el problema del Reporte de Suma de Registros

Debido al problema que se presentó se tuvo que incluir su solución como un objetivo del siguiente *sprint* lo que ocurrió cuando se hizo la planificación del mismo (claro que en el *sprint* había objetivos de otros módulos del sistema de los que se encargarían de cumplir otros desarrolladores).

Con formato: Fuente: Sin Negrit Cursiva, Color de fuente: Automático, Sin Versalitas

Con formato: Fuente: Sin Negrit Cursiva, Color de fuente: Automático, Sin Versalitas

Ahora se presentará la reestructuración a nivel técnico que ~~se hizo~~desarrollé para generar el Reporte de Suma de Registros. La diferencia entre el nuevo algoritmo y el anterior es que se realizan primero las sumas de la parte inferior del árbol, el resultado se va propagando hacia arriba, es decir, se obtiene el total de todas las categorías base y se agrupan las que tengan el mismo padre *A*, se suman sus totales, lo que nos dará otro conjunto de totales. De ese propio conjunto se agrupan las que tengan el mismo padre *B*, se suman sus totales, lo que al igual que antes nos dará otro conjunto de totales. Así en lo sucesivo, hasta llegar a las que no tengan padre, que esas serían las categorías de la parte superior del árbol. Para nuestro ejemplo, suponga que un usuario introdujo al sistema movimientos en todas las categorías, y que el conjunto inicial de resultados es el siguiente:

$$A = \{Departamentos = \$400, Individual = \$1{,}000, Compartida = \$700,$$
$$Persona = \$3{,}000, Préstamos = \$\,850,$$
$$Rifas = \$1{,}200, Impuestos_1 = \$8{,}000,$$
$$Personal = \$10{,}000,\ Impuestos_2 = \$10{,}000\}$$

Con formato: Conservar con el siguiente

ECUACIÓN 2

Con formato: Centrado

Note que solo se muestran los totales pertenecientes a las categorías base. Ahora según el algoritmo descrito más arriba, se deben agrupar los elementos que tengan un padre en común en *A*, y posteriormente sumar sus resultados.

Comentario [MDSGR4]: Ecuació algoritmo 1

$$B = \{Casas = \$400, Oficinas = \$1{,}700, Proveedores = \$3{,}000,$$
$$Pagos = \$850, Premios = \$1{,}200, Deducciones = \$8{,}000,$$
$$Sueldos = \$10{,}000, Cobros = \$10{,}000\}$$

ECUACIÓN 3

Nuevamente, se repite el algoritmo: se deben agrupar los elementos que tengan un padre en común en *B* y posteriormente sumar sus resultados.

$$C = \{Inmuebles = \$2{,}100, Ingresos = \$5{,}050, Egresos = \$10{,}000\}$$

ECUACIÓN 4

A partir de aquí~~,~~ ocurre un problema silencioso; para ilustrarlo, observe el conjunto *C*, en especial al total de *Egresos* de $10,000. Es incorrecto, ya que las subcategorías *Impuestos* y *Personal* son descendientes de esta categoría y su total debería ser $20,000. Aquí es muy importante notar que el número de agrupaciones que se hicieron debe ser igual a la mayor profundidad del árbol más uno. Porque en dado caso de que existiera un monto en la categoría más baja, ya sea *Rifas* o *Personal*, el agrupar tres veces (la profundidad de estas categorías es dos) propagará este monto hasta la parte superior del árbol. Con esto como base, para obtener los verdaderos totales de cada categoría hace falta calcular otro conjunto resultado, que denominaremos *U*:

Con formato: Conservar con el siguiente

Con formato: Centrado

Comentario [MDSGR5]: Ecuació algoritmo 1

Con formato: Conservar con el siguiente

Con formato: Centrado

$$U = A \cup B \cup C = \{Departamentos = \$400, Individual = \$1{,}000,$$
$$Compartida = \$700, Persona = \$3{,}000, Préstamos = \$\,850,$$
$$Rifas = \$1{,}200, Impuestos_1 = \$8{,}000,$$
$$Personal = \$10{,}000, Impuestos_2 = \$10{,}000, Casas = \$400,$$
$$Oficinas = \$1{,}700, Proveedores = \$3{,}000, Pagos = \$850,$$
$$Premios = \$1{,}200, Deducciones = \$8{,}000, Sueldos = \$10{,}000,$$
$$Cobros = \$10{,}000, Inmuebles = \$2{,}100, Ingresos = \$5{,}050,$$
$$Egresos = \$10{,}000\}$$

Con formato: Conservar con el siguiente

ECUACIÓN 5

Con formato: Centrado

En este último conjunto hay que repetir la operación de agrupación para los elementos que tengan el mismo padre y sumar los montos; esta es la última vez que se necesita esta operación, lo que nos da un conjunto *T*:

$$T = \{Departamentos = \$400, Individual = \$1{,}000, Compartida = \$700,$$
$$Casas = \$400, Oficinas = \$1{,}700, Inmuebles = \$2{,}100,$$
$$Persona = \$3{,}000, Préstamos = \$\,850,$$
$$Rifas = \$1{,}200,\ \ Impuestos_1 = \$8{,}000, Proveedores = \$3{,}000,$$
$$Pagos = \$2{,}050, Deducciones = \$8{,}000,$$
$$Personal = \$10{,}000, Impuestos_2 = \$10{,}000,$$
$$Sueldos = \$10{,}000, Cobros = \$20{,}000, Egresos = \$20{,}000\}$$

Con formato: Conservar con el siguiente

ECUACIÓN 6

Con formato: Centrado

Este conjunto nuevo sí contiene los totales requeridos para el reporte.

Suponga otra vez que el número de movimientos que registró el usuario en el sistema es igual, *t*; que cada categoría tiene *s* categorías hija y que la profundidad máxima para una categoría es *m*.

La siguiente expresión muestra el tiempo total en el que se realizaría el cálculo de las sumas para el reporte.

$$q_{total} = t \cdot q \cdot s^m + s \cdot q \cdot \sum_{i=0}^{m-1} s^i$$

ECUACIÓN 7

Nuevamente supondremos los siguientes valores para cada una de las variables: *t = 1000, q = 10 [ns], m = 4* y *s = 4*. Sustituyendo los valores en la expresión, se llega a un total de 2 563 400 nanosegundos, o aproximadamente 2.56 milisegundos. Esto representa una mejora en el tiempo de ejecución de hasta cinco veces. En la práctica la optimización fue incluso mejor.

En el ambiente productivo la consulta antigua gastaba tantos recursos de la base de datos que muchas veces no se era posible realizar un cálculo rápido ya que el tiempo de conexión se agotaba. En algunos casos su tiempo de ejecución llegaba a ser de más de 90 segundos o hasta cinco minutos, mientras que la nueva técnica tenía un tiempo de ejecución de 0.3 segundos (el tiempo era trescientas veces menor con un mejor uso de los recursos). Claro que esta diferencia entre el cálculo algebraico y los resultados que se obtuvieron en realidad pueden deberse a que SQL es un lenguaje declarativo, es decir, no se le *dice* a la computadora cómo realizar cierto ~~cálculo~~cálculo, sino que se le *dice* qué es lo que se quiere obtener, así que es muy probable que el DBMS hubiera realizado más optimizaciones para mejorar el rendimiento. Esto se trata de una aproximación al cálculo real.

Comentario [MDSGR6]: Ecuació o algoritmo 2

Con formato: Conservar con el siguiente

Con formato: Centrado

Comentario [MDSGR7]: ¿?

Implementé esta optimización; para las agrupaciones mencionadas en el algoritmo empleé la cláusula ***group by*** de SQL vista en el curso de Bases de Datos, aunque fue aplicada en combinación con reuniones externas (***outer join***) y otras operaciones relacionales que hacían falta para implementar la consulta. La solución fue llevada al ambiente productivo, lo que mejoró sustancialmente el funcionamiento del módulo del sistema que generaba el reporte.

Comentario [MDSGR8]: Cual?

Mejor aún, dado que lo que se encargaba de realizar este cálculo era el sistema de base de datos, entonces la proyección del resultado era igual a la proyección que realizaba el método anterior (la palabra proyección se usa en el sentido de bases de datos, recuerde que una operación de proyección es aquella que selecciona solamente los atributos especificados en una expresión de álgebra relacional[6]) así que solo fue cuestión de cambiar la consulta por la nueva sin necesidad de modificar código para el servidor.

A pesar de lo anterior, hubo cierto inconveniente sobre cómo funcionaba el código que generaba el reporte. Muy probablemente por cuestiones de tiempo se tuvo que replicar en muchos lados el mismo código para generar el Reporte de Suma de Registros, lo que resultó perjudicial pues como se ~~dijo más arriba~~comentó, los requerimientos de ciertos módulos podían cambiar con el tiempo, agregar características o quitar algunas. Esto se traducía en un problema pues cada vez que se agregara o quitara una característica habría que actualizar el código en todas las partes en las que se había replicado. Es bien sabido en ingeniería de software que la redundancia en el código aumenta la probabilidad de errores, así que por una buena práctica de programación y planeación siempre es bueno *refactorizar* bloques de código comunes en funciones.

Con formato: Fuente: Sin Negrit Cursiva, Color de fuente: Automático, Sin Versalitas

~~Debido a esto, *refactoricé* completamente el código de este módulo y lo coloqué en un grupo único de funciones para que solo se tuviera que actualizar~~

Con formato: Fuente: Sin Negrit Cursiva, Color de fuente: Automático, Sin Versalitas

[6] Silberchatz, A., Korth, H. y Sudarshan A. (2002: 59) *Fundamentos de Bases de Datos*: McGRAW-HILL

~~una sola sección con cada característica que se pudiera agregar en el futuro.~~Debido a lo anterior, el cumplimiento del objetivo del *sprint* se vio afectado, ya que los líderes de proyecto consideraron poco viable actualizar todas las réplicas del código de este reporte cada vez que se modificaran sus requerimientos. Por lo que el objetivo se modificó y ahora, aparte de la propia optimización del módulo tuve que *refactorizar* e incluso volver a codificar algunas de sus secciones.

Con formato: Fuente: Sin Negrit
Cursiva, Color de fuente:
Automático, Sin Versalitas

Capítulo 5 El problema de los Reportes de Detalle de Registros

5.1 El Reporte de Detalle de Registros

Diversos módulos del SRGP requerían la generación de otro tipo de reportes. Esta clase de reportes a diferencia del Reporte de Suma de Registros tenían que mostrar el detalle de los movimientos que había registrado el usuario, no solo la suma de estos. Incluso otros debían mostrar la suma de los registros junto con el detalle de los mismos.

Por otro lado, en otros módulos que formaban parte del SRGP el usuario tenía la oportunidad de registrar otro tipo de operaciones. Estas eran otras partes del sistema en las que el usuario podía registrar documentos que avalaran de cierta forma los registros que había efectuado. Ya sean~~fueran~~ fotos, ~~ya fueran~~ facturas o recibos, entre otros tipos de documentos. El usuario tenía la oportunidad de asociar una cantidad arbitraria de documentos a un solo registro.

Con esto se necesitaba que el sistema fuera capaz de ~~-~~crear reportes que contuvieran información sobre cada documento. El problema al parecer se reducía a solo generar una consulta al DBMS que recuperara los datos requeridos de cada registro. Un ejemplo ilustrativo sería como se muestra en la Figura 5.1.~~:~~

REPORTE DE DETALLE

Fecha de generación del reporte: 08/10/2016 21:42:15
Usuario que generó la descarga: Juan Rodríguez Pérez

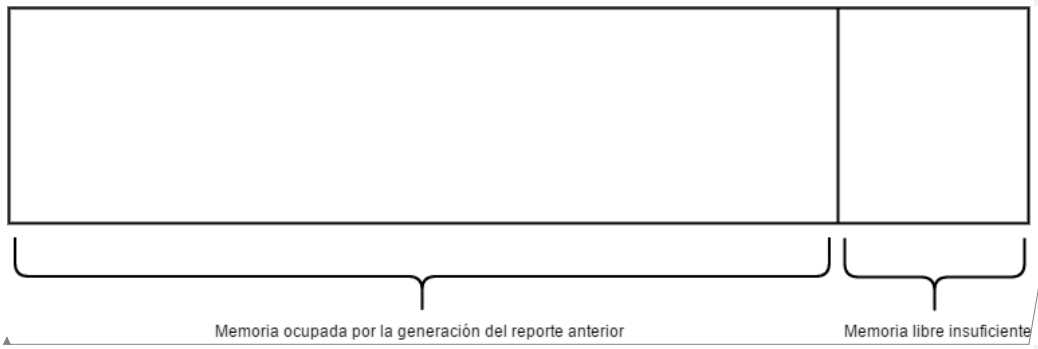
SUJETO OBLIGADO	PROCESO	AMBITO	ENTIDAD	FOLIO	PERIODO	ETAPA	MES	FECHA DE REGISTRO	ESTATUS
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	ENERO	10/05/2016 20:37:35	SIN EFECTO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	ENERO	10/05/2016 20:37:35	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	FEBRERO	11/05/2016 01:19:28	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	FEBRERO	11/05/2016 01:19:28	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	MARZO	11/05/2016 13:48:40	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	MARZO	11/05/2016 13:48:40	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	ABRIL	19/08/2016 15:16:07	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	MAYO	23/08/2016 12:42:25	SIN EFECTO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	MAYO	23/08/2016 12:42:25	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	MAYO	23/08/2016 12:42:25	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	JUNIO	22/08/2016 15:21:07	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	JUNIO	22/08/2016 15:21:07	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	JUNIO	22/08/2016 15:21:07	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	JULIO	12/08/2016 18:27:21	ACTIVO
PARTIDO FUNDAMENTAL	NORMAL	FEDERAL	JALISCO	1	2016	NORMAL	JULIO	12/08/2016 18:27:21	ACTIVO

FIGURA 5.1~~5.1~~ EJEMPLO DEL NIVEL DE DETALLE QUE SE PODÍA REQUERIR EN UN REPORTE. FUENTE: ELABORACIÓN PROPIA

En la fFigura 5.1 se muestra un ejemplo del nivel de detalle requerido para el reporte. El problema aparece cuando el volumen de información comienza a crecer desmedidamente. Cuando el número de registros en el sistema no sobrepasaba los 100 000 era viable aún usar las técnicas anteriores de generación.

Para ilustrar de mejor manera el problema, es importante destacar que el desarrollo del sistema era en Java, un lenguaje que utiliza la recolección de basura. Recuerde que la recolección de basura es el proceso de reclamar los trozos de almacenamiento a los que un programa ya no puede tener acceso[7].

La introducción del recolector de basura es una parte fundamental de muchos lenguajes actuales para librar al programador de la tarea de administrar la memoria mediante el código. Aunque tiene esta ventaja también tiene otras desventajas. Una desventaja del uso del recolector de basura es que puede ser un proceso muy lento. Es importante que no aumente en forma considerable el tiempo total de ejecución de una aplicación[8].



~~*FIGURA 5.2 MEMORIA INSUFICIENTE PARA ALMACENAR LOS NUEVOS OBJETOS. FUENTE: ELABORACIÓN PROPIA*~~

Con formato: Fuente: (Predeterminado) Arial, 12 pto, S Negrita, Color de fuente: Automático, Sin Versalitas

Comentario [MDSGR9]: Donde hablas de la figura 5.2

[7] V. Aho, S. Lam, Sethi y D. Ullman (2008: 465). *Compiladores. Principios, técnicas y herramientas*: Pearson. Addison Wesley.
[8] V. Aho, et al. (2008: 465).

-5.1.1 Generación anterior de este tipo de reportes

Con formato: Título 3

Al igual que con la generación del Reporte de Suma de Registros, el algoritmo anterior para la generación de estos reportes ya se había implementado. Anteriormente la generación de estos reportes era muy sencilla: se creaba una consulta que pudiera ejecutar el DBMS para obtener todos los registros necesarios, se realizaban ciertos cómputos sobre estos registros, se les aplicaba algún tipo de formato y finalmente se escribían en el archivo final.

Esta técnica funcionaba bien con poco volumen de información. Sin embargo, una vez que se sobrepasaron los 300 000 registros empezó a aparecer un problema: el recolector de basura parecía no ejecutarse con conveniencia. Al crear una vez el reporte este funcionaba de forma correcta, pero las veces siguientes el sistema no encontraba memoria para almacenar los registros traídos desde la base de datos porque la memoria aún estaba habitada por los objetos de la ejecución anterior, que aunque ya no fueran objetos alcanzables (en esteos términos contexto un objeto es alcanzable cuando aún puede llegar a referenciarse siguiendo una cierta ruta de *apuntadores* desde un conjunto de objetos conocido como conjunto raíz[9]). Los reportes de gran tamaño se volvieron difíciles de generar.

Y peor aún, al tratarse de un sistema web que era es usado a nivel nacional, este no tendríaía solamente un usuario que generara dichos reportes, podrían ser cientos de ellos. Como ejemplo, si cincuenta usuarios mandaban a hacer un reporte con 30-000 registros entonces iban a existir 1 500 000 registros en memoria del servidor en forma simultánea (por supuesto que todos tendrían que sincronizarse para que esto pasara). Y esto sin considerar que la generación

[9] V. Aho, et al. (2008: 466).

de reportes no era la única tarea que se podía realizar en el sistema, sino que podrían existir otros tipos, por ejemplo, introducción de información en el sistema.

Capítulo 6 Méetodo~~logía~~ para resolver el problema de los Reportes de Detalle de Registros

Como los problemas que surgieron averiaban el funcionamiento de los módulos de generación de estos reportes, su solución se tuvo que agregar a los objetivos de un *sprint*. En la junta de planificación de consideraron distintas opciones para solucionarlos, como una paginación de los registros, tratar de convencer a los usuarios del sistema de modificar las reglas de negocio para la generación de estos reportes, simplificar la estructura del reporte, entre otras soluciones. Como finalmente los usuarios no accedieron a modificar los requerimientos del reporte, se optó por la primera opción.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Es importante aclarar que las soluciones que se abordaron definitivamente no eran las mejores debido a que los administradores de bases de datos se encontraban trabajando en otras partes del sistema, por lo que no tenían tiempo de realizar las afinaciones pertinentes a la base de datos.

~~Hay varias soluciones que pueden ayudar a solventar los problemas que surgieron para la generación de los Reportes de Detalle de Registros. Una de ellas es~~Cabe mencionar que se podrían haber agrega~~a~~ado~~r~~ más recursos al sistema, de tal manera que existi~~era~~~~a~~ memoria suficiente y que la velocidad de procesamiento también ~~sea~~ fuera adecuada. Sin embargo, esta solución no era viable ya que requería un cambio en la infraestructura, pero como se explicó antes, no era posible ~~y no garantizaba que en el futuro siguiera siendo suficiente~~. Además de que esta solución es poco elegante. Así que solo quedaba buscar una técnica definitiva que permitiera solucionar el problema sin importar cuánto pudiera crecer la cantidad de información (claro que con límites en función de la finalidad del sistema).

~~Otra de las posibilidades involucraba cambiar la forma en que se generaban estos reportes; buscar mejores algoritmos.~~

~~Una solución~~Así que la solución en ese momento sería no cargar todos los registros a memoria, sino hacer una carga lenta de ellos (paginarlos): cargar unos

cuantos miles de registros, hacer un poco del reporte y volver a cargar otros miles para hacer otro poco del reporte. De esta forma hasta terminar.

Así implementé una clase genérica que se encargara de recuperar los registros poco a poco, de tal manera que ofreciera una interfaz de trabajo sencilla para todos los algoritmos que se utilizaran para generar los Reportes de Detalle de Registros y que se evitara implementar una y otra vez los algoritmos que recuperaran los registros de la base de datos de forma paginada. Esta clase solo cargaba en la memoria de la computadora unos pocos registros y cuando se terminaban de procesar, se cargaban más. En el algoritmo de generación del reporte se podía asumir que todos los registros estaban en la memoria aunque no fuera así.

Había dos funciones comunes que formaban parte de la interfaz de dicha clase y que los algoritmos de generación de reportes podían usar para *iterar* sobre el ~~set~~conjunto de resultados recuperado de la base de datos: *siguienteRegistro* y *avanzar*. En la sección anexa se puede consultar el ~~código~~Código 1 en donde se implementa esta clase y también el ~~código~~Código 2 en donde se implementa un ejemplo de su utilización.

En el ejemplo se crea una instancia de la clase *ResultSet*, se le envía como parámetro cualquier consulta de la que se pretenda paginar el resultado, y también el número de registros de la base de datos que *sí* estarán en memoria. Nuevamente se trata de solo una aproximación, ya que la paginación usada contenía más detalles que eran introducidos por los *frameworks* que se utilizaron para el sistema. Hay que destacar que en el algoritmo de ejemplo no se define ninguna clase de lógica que haga carga de los registros de la base de datos, solo hace uso de las funciones *siguienteRegistro* y *avanzar*. La clase *ResultSet* se encarga de esta parte de la lógica, de tal manera que la función que genere cualquier reporte puede asumir que todos los registros existen en memoria.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

La única condición que debían cumplir los algoritmos de generación de reportes era la de no necesitar registros que ya hubieran sido procesados para realizar cómputos sobre ellos ya que no se tendría la opción de retroceder en el ~~set~~conjunto de resultados. Podrían existir unos pocos de ellos en memoria para aplicar cómputos y obtener algún tipo de resultado; si fueran demasiados, la ventaja de la carga lenta se perdería; generalmente el cálculo de resultados que fueran función de la información de los registros se puede relegar a la base de datos, de este modo se elimina la necesidad de hacerlo en el servidor de aplicación. A pesar de que algunos reportes se tuvieran que implementar de nuevo para cumplir la condición anterior, esta forma de trabajar parece una solución plausible, pero más adelante se describirá un problema que surgió.

Imagine que posee una lista con elementos desordenados~~,~~ como en ~~el siguiente ejemplo~~la Figura 6.1~~:~~

Fernando
José
Roberto
Carlos
Erasmo
Julio
Jorge
Alberto
Octavio
Uriel
Simón
Adalberto
Rodrigo

FIGURA 6.1~~6.1~~ UN CONJUNTO DESORDENANDO DE REGISTROS. FUENTE: ELABORACIÓN PROPIA

Aunque son pocos datos, sirve para ilustrar el problema. Ahora imagine que necesita un reporte que contenga los nombres ordenados alfabéticamente. En este punto hay que hacer notar que en la base de datos no solo existían nombres sino otro tipo de datos que el usuario podría querer ordenar. Además de que los algoritmos de ordenamiento no serían directamente diseñados por un

programador, sino que al tratarse de un DBMS, este puede ordenarlos y entregar el resultado. El DBMS usado por el sistema tiene una característica muy astuta: cuando el número de resultados que se quiere ordenar es muy grande de tal manera que no cabe en memoria RAM hay que utilizar memoria virtual para poder ordenar este ~~set~~conjunto de resultados[10]. Esto tiene el problema de que este tipo de memoria es mucho más lenta que la memoria RAM.

Con la lista de la ~~figura~~Figura 6.1~~3.7~~, suponga que la necesita ordenada. Como se mencionó antes, paginar los registros y hacer un poco del reporte cada vez es una buena opción.

[10] Burleson (2015, 15 de octubre). "*Inside Oracle Sorting*". Exraída de http://www.dba-oracle.com/t_oracle_sorting.htm

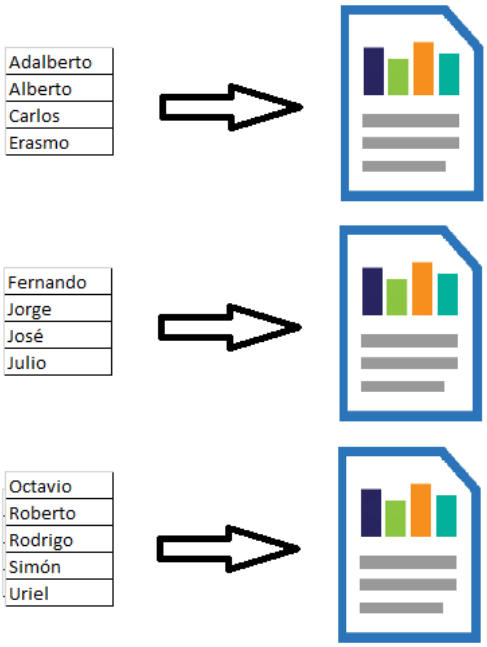


FIGURA 6.2~~6.2~~ EJEMPLO DE PAGINACIÓN DE REGISTROS OBTENIDOS. FUENTE: ELABORACIÓN PROPIA

Aunque es buena idea, cada vez que dentro de la clase *ResultSet* se mande~~a~~ a llamar a la función *obtenerSiguienteSet* este ordenamiento se realizaría~~á~~ una y otra vez. La solución para este problema se describirá más adelante. Antes de ahondar en ese problema, se aplicó otra optimización para acelerar el funcionamiento de esta parte del sistema.

Con formato: Fuente: Cursiva, Color de fuente: Automático

En una junta del *sprint* propuse una idea que concomitara la solución de la paginación:~~Se consideró~~ aprovechar el paralelismo de la base de datos. Ya que la mayoría de las bases de datos están especialmente diseñadas para atender las peticiones de múltiples usuarios, este paralelismo se puede aprovechar de alguna manera, como por ejemplo, mandar muchas peticiones de registros y que cada una recuperara una parte de los registros para la formación del reporte. Como el

Con formato: Fuente: Cursiva, Color de fuente: Automático

sistema va a esperar a que termine de responder la base de datos a una petición, entonces todas ellas se tendrían que hacer una tras otra.

Por ejemplo, en el caso de la Ffigura 6.2 se tendrían que recuperar los cuatro primeros registros, después, los cuatro siguientes y así en lo sucesivo. Para poder explotar el paralelismo de la base de datos hace falta que todas estas peticiones se realicen al mismo tiempo. Tomando como ejemplo la Ffigura 6.2, ejecutar hilos sería una elegante solución: que el primero recupere los primeros cuatro registros, que el segundo recupere los segundos registros y que el tercero recupere los últimos. Pero todo lo anterior cuidando que entre todos los hilos ejecutados el número total de todos los registros no supere unos cuantos miles para no ocupar demasiados recursos del servidor. Claro que programar esto agregaba más trabajo al *sprint* en curso; sin embargo, los líderes de proyecto aprobaron la idea, y me asignaron la labor de implementarla.

En la Figura 6.3 se muestra un esquema básico de la generación de estos reportes por medio de la recuperación en paralelo de la información de la base de datos.

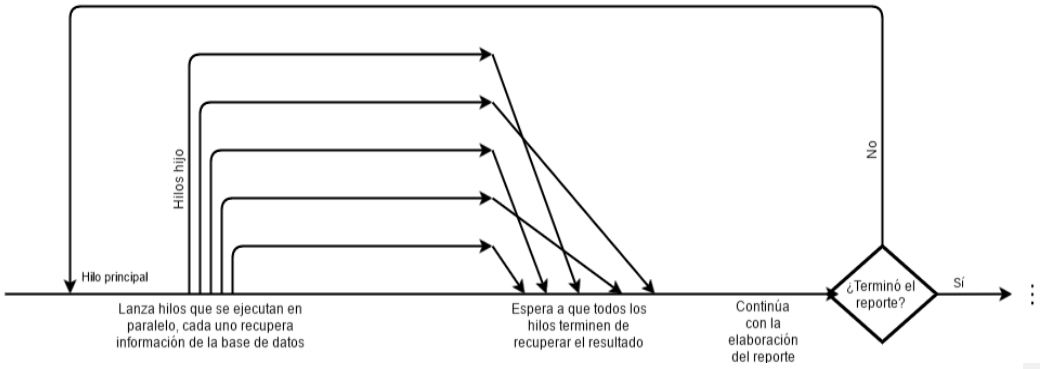


FIGURA 6.3~~6.3~~ FLUJO DE LA GENERACIÓN DE UN REPORTE USANDO HILOS. FUENTE: ELABORACIÓN PROPIA

Esta solución se había comportado bien en todos los ambientes de pruebas; el tiempo de generación se había reducido de unos quince minutos a

cinco. Por lo que se llevó al ambiente productivo; y se había considerado alcanzado el objetivo del *sprint*.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Aunque eras buena idea, lamentablemente surgieron dos grandes problemas con esta solución. La configuración de los servidores definía un límite en el número de peticiones que se podían realizar a la base de datos en un tiempo dado, de esta forma habría que lanzar pocos hilos a la vez. La Figura 6.4 es una representación básica entre la conexión a la base de datos y el servidor de aplicación. Cada flecha representa una conexión. Todas estas conexiones para un solo reporte. EAunado a esto, es casi seguro que no solo habría un usuario generando un reporte en un tiempo dado, así que había que procurar que el total del número de hilos no superara al número máximo de conexiones soportaba la base de datos.

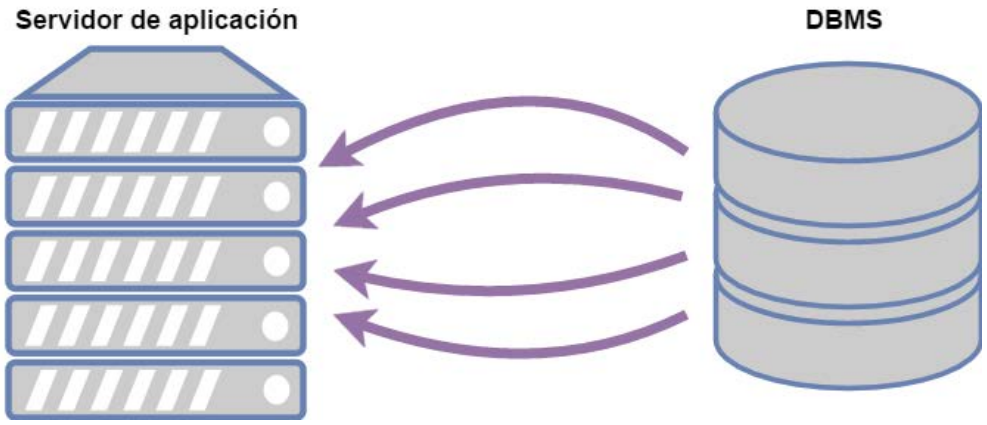


FIGURA 6.4~~6.4~~ EXTRACCIÓN DE LOS DATOS PARA UN SOLO REPORTE POR MEDIO DE HILOS. FUENTE: ELABORACIÓN PROPIA

Además para la generación de reportes se hacían consultas a tablas de la base de datos. Como los reportes solo reúnen la información que se ha registrado en la base de datos y no la modifican, únicamente se harían operaciones de lectura a estas tablas; en teoría, las operaciones de lectura no deberían bloquear tablas puesto que no hay peligro de corromper alguna secuencia o algún otro tipo de valor como en las operaciones de inserción, borrado o actualización; por esto,

estas operaciones pueden introducir bloqueos de tablas. Sin embargo, cuando se implementó esta solución en el sistema, se hacía uso intensivo de la tabla que almacenaba los movimientos que había registrado el usuario y esta comenzaba a bloquearse cuando había demasiados hilos haciendo una consulta en forma concurrente.

~~La magnitud del problema era enorme~~Esto ya se había convertido en un problema ya que el número de operaciones concurrentes en el ambiente productivo sería mucho más grande que en los ambientes de pruebas con los que se cuenta en la organización, así que los hilos comenzaron a dejar de ser opción.

Además de esto, comenzaba a surgir otro problema: recuerde que cada hilo mandaba a ejecutar la misma consulta a la base de datos, solo que cada uno recuperaba el rango de registros que debía traer. Pero como se requería cierto ordenamiento de los registros en el resultado recuperado, había otro detalle: al mandar ejecutar una vez por hilo la consulta este ordenamiento también se realizaba una vez por hilo, así que si se lanzaban diez hilos, este ordenamiento se repetiría diez veces.

Otra desventaja de realizar el mismo ordenamiento una y otra vez surge cuando el número de registros es muy grande, ya que cuando esto pasa, el algoritmo de ordenamiento del DBMS debe usar memoria virtual[11], lo que vuelve el ordenamiento de los registros mucho más lento a que si estuviera en memoria RAM debido a las operaciones de lectura/escritura a disco duro, y es mucho peor cuando la operación es repetitiva. Para ~~el~~ solucionar el último problema tenía que encontrarse una manera de realizar el mismo ordenamiento una sola vez. La Figura 6.5 es parecida a la Figura 6.4, pero en esta ocasión, se muestra que cada conexión (flecha) tiene un rango de registros a recuperar, y además debe ordenar los resultados por cada hilo.

[11] Burleson Consulting (2015, 15 de Octubre). "*Inside Oracle Sorting*". Extraída de http://www.dba-oracle.com/t_oracle_sorting.htm

Con formato: Fuente: 10 pto, Cursiva

Con formato: Español (México)

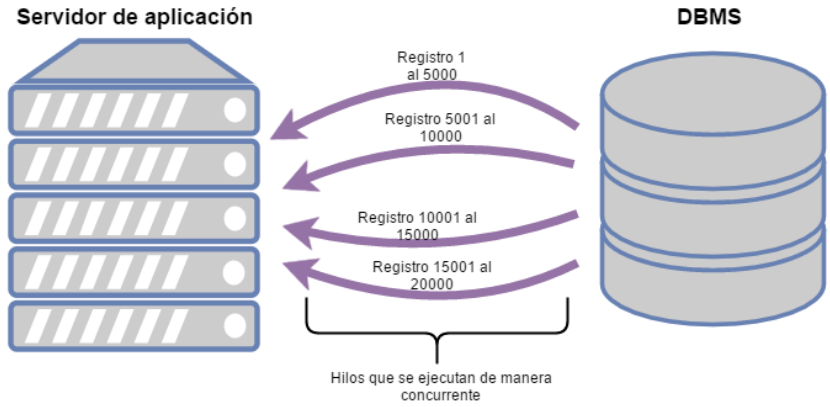


FIGURA 6.5~~6.5~~ HILOS QUE RECUPERAN LOS REGISTROS DE LA BASE DE DATOS DE MANERA CONCURRENTE PERO REALIZANDO UNA Y OTRA VEZ EL MISMO ORDENAMIENTO. FUENTE: ELABORACIÓN PROPIA

Además como se requería un gran nivel de detalle en el reporte, la información que se necesitaba no solamente residía en una tabla, sino que se debían hacer muchas reuniones o *joins* para poder recuperar todos los detalles necesarios. Las reuniones son operaciones muy costosas y además determinar el orden óptimo de las reuniones es un problema NP-hard porque el número de permutaciones posibles entre *n* tablas es *n!*; aquí cabe recordar que el orden en el que se realicen las reuniones en una consulta influye en su rendimiento[12]. Afortunadamente el DBMS utilizado tiene optimizaciones para ejecutar las reuniones en un orden conveniente: crea un conjunto de planes de ejecución y escoge el que tenga el menor costo y hace uso de los índices cuando las reuniones se realizan por medio de la llave primaria de la tabla. Así que por ese lado no se tenía mayor inconveniente, más que el de realizar una vez por cada consulta las mismas reuniones y ordenamientos.

Cuando el número de resultados era mayor a un millón el plan de ejecución elegido por el DBMS no parecía ser el óptimo, ya que dicho motor utiliza

[12] Burleson Consulting (2015, 8 de febrero). "*Oracle table join order tips*". Exraída de http://www.dba-oracle.com/t_table_join_order.htm

estadísticas basadas en la cardinalidad del conjunto de resultados en ejecuciones anteriores de la consulta. En el motor utilizado este tipo de operaciones se conoce como *retroalimentación de cardinalidad*.

Este tipo de optimización basada en la cardinalidad de los conjuntos devueltos por las consultas anteriores entraba en juego debido a que las consultas para la generación de los reportes se ejecutaban una y otra vez, y también se usa este tipo de optimización porque los predicados de las mismas consultas eran complejos para que el optimizador pudiera estimar el tamaño del ~~set~~conjunto de resultados para elegir el mejor plan de ejecución[13].

A pesar de que parece una buena solución, esta retroalimentación de cardinadidad afectaba el rendimiento de las consultas, ya que la segunda vez su ejecución se hacía extremadamente lenta a pesar de que ya tuviera la cardinalidad del conjunto de resultados.

En resumen, se ordenaban millones de registros por cada hilo, lo que hacía muy lento el proceso de extracción de datos; además ~~también~~ por cada hilo se hacían múltiples reuniones de muchas tablas, y no usando las reuniones en forma óptima. Todo para la generación de un solo reporte.

Por todo lo anterior, tuvo que haber una junta para discutir un medio para arreglar este problema; una idea fue crear una tabla temporal para insertar los registros. La solución precedente por lógica fue agregada como objetivo del nuevo *sprint*.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Así que comencé a trabajar en ~~una solución~~dicha solución. El~~:~~ crear una tabla temporal ~~en la que almacenar el resultado ya ordenado. De este modo,~~

[13] "*Cardinality Feedback*" (n. d.). Extraída de https://blogs.oracle.com/optimizer/entry/cardinality_feedback

Código de campo cambiado

~~a~~layudaba a tener las reuniones ya elaboradas y todos los movimientos ya ordenados en un solo ~~set~~conjunto de resultados, ~~este se almacenaría en una tabla para~~y así extraer la información después con una consulta simple cuyo único predicado sería el rango de registros que se requiere por petición a la base de datos (recuerde que, por ejemplo, en la implementación por hilos, cada hilo ahora ejecutaría una consulta sencilla con una cláusula ***where*** más fácil de evaluar). La Figura 6.6 es un ejemplo de una consulta con un predicado muy sencuillo.

```
SELECT *
  FROM tabla_rep_1
 WHERE renglon BETWEEN inicio AND fin
```

FIGURA 6.~~6.6~~ EJEMPLO DE UNA CONSULTA SENCILLA CON UN PREDICADO SIMPLE. FUENTE: ELABORACIÓN PROPIA

Es interesante el cómo se guardaban los resultados en una tabla temporal, es decir, si no era necesario extraerlos y después con la aplicación del servidor almacenarlos. Pero por mucha fortuna esto no es necesario ya que el DBMS ofrecía una forma de almacenar directamente en una tabla el resultado de una consulta sin necesidad de extraerlo y llevarlo a la aplicación. Esto era muy afortunado ya que los DBMS están muy optimizados para hacer este tipo de operaciones.

Implementé esta nueva~~la~~ idea para ~~de~~ crear una tabla temporal por reporte que contuviera los resultados ya ordenados. La Figura 6.7 es una representación de la transformación de una consulta con predicados complejos en una tabla temporal.

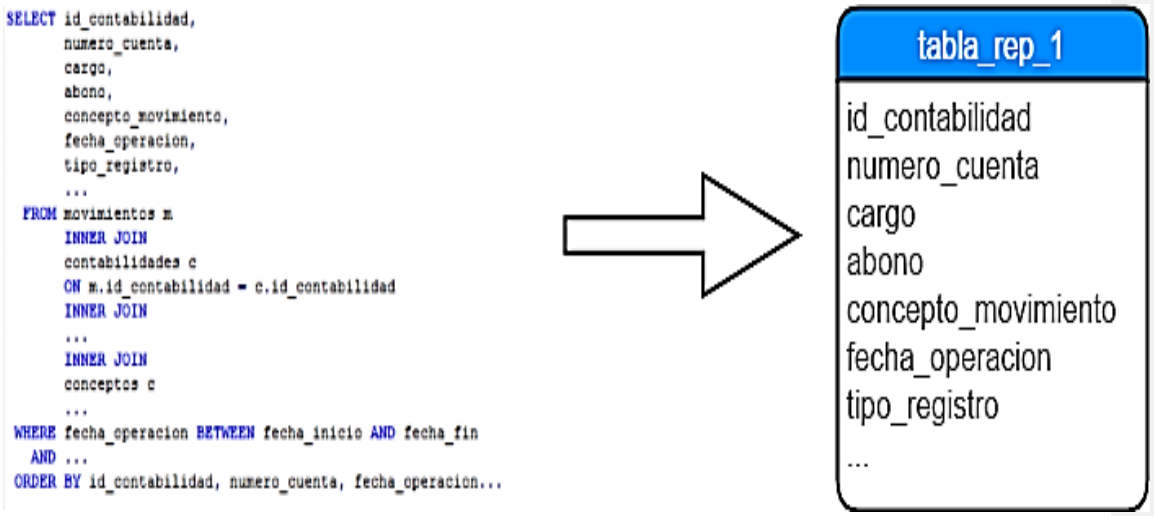


FIGURA 6.7~~6.7~~ CREACIÓN DE UNA TABLA TEMPORAL POR MEDIO DE UNA CONSULTA. FUENTE: ELABORACIÓN PROPIA

Pero había que tener muchas cosas en cuenta con la creación de la tabla temporal ya que esta para mantener el nivel de optimización esperado no debía tener restricciones (en el sentido de un esquema de base de datos, por ejemplo, restricciones de integridad referencial, de valor único, condicionales, entre otras); lo anterior debido a que la comprobación del cumplimiento de estas restricciones toma tiempo de ejecución[14]. Así que se tenía que buscar una forma de reducir al mínimo el tiempo de inserción de los registros obtenidos con la consulta en la nueva tabla temporal. Nuevamente el DBMS llegó a salvar la situación ya que ofrecía la posibilidad de realizar esta inserción por bloques.

La forma que hacía que funcionara este ~~modo~~ método de era insertando por bloques los resultados en la nueva tabla creada. El DBMS poseía una sintaxis especial para indicarle que las restricciones debían pasarse por alto y que la forma de inserción se escribiera directamente en los *data files* (archivos en los

[14] Silberchatz, A., et. al. (2002: 141-146).

que se guardan verdaderamente los registros en una base de datos). Esto mejoraba en forma intensa el rendimiento de la generación de reportes.

Así, los problemas del bloqueo de tablas, el ordenamiento y de las reuniones quedaban solventados de una forma elegante y aprovechando muy bien los recursos de ambas partes: el servidor de aplicación, al hacer una carga lenta de los registros de base de datos para evitar terminar con la memoria del servidor; y de la base de datos, al solo realizar una vez una consulta y luego recuperar estos registros con una consulta rápida a una sola tabla.

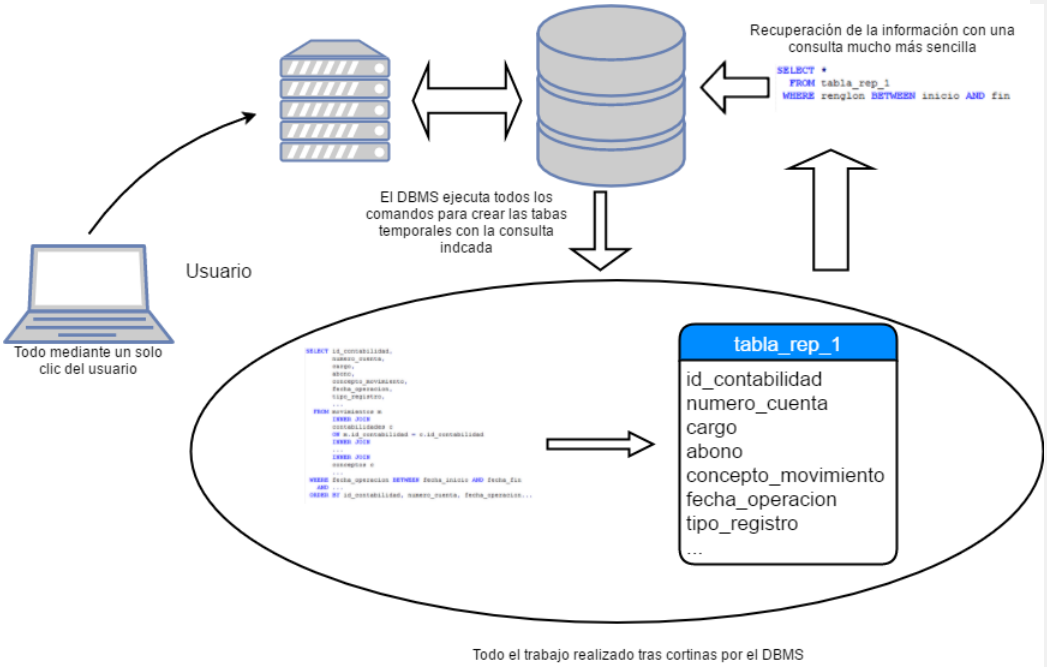


FIGURA 6.8~~6.8~~ RESUMEN DE LA TÉCNICA CREADA PARA LA GENERACIÓN RÁPIDA Y EFICAZ DE REPORTES. FUENTE: ELABORACIÓN PROPIA

A pesar de lo anterior había que tener mucho cuidado al generar los reportes ya que por cada reporte que se hiciera se tendría que repetir el proceso mostrado en la ~~figura~~Figura 6.8, cada vez que se terminara de generar un reporte se tendría que eliminar ~~la~~ su tabla temporal correspondiente. Y por cualquier interrupción que pudiera afectar al servidor, ya fuera un corte del suministro de

energía, un reinicio o cualquier cosa que pudiera interrumpir su operación, la tabla temporal se tendría que borrar a pesar de no haber terminado de generar el reporte. Sin embargo, el DBMS usado ofrece una característica para las tablas temporales: para este manejador solo existen en la sesión en la que se crearon, así que cuando el servidor finalizara la sesión que inició, esta tabla temporal se borraría.

Por supuesto que la clase *ResultSet* se modificó para no ejecutar la consulta compleja una y otra vez. Ahora esta efectuaría todo el trabajo de la creación de la tabla temporal y la recuperación de los registros. Una implementación aproximada al resultado se puede consultar en el ~~código~~Código 3 en la sección anexa.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Lo que cambia con respecto a la implementación del c~~c~~ódigo 2 es que ahora las consultas se hacen a la tabla temporal y son mucho más simples. Esta tabla se crea en cuanto se manda a llamar el constructor de *ResultSet*. Aunque no se muestre explícitamente la implementación de *obtenerDatosTablaTemporal*, esta función obtiene los datos de la tabla temporal, como el nombre, de tal manera que ocupe un nombre que no esté en uso por algún otro reporte que se esté generando en el mismo momento. La ventaja de la clase *ResultSet* es que obtenía los mismos resultados e implementaba las mismas funciones, así que los reportes que hacían uso de esta clase no debían modificar sus algoritmos ya que el cambio solo se realizó a nivel interno de la clase.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Todas las operaciones anteriores se realizaban cuando un usuario solicitaba un reporte. Podría surgir la pregunta de por qué no se generaban estos una vez, por ejemplo, en la madrugada, para que cuando el usuario los solicitara solo se obtuvieran del almacenamiento secundario del servidor. Aunque es una buena idea, esto no funcionaría ya que un requerimiento de estos era poder generarlos de una fecha arbitraria hasta otra fecha arbitraria, entonces estos solo mostrarían los registros que fueron almacenados en esas fechas. Además de que el usuario podría elegir otro tipo de opciones de filtrado, por ejemplo, que el

reporte solo contuviera registros de tipo *Ingresos*. Los montos mostrados en los reportes casi siempre cambiaban al elegir distintas opciones.

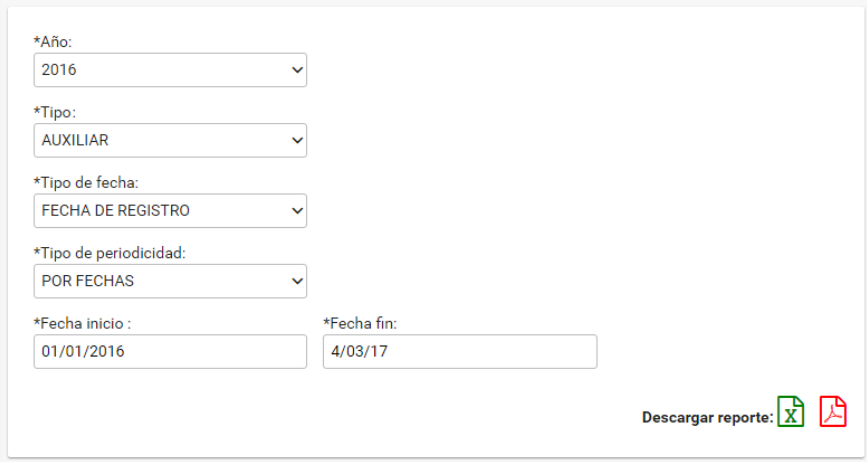


FIGURA 6.~~9~~~~6.9~~ EJEMPLO DE LAS OPCIONES QUE EL USUARIO PODÍA ELEGIR PARA GENERAR UN REPORTE. FUENTE: ELABORACIÓN PROPIA

Una vez más, esta solución se llevó al ambiente productivo y su comportamiento fue bueno, por lo menos en un principio. Con esto, este objetivo del *sprint* se daba por cumplido.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Aunque ~~esta~~ la técnica solucionaba el problema en parte, no es elegante la idea de crear una tabla cada vez que un usuario generara un reporte, además de que había ciertos problemas que se describirán más adelante y cuya solución también se abordará.

Hasta ahora solo se han revisado los aspectos de la base de datos para la generación de reportes, pero otra parte importante de la misma eran los cómputos que se hacían con los datos después de obtenerlos. Estos algoritmos debían ser muy eficientes y usar pocos recursos para evitar aumentar demasiado los tiempos de espera. Afortunadamente la complejidad de los algoritmos que implementé siempre fue *O(n)*, a saber, el crear los reportes solo implicaba recorrer el conjunto de resultados como si fuera una lista común de *n* elementos. ~~y a~~Además evitaba

realizar las operaciones matemáticas por medio del lenguaje de programación del servidor porque su modificación era riesgosa ya que al hacer un cambio en la lógica de un módulo este debía probarse hasta comprobar que no se hubiera afectado alguna otra funcionalidad. En cambio la tendencia era que en los algoritmos que desarrollé la base de datos se encargara de efectuar dichas operaciones, por lo que estas se hacían en las consultas empleadas.

Como se puede notar en la Ffigura 6.93.15, había dos formatos en los que se podía descargar un reporte. El primero de ellos no era muy complejo de generar y su estructura se formaba a la par con la recuperación de registros: crear un poco del reporte, ir por más información, crear otro poco, ir por información, crear otro poco, y así en lo sucesivo hasta terminar. Pero el segundo formato, PDF, hasta antes de que existiera demasiada información en el sistema se generaba de la siguiente manera: se recuperaban *todos* los registros de la base de datos, se aplicaban ciertos cómputos sobre ellos y por lo general se almacenaban en una lista en la memoria del servidor que se mandaba como parámetro a una función una biblioteca la cual se encargaba de crear el PDF final. Hasta ahora ya se podrá notar el problema cuando se trataba de muchos registros. No era viable ya almacenarlos en memoria RAM.

Afortunadamente Java, el lenguaje usado para programar el sistema, es orientado a objetos y tiene otros aspectos comunes de la mayoría de estos lenguajes: herencia, y polimorfismo. Estos dos aspectos fueron clave. Eran importantes porque la biblioteca que contiene la definición de las listas para el lenguaje Java tiene una clase base (en realidad es una interfaz) de la que heredan diversas clases que implementan listas, por lo que estas deben ofrecer la implementación los métodos de esta clase.

Por otro lado, la lista que se debía mandar como parámetro a la función que generaba el PDF (la que pertenecía a una biblioteca de terceros) debía ser de un tipo que heredara de la clase base de todas las listas (en Java cuando una clase implementa una interfaz debe implementar todos los métodos especificados

en ella, a pesar de que no se haga uso de estos: en casos como estos, la implementación puede ser una función con el cuerpo vacío).

~~Esta~~ La función de esta biblioteca que genera archivos PDF ~~de esta biblioteca~~ internamente realiza operaciones que no se especificarán aquí, pero itera sobre la lista recuperando sus elementos uno por uno para aplicar ciertos cómputos sobre ellos y así presentarlos en el resultado final. Esta función recupera un iterador sobre la lista que se le mande, así que llamaba a la función de la lista: *lista.iterator()*. Un iterador es un tipo de *patrón de diseño* que ofrece una forma estándar de acceder a los elementos de forma segura, a saber, es un objeto que se utiliza para recorrer un contenedor de otros objetos sin exponer su implementación[15]; los diversos algoritmos utilizados para recorrer estructuras de datos necesitan hacerlo de diversas formas, así que hay distintos tipos de iteradores, pero el ocupado por la función para crear el archivo PDF es uno que solo ofrezca los objetos «consecutivos» almacenados en la lista que se la mande como parámetro. La firma de la función *iterator* está en la interfaz que implementan las listas, *List*, y devuelve como valor un objeto que implementa la interfaz base de todos los iteradores: *Iterator*; al igual que con la clase interfaz *List*, las clases que implementen a *Iterator* deben implementar todos las funciones definidos en esta. Hay dos funciones de la interfaz *Iterator* que se usa la biblioteca para generar archivos PDF para recuperar los objetos de la lista: *next* y *hasNext*, los que se utilizan para recuperar el siguiente objeto de la lista y para comprobar que hubiera más objetos en la lista, respectivamente. Lo siguiente a que tenía que realizar quedaba claro: elaborar una clase y una interfaz, *ResultSetList* y la clase *Transformer*, que heredaran de *List* y de *Iterator*, respectivamente.

```
public class ResultSetList<T> implements List {

    Transformer<T> t;
```

[15] "*Introducción a los iteradores*" (n.d.). Extraída de http://arco.esi.uclm.es/~david.villa/pensar_en_C++/vol1/ch16s07.html

Con formato: Fuente: Sin Cursiv Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: Cursiva, Color de fuente: Automático
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)

```
      public ResultSetList(Transformer t) {
            this.t = t;
            //...
      }
      public Iterator<T> iterator () {
            return t;
      }
    //Otras funciones...
}


public interface Transformer<T> extends Iterator {
     ResultSet t;
     public Transformer<T>(ResultSet t);
}
```

CÓDIGO 4. LA IMPLEMENTACIÓN DE LA CLASE RESULTSETLIST

La clase *ResultSetList* solamente implementaba a *List* aprovechando la característica de la programación orientada a objetos ***es un***, es decir, todos los *ResultSetList* ***son*** listas en el sentido de la POO. Aunque en realidad no se trataría de una lista sino de una *interfaz* (no en el sentido de las interfaces de Java, sino en el sentido propio de la palabra) que se comportara como una lista, y haría una carga paulatina de los registros desde la base de datos como se define en la clase *ResultSet*. Así la biblioteca empleada para generar los reportes en formato PDF solo *vería* una lista, pero el comportamiento de esta lista se podría modificar mediante el polimorfismo.

La interfaz *Transformer* convertía los objetos de la clase Registro a objetos del tipo de requería la biblioteca PDF. Podría surgir la pregunta de por qué no se creó una clase específica en lugar de una interfaz. Recuerde que hay más de un Reporte de Detalle de Registros, por lo que la información recuperada de la base de datos sería distinta, así que cada reporte crearía su implementación específica de la interfaz *Transformer*, y como bien el objetivo de este trabajo lo menciona, la solución debe ser reutilizable.

El ~~código~~Código 5 en la sección anexa es un ejemplo de cómo se generaba un reporte PDF sin necesidad de cargar todos los registros en la memoria del servidor sino haciendo una carga lenta de estos. *DTOReporte* es una clase del tipo que es requerido por PDF para crear los reportes. Es importante notar que aunque herede de una lista *ResultSetList* no es necesariamente una lista, su comportamiento fue especificado para paginar los registros que residían en la base de datos.

A pesar de que surgieron nuevos problemas con las soluciones empleadas, las técnicas descritas hasta este punto siguieron siendo útiles para soluciones futuras.

Aunque ya se había logrado un avance importante en la mejora de los tiempos de ejecución para la generación de reportes (algunos tardaban hasta quince minutos y estos no tenían más de 200 000 registros, el tiempo con las nuevas técnicas ya era de unos tres minutos con 500 000 movimientos, lo que representa una mejora del 12.5 veces) y que la memoria se usaba de mejor forma, aún ocurrían comportamientos inesperados, por ejemplo, la creación de la tabla temporal a veces solo tardaba unos segundos, pero otras, minutos. Estos comportamientos anormales se debían a los distintos planes de ejecución que utilizaba el DBMS para las consultas que llenaban la tabla temporal del reporte.

Por otro lado las consultas usadas para recuperar los datos de la tabla temporal eran muy rápidas debido a su simpleza, pero aún existía otro problema,

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

que muy probablemente es de los más importantes: el canal de comunicación entre el servidor de base de datos y el servidor de aplicación. Una vez que se tenían los datos, se debían enviar al servidor de aplicación; muchas veces este proceso era muy lento, así que no importaba la velocidad de la consulta, existía la limitante del tiempo de transferencia de los datos. Así que tenía que haber una forma de evitar transferir los datos o de solo ~~mandar~~ transmitir unos pocos datos al servidor de la aplicación.

Los registros que se recuperaban de la base de datos eran muy distintos unos de otros. Es decir, no había datos que tuvieran muchos de ellos en común como para solamente recuperar esos datos una vez y así evitar la carga de muchos datos desde la base de datos. Por ejemplo, que un reporte debiera mostrar en el detalle de los registros un campo que se repitiera en muchos de ellos. Generalmente esto no pasaba y no había forma de «factorizar» la información repetitiva.

Con formato: Fuente: Sin Cursiv
Color de fuente: Automático

Así que no había manera de evitar la carga completa de los datos. Entonces mandar pocos datos al servidor quedó descartado como opción.

Debido a los comportamientos irregulares antes descritos, muchas veces el reporte se generaba muy rápido, otras, demasiado lento. En el último caso, incluso perduraba la anomalía de la terminación del tiempo de conexión porque a veces tardaba más de cinco minutos la tarea. Esto provocó que se agregara otra optimización como objetivo del siguiente *sprint* la cual se describirá más adelante.

Con formato: Fuente: Cursiva,
Color de fuente: Automático

~~Aun así h~~La optimización ideada era la siguiente~~abía otra posibilidad~~: no hacer la consulta a la base de datos cada vez que se requiriera un reporte, pero sí almacenar los datos en el servidor con el orden requerido. Antes de ahondar en esta última idea cabe recordar algo: la redundancia de datos generalmente no es deseable, ~~aunque~~ pero muchas veces se convierte en opción para aumentar el rendimiento. Aunque como se dijo, los datos almacenados en el servidor serían una copia de los datos almacenados en la base de datos, así que la información

se almacenaría de forma redundante. Pero las reglas de negocio del SRGP indicaban que los registros mostrados en los Reportes de Detalle de Registros no debían cambiar, es decir, una vez almacenado un registro que proviniera de cualquier tipo de captura (por ejemplo, la captura de los movimientos en las categorías en el Reporte de Suma de Registros) sería siempre la misma, por lo que la información almacenada en el servidor siempre coincidiría con la información almacenada en la base de datos.

Por lo anterior, tendría que haber alguna clase de funcionalidad en el sistema que se ocupara de copiar en algún momento los registros de la base de datos al almacenamiento secundario del servidor; por supuesto que aquí se podría pensar que el espacio en el servidor podría no ser suficiente, pero afortunadamente la infraestructura proveía de un almacenamiento secundario que se podía expandir según las exigencias del sistema~~virtualmente ilimitado~~.

Pero esta copia de los registros desde la base de datos al servidor se tendría que realizar de manera periódica para que los archivos contuvieran los nuevos registros introducidos al sistema. Y así es como lo implementé: la medianoche de todos los días un proceso automático se encargaba de crear estos archivos para que al día siguiente estos estuvieran actualizados con la información registrada hasta el día anterior. Por ejemplo, la medianoche del día 3 de abril de 2016 se recuperarían todos los registros almacenados en la base de datos desde el primero de enero; durante el día los Reportes de Detalle de Registros se podrían generar con la información existente en el archivo, y una vez llegada la media noche del 4 de abril de 2016 el archivo se reemplazaría por uno nuevo que contuviera todos los movimientos introducidos ese día. Aquí podría surgir la pregunta de por qué reemplazar todo el archivo y no solamente anexar al final los nuevos registros; esto no era opción ya que el orden en el que se almacenaban los registros en los archivos era el mismo que el orden en el que se mostraban en el reporte generado (esto para evitar ordenar los registros cada vez que se creara un reporte), pero no estaba determinado por la fecha de inserción del registro en la base de datos. Este orden era definido por las reglas de negocio

y tenía una consecuencia: los registros introducidos el último día tenían datos que, según los criterios para ordenar, podían aparecer antes que registros guardados en los primeros días. La ~~figura~~Figura 4.16 ilustra esta situación.

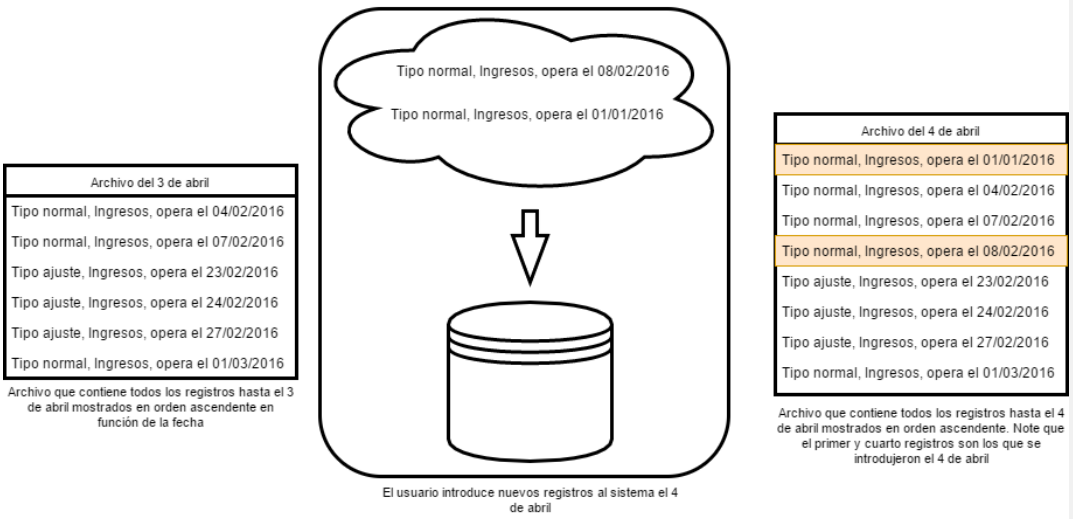


FIGURA 6.10~~6.10~~ REPRESENTACIÓN DEL ARCHIVO ALMACENADO EN EL SERVIDOR. FUENTE: ELABORACIÓN PROPIA

La primera de las tres figuras en la F~~f~~igura 6.10 es una representación del archivo almacenado en el servidor, la segunda es una representación de los datos introducidos durante el día y la tercera es cómo debería estar el archivo que incluyera ya estos nuevos registros. Por supuesto que la imagen es solamente ilustrativa porque realmente el archivo guardado en el servidor llegó a contener más de dos millones de registros y en un día promedio se introducían hasta 10 000 nuevos registros al sistema; otros incluso se introducían más de 30 000 nuevos registros.

La siguiente figura es un ejemplo más detallado del orden en el que debían mostrarse los registros en un reporte.

Mes	Tipo de póliza	Subtipo de registro	Fecha de Registro	Usuario	Concepto del Movimiento
ABRIL	NORMAL	EG	20/05/2016	martha.escudero	CH-33687 APERTURA DE FONDO FIJO, SRIA. DE COMUNICACION
ENERO	AJUSTE	IG	19/04/2016	guadalupe.arcos	DEPOSITO CAJA CHICA
FEBRERO	NORMAL	EG	09/05/2016	adriana.rodrigue	CH-33347 APERTURA DE FONDO FIJO, COORD. DE SERV. GENERALES
FEBRERO	NORMAL	IG	25/04/2016	guadalupe.arcos	DEVOLUCION DE CAJA CHICA ROSA MARIA CANTERO
JULIO	AJUSTE	IG	22/07/2016	guadalupe.arcos	CANCELACION CAJA CHICA LAURA BARTELT HOFER
JULIO	AJUSTE	IG	22/07/2016	guadalupe.arcos	CANCELACION CAJA CHICA BENITO GERARDO CARRASCO ORTIZ
JULIO	AJUSTE	IG	22/07/2016	guadalupe.arcos	CANCELACION CAJA CHICA RAFAEL PEREZ CUELLAR
JULIO	NORMAL	IG	22/07/2016	guadalupe.arcos	CANCELACION CAJA CHICA JOSE FRANCISCO BERNES ZAVALA
JULIO	NORMAL	IG	26/07/2016	guadalupe.arcos	DEV. DE CAJA CHICA JUAN NAVA CORNEJO
JULIO	NORMAL	IG	26/07/2016	guadalupe.arcos	DEV. DE CAJA CHICA EDUARDO IGNACIO SELDNER AVILA

FIGURA 6.11~~6.11~~ EJEMPLO DE ORDENAMIENTO FINAL DE LOS REGISTROS INCLUIDOS EN UN REPORTE. FUENTE: ELABORACIÓN PROPIA

Note que en la ~~figura~~Figura 6.11 está ordenado por niveles; por ejemplo, primero por *Mes*; para todos los registros de un mismo mes se ordena por *Tipo de póliza*; para todos los registros con el mismo tipo de póliza se ordena por subtipo de registro; y finalmente para todos los registros que tienen el mismo subtipo de registro se ordena por *Fecha de registro*. Tal como lo haría una cláusula ***order by*** del lenguaje SQL.

Surgen muchos detalles con esta idea; uno de ellos se debe a lo que anteriormente se mencionó: un requerimiento de los reportes era poder generarlos de una fecha arbitraria hasta otra fecha arbitraria, entonces estos solo mostrarían los registros que fueron almacenados en esas fechas. Se puede ver que los archivos en los que se almacenarían los datos de estos reportes contendrían todos los registros, no solo los que pertenecieran a los rangos de fechas que introdujo el usuario.

Antes de continuar hay que recordar que el sistema se creó para que los sujetos obligados pudieran registrar de una manera homogénea los gastos que realizan así como los ingresos que reciben. Asimismo también podrían registrar otro tipo de operaciones, pero para los fines actuales el primer propósito mencionado es especialmente útil. Retomemos nuestro ejemplo del Partido para México. Imagine que el PM tiene divisiones para todas las entidades federativas del país, es decir, existe el PM Chihuahua, el PM Guanajuato, etc. A cada una de estas divisiones la designaremos como contabilidad; por ejemplo, nos referiremos al PM Jalisco como la contabilidad del PM Jalisco; en el sistema a esta se le

asignaba un número identificador. Cada contabilidad podía efectuar operaciones en el sistema, por ejemplo, registrar los gastos que hizo para la campaña, guardar proyectos, eventos, entre otros. El identificador asignado sería como una llave para saber todas las operaciones que había registrado esta contabilidad. Del mismo modo, cada contabilidad contaba con un tipo especial de usuario llamado capturista, que, como su nombre lo indica, se encargaba de capturar los datos en el sistema, ya fuera registrar un proyecto o evento, ya fuera registrar sus ingresos, egresos, etc. Los capturistas, como lo indicaba la regla de negocio, solo podían ver los datos de su contabilidad. De este modo, si el capturista del PM Jalisco ingresaba al sistema y quería descargar el Reporte de Suma de Registros de su contabilidad especificando, por decir, del 1 de enero de 2016 al 19 de marzo de 2016 podía hacerlo. También un capturista podía descargar un Reporte de Detalle de Registros de su contabilidad. Recuerde que para acelerar el proceso de generación de estos últimos reportes la propuesta fue crear archivos que contuvieran todos los registros de la base de datos en el orden requerido. Y se podrían crear tantos archivos como contabilidades existieran en el sistema para contener los registros de cada contabilidad.

Con esta solución solamente era cuestión especificar los rangos de fechas entre otros tipos de filtros (aquí la palabra filtro se refiere a algún tipo de condición que elegía el usuario, por ejemplo, que el reporte contuviera solo los movimientos de *Ingresos* en lugar de todos los tipos de movimientos, o que solo contuviera los registros de mayo), abrir el archivo de la contabilidad y que el algoritmo que generara el reporte se encargara de discriminar los registros guardados en el archivo en función de los filtros elegidos por el usuario. Así se ahorraría el hacer la consulta a la base de datos y solamente leería un archivo. Sin embargo existe otro problema ¿y si el usuario introdujera algún registro e inmediatamente lo quisiera ver reflejado en el reporte de detalle de registros? Con lo descrito hasta ahora esto no sería posible porque con la idea del archivo, este solamente contendría los datos hasta la medianoche del día anterior por lo que el reporte no podría contener el nuevo registro.

La solución a este problema es bastante razonable: generar el reporte con los registros del archivo más los registros recuperados de la base de datos. Es decir, elaborar una consulta que recuperara los registros introducidos a partir de la medianoche y agruparlos con los del archivo. Como ambos conjuntos de resultados ya estarían ordenados sería un proceso bastante rápido mezclarlos. Solo había que determinar en función del ordenamiento definido por la regla de negocio, si el siguiente registro a colocar en el reporte sería uno recuperado de la base de datos o el siguiente registro recuperado del archivo. En este punto conviene mirar a ambos conjuntos de datos como un par de listas ordenadas (aunque no se tratara de listas); lo siguiente a realizar sería muy parecido a un *ordenamiento por mezcla*, pero en solamente el último paso: la mezcla de dos conjuntos de resultados ya ordenados. Si se tiene la lista *A* y la lista *B* y se puede acceder a cualquier elemento de ellas mediante un subíndice, por decir *A(i)* para acceder al i-ésimo elemento de *A*, entonces la mezcla de las listas sería parecida ~~al siguiente pseudocódigo~~al pseudocódigo de la Figura 6.11:

```
i = j = k = 0

mientras i < tamaño de A o j < tamaño de B

    Si i < tamaño de A y j < tamaño de B

        Si A(i) < B(j)

            C(k) = A(i)

            i = i + 1

        otro caso

            C(k) = B(j)

            j = j + 1

    o si i < tamaño de A

        C(k) = A(i)
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

```
        i = i + 1

    otro caso

        C(k) = B(j)

        j = j + 1

    k = k + 1
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

FIGURA 6.12~~6.12~~ MEZCLA DE DOS LISTAS ORDENADAS. FUENTE: ELABORACIÓN PROPIA

Al final la lista mezclada queda almacenada en la lista *C*. Podemos ver que la complejidad del algoritmo es *O(n + m)* en donde *n* y *m* son las longitudes de las listas que se van a mezclar. Claro que el pseudocódigo muestra cómo sería esta función de mezcla para, por ejemplo, dos listas de números; pero los registros de la base de datos no serían comparados mediante un operador relacional como lo es el < (menor que), sino mediante el criterio de ordenamiento especificado en la regla de negocio. Esto introduce un poco más de dificultad al cálculo de la complejidad del algoritmo ya que el tiempo de ejecución de un procedimiento que comparara podría bien variar en función de los dos registros que se estén comparando en ese momento. Para comparar estos registros se hacía uso de una función *comparar(registroA, registroB)* que devolvía el siguiente registro que debía ser introducido en el reporte.

Así esta fue una solución definitiva porque ya solo se mandarían pocos registros al servidor de aplicación (solamente los introducidos al sistema durante un día, que pertenecieran a una contabilidad y que cumplieran con el criterio definido por los filtros que introdujera el usuario) y este ya tendría todos demás registros en un archivo, solo que algún algoritmo se encargaría de agrupar ambos conjuntos de resultados en uno solo utilizando el último paso del *ordenamiento por mezcla* y aplicar a los registros almacenados en el archivo los mismos criterios de discriminación que se aplicaran a los registros almacenados en la base de datos.

Para llevar los registros desde el servidor de base de datos al servidor de aplicación se aprovecharon dos técnicas de optimización. La primera sería utilizando el archivo anteriormente generado y crear el nuevo archivo, todo esto con la ayuda de la clase *RetrieveResult*, que toma un archivo con los registros almacenados en el sistema desde el comienzo hasta la medianoche y los combina con los que se introdujeron al sistema desde la medianoche; la segunda sería la misma técnica que se había estado manejando para la generación de los reportes: utilizar una tabla temporal que almacenara todos los registros de la base de datos que cumplan el criterio de la consulta que recuperar los datos como *sí* se necesitan el reporte (recuerde que esta consulta es compleja, contiene reuniones y define los ordenamientos) y después recuperar la información de ella con una consulta más sencilla. No conviene usar una consulta compleja y hacer paginación sobre ella para realizar esta tarea porque introduce una sobrecarga en la base de datos: el ordenar una y otra vez los registros. Recuerde el diagrama de la ~~figura 3.14~~Figura 3.14 en donde toda la creación de la tabla temporal se hace cuando un usuario da un clic (esto también tiene un inconveniente: si se hicieran 20 reportes en una sola vez se tendrían que crear 20 tablas temporales). Ahora en el diagrama se reemplazaría el usuario por un proceso automático. La razón de utilizar estas dos técnicas en conjunto y no solo la del archivo en el servidor es que se necesitaba algo que almacenara un primer archivo con todos los registros insertados hasta la fecha, y esto se haría con la técnica de la tabla temporal; o por si se cambiaba algún requerimiento de los reportes y debía modificarse su contenido, deberían regenerarse completamente estos archivos, ya que pudieran no contener la nueva información requerida o tener un orden de registros ya obsoleto.

Con formato: Fuente: Cursiva, Color de fuente: Automático

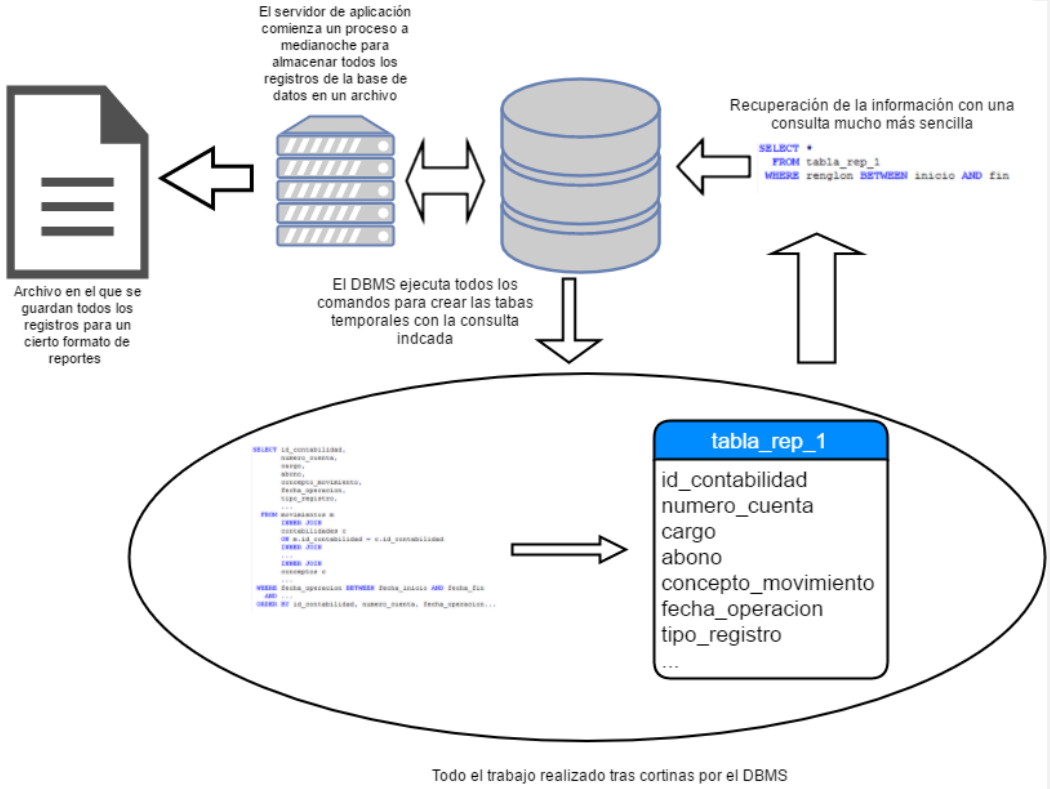


FIGURA 6.13~~6.13~~ PROCESO LLEVADO A CABO A MEDIANOCHE. FUENTE: ELABORACIÓN PROPIA

La ~~figura 6.13~~ Figura 6.13 muestra una representación del proceso que se llevaba a cabo la medianoche de todos los días (en realidad no era necesario realizar el proceso a la medianoche) para almacenar los registros de la base de datos en un archivo. Este método al igual que el anterior hace uso de la clase *ResultSet*, aunque la implementación con la clase *ResultSet* tiene el inconveniente de los comportamientos inesperados en cuanto a tiempo de ejecución (a veces tardaba más y a veces menos), esto ya no importaba. Ahora surge la cuestión de si no habría problema al tratar de generar un reporte mientras se estuviera creando el archivo; la respuesta es que sí, pero la solución es simple: cada medianoche comenzar el procedimiento de almacenamiento del archivo, nombrar a este archivo con un nombre temporal de modo que no tenga el mismo nombre que el archivo al que va a sustituir. Cuando se termine de crear este archivo entrar a un bucle que se encargue de verificar que el otro archivo (el

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

generado hasta el día anterior) no se encuentre en uso; cuando se verifique que el archivo ya no está en uso, borrarlo y cambiar al nombre temporal del archivo nuevo al del archivo que se acaba de borrar (por este motivo no es necesario que se cree a medianoche el archivo, pero es cuando el servidor tiene menos carga). De este modo no hay interrupciones a la operación de los módulos de generación de estos reportes. Sería transparente para el usuario el reemplazo del archivo. Las nuevas ventajas con este sistema son que todos los Reportes de Detalle de Registros solo abrirían sus respectivos archivos, los datos ya estarían ordenados y no habría necesidad de hacer reuniones entre tablas, pues estas ya se habrían llevado a cabo a medianoche.

Si muchos usuarios quisieran generar el mismo reporte, las operaciones de ordenamiento y de reunión ya se habrían realizado solo una vez para todos ellos; mucho mejor que si cada usuario mandara ejecutar una y otra vez una consulta con muchas reuniones de tablas para obtener los datos ordenados, así que se reduce considerablemente la carga sobre la base de datos. Todo lo reduce a un cálculo llevado a cabo cuando muy probablemente nadie estuviera ocupando el sistema.

Hay que resaltar un aspecto negativo de esta estrategia: el archivo en el servidor contenía todos los registros sin ningún filtro aplicado; por lo anterior, se tenía que relegar la tarea de aplicación de filtros para estos reportes al servidor, que es algo que solo debía hacer la base de datos. La carga debida a este problema no era demasiada, por lo que es viable.

Finalmente la clase *ResultSet* (recuerde que es la que crea la tabla temporal con los datos ya ordenados) ya no se utiliza más para generar los reportes directamente porque a pesar de que hace un mejor uso de memoria, su comportamiento es tan inestable que no se puede predecir cuál será el tiempo en el que se genere un reporte. Aun así los archivos se generarían por medio de esta clase, y como es una tarea automática, no es malo que pueda demorarse incluso

Con formato: Fuente: Cursiva, Color de fuente: Automático

horas, es decir, como no hay alguna conexión de red que se pudiera terminar o algún usuario que no desee esperar tanto tiempo, entonces es una mejor opción.

De este modo, esta nueva técnica que se diseñó ahora se podía implementar, y como siempre, su ~~implementación~~ desarrollo es genéric~~o~~a para que todos los reportes solo *vean* una interfaz como la que expone la clase *ResultSet*. De hecho en la reestructuración de estos módulos el objetivo era reemplazar los objetos de la clase *ResultSet* por los objetos de la clase *RetrieveResult*. En el ~~código~~Código 6 de la sección anexa se muestra la implementación de la clase *RetrieveResult*. Cabe señalar que el ~~código~~Código 6 solo es una aproximación al código que se utilizó en el SRGP. Note que cuenta con las mismas funciones de interfaz que la clase *ResultSet*, solo que las implementaciones de las funciones cambiaron completamente. El propósito de conservar la misma interfaz es procurar que las clases que hagan uso de esta clase *esperen* el mismo comportamiento de la clase *ResultSet* (que de hecho es el mismo) para que todos los algoritmos que hacían uso de esta última ahora utilicen la nueva clase sin necesidad de ser modificados.

Al crear un objeto de la clase *RetrieveResult*, el constructor de esta en seguida recupera los resultados de la base de datos que se introdujeron al sistema después de la medianoche y que cumplan con los criterios que haya elegido el usuario, entre otras tareas, como la apertura del primer archivo. Note que el nombre del primer archivo es una cadena común, *archivo*, más el identificador de la contabilidad. Por ejemplo, el nombre de los archivos pertenecientes al Reporte de Detalle de Registros de Gastos podría ser el siguiente: *rep_det_reg_gast_34*. Lo que se está sugiriendo aquí es que cada tipo de Reporte de Detalle de Registros debería de tener sus propios archivos para cada una de las contabilidades existentes. Se concatenaba el identificador de la contabilidad al final del prefijo común, en este ejemplo, *rep_det_reg_gast_* para identificar a cada archivo. Se podría pensar que es un problema pues habría muchos archivos que ocuparían espacio, pero recuerde nuevamente que se contaba con almacenamiento secundario virtualmente ilimitado.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

La función *siguienteRegistroSinAvanzar* es la que se encarga de *mezclar* los registros ya ordenados que existen en el archivo y los recuperados en la consulta de la base de datos. Esta función implementa implícitamente el algoritmo 1, solo que en lugar de usar algún operador relacional utiliza un criterio de ordenamiento definido por la regla de negocio. La función *avanzar* solo establece a *null* la variable de clase *registroActual* debido a siempre que que se llame a la función *siguienteRegistro* esta va a entregar el valor de esta variable si no es nula, de lo contrario sí recuperará el siguiente registro, ya sea del archivo o del resultado complementario recuperado de la base de datos. Note que cada vez que el siguiente resultado del archivo es nulo, el algoritmo abre el archivo de la siguiente contabilidad si es que hay más en la lista enviada como parámetro.

Hay una parte que puede ser la principal de este modo de trabajar: note la variable de clase del tipo Predicado en la clase *RetrieveResult*. Predicado es una interfaz permite que no se replique el código de este algoritmo cada vez que se desee desarrollar un nuevo reporte por medio de la sobrecarga de funciones. El programador que use estas clases solo tendrá que proveer las definiciones de estas funciones. Entre todas sus funciones, define a *leerObjeto*, *comparar* y *prueba*, estas se encargan de leer el siguiente registro del archivo, comparar dos registros (usualmente uno recuperado de la base de datos y otro recuperado del archivo) y ejecutar una prueba sobre un registro recuperado del archivo, respectivamente.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Para su uso el programador debía proveer la implementación de la función *leerObjeto* debido a que los Reportes de Detalle de Registros pueden requerir diferente información, por ejemplo, uno podría necesitar una fecha y dos nombres; mientras que otro, siete fechas, un subtipo de movimiento, algún estatus del registro, etc., por lo que se almacenan de distinta forma en el archivo y esta función tiene que *conocer* su formato de almacenamiento para poder traducirlos a un objeto Registro. Asimismo debe proveer la implementación de la función *comparar* ya que los criterios pueden variar en función del reporte. Por ejemplo, un reporte podría requerir que los registros estén ordenados por fecha, luego por

el nombre de quien los introdujo; mientras que otro podría requerir que estén ordenados por tipo de un movimiento y después un ordenamiento lexicográfico de la descripción de un movimiento. En síntesis, la clase *RetrieveResult* utilizaba esta función para saber cuál registro *va primero* si el recuperado de la base de datos o el recuperado del archivo. Por último el programador debe proporcionar la implementación de la función *prueba* ya que esta, según este esquema de trabajo, es la que se encarga de aplicar los filtros que introdujo el usuario en la pantalla de generación de reportes; por ejemplo, que los registros se hayan guardado en el sistema en un rango de fechas dado, o que solo traiga los movimientos de la cuenta bancaria 345, entre otros criterios.

Con formato: Fuente: Cursiva, Color de fuente: Automático

En el ~~código~~Código 7 de la sección anexa se muestra una clase que implementa la interfaz predicado que provee las funciones necesarias para que se pueda utilizar junto con el ~~código~~Código 6. Por ejemplo, la función *prueba* proveída verifica que la fecha del registro esté dentro de un rango de fechas y que pertenezca a un cierto número de cuenta. La implementación de la función *comparar* compara del ordenamiento de mayor prioridad, que este caso es la descripción; posteriormente compara por medio de la fecha del registro y después por la cuenta a la que pertenece el registro. Como esta función se llama al mezclar los dos conjuntos de registros el resultado final es como si se hubiera aplicado la cláusula ***order by descripcion, fecha, cuenta*** de SQL sobre los registros directamente en la base de datos sin utilizar ningún archivo. La implementación de la función *leerObjeto* lee del flujo de entrada carácter por carácter hasta encontrar un tabulador lo que significaría que debe interpretar todos los caracteres leídos desde el tabulador anterior, salto de línea o inicio del archivo hasta el momento y convertirlos en el tipo de dato esperado, así hasta encontrar otro salto de línea, lo que significa que ha terminado de leer el registro actual. Por supuesto que esta es una aproximación a cómo funcionaba en el SRGP, pero conceptualmente es igual.

Aquí cabe destacar una cosa: estas funciones tenían que ser rápidas, es decir, que no demoraran tanto como para que la nueva implementación con el

archivo incrementara demasiado el tiempo de ejecución. Si esto hubiera pasado, tal vez podría desaparecer la ventaja que esta nueva técnica ofrecía sobre las técnicas anteriores.

Por otro lado vea que, por ejemplo, para determinar si un registro pertenece a una cierta cuenta de una lista de cuentas que haya especificado el usuario, se tiene que realizar una búsqueda en esta lista; si se hace lineal, el tiempo de ejecución incrementará. Por esa razón en el ejemplo se utiliza una tabla *hash* en la que la llave de búsqueda es el número de cuenta, esto porque su tiempo de búsqueda generalmente es *O(1)* (generalmente porque puede haber colisiones y, dependiendo de la implementación, la solución más popular es hacer que cada celda sea un recipiente *hash*; generalmente se asocia una lista a cada clave).

Con formato: Fuente: Cursiva, Color de fuente: Automático

Para utilizar estas nuevas técnicas vea el ejemplo del ~~código~~Código 2: solo sería necesario cambiar los objetos de la clase *ResultSet* por objetos de la clase *RetrieveResult* sin cambiar alguna parte del algoritmo de generación del reporte.

Con formato: Fuente: Cursiva, Color de fuente: Automático

Con formato: Fuente: Cursiva, Color de fuente: Automático

Implementé esta nueva técnica de generación de reportes. Recuerde que en un principio los reportes de no más de 200 000 registros llegaban a tardar más de quince minutos. Con esta nueva técnica el tiempo se mejoró bastante; en realidad reportes de aproximadamente 680 000 registros pasaron a tomar solamente 20 segundos (este es el mejor tiempo ya que este es función de la carga que haya en el servidor), es decir, esta técnica aparte de ocupar mucho menos recursos era 153 veces más rápida. Otra gran ventaja es que su tiempo de ejecución era lineal, es decir, si el número de registros era el doble, el tiempo era el doble; contrario a las implementaciones anteriores, que el tiempo parecía incrementar de forma no lineal. Además de que permite la generación de reportes de gran tamaño sin terminar con los recursos de ambos servidores: el de base de datos y el de aplicación.

Con formato: Normal, Justificad Interlineado: 1.5 líneas

Así, se dio por cumplido este objetivo del *sprint*. Finalmente no surgieron más comportamientos impredecibles en la generación de reportes al llevar esta técnica a producción.

Con formato: Normal, Justificad
Sangría: Primera línea: 1.25 cm,
Interlineado: 1.5 líneas

Con formato: Fuente: Cursiva

Los resultados obtenidos

Las optimizaciones que realicé a estos módulos del sistema disminuyeron en gran medida el tiempo que debían esperar los usuarios finales para obtener los documentos que necesitaban. Por ejemplo, en el Reporte de Suma de Registros la mejora fue de hasta trescientas veces, lo que solucionaba el problema de la terminación de la conexión; además tardaba menos de un segundo la generación de este reporte. Por otro lado, un problema de naturaleza mucho más compleja, en el sentido de comportamientos inesperados, era el del tiempo que demoraba el Reporte de Detalle de Registros; en primera instancia porque al tratar de mejorar sus técnicas de generación surgían nuevos problemas sobre la marcha, como al tratar de obtener toda la información en una sola consulta, el servidor agotaba su memoria para un solo reporte, y que al tratar de paginar el resultado, se descubrió que se algunos cómputos se realizaban de forma repetitiva, tales como ordenamientos o reuniones de tablas; después, al tratar de solventar este problema con la tabla temporal, el llenado de la misma variaba mucho en cuestión de tiempo (podía tardar desde diez segundos hasta varios minutos). Además de que la transmisión de los datos de un servidor a otro tomaba más tiempo que la propia consulta. Puesto que con la solución final, el archivo en el servidor, ya no aparecieron nuevos problemas, se considera una solución definitiva. Además de que ya era casi imposible generar reportes con más de 200 000 registros, los que tardaban quince minutos o en algunas ocasiones no se alcanzaban a generar debido a que se agotaba el tiempo de conexión ya con la técnica final se pudieron generar reportes de más de 600 000 registros en menos de medio minuto, y que llegaron a pesar hasta 50 megabytes en un servidor con muchos usuarios conectados, cada uno efectuando registros o consultas.

Conclusiones

Considero que las técnicas que desarrollé durante mi estancia en la organización pudieron solventar de manera satisfactoria la necesidad que tenía el ~~SGRP~~SRGP de consultar información, dado que estas redujeron en forma considerable los tiempos de espera, por lo que se cumplió el objetivo de consultar grandes cantidades de información rápidamente. Además como se trataba de soluciones generales, estas podrían utilizarse para otro tipo de problemas que pudieran aparecer al definir nuevos módulos en el SRGP que requirieran la consulta de grandes cantidades de información que se necesitara ordenada de alguna manera. Por esto último, se cumple el objetivo de crear soluciones reutilizables.

Aunque muy probablemente este tipo de métodos ya no sean aplicables para más información, hay que recordar que se trata de un sistema web, y por lo general en este tipo de sistemas no suelen existir procesos pesados que accione el usuario al disparar eventos desde el lado del cliente. Así que se concluye que cumplen con la función y el problema quedó solucionado, pues se prevé que por cada año el número máximo de registros en un reporte no vaya más allá de los 800 000.

Referencias

Bibliografía

Tanenbaum, A. (2009) Sistemas Operativos Modernos México: Pearson, Prentice Hall.

Silberchatz, A., Korth, H. y Sudarshan A. (2002) Fundamentos de Bases de Datos: McGRAW-HILL

V. Aho, S. Lam, Sethi y D. Ullman (2008: 465). Compiladores. Principios, técnicas y herramientas: Pearson. Addison Wesley.

Mesografía

Schwaber, K. y Sutherland, J. (2013, julio) “La Guía de Scrum” Recuperado de http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-es.pdf

Nanda, A. (2009, marzo) “Líneas de Base y Planes más Favorables”. Recuperado de http://www.oracle.com/technetwork/es/articles/sql/o29spm-098177-esa.html

Burleson (2015, 15 de octubre). “Inside Oracle Sorting”. Recuperado de http://www.dba-oracle.com/t_oracle_sorting.htm

Burleson Consulting (2015, 8 de febrero). “Oracle table join order tips”. Recuperado de http://www.dba-oracle.com/t_table_join_order.htm

“Cardinality Feedback” (n. d.). Recuperado de https://blogs.oracle.com/optimizer/entry/cardinality_feedback

“Introducción a los iteradores” (n.d.). Recuperado de http://arco.esi.uclm.es/~david.villa/pensar_en_C++/vol1/ch16s07.html

Anexo de códigos

En este anexo se muestran los códigos a los que se hace referencia en varias partes del presente trabajo

```
public class ResultSet {

    /*Atributo que almacena la consulta que se va a ejecutar para

      realizar la paginación de los registros*/

    private String consulta;

    /*Atributo que indica el número máximo de registros que deben existir en
memoria*/

    private Integer numRegistros;

    /*Atributo que indica el registro actual en el que se va en la consulta*/

    private Integer desde;

    /*Lista en la que se almacenan los registros que están actualmente en
memoria,

      pertenecientes a la paginación actual.

      Como se recuperan poco a poco los registros, los pocos que se recuperan
cada

      vez se almacenan en esta lista*/

    private ArrayList<Registro> listaRegistros;

    /*Atributo que es el índice de la lista en la que se almacenan los registros

      que sí están en memoria*/

    private Integer indiceRegistros;

    public ResultSet(String consulta, Integer numRegistros) {

        this.consulta = consulta;

        this.numRegistros = numRegistros;

        desde = 0;
```

```
        recuperarSiguienteSet();

        ...

    }

    public Registro siguienteRegistro() {

        if (indiceRegistros < listaRegistros.size())

            return listaRegistros.get(indiceRegistros);

        indiceRegistros = 0;

        recuperarSiguienteSet();

        if (listaRegistros.size() > 0)

            return listaRegistros.get(indiceRegistros);

        return null;

    }

    public void avanzar() {

        indiceRegistros++;

    }

    private void recuperarSiguienteSet() {

        String cadenaConsulta =   "SELECT * "

                                + "  FROM (" + consulta + ") "

                                + " WHERE NUM_ROW BETWEEN " +

                                                        desde + " AND " +

                                                        (desde + numRegistros);

        ConsultaSQL consulta = new ConsultaSQL(cadenaConsulta);

        listaRegistros = consulta.lista();
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

```
        indiceRegistros = 0;

        desde += numRegistros;

    }

    //Otras funciones...

}
```

CÓDIGO 1. PRIMERA IMPLEMENTACIÓN DE LA CLASE RESULTSET

```
public byte[] hacerUnReporte(...) {

      ...

      ResultSet r = new ResultSet("SELECT...", 100);

      Reporte rep;

      ...

      Registro actual;

      while ((actual = r.siguienteRegistro) != null &&

             /*Otras condiciones*/…) {

            //Algunos cómputos con el registro

            ...

            r.avanzar();

      }

      ...

      return rep.reporteABytes();

}
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

CÓDIGO 2. EJEMPLO DEL USO DE LA CLASE RESULTSET

```
public class ResultSet {

    /*Mismos atributos que en el código anterior más otros nuevos…*/

    public ResultSet(String consulta, Integer numRegistros) {

        this.consulta = consulta;

        this.numRegistros = numRegistros;

        desde = 0;

        obtenerDatosTablaTemporal();

        crearTablaTemporal();

        recuperarSiguienteSet();

        ...

    }

    private void crearTablaTemporal() {

        String crearTabla =  “CREATE TEMPORARY TABLE ” + nombreTablaTemporal

                            + “(campos…)”;

        ConsultaSQL crearTablaSQL = new ConsultaSQL(crearTabla);

        crearTablaSQL.ejecutar();

        String cadenaConsulta = ”INSERT INTO ”

                                + nombreTablaTemporal + “ ”

                                + consulta;

        ConsultaSQL consulta = new ConsultaSQL(cadenaConsulta);

        Consulta.ejecutar();

    }

    public   Registro   siguienteRegistro()   {/*Misma   implementación   que   la
anterior*/}
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

```
    public void avanzar() {/*Misma implementación que la anterior*/}

    private void recuperarSiguienteSet() {

        String cadenaConsulta =   "SELECT * "

                                  + "  FROM (" + nombreTablaTemporal + ") "

                                  + " WHERE NUM_ROW BETWEEN " +

                                                     desde + " AND " +

                                                     (desde + numRegistros);

        ConsultaSQL consulta = new ConsultaSQL(cadenaConsulta);

        listaRegistros = consulta.lista();

        indiceRegistros = 0;

        desde += numRegistros;

    }

    //Otras funciones...

}
```

CÓDIGO 3.MEJORA DE LA IMPLEMENTACIÓN DE LA CLASE RESULTSET

```
public byte[] HacerUnReporte(...) {

      ...

      ResultSet r = new ResultSet("SELECT...", 100);

      Reporte rep;

      ...

      Transformer<DTOReporte> t = new Transformer<DTOReporte>(r) {

            public Transformer<DTOReporte>(ResultSet r) {

                  this.r = r;
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estac Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: Color de fuente: Automático, Inglés (Estac Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: Color de fuente: Automático, Inglés (Estac Unidos)
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)
Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)
Con formato: Inglés (Estados Unidos)
Con formato
Con formato: Inglés (Estados Unidos)
Con formato
Con formato: Inglés (Estados Unidos)
Con formato

```
            }
            public DTOReporte next() {
                  Registro reg = r.siguienteRegistro();
                  DTOReporte dtoRep = new DTOReporte();
                  //Cómputos para convertir los datos
                 //de reg en un DTOReporte
                  ...
                  reg.avanzar();
                  return dtoRep;
            }
            public Boolean hasNext() {
                  return r.siguienteRegistro() != null;
            }
      }
      ResultSetList rsl = new ResultSetList(t);
      ...
      rep = BibliotecaPDF.crearPDF(rsl);
      return rep.reporteABytes();
}
```

CÓDIGO 5. EJEMPLO DEL USO DE RESULTSETLIST Y TRANSFORM

```
public class RetrieveResult {

    /*Atributo que almacena una consulta complementaria
```

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: 10.5 pto, Color de fuente: Automático, Ing (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

Con formato

```
      para "complementar" los registros que están

      almacenados en los archivos*/

    private ArrayList<Registro> resultadoComplementario;

    private Integer indiceResultadoComplementario;

    /*Flujo de entrada para el archivo del cual se

      extraerá la información*/

    private FileInputStream archivoEntrada;

    /*La lista de las contabilidades de la que se extraerá la
información*/

    private ArrayList<Integer> listaContabilidades;

    private Integer indiceListaContabilidades;

    /*Esta es una variable en la que se almacena

      el útlimo registro recuperado

      ya sea del archivo, ya sea de la base de datos*/

    private Regitro registroActual;

    /*Variable que almacena el último regitro Recuperado del archivo*/

    private Registro regArchivo;

    public RetrieveResult(String consulta, String archivo,

            ArrayList<Registro> listaContabilidades,

            Date fechaDesde, Predicado p) {

        ConsultaSQL consultaSQL = new ConsultaSQL(consulta);

        consultaSQL.setDate("fechaDesde", fechaDesde);

        setResultadoComplementario(consultaSQL.lista());
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

```
        setIndiceListaContabilidades(0);

        setIndiceResultadoComplementario(0);

        setRegistroActual(null);

        archivoEntrada = new FileInputStream(archivo +
listaContabilidades.

        get(indiceListaContabilidades));

        ...

    }

    public Registro siguienteRegistro() {

        if (getRegistroActual() == null)

            setRegistroActual(siguienteRegistroSinAvance());

        return getRegistroActual();

    }

    public Registro siguienteRegistroSinAvance() {

        Registro complementario = null;

        if (indiceResultadoComplementario <

            resultadoComplementario.size())

            complementario = resultadoComplementario.

            get(indiceResultadoComplementario);

        if (regArchivo == null)

            do {

                regArchivo = siguienteRegistroArchivo();

            } while(regArchivo != null && !p.prueba(regArchivo));
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

```
        if (complementario != null && regArchivo != null) {
            Registro registro = p.comparar(complementario, regArchivo);
            if (registro == complementario)
                indiceResultadoComplementario++;
            else if (registro == regArchivo)
                regArchivo = null;
            return registro;
        }
        else if (regArchivo != null) {}
            Registro aux = regArchivo;
            regArchivo = null;
            return aux;
        }
        else if (complementario != null)
            return resultadoComplementario.
            get(indiceResultadoComplementario++);
        return null;
    }
    private void siguienteRegistroArchivo() {
        Registro regArchivo;
        Boolean prueba;
        do {
            regArchivo = p.leerObjeto(archivoEntrada);
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

```
            prueba = regArchivo == null
            && indiceListaContabilidades < listaContabilidades.size();
            if (prueba) {
                archivoEntrada.close();
                indiceListaContabilidades++;
                archivoEntrada =
                new FileInputStream(archivo + listaContabilidades.
                get(indiceListaContabilidades));
            }
        }while (prueba);
        return regArchivo;
    }
    public void avanzar() {
        setRegistroActual(null);
    }
    //Otras funciones...
    //Getters y setters
}
```

CÓDIGO 6. IMPLEMENTACIÓN DE LA CLASE RETRIEVERESULT

```
public class ReporteAnexoGastos implements Predicado {
      private Date inicio, fin;
      private HashMap<String, Integer> cuentas;
```

```
	public ReporteAnexoGastos(...){...}

	public Boolean prueba(Registro r) {

		if (!(r.fecha.after(inicio) && r.fecha.befor(fin) ||

		    r.fecha.equals(inicio) || r.fecha.equals(fin)))

			return false;

		if (!cuentas.containsKey(r.cuenta))

			return false;

		return true;

	}

	public Registro comparar(Registro a, Registro b) {

		if (a.descripcion.compareTo(b.descripcion) < 0)

			return a;

		else if (a.descripcion.compareTo(b.descripcion) > 0)

			return b;

		if (a.fecha.before(b.fecha))

			return a;

		else if (a.fecha.after(b.fecha))

			return b;

		if (a.cuenta.antes(b.cuenta))

			return a;

		else if (a.cuenta.despues(b.cuenta))

			return b;

		return a;
```

Con formato

```
    }
    public Registro leerObjeto(InputStream is) {
        for (int i = 0; i < 10; i++) {
            char siguiente;
            StringBuilder sb = new StringBuilder();
            while ((siguiente = is.read()) != '\n') {
                if (siguiente != '\t')
                    sb.append(siguiente);
                else {
                    switch (i) {
                        case 1:
                            registro.cuenta =
                            Integer.valueOf(sb.toString());
                            break;
                        //Los otros casos para
                        //las otras etiquetas...
                        case 9:
                            registro.descripcion =
                            sb.toString();
                            break;
                        default:
                    }
                    break;
```

Con formato: Fuente: Color de fuente: Automático, Inglés (Estados Unidos)

Con formato: Inglés (Estados Unidos)

Con formato

```
                        }

                    }

                }

            }

    //Otras funciones...

}
```

Código 7. Una implementación de la interfaz Predicado