InCo – Facultad de Ingeniería
Universidad de la República

Diseño e implementación de una herramienta para la Evolución de un DataWarehouse Relacional.

Informe final

TALLER V

Integrantes:

Andrés Alcarraz CI 2668955-9
Martín Ayala CI 2915819-5
Pablo Gatto CI 3084126-6

Docentes:

Adriana Marotta
Verónika Peralta

Indice

1. Introducción

La creciente necesidad de información para la toma de decisiones ha hecho que los sistemas decisionales cobren mayor importancia. En el proyecto presentado en 1999 se plantea la construcción e implementación de una herramienta gráfica, orientada a asistir en el diseño de Datawarehouses relacionales, basándose en operaciones de transformación de esquemas mediante primitivas. Aquí presentaremos una herramienta que permita una vez realizado el diseño del DataWarehouse la evolución del mismo frente a un cambio en la base fuente.

1.1. Contexto

La herramienta desarrollada en el taller5 de 1999 (de acá en más DWD 99), provee los mecanismos para el diseño de un DataWarehouse (DW) a partir de una base de datos fuentes integrada. Una vez terminado el proceso de diseño en el cual se obtiene el DW, puede ser necesario aplicar cambios en el esquema fuente, por lo que estos cambios tendrán su impacto en el DW.

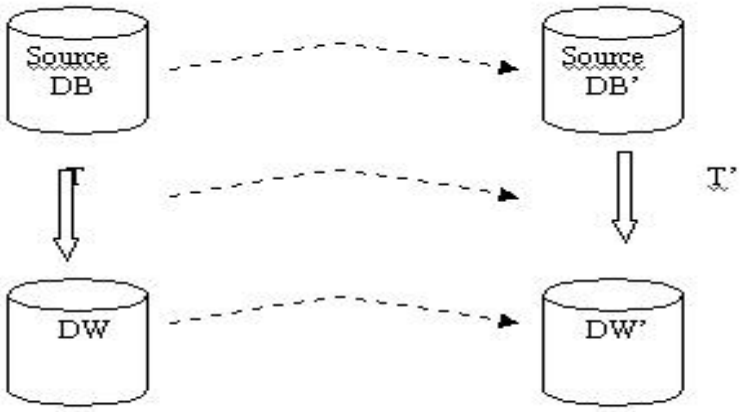


Figura 1.1

En la figura 1.1 tenemos el diseño de un DW , a partir de una base de datos fuente (Source DB) . Cómo se explica en [2] se llega al DW luego de haber aplicado las primitivas para el diseño. La secuencia de estas primitivas, forman una traza T (orden en que en que son aplicadas las primitivas); teniendo la traza T y la base de datos fuente , se llega al DW diseñado.

Supongamos ahora que es necesario realizar un cambio en el esquema de la base fuente , modificándose la base original a Source DB' . Estas modificaciones pueden tener un impacto sobre el DW, este impacto genera un nuevo DW' y una nueva traza T'.

1.2. Definición del problema

La herramienta DWD 99 provee funcionalidades para el diseño de un DW a partir de una base de datos relacional integrada.

Una vez terminado el proceso de diseño en el cual se obtiene el DW, puede ser necesario aplicar cambios en el esquema fuente. Los posibles cambios se discutirán en la sección 2.3 , estos cambios deben realizarse en base a las reglas de propagación explicadas en [1] .

Los cambios pueden afectar a :

- relaciones del DW
- las relaciones intermedias para obtenerlas
- primitivas y sus parámetro.
- Traza de transformación

Las reglas de propagación se basan en las dependencias de los cambios en la base fuente con respecto a los elementos del DW. Las dependencias se deducen de la traza de transformación.

1.3. Objetivos

El objetivo del proyecto es implementar una herramienta para propagar los cambios hacia el DW cuando hay evolución del esquema fuente. La herramienta además de propagar los cambios en el diseño del DW en forma automática de acuerdo a las reglas de propagación especificadas en [1], deberá notificar al usuario como afectará el DW ese cambio que se quiere realizar , y como dependen los atributos y relaciones del DW de estos cambios. Esta herramienta además debe ser una extensión de DWD 99 , o sea una continuación de la misma.

Para lograr estos objetivos se debe:

- Diseñar e implementar
  - Operaciones básicas (descriptas en [1])
  - Expresar primitivas en términos de operaciones básicas
  - Implementar las reglas de propagación
- Proveer una interfaz gráfica que permita
  - aplicar cambios en el esquema fuente
  - visualizar dependencias de los atributos y relaciones del esquema del DW
- Implementar modular y abiertamente la herramienta. De manera de permitir su mantenimiento y extensión por medio de cambios y agregados en el conjunto de cambios posibles en la base fuente.

2. Conceptos y trabajos relacionado

En este capítulo mencionaremos los conceptos básicos sobre DB, DW y evolución , también el trabajo en que se basa el proyecto y cómo continúa este la herramienta DWD 99.

2.1. Introducción

Esta sección trata de exponer en forma resumida algunos conceptos básicos , como ser evolución en los esquemas de base de datos en general y evolución de un DW. También se expondrá la tesis en que se basa el proyecto y su relación con el proyecto anterior DW 99.

2.2. Evolución

2.2.1. Evolución de BD en general

Llamamos evolución de esquema, a cualquier cambio que ocurra en él . Por ejemplo , agregar atributos a una relación , eliminar uno , eliminar relaciones , agregar , etc.

En general encontramos 2 aspectos importantes a tener en cuenta tras la evolución de un esquema en una base de datos respecto al estado en que quedará esta , estos son :

I) la consistencia estructural (entre la base y el esquema).

II) la consistencia en el comportamiento (esto significa mantener la consistencia para que las aplicaciones que trabajan con ella sigan funcionando).

Por otro lado , encontramos dos enfoques para manejar la evolución de esquemas : adaptativo y de versiones. En el enfoque adaptativo cuando el esquema es modificado , el estado anterior al cambio es perdido , y el resultado de la evolución es solamente la estructura del nuevo esquema. Las instancias

existentes deben ser adaptadas . En el enfoque de versiones , las modificaciones de esquemas no son aplicadas directamente sobre esta, sino que una nueva versión del esquema es creada. En este caso las instancias existentes no necesariamente necesitan ser transformadas para satisfacer el nuevo esquema. Además las aplicaciones continuarán funcionando sobre la versión de la base previa , no necesitan ser adaptados al nuevo esquema.

Cuando se toma el enfoque adaptativo para manejar la evolución de esquemas, surge la siguiente pregunta , ¿ cómo manejar la actualización de los datos existentes respecto al nuevo esquema ? Una de las formas de encararlo es la de actualización inmediata o sea los datos son actualizados ni bien se tiene disponible el nuevo esquema , la otra forma es la forma perezosa en el que los datos son actualizados en el momento de usarlos. Para la actualización de la base , el diseñador debe proveer las funciones de conversión . Dependiendo de la complejidad de las funciones se deberá determinar cual es la mejor estrategia a aplicar (perezosa o inmediata). Ambas estrategias buscan llegar al mismo final , llegar a un estado consistente para el nuevo esquema.

Al utilizar el enfoque de versiones , sólo la última versión puede ser modificada. Este mecanismo nos permite tener distintos estados en el esquema , que nos da la posibilidad de volver atrás (a estados previos) si algunos de los cambios que introducimos no da el resultado esperado. Además las aplicaciones seguirán corriendo en los estados anteriores. El problema que se plantea acá , es el de cómo compartir datos entre las diferentes versiones de esquemas. Existen tres alternativas para esto : i) IAS (Instance Access Scope) , es la parte de la base visible a través de esta nueva versión , contiene instancias de lo que ha sido creado en esta nueva versión y las instancias propagadas de otras versiones , ii) funciones de conversión , son usadas para transformar los datos a la nueva versión del esquema , están las funciones de conversión hacia delante (f.c.f) y las funciones de conversión hacia atrás (b.c.f) , como ejemplo supongamos que queremos leer datos viejos a la nueva versión ; para esto debemos leer y luego transformarlos (f.c.f) . iii) Propagación de flags , el diseñador define (con las flags) que parte de la superversión de la base será compartida por la subversión y que clase de operaciones le será posible aplicar sobre la base.

Para elegir uno de los enfoques se debe tener en cuenta que el adaptativo puede invalidar aplicaciones que están corriendo sobre la base , mientras que el de versiones agrega un overhead en el sistema. La idea es aplicar el enfoque adaptativo en los casos donde no se corrompan las aplicaciones y la de versión en los demás casos.

2.2.2. Evolución de un DW

DW son sistemas complejos que consisten en un almacenamiento de datos (altamente agregados) para ayudar en las decisiones , por ejemplo de una organización. La mayoría de los requerimientos de estos vienen de parte de los directivos de la empresa o analistas , la idea es que el DW los asista para tomar decisiones ,por ejemplo ,de negocios. La naturaleza de estos requerimientos está

continuamente cambiando y son altamente subjetivos. Además dentro de los requerimientos no sólo se exige una rápida respuesta a las consultas, sino que de continuo se está pidiendo más información (que antes no se presentaba en el DW) , que se aumente la calidad en los datos , etc. Por esto , un DW no puede ser diseñado en un paso , comúnmente evoluciona durante los años.

En un DW pueden ocurrir cambios o se requiere que ocurran en distintas situaciones (cambios en los requerimientos) .En cualquier caso el DW debe ser adaptado a los cambios que ocurrieron

- cambios en el esquema (lógicos o conceptuales)
- cambios en la propiedades físicas de los datos (en la ubicación de los datos)
- cambios en los tiempos para la extracción de los datos de la fuente (performance).

El problema de la evolución en un DW tiene dos perspectivas diferentes. La primera es la que respecta a la evolución del esquema fuente, para esta se proveen distintos algoritmos para reflejar los cambios (en este proyecto nos concentraremos en las reglas propuestas en [1]) en base a posibles cambios sobre el esquema fuente. En este tipo de perspectiva es necesario versionar los esquemas del DW ya que hay aplicaciones que podrían no adaptarse a la nueva y además para poder mantener los datos históricos. La segunda es el problema de mantener los esquemas de un DW, es el caso de mantener un cubo multi-dimensional en el caso de que se actualice una dimensión de este.

2.3. Tesis: "A Transformations Based Approach for Designing the Datawarehouse".

El proyecto completo se basa en la tesis de maestría por Adriana Marotta: "A Transformations Based Approach for Designing the Datawarehouse". Dicha tesis se concentra en proponer soluciones para diseño y evolución de un DW.

En la tesis se propone una taxonomía para representar los posibles cambios en el esquema fuente. La taxonomía de cambios es la siguiente:

1- Renombrar atributo
2- Agregar atributo
3- Eliminar atributo (el atributo no puede ser clave primaria)
4- Cambiar la clave de una relación
5- Renombrar relación
6- Agregar relación
7- Borrar relación

En el contexto del proceso de transformación se consideran al esquema fuente y final (DW) como una unión de sub-esquemas. En el esquema fuente

estos sub-esquemas se toman como entrada para la aplicación de primitivas y en el esquema final estos sub-esquemas son salida de primitivas.

En la mayoría de los casos el sub-esquema final es obtenido a través de la composición de varias primitivas. Esto se logra aplicando una primitiva a un sub-esquema, luego otra primitiva al resultado de la aplicación previa y continuando el proceso hasta que el esquema deseado es obtenido. A este proceso le llamamos una secuencia de aplicación de primitivas.

Entonces, por cada elemento (relación o atributo) del esquema final existe una traza de transformación que puede ser vista como el camino seguido para obtener dicho elemento comenzando del esquema fuente. Esta traza debe proveer la información sobre la secuencia de primitivas que fueron aplicadas.

Luego de aplicar alguno de los cambios de la taxonomía de evolución en el esquema fuente se deben deducir usando la traza que elementos del DW se ven afectados con este cambio, en otras palabras se deben deducir cuales son los elementos del DW que "dependen" del elemento al cual se le aplicó el cambio. Se introduce entonces aquí el concepto de *dependencia:* decimos que un elemento B del DW tiene una dependencia respecto al elemento modificado del esquema fuente A si existe un camino en la traza desde A hacia B.

La tarea de evolución de un DW puede descomponerse en dos subproblemas principales dado un cambio en el esquema fuente:

- Determinar los cambios que deben ser hechos al DW y a la traza
- Propagar los cambios correspondientes a la traza y al DW.

Para deducir las dependencias dado un cambio en el esquema fuente lo que se hace es descomponer las primitivas en operaciones básicas y se debe recorrer el camino desde la relación modificada en el esquema fuente hasta el DW para saber cuales son las relaciones y los atributos del DW que se ven afectados por un cambio en el esquema fuente. Además de saber que partes del DW se ven afectadas se necesita también conocer *que tipo de dependencia tiene* cada atributo del DW con respecto al esquema fuente. Para describir los tipos de dependencias suponemos que el atributo B del DW depende del atributo A del esquema fuente. Tenemos tres tipos diferentes de dependencias:

1) Copiado. Esto significa que el atributo B del DW es una copia del atributo A del esquema fuente.

2) Usado en el calculo o calculado. Esta dependencia significa que el atributo B es calculado usando una determinada función de cálculo f que toma como parámetro al atributo A, o sea que B = f(A).

3) Requerido para el cálculo. Esto nos dice que si bien el atributo B es calculado, la función de cálculo f no toma como parámetro al atributo A. En este caso el atributo A se comporta como un atributo de 'join', permite unir dos relaciones para derivar un atributo de una relación desde atributos de la otra relación. Por eso el nombre 'Requerido', si bien no es un parámetro de la función de cálculo, el atributo se requiere para poder aplicar la función.

Para determinar los cambios que deben ser hechos al DW se proponen una serie de reglas de propagación que dependen de cual haya sido el cambio en el esquema fuente (de los cambios posibles que aparecen en la taxonomía de evolución) y de cual sea la dependencia del atributo en el DW con respecto al atributo del esquema fuente. Para cada combinación de cambio-dependencia se tiene una regla de propagación.

Ejemplo:

Supongamos que borramos un atributo A del esquema fuente. Al hallar las dependencias vemos que ese atributo A es requerido para el cálculo de cierto atributo B en el DW. Buscamos en la tesis la regla de propagación para el cambio 'Remover Atributo' y la dependencia 'Requerido' y vemos que la regla nos dice que debemos borrar el atributo B del DW y borrar el camino desde A hasta B en la traza.

♦

En este proyecto se implementan las reglas de propagación que se aplicarán dado un cambio de la taxonomía de evolución y una dependencia determinada.

2.4. Proyecto Anterior: DWD 99

El objetivo de esta sección es explicar que hace el proyecto DWD 99 y en que situación se encuentra el mismo para ser continuado.

2.4.1. Características y funcionalidades.

Como dijimos en la sección anterior, el proyecto llamado DWD 99 diseña e implementa lo que se especifica en el capítulo tres de [1]. Se construye un prototitpo de ayuda en el diseño de un DW relacional. La herramienta permite diseñar un DW a partir de un esquema fuente plicando las primitivas especificadas. Se implementaron las 14 primitivas propuestas. A medida que se van aplicando las mismas para diseñar el DW se va creando la traza de transformación. La herramienta permite visualizar en cualquier momento la traza así como el DW y el repositorio (conjunto de todas las relaciones implicadas en el diseño ya sea que pertenezcan al esquema fuente, al DW o sean relaciones intermedias). El esquema fuente a partir del cual se quiere diseñar el DW es obtenido por la herramienta a partir de un archivo con formato especial donde se

especifican las relaciones del esquema, los atributos de cada relación, las claves primarias y las claves foráneas.

Para diseñar un DW usando la herramienta DWD 99 se deben seguir los siguientes pasos:

-Se genera el archivo con las relaciones del esquema fuente, sus atributos y todas las claves primarias y foráneas. Si el archivo ya fue generado se puede abrir del disco como se muestra en la figura 2.1:

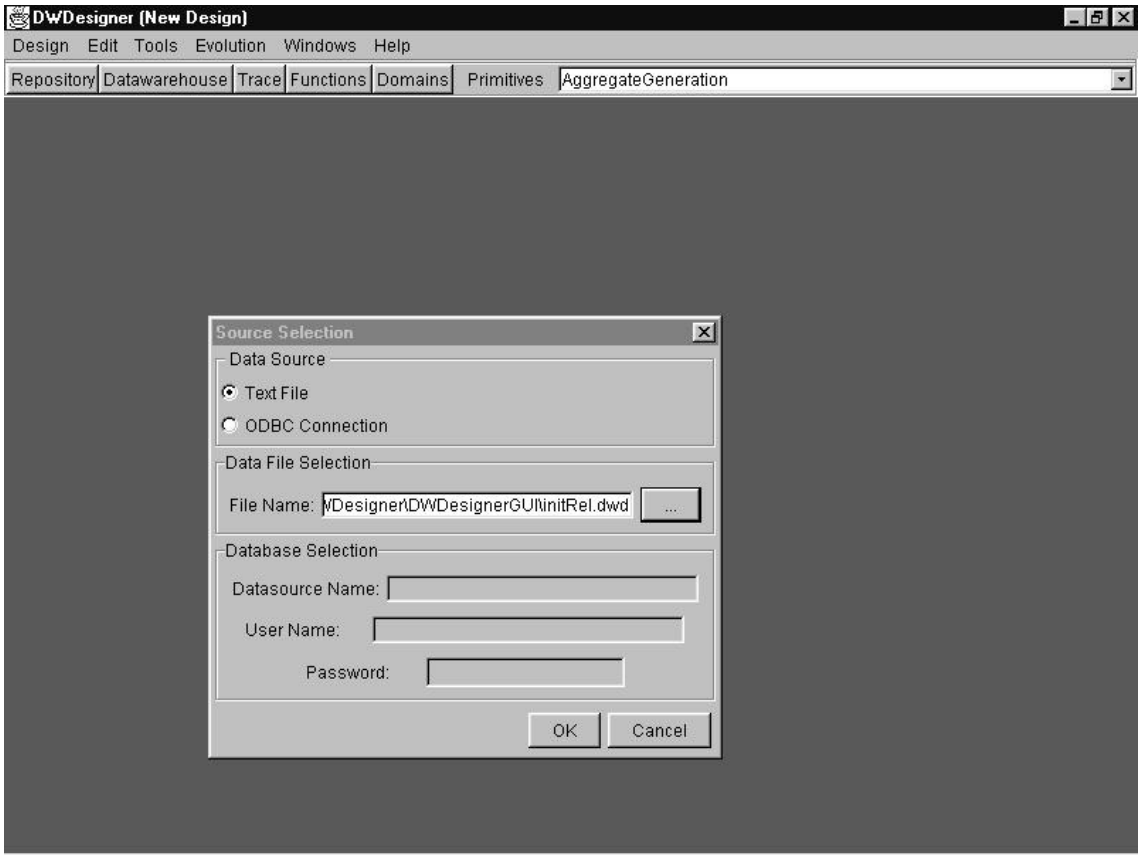


Figura 2.1

-Se ingresa un nombre para el diseño y se clasifican las relaciones de entrada en uno de los cuatro tipos siguientes: dimensión, cruzamiento, medida y jerarquía. Las relaciones de dimensión son las que representan información descriptiva del mundo real. Las relaciones de cruzamiento son relaciones que representan relaciones o combinaciones entre los elementos de un grupo de dimensiones. Las relaciones de medida son relaciones de cruzamiento que tienen por lo menos un atributo de medida. Por último, las relaciones de jerarquía son las relaciones de dimensión que contienen un conjunto de atributos que constituyen una jerarquía.

Es importante para un buen diseño hacer una buena clasificación inicial de las relaciones del esquema fuente. Para esto es bueno tener claro que es lo que

hace cada primitiva para saber que tipo de relaciones deben tomar como entrada. Por ejemplo la primitiva Hierarchy Roll Up dada una relación de medida R1 (por lo menos tiene que tener un atributo de medida) y otra de jerarquía R2 hace un 'roll up' de R1 por alguno de sus atributos siguiendo la jerarquía de R2. Es necesario entonces para que la aplicación de la primitiva que R1 sea una relación de medida y R2 sea una relación de jerarquía.

Existen primitivas para generar determinado tipo de relaciones. Por ejemplo para generar relaciones de jerarquía tenemos el grupo de primitivas Hierarchy Generation. Dicho grupo de primitivas genera jerarquías de tres tipos: denormalizada, totalmente normalizada (copo de nieve) o particionada de acuerdo a los requerimientos del diseñador (descomposición libre).

La ventana que permite clasificar las relaciones de entrada se muestra en la figura 2.2.

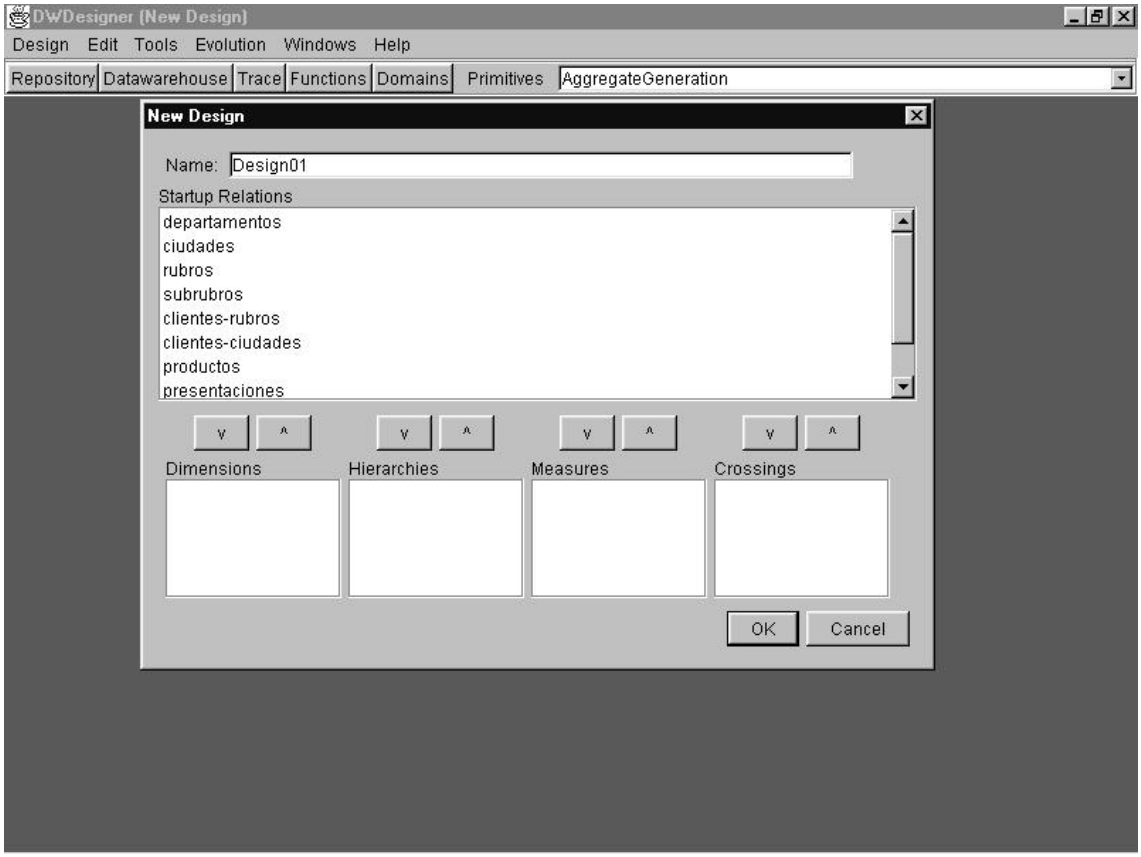


Figura 2.2

-Se aplican primitivas a las relaciones del repositorio para generar el DW. En un principio el repositorio va a estar constituido solo por las relaciones del esquema fuente. Luego, a medida que se van aplicando primitivas las salidas de estas quedan en el DW y en el repositorio. Al querer aplicar una nueva primitiva podemos aplicarla tanto a relacionesdel DW, como del esquema fuente o intermedias. Estas relaciones están todas en el repositorio y se puede tomar cualquier relación como entrada de una primitiva. Si se aplica una primitiva a una

relación que estaba en el DW ya no va a estar mas en éste, la relación de entrada pasa al repositorio y la de salida al DW.

Para clarificar los conceptos de DW, esquema fuente y repositorio ponemos un ejemplo.

Ejemplo

Supongamos que en un principio tenemos una cierta relación R en el esquema fuente y queremos aplicar primitivas a esta relación para fomar un DW con dos relaciones. En un principio el esquema fuente contiene a la relación R, el DW está vacía y el repositorio contiene también a R, o sea:

source = { R }, repository = { R }, DW = {}

Supongamos que aplicamos la primitiva p1 a R y generamos como relación de salida R2, esta se introduce en el DW y en el repositorio:

source = { R }, repository = { R, R2 }, DW = { R2 }

Ahora supongamos que aplicamos primero cierta primitiva p2 tomando R2 como entrada y luego otra primitiva p3 también tomando R2 como entrada. Supongamos que p2 y p3 generan las relaciones R3 y R4 respectivamente. El estado es el siguiente:

source = { R }, repository = { R, R2,R3, R4 }, DW = { R3, R4 }

Observamos que se insertaron las relaciones R3 y R4 del DW y se removió R2 mientras que en el repositorio se guardaron todas las relaciones y el esquema fuente permaneció incambiado. En la figura 2.3 se muestra como sería el proceso de diseño explicado anteriormente:

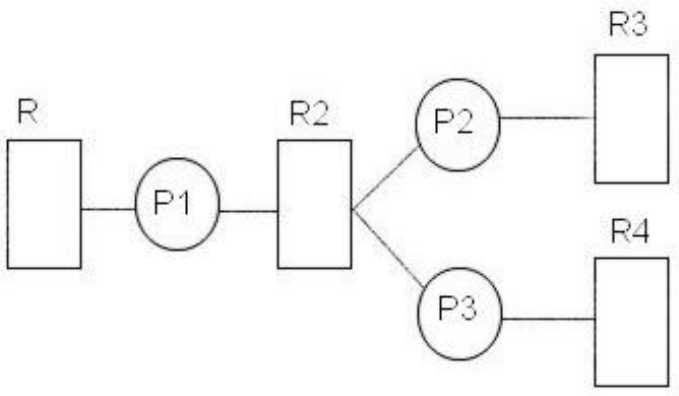


Figura 2.3

♦

2.4.2. Relación con el proyecto actual

La herramienta DWD 99 nos permite diseñar el DW. Habiendo logrado este objetivo, se quiere que la herramienta permita aplicar en el esquema fuente los cambios según la taxonomía presentada en [1] y propagar los mismos hacia el DW.

Los objetivos de ambos proyectos son claros y bien diferenciables. Por un lado la primera parte (DWD 99) implementa una herramienta para el diseño de un DW y este proyecto continúa la implementación de la primera permitiendo hacer modificaciones en el esquema fuente y reflejar las repercusiones de los mismos en el DW. Sin embargo ambos proyectos se encuentran altamente relacionados ya que las tareas de diseño y evolución no son disjuntas, es más: no se considera terminado el diseño cuando se comienza con la tarea de evolución sino que más bien ambas cosas actúan conjuntamente. Cuando el usuario hace algún cambio en el esquema fuente puede querer continuar con el diseño. Esto es de particular interés cuando se aplica alguno de los siguientes cambios de la taxonomía sobre el esquema fuente: agregar atributo o agregar relación. Luego de agregar una relación es deseable que se pueda seguir diseñando a partir de ella. Probablemente si el usuario deseó agregar una relación fue para seguir diseñando el DW a partir de la nueva relación. Lo mismo ocurre si se agrega un atributo ya que es probable que se quiera aplicar nuevas primitivas a la relación del esquema fuente que contiene el nuevo atributo.

No solo es deseable aplicar nuevas primitivas a las nuevas relaciones sino que se permita usar el esquema fuente entero para continuar con el diseño. Por eso también resulta útil que se pueda pasar de una herramienta a la otra, de diseño a evolución y viceversa. Un caso concreto de esto es cuando el usuario agrega una nueva relación y luego aplica una primitiva que toma dos entradas: la nueva relación y otra ya existente. En este caso se hizo un diseño determinado, luego se aplicó evolución y nuevamente se continuó con el diseño aplicando primitivas tanto a relaciones nuevas como a las agregadas por la tarea de evolución.

3. Análisis

La mayor parte del análisis para la realización de esta herramienta se encuentra en el capitulo 4 de [1], donde se especifican los cambios posibles en el esquema fuente y las reglas de propagación para dichos cambios. En nuestro trabajo encontramos que en algunos casos pueden generarse inconsistencias en la traza de diseño al propagar los cambios según las especificaciones de [1].
En la siguiente sección presentamos los problemas que pueden presentarse.

3.1. Introducción

En esta sección se estudian los posibles efectos que puedan surgir de aplicar la evolución en el DW. Luego de haber usado las primitivas para diseñar la misma pueden ocurrir cambios en la base fuente que produzcan inconsistencias en el DW. A continuación se hace un estudio de los efectos que podrían generar los posibles cambios. Se efectúa un estudio por primitiva y solo se listan aquellas operaciones que puedan causar inconsistencias en la traza de diseño del DW. Por ejemplo la operación Renombrar atributo no se generan inconsistencias, el DW ni siquiera se ve afectado ya que la traza absorbe los cambios. Sin embargo al borrar un atributo se debe tener más cuidado en evitar efectos secundarios que puedan hacer que se pierda la semántica de alguna primitiva o que la aplicación de la misma ya no tenga sentido o quizás haya que eliminarla porque ni siquiera es posible que la primitiva sea aplicable después de realizar los cambios. Por ejemplo, es posible que se borre un Atributo que sea usado como único parámetro en una función de cálculo. Es posible que esa función sólo pueda ser calculada con ese atributo como parámetro y sea imposible sustituir el atributo que es parámetro de entrada de la función por otro. Si el atributo se borra, la función no puede aplicarse más y es posible que la primitiva deje de tener sentido.

Los cambios al esquema fuente que se aplican sobre relaciones no se analizan en esta sección. Al aplicar alguna de estas operaciones se podrán agregar o borrar primitivas pero nunca se producirá algún efecto inesperado.

En esta sección se hace referencia a las reglas de propagación especificadas en [1] para cada cambio en el esquema fuente.

3.2. Estudio de las posibles inconsistencias

A continuación se efectúa el estudio de las situaciones que pueden generar inconsistencias.

2 Data Filter

Remover atributo. Si aplicamos la operación Remover atributo en la relación de entrada de una instancia de Data Filter pueden pasar dos cosas: 1- que el atributo eliminado de la relación de entrada haya sido filtrado por la primitiva en cuyo caso la relación de salida no se ve afectada y 2- que el atributo eliminado no haya sido filtrado en cuyo caso se elimina el mismo atributo de la relación de salida, en ese caso la primitiva seguiría actuando de la misma forma. Podrían encontrarse casos de borde donde por ejemplo el atributo a eliminar sea el único que no se filtra, al eliminarse éste, la relación de salida queda vacía y como consecuencia se elimina la primitiva y posiblemente varios caminos en la traza.

3 Temporalization

Cambiar la clave. La Primitiva Temporalization nos da la opción de extender la clave agregando el nuevo atributo de tiempo a la clave primaria. En este caso podrían surgir problemas al cambiar la clave que no estén contemplados en ninguna regla de propagación. Para ilustrar se muestra un ejemplo muy simple.

Ejemplo

Supongamos que tenemos la relación R1 con los siguientes atributos:

R1 = { Nombre, Apellido, Teléfono } PK = [Nombre]

y que queremos aplicarle a esta relación la primitiva Temporalization agregando un nuevo atributo llamado Time e incluirlo como parte de la clave. La relación de salida la llamamos R2 y tiene los siguienes atributos:

R2 = { Nombre, Apellido, Teléfono, Time } PK = [Nombre, Time]

Supongamos que queremos ahora cambiar la clave de R1 de Nombre a Apellido. El atributo Nombre es copiado en R2 y la regla de propagación nos pregunta si el atributo Nombre es clave primaria o foránea en R2 y vemos que ni es clave primaria ni foránea ya que es parte de la clave primaria pero no es clave porque se agregó el atributo Time a la clave. No tenemos en este caso ninguna regla de propagación.

♦

4 Group Key Generalization

4.2 Key Extension

Cambiar la clave. Si cambiamos la clave en una relación que sea entrada de una primitiva Key Extension debemos aplicar la regla de propagación R6 ya que todos los atributos de la relación de salida son copiados de la relación de entrada. En ésta regla se plantean dos acciones diferentes: una si el atributo del DW es clave de la relación en el esquema fuente y otra si es clave foránea en el mismo. Éste caso no se encuentra dentro de ninguno de esos dos, en el DW la clave a original es parte de la clave actual ya que la finalidad de la primitiva es justamente extender la clave. Este problema es similar al que puede ocurrir en Temporalization.

6 Group DD-Adding

6.1 DD-Adding1-1

Remover Atributo. En éste caso al remover un atributo se aplican las reglas de propagación R3 y R4.La aplicación de R3 es trivial por lo que se discute la aplicación de R4. Los Atributos de la base fuente son usados en el cálculo del Atributo a agregar por lo que si se borra uno, en el caso que éste participe en la función de cálculo el usuario debe decidir si cambia la función de cálculo o elimina el atributo calculado. Cambiar la función de cálculo no siempre es posible por lo que habría casos en que el atributo agregado necesariamente debería ser borrado. Esto es consistente con la regla de propagación pero sin embargo si se borrara el Atributo agregado por DD-Adding la primitiva aplicada dejaría de ser ésta y pasaría a tener la semántica de la identidad ya que el único atributo que se agregó ahora se borra. En este caso podría ser preferible eliminar la primitiva ya que su aplicación puede dejar de tener sentido.

6.2 DD-Addingn-1

Remover Atributo. Si se borra un atributo que participa en la función de cálculo y no es posible quitar este atributo de la función estamos en las mismas condiciones que en el caso 6.1. Si se borra el atributo de join entre las relaciones se aplica la regla de propagación R5 y la primitiva entera debe eliminarse ya que no es posible su aplicación sin un atributo de join entre las relaciones.

6.3 DD-Addingn-n

Éste caso es análogo al anterior

8 Hierarchy Roll Up

Remover Atributo. En éste caso puede haber problemas si el atributo que se borra es calculado en el DW y ninguna de las dos acciones a tomar en las reglas de propagación R4 es adecuada (Remover el atributo en el DW o cambiar la función de cálculo).

Ejemplo

En el capítulo 3 de [1] se ilustra un ejemplo donde se hace un Hierarchy Roll Up desde DATE hasta MONTH (se escalan dos niveles en la jerarquía). El Atributo MONTH si bien se copia igual de la Relación origen es un Atributo calculado usando como función de cálculo la igualdad, no se copia como Att_cpy. Esta diferencia radica en que se requieren mas atributos (DATE en este caso) y no es sino la operación Att_calc que permite éste tipo de cálculos. Si deseáramos borrar el atributo MONTH del esquema fuente aplicamos entonces la regla de propagación R4. En las acciones nos da la opción de borrar el atributo correspondiente en el DW o cambiar la función de cálculo.

Si decidiéramos optar por borrar el atributo correspondiente en el DW deberíamos borrar el atributo MONTH y su camino desde el esquema fuente. Esta decisión no es correcta porque el DW quedaría con la relación pero el atributo MONTH desaparece que es un atributo fundamental de la relación porque es justamente el que se eligió para hacer el Roll Up. En éste caso la primitiva permanecería pero su resultado no sería el esperado y la aplicación de la primitiva dejaría de tener sentido.

La segunda opción tampoco sería correcta porque la única función adecuada para función de cálculo es la igualdad y hay atributos requeridos que no permiten que se aplique la operación básica Att_cpy.

♦

Cambiar la clave. Podría suceder en algunos casos que se agregue un nuevo atributo a una relación y que luego se cambie la clave a éste nuevo atributo. En caso de que la antigua clave sea requerida para el cálculo de un atributo del DW se nos presentan dos opciones, eliminar el atributo del DW o cambiar el atributo requerido. Puede haber casos en donde ninguna de estas alternativas sea la adecuada.

Ejemplo

Usando el mismo ejemplo anterior supongamos que se agrega un nuevo atributo Time_Key en la Relación TIME y luego se cambia la clave desde DATE a Time_Key. Al hacer éste cambio se debe aplicar la regla de propagación R7 ya que DATE es requerido para el cálculo de MONTH en el DW como veíamos en el ejemplo anterior. Ésta regla permite los dos tipos de acciones mencionados anteriormente pero resulta que ninguno de ellos sirve ya no podemos borrar el atributo MONTH o quedaríamos en las mismas condiciones del ejemplo anterior. Tampoco podríamos cambiar el atributo requerido en la función de cálculo ya que Time_Key (el nuevo Atributo agregado) no existe en la relación MONTH_SALES del DW y si observamos los atributos requeridos en el DW vemos que también se

necesitaría que el atributo Time_Key figure en MONTH_SALES (ver la descomposición de Hierarchy Roll Up en operaciones básicas en [1]).

En caso que se dé esta situación tenemos dos alternativas: o bien no se cambia la clave o en el caso de que se decida efectuar el cambio se debe eliminar toda la primitiva ya que esta pasa a un estado inconsistente. Además de eliminarse la primitiva se debe eliminar todo lo que involucre a esta ya que luego pueden venir más primitivas y estas probablemente tampoco puedan aplicarse.

♦

14 New Dimension Crossing

Remover Atributo. Para ser aplicada ésta primitiva se exige que las relaciones de entrada tengan un atributo en común. Si éste atributo se elimina de dejarían de cumplir los requerimientos para la aplicación de la primitiva y ésta debería eliminarse. Las reglas de propagación no son aplicables en éste caso.

Ejemplo

En el capítulo 3 de [1] se aplica New Dimension Crossing con dos Relaciones cuyo único Atributo en común es COURSE, si éste atributo se elimina de alguna de las dos relaciones, el atributo Z que se exige que esté en ambas relaciones ya no existe y por lo tanto la aplicación de la primitiva ya no es posible. Éste atributo COURSE no es ni calculado, ni requerido para el cálculo ni copiado por lo que lo clasificaríamos como atributo sin dependencias en el DW y por lo tanto no se aplicaría ninguna regla de propagación.

♦

Cambiar la clave. Al cambiar la clave hay que tener en cuenta lo que hace la primitiva, definir una clave en la nueva relación que es la unión de las claves de las relaciones del esquema fuente. En éste caso correspondería aplicar la regla de propagación R6, sin embargo habría que ver cuales son los pasos a seguir si el atributo en el DW es parte de la clave y no la clave entera. Esto se podría dar al aplicar New Dimension Crossing porque se define como clave de la relación resultante a la unión de claves de las relaciones originales y en la regla R6 se estudia si el atributo es clave primaria o clave foránea en el DW. Habría que adaptar las acciones cuando el atributo es parte de la clave primaria que no es ninguno de estos dos casos. Esta situación es análoga a Cambiar la clave en la primitiva Key Estension (4.2) y Temporalization (3).

3.3. Soluciones

En el punto anterior se introdujeron varias posibles inconsistencias que se pueden dar en el DW al aplicar cambios en el esquema fuente. El próximo paso luego de encontradas estas inconsistencias es encontrar la solución mas adecuada.

Podríamos pensar en una primera posible solución para evitar que surjan inconsistencias, impedir al usuario que realice los cambios en el esquema fuente si sabemos que se está en alguna de las situaciones críticas. Esta posibilidad asegura que no se producen inconsistencias en la traza de diseño del DW, sin embargo es una solución demasiado restrictiva para el usuario el cual quizás no tenga mas remedio que hacer cambios en la base fuente.

Una segunda alternativa podría ser que el usuario tenga total libertad para hacer los cambios que desee, si se observa que se produce alguna inconsistencia, todas las relaciones que queden en ese estado son eliminadas, además de eliminar también de la traza los caminos que ya no se necesitan.

Si bien aplicando ésta última solución el usuario tiene total libertad sobre el esquema fuente, éste podría no ser conciente de los problemas que puedan ocurrir ya que algunos problemas son poco intuitivos y difíciles de ver por lo que aplicar éste enfoque podría ser peligroso por lo que optamos por usar un híbrido de las posibles soluciones mencionadas anteriormente. Si el usuario desea hacer algún cambio, éste tiene toda la libertad de hacerlo pero si el cambio que va a efectuar puede llevar a que ocurra alguna inconsistencia se le avisa al usuario el problema que puede ocurrir y éste decide lo mas conveniente en cada caso. Si de todas maneras decide hacer el cambio, se elimina todo lo que pueda quedar inconsistente, sin embargo en éste caso el usuario fue avisado y él decidió hacer el cambio de todas maneras. De ésta forma el usuario no necesita pensar en los posibles efectos que puedan ocurrir, simplemente en caso que ocurra alguno éste se entera y toma la decisión mas adecuada de acuerdo a la situación.

4. Diseño

En este capítulo, presentamos el diseño del proyecto. Se explican los principales conceptos del mismo, y como estos conceptos se traducen en las clases y módulos implementados.

4.1. Introducción

Lo primero que tuvimos que hacer fue estudiar el diseño de DWD 99, para que nuestro diseño sea compatible. La política que tomamos fue de modificar lo menos posible el diseño original, y que nuestro aporte fuera agregando componentes y no modificando los existentes. Aún así hubo que modificar algunas de las clases originales para que interactúen con las nuestras. Otro tipo de modificaciones que se hicieron al diseño anterior fue agregar funcionalidad a las clases originales.

4.2. Principales Conceptos

En esta sección mostramos los conceptos fundamentales de nuestra aplicación, y a partir de ellos deducimos las clases que conforman nuestro diseño.

1- **Traza Detallada:** dado el diseño del DW en función de las primitivas, se descompone en un conjunto de operaciones básicas, que permiten deducir las dependencias a nivel de atributo entre elementos del DW y elementos de la base fuente. A este conjunto de operaciones básicas y sus interrelaciones le llamamos “Traza detallada”.
2- **Dependencias:** Cuando se aplica una primitiva a una relación o más, los atributos de la relación de salida pueden depender de diferentes formas respecto a los atributos de las relaciones de entrada o no tener dependencia alguna

3- **Propagación de los cambios:** Al producirse un cambio en el esquema fuente, se debe propagar el cambio de acuerdo a las reglas de propagación especificadas en [1].
4- **Visualización de la traza detallada:** Proveer una interfaz gráfica para ver las dependencias entre relaciones del DW y las relaciones de la base de datos fuente.

4.3. Arquitectura

4.3.1. Mecánica de Funcionamiento

Por ser nuestro proyecto una continuación de otro, nuestra arquitectura se adapta a la arquitectura ya existente.

Cómo se desprende de la sección anterior el diseño se divide en 4 partes.

A continuación mostramos como se deducen las distintas clases implementadas en el sistema.

Traza detallada

Por cada aplicación de una primitiva tenemos una Traza Detallada, que está representada por un objeto de la clase **DetailedTrace**, esta clase contiene un subgrafo asociado a la primitiva.

En el diseño de DWD 99 la traza está representada por un grafo en el cual sus nodos contienen un objeto de la clase **TraceEntry**, estos objetos contienen información sobre la aplicación de una primitiva a un conjunto de relaciones, la información que contienen es acerca de las relaciones de entrada y de salida, y también contiene una referencia a la primitiva que fue aplicada.

La primitiva al aplicarse a un conjunto de relaciones de entrada con un determinado conjunto de parámetros es la responsable de generar este objeto (**TraceEntry**) que describe la acción realizada por ella, así es la primitiva también quien genera la traza detallada.

Para recorrer toda la traza detallada hay que recorrer la traza de transformación, y para cada nodo de ésta recorrer la traza detallada que contiene su **TraceEntry**. El grafo que representa esta traza consta de objetos de las siguientes clases **DetailedEdge** (Arista detallada) y **DetailedNode** (Nodo detallado), donde los segundos son los nodos de la traza detallada y los primeros son las aristas que unen los nodos. Los nodos se dividen en tres subclases de **DetailedNode**, estas son, **InputNode**, **OutputNode** e **InternalNode**, **InputNode** y **OutputNode**, vinculan la traza detallada con la traza de transformación, los nodos internos (**DetailedNode**) son los que contienen las operaciones básicas. Cada

traza detallada contiene una lista ordenada de nodos de entrada y otra de nodos de salida, la cantidad de nodos de entrada y de salida de una traza detallada debe coincidir con la cantidad de aristas de entrada y salida respectivamente, del **TraceEntry** al que pertenece, esta es la forma de saber a que parte de la traza detallada apunta una arista de la traza general. Por más información sobre el diseño ver el anexo “diagramas de clases”.

Dependencias

Los atributos de las relaciones del DW pueden depender de distintas formas de los atributos de la base de datos fuente.

La deducción de estas dependencias implica recorrer la traza detallada siguiendo los caminos del atributo que interesa modificar hasta el DW, se tiene una estructura que permite almacenar el camino que lleva a cada dependencia y la operación básica que origina la dependencia más fuerte dentro de cada camino, así en los casos en los que la operación básica debe ser modificada, por ejemplo cuando la dependencia es de cálculo y se quiere cambiar la función de cálculo, esto se puede hacer sin tener que volver a recorrer la traza. Para esto creamos la clase **DependenciesList**, que contiene:

- la relación y el atributo modificados.
- caminos que van desde el atributo modificado a los atributos del DW que dependen de éste.

Los primeros son para almacenar respecto a quien son las dependencias de la lista.

Para cada atributo del DW que depende del atributo a modificar en el esquema fuente, se almacena el camino que lleva del atributo a los del DW en forma de una lista de dependencias. El primer elemento de esta lista contiene la dependencia más fuerte, y el resto el camino. Los elementos de esta lista son de la clase **Dependency**.

Los objetos de la clase **Dependency** son generados por las operaciones básicas al consultar las dependencias de uno de los atributos de sus relaciones de entrada y son retornados por su método getDependencies().

La clase **Dependency** contiene información de la dependencia de la salida de una operación básica en particular respecto a un atributo de una de las relaciones de entrada a dicha operación, la información que aporta es que atributo de que relación de salida depende del atributo solicitado, cual es la dependencia y también una referencia a la operación básica que la generó.

Propagación de cambios

Para propagar los cambios sólo tenemos que recorrer la lista de todas las dependencias, y notificar a las operaciones básicas del cambio producido para que pueda cambiar sus datos internos ,quedando en un estado consistente.

Visualización de la traza detallada

Como dijimos, la traza detallada es una estructura dispersa, es decir, no está completamente almacenada en un objeto, sino que cada parte está almacenada en un nodo de la traza de transformación.

Al dibujar la traza detallada nunca se dibuja la traza completa, debido a que esta puede llegar a ser tan grande que no se puede visualizar correctamente, por esta razón se decidió que solo se dibuje la parte de la traza que resulta de comenzar por una relación del esquema fuente.

4.3.2. Diseño modular

Siguiendo los pasos de [2] nuestro proyecto se implementó agregando un módulo más y modificando algunos de los módulos del DWD99.

Así nuestro esquema de diseño por módulos queda como se muestra en la figura 3.1.

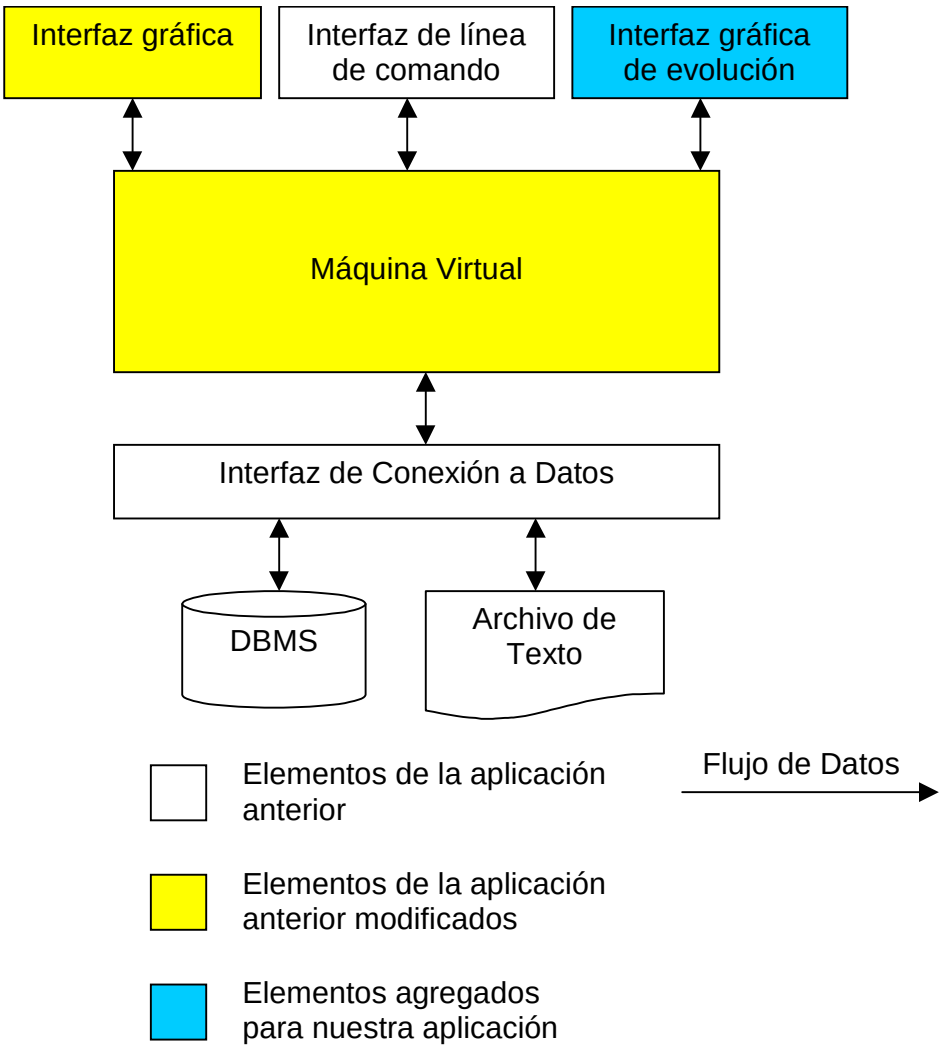


Figura 3.1: Módulos que componen la herramienta

- DBMS.

Este módulo representa el manejador de base de datos que almacena la base de datos origen. Cabe recordar que los lineamientos de diseño, usados por el trabajo [1] implican bases de datos relacionales, por lo que nuestra herramienta fue desarrollada pensando que el DBMS anterior es un manejador relacional.

- Archivo de Texto.

Representa tanto el archivo con la especificación del esquema inicial (ver Anexo 1: "Manual de Usuario", sección Creación de un nuevo diseño en [2]) como el de código de creación del esquema final.

- Interfaz de Conexión a Datos

Este módulo cumple la función de proveer una interfaz estándar a las posibles fuentes de datos.

Este módulo aísla a la Máquina Virtual del impacto producido al cambiar el origen de datos, minimizando los cambios necesarios al sistema.

- Máquina Virtual

  Este módulo de la herramienta cumple las siguientes funcionalidades:
  - Contiene estructuras de datos necesarias para representar los conceptos de bases de datos, y en particular de DW.
  - Proporciona los servicios necesarios para aplicar las primitivas sobre estas representaciones.
  - Proporciona todas las estructuras y algoritmos necesarios para cumplir con la funcionalidad de trazabilidad antes descripta.
  - Las estructuras para almacenar la traza detallada, así como las operaciones básicas.

- Interfaz de línea de comando

  Este módulo permite la interacción entre el usuario y la Máquina Virtual. Esta interacción se llevará a cabo a través de comandos ingresados mediante una interfaz de línea de comando (al estilo DOS o Unix), esto está habilitado solo para la parte de diseño.

- Interfaz gráfica

  La función de este módulo es análoga al anterior, solo que la interacción con los usuarios es realizada en forma gráfica, utilizando interfaces orientadas a ventanas y eventos (al estilo Windows).

- Interfaz gráfica de evolución

  Este es el módulo que maneja toda la parte de evolución en forma gráfica. Desde esta interfaz el usuario puede aplicar cualquiera de los cambios sobre el esquema fuente, ver las dependencias en el DW a partir de esos cambios y decidir si propagarlos o no. Además de eso se permite visualizar la traza detallada completa para observar los caminos que interesen en ella desde el esquema fuente hasta el DW.

4.4. Diseño Conceptual

Un usuario de la Máquina Virtual realiza sucesivas transformaciones sobre la representación interna de la base de datos origen, hasta obtener una representación satisfactoria de lo que representa el DW final.

Una vez que el proceso se ha terminado y el diseño ha sido guardado, cuando por alguna razón se deba modificar el esquema de la base de datos fuente, el usuario puede usar esta herramienta para realizar modificaciones en el mismo. Luego éste será guiado en el proceso de realizar los cambios.

Una vez de terminada la estructura del nuevo esquema del DW, se generará y se mantendrá una nueva versión del DW. como se muestra en la Figura 3.2 el módulo “Generador de Versiones del DW y funciones de conversión” genera las funciones de transformación de una versión del DW a otra, este módulo se debe implementar (ver capítulo 6. Conclusiones).

Los módulos que generan el nuevo esquema y el código para los nuevos procesos de carga son “Generador de Código de creación de esquema” y “Generador de código de carga de datos” respectivamente, ver capítulo 3 de [2].

4.4.1. Máquina Virtual

Esta es la capa más importante del sistema, es la que contiene toda la arquitectura para deducir las consecuencias de la evolución y además contiene los generadores de código, para generar y cargar el DW en el sistema de base de datos. A continuación explicamos lo nuevo de la máquina virtual

Conceptos generales

Descripción

La Máquina Virtual, es la parte del sistema que contiene las distintas operaciones que se pueden realizar, la secuencia de operaciones realizadas sobre la base de datos fuente para generar el DW, los distintos cambios sobre el esquema fuente para los cuales está implementada la actualización automática del DW, contiene también todos los objetos encargados de propagar dichos cambios.

Contiene el estado del sistema, es decir que en ella está toda la información sobre lo que el usuario hizo en el sistema, por ejemplo si salvamos en determinado momento la información que contiene la Máquina Virtual a disco y luego la levantamos en otra instancia de la aplicación, el usuario podrá seguir trabajando, y continuar el trabajo desde el punto en el que estaba cuando se salvó la Máquina Virtual.

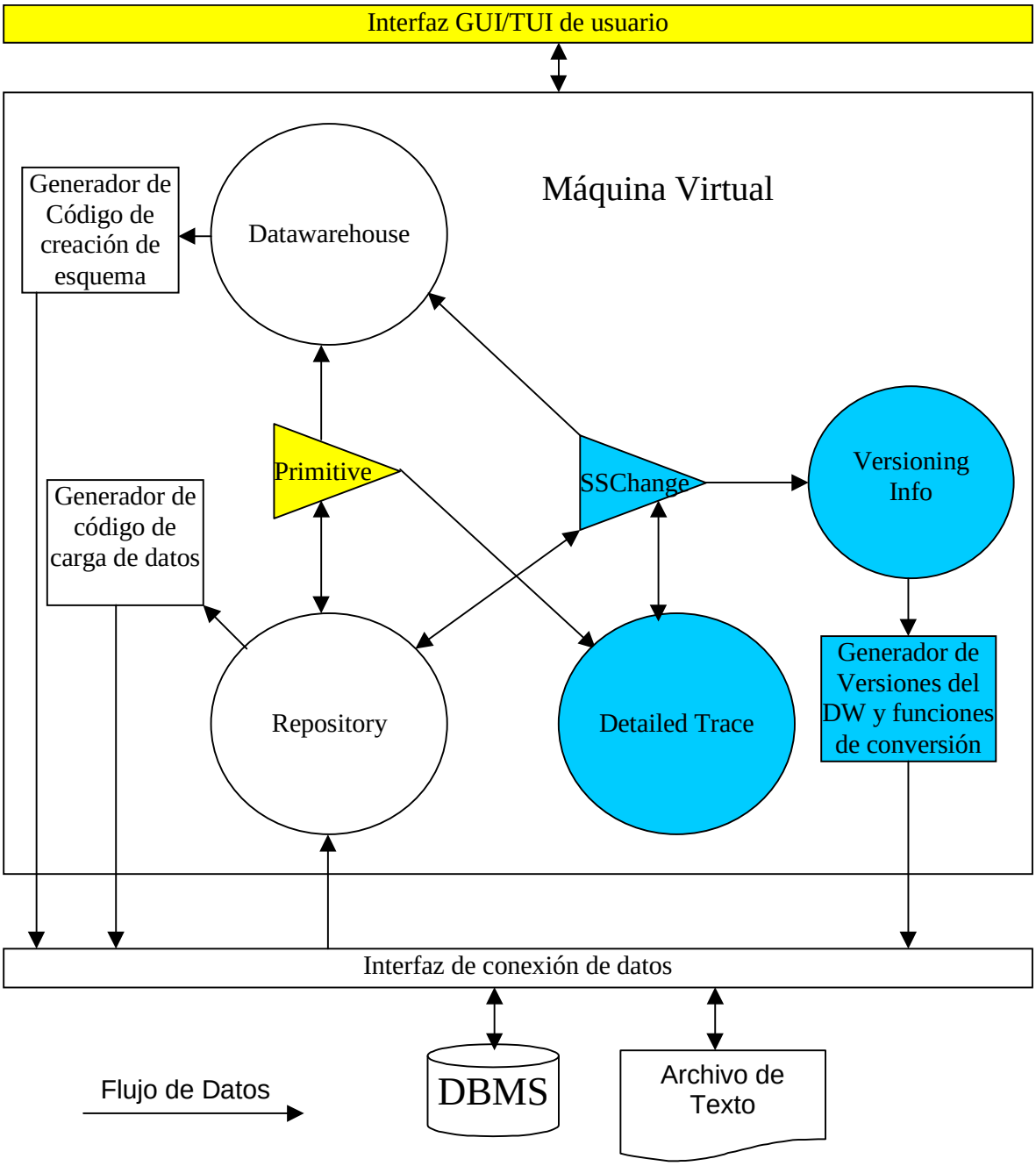


Figura 3.2 módulos que componen la herramienta

A los objetivos anteriores se le agrega mantener el DW actualizado y consistente a medida que el esquema fuente va cambiando, mediante la propagación de los cambios a través de la traza.

Principales componentes

La Máquina Virtual, se descompone principalmente en tres sub-módulos que realizan distintas funciones.

- *Herramientas para deducir la evolución del DW.*

  Uno de los principales propósitos de la Máquina Virtual, incluído por esta herramienta, es hacer que el DW evolucione consistentemente y en paralelo con el esquema fuente. Esta funcionalidad se realiza a través de los Cambios en el Esquema Fuente (SSChanges, Source Schema Changes) y las operaciones básicas (Basic Operations).

- *Traza de transformación*

  La trazabilidad del diseño se realiza mediante este sub-módulo. Se registra cada primitiva aplicada junto con las relaciones de entrada y salida. Las primeras relaciones en ser utilizadas como entrada son las relaciones iniciales.

- *Traza Detallada*

  Traza a nivel de operaciones básicas, esta es la traza que hay que seguir para saber que cambios son necesarios en el DW y en el camino para generarlo, para que siga siendo consistente con el esquema fuente cuando éste cambia.

Herramientas para deducir la evolución del DW.

Aquí describimos los sub-módulos responsables de deducir las dependencias y propagar los cambios desde el esquema fuente hacia el DW, es decir, aquellos que deducen la evolución que debe tener el DW para que siga siendo consistente con la base de datos fuente.

Listas de Dependencias

En estas listas almacenamos las dependencias que se producen entre las entradas y las salidas de las operaciones básicas que hay en los caminos que van desde el elemento modificado al DW, como describimos en la sección 4.3.1 Mecánica de Funcionamiento

Operaciones Básicas

Las operaciones básicas permiten describir las primitivas en función de operaciones más elementales y sencillas de entender, identificando bien el conjunto de operaciones básicas necesario para describir cualquier primitiva y luego la descomposición en operaciones básicas, permiten obtener las dependencias que hay entre sus entradas y sus salidas.

La clase que implementa las operaciones básicas es *BasicOperation* donde cada tipo de operación básica es una subclase de *BasicOperation*.

Cambios al Esquema Fuente

Los cambios al esquema fuente, son los objetos de la clase *SSChange*, la clase *SSChange* es abstracta y las subclases concretas van a representar los distintos cambios, la idea de los cambios es que sean extensibles, es decir que se puedan definir y agregar nuevos cambios, la forma de hacerlo es simplemente crear una pantalla que permita ingresar sus parámetros, implementar una subclase concreta de *SSChange* que aplique el cambio deseado, y agregar el cambio en la lista de cambios del sistema.

Core

Esta entidad es la Máquina Virtual. Ella contiene cada uno de los componentes necesarios para realizar las transformaciones de las primitivas. Algunos de esto componentes son:

- *PrimitiveDirectory*, contiene las primitivas presentes en la herramienta
- *FunctionDirectory* , contiene las funciones de que dispone el implementador.
- *DomainDirectory*, contiene los dominios de que dispone el implementador.
- *Repository* y *Datawarehouse*
- *TransformationTrace* , la traza

Cómo solo tiene sentido un objeto de esta clase en toda la aplicación, dado que todas las clases deben acceder a los mismos componentes, se utilizó una adaptación del pattern Singleton de [4], el Pattern provee una forma de realizar una clase de la cual nos aseguramos que solo hay una instancia en todo el sistema, para no modificar el funcionamiento de las partes de DWD 99 se utilizó una adaptación que si bien no nos asegura que todos los objetos del sistema usen la misma máquina virtual, nos asegura que cuando creamos una nueva Máquina Virtual, utilice exactamente los mismos componentes, esto se hizo poniendo como estáticas todas sus variables miembro.

Se definieron métodos para salvar este objeto a disco y después poder leerlo.

5. Implementación

Este capítulo describe las decisiones que consideramos más importantes tomadas a la hora de implementar y los cambios que hicimos en el proyecto previo para realizar nuestra tarea en forma más eficiente.

5.1. Introducción.

La tarea de implementación requirió un gran porcentaje de tiempo de dedicación al proyecto. Este capítulo habla de cómo implementamos nuestros principales objetivos: la deducción de dependencias frente a un cambio en el esquema fuente y la propagación de los cambios. Parte de las decisiones de implementación consistieron en realizar cambios a la herramienta existente. Decisiones tomadas en el proyecto anterior estaban realizadas con el fin de optimizar el diseño de un DW. Sin embargo, esas decisiones no nos resultan cómodas a la hora de realizar nuestra tarea lo que nos hizo cambiar algunas de ellas así como agregar objetos nuevos para poder concentrarnos mejor en la tarea de evolución.

En la sección 4.2 se habla del ambiente de desarrollo utilizado. En las secciones 4.3 y 4.4 se habla de deducción de dependencias y propagación de cambios respectivamente. En la sección 4.5 se explica el método que utilizamos para recorrer la traza. En la sección 4.6 comentamos los agregados y modificaciones que le hicimos al proyecto anterior. Finalmente en la sección 4.7 se habla de cómo hacemos para que la herramienta sea extensible.

5.2. Lenguaje de programación y ambiente de desarrollo.

La herramienta fue totalmente desarrollada en Java. Esto era un requerimiento del proyecto ya que la primer parte del mismo fue desarrollada en Java y debímos usar el mismo lenguaje.

La herramienta de diseño usada también fue la misma: Rational Rose®. Ésta herremienta permite realizar diseños en UML de manera sencilla y además generar parte del código (cabezales de las clases Java) en forma automática.

Como entorno de desarrollo se utilizó JBuilder 3® de Borland al igual que en la primer parte del proyecto. Éste producto posee una gran facilidad de uso, varias facilidades para debug del código y permite diseñar interfaces gráficas en forma sencilla. JBuilder 3® permite diseñar ventanas en forma visual y genera código standard lo que nos resultó de mucha ayuda. Un gran competidor a la hora de elegir el ambiente de desarrollo fue VisualAge for Java®. Ésta herramienta posee las facilidades descriptas anteriormente y aún más. Sin embargo no era compatible con JBuilder 3®(entorno usado en el proyecto anterior) por lo que fue descartada.

5.3. Operaciones básicas

El primer paso en la implementación de nuestro proyecto fue implementar las operaciones básicas según las especificaciones de [1], ya que el diseño de un DW se hace en base a aplicaciones de primitivas, pero estas primitivas pueden expresarse en pequeñas unidades llamadas operaciones básicas. Cada primitiva es equivalente a aplicar una secuencia de operaciones básicas (en el apéndice B de [1] se especifica la equivlencia entre primitivas y operaciones básicas).

Las operaciones básicas son una parte fundamental de nuestro proyecto, ya que el impacto de las modificaciones en el esquema fuente (eliminar atributos, agregar atributos, cambio de claves, renombrar atributos, etc.) sobre el DW se realiza modificando los parámetros de las operaciones básicas.

Los objetivos que tenemos al implementarlas son que cumplan las especificaciones según [1] y que sean extensibles. Es decir, al igual que las primitivas son extensibles, lograr lo mismo con las operaciones básicas, para que en el futuro si se desea agregar alguna sea fácil de incorporarla.

5.4. Traza detallada

Como se mencionó en la sección anterior, cada primitiva es equivalente a una secuencia de operaciones básicas. Esto es necesario desde 2 puntos de vista, primero, porque cuando ocurre un cambio en el esquema fuente, es necesario obtener las dependencias del mismo respecto al DW, que sólo se puede lograr a nivel de operaciones básicas. El segundo motivo es para propagar los cambios una vez que se producen en el esquema fuente, las modificaciones también se propagan a nivel de operaciones básicas (modificación de prarámetros según las reglas de propagación provistas en [1]).

La estructura usada para representar la secuencia de aplicación de cada primitiva en términos de operaciones básicas fue la de un grafo. Cada grafo de operaciones básicas en una primitiva lo llamamos traza detallada. Para obtener las dependencias o propagar cambios es necesario recorrer la misma (y alterarla si es necesario). La figura 4.1 representa un esquema de una traza detallada en una primitiva.

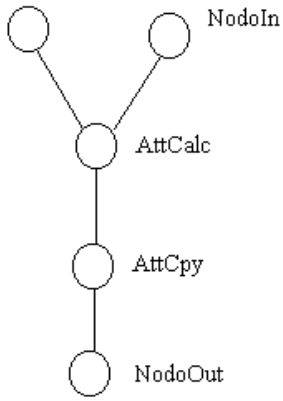


Figura 4.1

La traza detallada que muestra la figura es la que se incluye dentro de cada primitiva. La idea de la traza es tener 3 tipos de nodos

- NodosIn. Tienen como entrada las relaciones que son entrada de la primitiva.
- Nodos Internos. Contienen operaciones básicas con los parámetros correspondientes.
- NodosOut. Tienen como salida las relaciones de salida de las primitivas.

Otra aclaración importante sobre la traza es que las aristas que conectan los nodos tienen la información de la relación que representan. Por ejemplo, una arista entre 2 nodos internos tendrá la información de la relación de salida de la operación básica del nodo de entrada.

Incluimos la traza detallada dentro de cada primitiva en el momento en que ésta se aplica. Ya que para cada primitiva se tiene una traza detallada en particular y es en el momento de aplicarla en el que se disponenen de todos los parámetros para generarla.

Hasta el momento se cuenta con 2 niveles, uno es el nivel de las primitvas utilizado para el diseño del DW, el otro nivel, es un nivel más fino a nivel de operaciones básicas que se encuentra dentro de cada primitiva (traza detallada), este nivel es utilizado para manejar los cambios en el DW a partir de los cambios en el esquema fuente. En la siguiente figura 4.2 se muestran los 2 niveles mencionados.

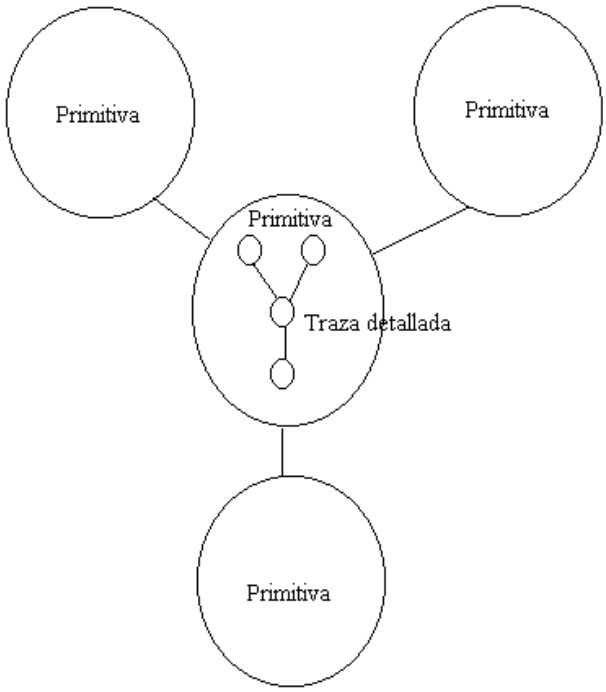


Figura 4.2

5.5. Deducción de dependencias

5.5.1. Motivación

Una parte fundamental de nuestro proyecto fue como a partir de un cambio en el esquema fuente deducir las dependencias en el DW. Necesitamos saber que atributos y/o relaciones del DW deben ser modificadas luego de aplicar el cambio. Además de eso nos interesa actualizar también el camino en la traza para que también las relaciones intermedias queden consistentes.

A modo de ejemplo, supongamos que tenemos el siguiente diseño:

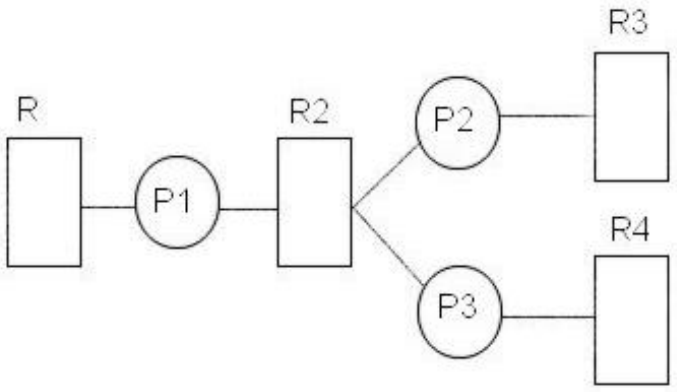


Figura 4.3

Dada una relación de entrada R, le aplicamos una primitiva P1 y obtenemos una relación de salida R2. A R2 le aplicamos las primitivas P2 y P3 y obtenemos las relaciones R3 y R4 respectivamente. Nuestro DW está formado en este caso por R3 y R4. La pregunta que queremos responder ahora es la siguiente:

Si hacemos un cambio en R (relación del esquema fuente) ¿qué debemos modificar en R2 (relación intermedia), en R3 y R4 (relaciones del DW)? En la siguiente sub sección intentamos contestar esta pregunta.

5.5.2. Solución

En la figura 4.3 observamos que nuestro diseño está formado por relaciones del esquema fuente, relaciones que pertenecen al DW y relaciones intermedias (R2 en este caso). Por lo tanto no nos alcanza sólo con actualizar el DW sino que debemos actualizar también todas las relaciones intermedias. Para esto debemos tener guardados los caminos que van desde la relación R modificada en el esquema fuente hasta las relaciones del DW que tienen alguna dependencia con R.

Lo que hacemos para solucionar este problema es tener listas con los caminos desde el esquema fuente al DW. En el ejemplo anterior tenemos dos listas de dependencias, una que va desde R a R3 y la otra que va desde R a R4, ambas pasando por R2. Cada vez que se aplica un cambio en el esquema fuente obtenemos estas listas de dependencias, las mostramos al usuario y propagamos los cambios si el usuario decide continuar. Para implementar esta lista se usa una estructura de datos que nos permita mostrar al usuario todo lo que este necesite ver y además usamos la misma lista para propagar los cambios.

5.5.3. Implementación de la lista de dependencias

La estructura de datos seleccionada se describe con detalle a continuación:

A la hora de hablar de dependencias tenemos tres niveles diferentes de granularidad:

1) A nivel de una sola operación básica.
2) A nivel de una primitiva.
3) A nivel del diseño completo.

1) En un primer término estudiamos la dependencia que puede tener un atributo cuando a éste se le aplica una operación básica. Si el atributo pertenece a las relaciones de entrada de la operación entonces debe tener una de las siguientes dependencias en la relación de salida:

- Copiado
- Usado en el cálculo
- Requerido para el cálculo
- No tiene dependencia

La clase operación básica cuenta con un método getDependency() el cual nos devuelve un arreglo de objetos de la clase Dependency que nos da toda la información necesaria si queremos deducir las dependencias de un atributo en una operación básica.

El objeto Dependency contiene cuatro campos:

- Tipo de dependencia (una de las cuatro mencionadas anteriormente)
- Relacion de salida
- Atributo de salida
- Una referencia a la operación básica misma

Ejemplo

A continuación ponemos un ejemplo que usa el método getDependency() para devolver un objeto de la clase Dependency:

Supongamos que tenemos una cierta relación R1 con los siguientes atributos:

R1 = { nombre, apellido, dirección, teléfono }

y que a esta relación le aplicamos la operación básica AttRem con el siguiente conjunto de atributos a borrar: X = { dirección, teléfono }

La relación de salida luego de aplicada la operación básica sería la siguiente (suponemos que tiene nombre R2):

R2 = { nombre, apellido }

Supongamos que queremos saber que dependencia tiene el atributo Apellido en la operación básica, para eso llamamos a getDependency() con la relación R1 y el atributo Apellido como parámetros. Esto nos devuelve un objeto de la clase Dependency que se muestra a continuación:

Lo que nos dice este objeto es que el atributo Apellido está copiado en R2 y también se guarda una referencia a la operación básica que va a ser parte del camino que luego construiremos uniendo las operaciones básicas. Si hubiéramos querido obtener la dependencia en la operación básica del atributo Dirección nos hubiera dado NODEPENDENCY ya que el atributo fue eliminado en la operación básica. En este caso el arreglo de dependencias contiene un solo elemento aunque puede haber casos donde tengamos más de uno.

♦

Si bien en el ejemplo anterior el método getDependency() devuelve una sola dependencia cuando usamos la operación básica AttCalc puede haber más de una, por eso el método devuelve un arreglo de dependencias. Este caso se puede dar cuando usamos alguna función de cálculo para generar un nuevo atributo y tanto el atributo de salida como los de entrada pertenecen a la misma relación.

Para fijar más las ideas ponemos otro ejemplo:

Ejemplo

Supongamos que tenemos la relción R1 con los siguientes atributos: producto, unidades_vendidas e ingresos_totales y que queremos agregar un nuevo atributo a esta relación que se llame ingreso_promedio y sea justamente eso: el total de ingresos sobre unidades vendidas.

R1 = { producto, unidades_vendidas, ingresos_totales }

Para eso debemos aplicar la función de cálculo COCIENTE con los atributos ingresos_totales y unidades_vendidas como parámetro y esto generará el nuevo atributo. Este cálculo se podría hecer con la operación básica AttCalc especificada en [1] y se genera una relación R2 con los tres primeros atributos y además con el nuevo.

R2 = { producto, unidades_vendidas, ingresos_totales, ingreso_promedio }

¿Qué pasaría si quisiéramos saber que dependencia tiene unidades_vendidas de R1 en la aplicación de esa operación básica? Por un lado está copiado tal cual en R2 pero además participa en la función de cálculo de ingreso_promedio o sea que tiene dos dependencias: una como copia y la otra como usado en el cálculo. En este caso dada una relación de entrada y una de salida tenemos dos dependencias y por lo tanto en la implementación deberíamos devolver dos objetos Dependency. Si ocurre esta situación cada dependencia se trata por separado, si hay dos se armarán dos caminos en la lista de dependencias.

♦

2)Ya hablamos de cómo se deducen las dependencias al aplicar una operación básica, ahora nos podemos preguntar como guardar las dependencias en una primitiva completa. Aquí encontramos más complicaciones que en el caso anterior ya que las primitivas están compuestas en general de varias operaciones básicas y pueden tener más de una relación de salida. Justamente como las primitivas están compuestas de operaciones básicas resulta interesante usar el método anterior getDependency() reiteradas veces en una primitiva (para cada operación básica) e ir armando los caminos.

Esos fueron los motivos que nos llevaron a elegir la siguiente estructura de datos llamada DependenciesList, para guardar las dependencias en una primitiva. Ésta estructura se compone de los siguientes campos:

relation relIn;
Attribute attIn;
ArrayList[] Dependency;

El objeto se podría ver como en la figura 4.4:

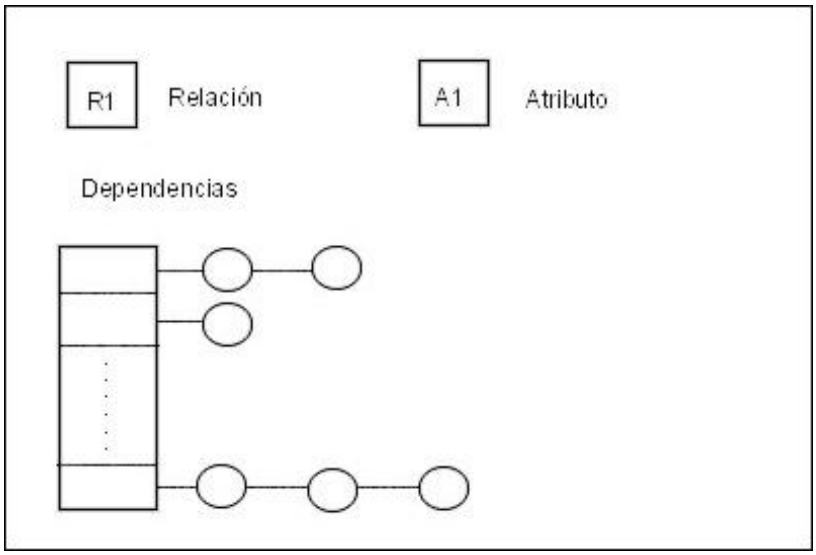


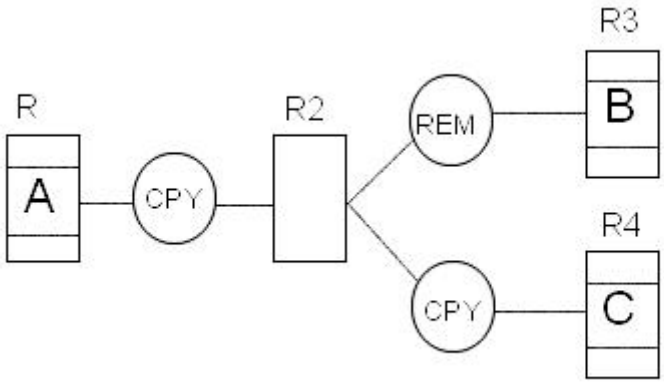
Figura 4.4

Lo primero que vemos en el objeto de la clase DependenciesList es que contiene a una relación y a un atributo. Este atributo es un atributo de la relación R1 y es el atributo de la base fuente al que se le aplicó uno de los cambios de la taxonomía de evolución (por ejemplo: a R1 se le renombró el atributo A1). En el caso que el cambio de la taxonomía involucre a una relación completa (por ejemplo se renombró una relación y no interesan sus atributos) el atributo A1 es nulo. En otras palabras, la relación R1 y el atributo A1 son los objetos del esquema fuente a los cuales se les aplicó el cambio.

El otro objeto (Dependencias) en la figura es un arreglo de listas de dependencias. Cada lista del arreglo es una lista de objetos de la clase Dependency ya mencionada y cada lista contiene un camino de dependencias de principio a fin de la traza detallada.

En cada primitiva para hallar la lista de dependencias se usa el método getDependencies(), éste método se invoca en cada primitiva, recorre la traza detallada y llama a getDependency() en cada operación básica.

El método getDependencies() funciona de la siguiente manera: se invoca desde la traza detallada (que está en una primitiva) y este método se encarga de recorrer la misma. Cada vez que este método encuentra una operación básica en el grafo de la traza detallada llama al método getDependency() y obtiene un objeto de la clase Dependency (excepto si la operación básica es AttCalc que obtiene varios pero el procedimiento es el mismo). Con estos objetos va armando la lista de dependencias y va a haber una lista de dependencias por cada camino desde el atributo de entrada a alguna relación de salida. El funcionemiento del método getDependencies() se describe con un ejemplo a continuación:



Ejemplo

Supongamos que tenemos el diseño que se muestra en la figura 4.5 (ilustrado a nivel de operaciones básicas) y que queremos deducir las dependencias de un determinado atributo A de la relación R el cual tiene dos dependencias en la traza detallada, una hacia un atributo B de la relación R3 y otra hacia un atributo C de la relación R4. Entonces cuando llamemos desde esta traza detallada al método getDependencies() nos va a devolver un objeto de la clase DependenciesList conteniendo la siguiente información:

El objeto relación contiene R y el objeto atributo contiene A. Esto nos dice que se quieren hallar las dependencias en esta traza detallada a partir del atributo A de la relación R (este método trata a esta traza detallada como si fuera un diseño completo y deduce las dependencias en él. Luego nos encargaremos de juntarlo con el resto de los caminos). En la lista de dependencias va a haber dos caminos: uno conteniendo el que va desde A hacia el atributo B de R3 y el otro que va desde el atributo A hacia el atributo C de R4. Ahora tenemos entonces toda la información necesaria sobre los atributos B y C de R3 y R4 respectivamente que son los que dependen del atributo A de R que era el atributo a partir del cual queríamos hallar las dependencias en la traza detallada. Nuestro próximo paso es ver como ir uniendo estas listas partiendo de la base fuente y llegando al DW para obtener las dependencias a lo largo de toda la traza de transformación.

♦

3) Para deducir las dependencias en el diseño completo (esto es, a partir de un atributo en el esquema fuente hallar las dependencias en el DW) contamos con un tercer método llamado getAllDependencies() que devuelve un objeto del tipo DependenciesList (el mismo que se describió anteriormente), éste método obtiene las dependencias en todo el camino y se invoca para atributos o relaciones que pertenezcan al esquema fuente.

La secuencia típica de llamadas es la siguiente:

a) Se efectúa un cambio en el esquema fuente y se quieren deducir las dependencias

b) Se llama al método getAllDependencies() con un atributo o relación del esquema fuente (depende de cual sea el cambio de la taxonomía de evolución). Éste método es recursivo, recorre la traza de transformación hacia el DW y para cada primitiva que encuentra en el camino llama al método getDependecies(), éste calcula las dependencias en cada primitiva recorriendo la traza detallada y llama al método getDependency() para cada operación básica que encuentra en el camino. El método getAllDependencies() se encarga de armar la estructura de datos uniendo las listas que le devuelve el método getDependencies() y luego que termina la recursión devuelve la estructura de datos completa.

c) Luego de obtenidas las dependencias, estas se muestran al usuario y éste decide aceptar o cancelar la propagación de los cambios.

Como se explicó anteriormente, el proceso de deducción de dependencias opera a varios niveles contando con tres métodos, cada uno de ellos trabaja con un nivel de granularidad diferente. El método getAllDependencies() opera en el diseño completo y recorre el camino desde el esquema fuente hasta el DW. Sin embargo éste método no se introduce dentro de las primitivas para hallar las dependencias. Simplemente se encarga de recorrer la traza de transformación y cada vez que encuentra una primitiva llama al método getDependencies() que es el encargado de hallar las dependencias al nivel de primitivas. Éste método a su vez recorre como dijimos la traza detallada y cada vez que encuentra una operación básica llama a getDependency() y es éste el único método encargado de introducirse dentro de las operaciones básicas y hallar las dependencias.

5.6. Propagación de los cambios

Luego de haber deducido las dependencias y saber que partes del DW se ven afectadas al hacer un cambio en el esquema fuente el usuario puede aceptar propagar los cambios o cancelar la operación. Si se opta por propagar los cambios debe actualizar la traza detallada y el DW si es necesario. La tarea de propagar los cambios depende del cambio realizado y de la dependencia de cada atributo. Estos cambios se especifican en [1].

Para propagar los cambios se usa la estructura DependenciesList que fue construida al deducir las dependencias. Luego de construida esta estructura se tiene en ella toda la información necesaria para propagar los cambios. A continuación se describe la forma en que se implementa la propagación de cada cambio.

Remover atributo

Para este cambio tenemos varias reglas de propagación. Las mismas difieren según el tipo de dependencia que se tenga. Para simplificar la explicación nos concentramos en el caso que el atributo del DW fue copiado del atributo del esquema fuente o sea que la dependencia es de copia (por más información ver [1]). En este caso la regla de propagación nos dice que se deben borrar todos los atributos que dependen del atributo borrado junto con los respectivos caminos en la traza. Para implementar la solución a este problema usamos la lista de dependencias previamente construida. Ya tenemos en la misma los caminos a seguir para la propagación de los cambios. Desde RemoveAttribute se llama al método propagate() que se encarga de hacer dos cosas: primero actualiza los parámetros en cada operación básica y luego borra físicamente el atributo de cada relación de los caminos. Lo que se hace primero que nada es actualizar los parámetros de las operaciones básicas para que los mismos queden consistentes.

Ejemplo

En una operación básica AttCpy(R1,R2,X) la cual copia el conjunto X de atributos de R1 a R2, el conjunto X de atributos debe permanecer consistente. Si aplicamos un cambio en el esquema fuente borrando un atributo que tiene una dependencia en la operación de copia en uno de los atributos del conjunto X el atributo debería ser borrado no solo de la base fuente y del DW sino también del conjunto X y eso sólo se puede hacer entrando en la operación básica. Esto es justamente lo que en [1] se especifica como remove path. Para mantener actualizada la traza detallada se recorre la lista de dependencias que contiene todas las operaciones básicas de los caminos que nos interesan. Luego de eso se borran los atributos de las relaciones o sea se actualizan el esquema fuente y el DW además de toda relación intermedia que esté involucrada en el camino. Esto equivale a Att_rem en [1]

En el caso de que la dependencia sea 'requerido' el procedimiento es el mismo que si es 'copiado'. Si la dependencia es 'usado en el cálculo' el usuario tiene la posiblidad de borrar el atributo que fue calculado o de cambiar la función de cálculo. Si se elige cambiar la función de cálculo el usuario aplica otra función que no tenga el atributo elimiando como parámetro. Si se elige borrar el atributo el procedimiento es el mismo que en los casos anteriores.

♦

Renombrar atributo.

Nuestra intención es hacer que el DW cambie lo menos posible por lo que al renombrar un atributo en el esquema fuente no deseamos que los atributos del DW que tienen dependencias para con éste cambien de nombre. Sin embargo queremos que las dependencias se mantengan aunque los atributos del DW que dependen del atributo del esquema fuente no cambien de nombre. Por ejemplo, si

queremos cambiar el nombre de determinado atributo de APELLIDO a LASTNAME, sólo se verá el cambio en el esquema fuente y no en el DW. Sin embargo si luego queremos hacer otro cambio estamos interesados en que el atributo mantenga sus dependencias aunque haya cambiado de nombre. Esto no nos es difícil de implementar ya que nosotros trabajamos con el objeto atributo y no con los nombres por lo que las dependencias de un atributo siguen siendo las mismas aunque este cambie su nombre.

Lo que debemos hacer es guardar en algún lado un mapeo entre el nombre viejo del atributo y el nuevo. Para eso incluimos en la primer operación básica de las primitivas que tomaban relaciones del esquema fuente una tabla de hash que realiza ese mapeo y guarda la correspondencia entre los nombres viejos y nuevos. De ésta manera, al hacer el mapeo lo antes posible no sólo evitamos que el DW cambie lo menos posible sino que también lo haga la traza detallada que es el objeto que mantiene actualizado el DW.

Agregar atributo.

Al agregar un atributo en el esquema fuente no tenemos reglas de propagación ya que al ser nuevo el atributo en el esquema no tenemos dependencias. El usuario puede aplicar primitivas a la relación que contiene el atributo y de esa manera genera nuevas dependencias del nuevo atributo hacia el DW.

Agregar relación.

Si agregamos en la base fuente una relación, la misma no tiene dependencias en el DW. Sin embargo el usuario puede aplicar primitivas para que se generen dependencias a partir de la relación recientemente creada.

Lo que hacemos al agregar una relación es simplemente preguntar el nombre de la misma, los atributos que van a estar contenidos en ella y que se defina la clave primaria de la relación.

Renombrar relación.

Las relaciones se identifican por su nombre. No puede haber en el repositorio (conjunto de todas las relaciones) dos relaciones con el mismo nombre. Entonces renombrar una relación es sencillo ya que no hay cambios que propagar, simplemente se cambia el nombre de la relación del esquema fuente. Como las dependencias son manejadas al nivel de objetos y no de nombres, el hecho de que se renombre una relación no afecta las dependencias por lo que no hay que hacer nada más. La relación cambió de nombre pero las dependencias van a ser siempre las mismas ya que se deducen a partir del objeto relación independientemente de su nombre.

Borrar relación.

Cuando se borra una relación en el esquema fuente se borran todas las relaciones del DW que sean salida de primitivas que tomaron como entrada la relación eliminada. También se eliminan todas las relaciones intermedias. A continuación se describe un ejemplo:

Ejemplo

Sea R1 = { Nombre, Apellido, CI, Dirección, Telefono, Fecha }

Supongamos que a R1 se le aplica cierta primitiva P1 la cual genera como salida una relación R2:

R2 = { Nombre, Apellido, CI, Dirección, Teléfono, Mes }

A su vez a R2 se le aplica otra primitiva P2 la cual devuelve como entrada R3:

R3 = { CI, Dirección, Teléfono , Mes }

Esta relación R3 pertenece a nuestro DW. Si luego deseamos eliminar R1 del esquema fuente la regla de propagación eliminará R2, R3 y ambas primitivas ya que la aplicación de las mismas no puede existir si se eliminan las relaciones involucradas.

♦

5.7. Recorrida de la traza

Para realizar la tarea de evolución, la traza es de fundamental importancia. Cada vez que se hace un cambio es necesario recorrer la traza para actualizarla y saber como llegar al DW para modificar los atributos y/o relaciones necesarias. Para esto necesitamos escoger una estructura de datos apropiada para la traza de transformación. La misma ya estaba implementada pero tenía un enfoque distinto al que nosotros necesitábamos, estaba hecha de forma que el diseño fuera fácil de manejar en la traza y no estaba planeado para una futura evolución de esquemas. La traza de transformación estaba originalmente implementada por niveles (se manejaba en forma similar a los niveles en una estructura de árbol) y cada vez que se generaba una primitiva teníamos niveles nuevos en la misma (cada nueva primitiva se insertaba en un nuevo nivel). Esta manera de implementar la traza optimizaba el proceso de diseño pero hacía difícil su manejo cuando queríamos propagar cambios a la hora de cambiar el esquema fuente. Por un lado la estructura de datos mencionada anteriormente era usada para el diseño del DW por lo que un cambio en la estructura de datos resultaba bastante complicado de implementar y mantener. Por otro lado la estructura existente no nos molestaba, sino que necesitábamos una estructura más fácil de recorrer. Fue por eso que decidimos mantener la estructura de datos original para la traza de

transformación pero agregando referencias para que nos fuera más fácil recorrerla a la hora de propagar los cambios.

Para realizar esta tarea nos interesa saber dada una relación cuales son la primitivas que la toman como entrada y dada una primitiva cuales son las relaciones de entrada y salida a la misma. Para mantener guardada esta información lo que hicimos fue tener en cada relación dos listas, una conteniendo referencias a las primitivas que toman como entrada a dicha relación y otra con las primitivas que la devuelven como salida. Teniendo estas listas en cada relación se nos hace más fácil recorrer la traza para propagar los cambios, lo hacemos procediendo de la siguiente manera: dado un atributo en una relación del esquema fuente podemos saber cuales son las primitivas que toman como entrada a la relación a la que pertenece el atributo, simplemente buscamos en la lista. Luego de obtenidas las primitivas que toman a dicha relación como entrada nos introducimos en cada una de ellas recorriendo la traza detallada y hallando las dependencias, cuando llegamos al final de cada primitiva, podemos acceder a las relaciones de salida y luego buscar nuevamente las primitivas que toman a esas relaciones como entrada repitiendo el proceso en forma recursiva hasta llegar al DW.

El proceso es sencillo ya que sólo se necesita buscar en las listas para saber las primitivas dentro de las cuales debemos introducirnos, luego en cada primitiva solo tenemos que llamar a los métodos de traza detallada que son los que se introducen dentro de las operaciones básicas. Por supuesto que las listas anteriores necesitan actualizarse. Cada vez que se agregue o se borre una primitiva (la eliminación de primitivas puede pasar por ejemplo al remover un atributo con gran cantidad de dependencias) hay que refrescar esta lista para que apunte a nuevas primitivas o deje de hacerlo si alguna fue eliminada. Sin embargo, con estas listas podemos abstraernos de la estructura de datos que se usó en el proyecto anterior para construir y recorrer la traza. Si bien el DWD 99 permite a partir de una relación saber cuales son las primitivas que la toman como entrada, esta estructura resulta bastante compleja de usar para aplicar cambios y hacer la tarea de evolución. Con estas listas que implementamos, podemos tener una manera sencilla de recorrer la traza para hallar las dependencias y propagar los cambios.

Con esta estructura de datos también se hace sencillo saber si determinada relación pertenece al esquema fuente o al DW. Una relación pertenece al esquema fuente si y sólo si la lista de primitivas que devuelven como salida a dicha relación es vacía. De la misma manera, una relación pertenece al DW si y sólo si la lista de primitivas que toman a dicha relación como entrada es vacía.

5.8. Implementación subyacente

La mayor parte de la implementación de nuestro proyecto se basa en agregados al proyecto anterior por lo que el mismo posee una gran cantidad de código de base. Gran parte de la implementación está hecha en clases que no poseen interfaz gráfica, es más, una gran parte del código implementado está agregado en clases previamente existentes, un ejemplo de esto es la creación de la traza detallada en cada primitiva. Cada primitiva está compuesta de operaciones básicas que forman la traza detallada. Cada vez que se instancia una primitiva se construye la traza detallada asociada a la misma por lo que el código de construcción de esta fue agregado en cada primitiva quedando totalmente subyacente. Además de éste agregado tuvimos que agregar estructuras de datos que nos facilitaran la implementación de nuestra tarea ya que nuestra herramienta debía interactuar con DWD 99 y estaba pensado para optimizar esa tarea y no para evolución (un ejemplo de esto son las listas de relaciones que mencionamos en la parte anterior). Por ese motivo gran parte del código queda detrás de lo que ya se había implementado.

Además de eso se hicieron agregados en el proyecto DWD 99 que nos resultaron útiles para nuestra tarea. Un ejemplo de esto fue la implementación de las funcionalidades "Save Design" y "Open Design". Al implementar esto se permite guardar el diseño de un DW en cualquier estapa del mismo, incluso después de haber aplicado cambios al esquema fuente. Esto es de gran ayuda ya que permite ahorrar tiempo en el diseño y poder tener gran cantidad de versiones del mismo.

5.9. Extensibilidad de la herramienta.

El diseño fue pensado para que la herramienta fuera extensible y eso debía estar reflejado en la implementación. Desde la primera parte del proyecto se pensó en la extensibilidad de la herramienta, DWD 99 estaba diseñado de tal forma que agregar nuevas primitivas fuese fácil. La orientación a objetos de Java permitió que esto se implementara en forma sencilla. Tratamos de mantener la misma postura y también pensamos en la posible extensión de la herramienta, sin embargo la decisión fue diferente. Si bien el diseño está hecho para que se puedan agregar operaciones básicas (es casi tan simple como agregar una nueva clase que herede de BasicOperation y redefinir los métodos en forma apropiada) la especificación fue pensada para que todas las primitivas y otras que pudieran ser agregadas pudieran ser construidas con las operaciones básicas existentes, entonces una extensibilidad en cuanto a operaciones básicas no resulta de gran utilidad (aunque de todas maneras se encuentra presente). Lo que resulta más útil es pensar en la extensibilidad en cuanto a los cambios en el esquema fuente, esto es más factible ya que nuevos cambios más complicados podían ser diseñados e implementados en forma sencilla.

Cada cambio en el esquema fuente está implementado como una clase que hereda de SSChange, si se desea agregar un nuevo cambio lo que se debe hacer

es una clase que herede de esta y redefinir los métodos para obtener las dependencias y propagar los cambios. Luego de eso se debe pasar a la interfaz gráfica. Cada cambio en el esquema es manejado con un panel el cual contiene los parámetros necesarios para la aplicación del mismo, estos paneles se encargan además de manejar los eventos de los botones que contienen y llamar a los métodos de deducción de dependencias y propagación de cambios. Si se quisiera agregar un nuevo cambio el segundo paso sería crear un nuevo panel de este tipo y agregarlo a la lista de paneles. Luego de terminar con estos dos pasos el nuevo cambio quedaría listo e incorporado a la herramienta.

5.10. Límites de la implementación.

En esta sección se describe la implementación que quedó fuera del alcance de nuestro proyecto para que sea tenido en cuenta para la continuación del mismo.

5.10.1. Futuras extensiones

En [1] se especifican 14 Primitivas (20 si contamos las subprimitivas) que fueron implementadas en DWD 99. Nuestra tarea era implementar la traza detallada de cada primitiva. Se logró implementar la traza detallada para el 80% de las primitivas. A continuación se nombran dichas primitivas:

P1. Identity
P2. Data Filter
P3. Temporalization
P4.1 Key Extension
P4.2 Version Digits
P5 Foreign Key Update
P6.1 DDAdding11
P6.2 DDaddingN1
P6.3 DDaddingNN
P7 Attribute Adding
P8 Hierarchy Roll Up
P9 Agregate Generation
P11.1 Vertical Partition
P11.2 Horizontal Partition
P12.1 Denormalizad Hierarchy Generation
P14 New Dimension Crossing

Las primitivas que no fueron implementadas son las siguientes:

P10 Data Array Creation
P12.2 Totally Normalized Hierarchy Generation (SnowFlake)
P12.3 Free Decomposition
P13 Minidimension Break Off

6. Conclusiones

El objetivo principal de este proyecto fue la implementación de una herramienta para la evolución de un DW basado en [1], como extensión de la herramienta DWD 99. Para lograr este objetivo pasamos por las etapas de análisis, diseño e implementación.

En la etapa de análisis además de estudiar las especificaciones de la herramienta que se debía implementar y la viabilidad del proyecto, realizamos un estudio de los casos de bordes en la evolución de un DW. Este análisis es un aporte teórico a [1], en él se estudian los posibles cambios que se pueden realizar sobre el esquema fuente dejando inconsistente la semánticas de las primitivas (aparece en el capítulo 3 de este documento).

Luego debíamos diseñar los componentes de la herramienta como ser : operaciones básicas, traza detallada , posibles cambios sobre el esquema fuente, las reglas de propagación y las interfaces. Al inicio de esta fase tuvimos que tomar una decisión importante, que afectaría el diseño y la herramienta misma. Las opciones eran:

- que la nueva herramienta sólo permita evolucionar el diseño del DW obtenido con el DWD 99, sin ofrecer la posibilidad de volver al diseño del DW con DWD99

- permitir que el usuario pase del diseño del DW obtenido con el DWD99 a la herramienta de evolución, aplique los cambios, y vuelva a diseñar con el DWD99 una vez aplicados estos cambios.

A pesar de que el objetivo principal del proyecto era construir una herramienta que fuera una extensión del DWD99 para la evolución de un DW según [1] y nos hubiera resultado más sencilla la primera opción (porque podíamos realizar un diseño e implementación que no dependiera del diseño de DWD99), elegimos la segunda opción. Esta opción, si bien el diseño y la implementación es más difícil (porque tiene que adaptarse al proyecto anterior), nos parece la más acertada por que así la herramienta estaría completamente integrada al DWD99 dándole más libertad de acción al usuario. Una vez tomada

esta decisión, los principales desafíos fueron: expresar las primitivas en términos de operaciones básicas, diseñar la traza y la propagación de los posibles cambios del esquema fuente al DW. Otro tema importante a al hora del diseño fue el de lograr la extensibilidad de la herramienta. Los posibles cambios en el esquema fuente son fácilmente extensibles, este era el aspecto más importante en cuanto a la extensibilidad ya que la herramienta lo que hace es aplicar los cambios en el esquema fuente y propagarlos por lo que podrían surgir nuevos cambios. En cuanto a la extensibilidad de las primitivas, se sigue manteniendo la misma que la herramienta DWD99 sólo que ahora cada primitiva que se agrega, debe proveer además de sus funcionalidades, su expresión en términos de operaciones básicas.

En la implementación fue necesario modificar algunos módulos del DWD99, estos cambios no afectaban el funcionamiento del DWD99, pero era necesario hacerlos para nuestro proyecto. Uno de ellos fue que en el momento que el usuario aplica una primitiva, ahí se tienen todos los parámetros de la misma (porque no se conservan en ningún lado) por lo que tuvimos que modificar sus primitivas para que cuando estas se aplican, construyan su expresión en términos de operaciones básicas, de acuerdo a los parámetros ingresados por el usuario. También tuvimos que realizar correcciones, ya que encontramos primitivas que no funcionaban correctamente o no funcionaban. Se implementaron un 80% de las primitivas (por razones de tiempo), de 20 en total. Se implementaron tanto primitivas de complejidad baja como las de complejidad más alta. La implementación de la propagación de los cambios del esquema fuente al DW fue bastante compleja, por que para propagar los cambios es necesario recorrer y propagar los cambios en la traza de transformación y dentro de cada nodo de esta traza recorrer y propagar los cambios de su traza detallada. En cuanto a interfaz gráfica, se implementaron de acuerdo a los patrones elegidos para DWD99, ya que queríamos que las interfases fueran comunes para una mayor integración. Las interfaces implementadas fueron:

- para permitir aplicar los cambios, aquí el usuario selecciona el cambio a aplicar y luego sobre el objeto del esquema fuente que lo aplicará
- visualizador de dependencias, esta interfase muestra al usuario las dependencias del cambio que quiere aplicar en el DW, ofreciéndole volver atrás o aplicar los cambios permanentemente para visualizar los mismos en la traza detallada

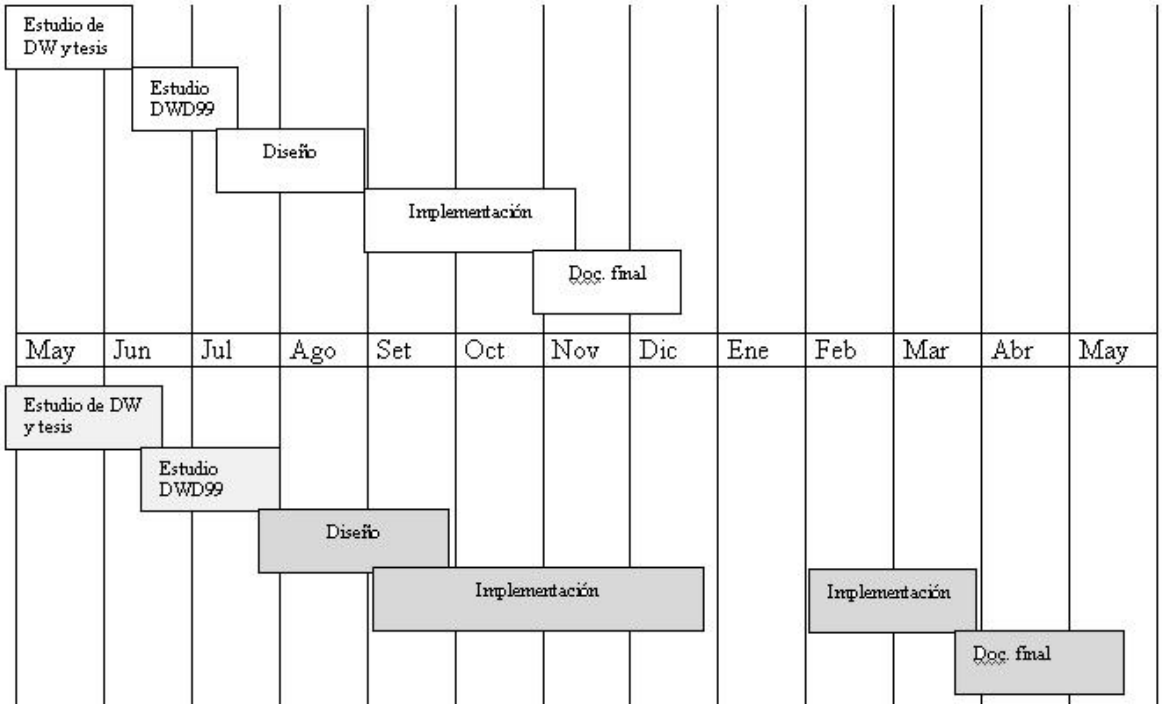
También se agregó la funcionalidad de salvar y cargar un diseño, lo que es sumamente importante para el usuario.

Finalmente se cumplió el objetivo principal de tener una herramienta que es una extensión del DWD99 (está integrada en un mismo producto) para la evolución de un DW de acuerdo a las especificaciones en [1]. Esta herramienta permite a un diseñador de DW, aplicar cambios sobre el esquema fuente y propagarlos al DW sobre un diseño en forma gráfica. Luego puede seguir diseñando, aplicar más cambios o salvar el diseño que obtuvo.

Desarrollo del proyecto

En cuanto a los tiempos que se manejaron, los mismos se desarrollaron según lo planeado. Excepto el de implementación que se extendió más de lo planificado, como se muestra en los siguientes diagramas :

Tiempo estimado



Tiempo Real

Finalmente se documentó el proyecto y se evaluaron los resultados obtenidos. El tiempo de la documentación fue de aproximadamente 2 meses (ya que algo se fue documentando en las distintas etapas). En la documentación además de diseño y aspectos de la herramienta se incluye los siguientes anexos

1. “Manual de Usuario”
2. “Diagramas de Clase”
3. “Caso de estudio”.

Sobre cosas que se pudieron mejorar en el desarrollo de esta herramienta , pensamos que la principal es el tiempo que le dedicamos al estudio del DWD 99 y al diseño , con el tiempo dedicado no alcanzamos a comprender en detalle el diseño de DWD 99 , cosa que comprendimos bien a la hora de implementar y nos encontramos con algunas dificultades , que nos hicieron que fuera más lenta la implementación. Incluso en algunos puntos , fue necesario modificar el diseño original que nos demoraron las etapas de implementación de acuerdo a los tiempos estimados de antemano.

Trabajos futuros

- Queda por implementar el 20% de las primitivas restantes (expresarlas en términos de operaciones básicas)
- Carga de los datos en el DW diseñado a partir del esquema fuente.

- Implementar una herramienta para el manejo de versiones , si se le quiere dar un enfoque de versiones para la evolución . Esta herramienta debería proveer funciones de conversión de la versión vieja a la actual y viceversa.

- Un caso de estudio real.

7. Bibliografía

[1] Marotta, A.: "Data Warehouse Design and Maintenance through Schema Transformations". Master Thesis. InCo - Pedeciba, Universidad de la República, Uruguay, 2000.

[2] Garbusi, P. Piedrabuena, F. Vázquez, G.: "Diseño e Implementación de una Herramienta de ayuda en el Diseño de un Data Warehouse Relacional". Undergraduate project. InCo, Universidad de la República, Uruguay, 2000.

[3] R. Kimball. "The Datawarehouse Toolkit." J. Wiley & Sons, Inc. 1996

[4] C. Larman "Applying UML and Patterns" Prentice Hall 1998

[5] Panos Vassilidis , Mokrane Bouzeghoub , Cristoph Quix "Towards Quality – Oriented Data Warehouse Usage and Evolutions" paper 199

[6] Cristoph Quix "Repository Support for Data Warehouse Evolution" paper 1999

[7] Fabrizio Ferrandina , Seven – Eric Lauttemann "An Integrated Approach to schema evolution for Object Datawarehouse" paper 1996