

InCo – Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

***Diseño e implementación de una
herramienta para la evolución de un
Datawarehouse Relacional.***

Informe final
TALLER V

Integrantes:

Andrés Alcarraz
Martín Ayala
Pablo Gatto

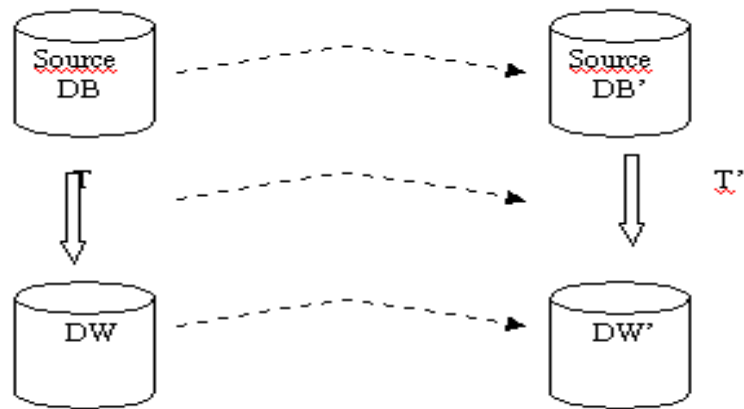
1. Introducción

En la tesis presentada en 2000 [\[referencia al taller viejo\]](#) se hace referencia a la importancia de los sistemas decisionales. 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 presentadas en [\[referencia a las primitivas de ellos\]](#). 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 (DWDesigner) , provee los mecanismos (primitivas) para el diseño de un DataWarehouse 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 que sea necesario aplicar cambios en el esquema fuente , por lo que estos cambios tendrán su impacto en el DW.

Aquí nos concentraremos en resolver los impactos que tienen los cambios en el esquema fuente , sobre el DW.



En la figura tenemos un diseño de un DW, a partir de una base de datos fuente (Source DB). Como se explico en el taller5 de 1999 se llega al DW luego de haber aplicado las primitivas para el diseño. La secuencia de estas primitivas, forman una traza T (forma en que fueron aplicadas las primitivas); teniendo la traza T y la base de datos fuentes, 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 tienen un impacto sobre el DW, este impacto genera un nuevo DW' y una nueva traza T'. Como ejemplo, se puede pensar que se remueve un atributo A de una relación R en el esquema fuente (de SourceDB). Este atributo participa en la obtención de un atributo A' de R' en DW, al eliminar el atributo A tendrá un impacto en el DW. El impacto de borrar este atributo nos da como resultado un cambio en el esquema del DW (el nuevo es DW') y un cambio en la traza T (la nueva T')

Como explicaremos en las siguientes secciones, en este proyecto nos preocuparemos en el tema de evolución de un DW debido a un cambio en el esquema fuente (de la base fuente integrada) de acuerdo a [1].

1.2. Definición del problema

La herramienta provista en [\[referencia al taller viejo\]](#) provee herramientas para el diseño de un DataWarehouse 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 que sea necesario aplicar cambios en el esquema fuente, por lo que estos cambios tendrán su impacto en el DW. Los cambios que pudieran ocurrir en la base fuente con el correr del tiempo los llamamos evolución del esquema fuente. Por ejemplo puede ser que un atributo en una relación de la base fuente se desee eliminar, por lo que sin duda la eliminación de ese repercutirá en el DW. Los posibles cambios se discutirán en la sección [\[sección de cambios en la taxonomía\]](#).

Cada vez que hay un cambio en el esquema fuente estos implican una modificación en el esquema del DW, estos cambios deben realizarse en base a las reglas de propagación explicadas en la tesis [\[referencia a la tesis\]](#), estos cambios afectan a las relaciones del DW, las relaciones intermedias para obtenerlas, parámetros de primitivas y las primitivas mismas (Transformation Trace [\[referencia a la transform. trace\]](#)).

El objetivo de este proyecto es ampliar la herramienta de diseño para un DW, para permitir que una vez obtenido el diseño, se pueda evolucionar el esquema fuente. Al evolucionar, los cambios en el diseño se deben propagar de acuerdo a las reglas de propagación del mismo, notificando como dependen de ese cambio los atributos y relaciones del DW (también el impacto que tendrán esos cambios antes de ser aplicados).

1.3. Objetivos

Como se mencionó el objetivo del proyecto es implementar una herramienta para manejar los cambios en el diseño del DW, según la evolución del esquema fuente. La herramienta además de aplicar los cambios en el diseño del DW según las reglas de [\[ref. reglas de cambios\]](#), deberá notificar al usuario como afectará el diseño ese cambio que se quiere realizar, y como dependen los atributos y relaciones del DW de estos. Esta herramienta además debe ser una extensión de [\[taller viejo\]](#), o sea una continuación de la misma.

Para lograr estos objetivos se debieron cubrir los siguientes puntos :

- Implementar las operaciones básicas según la especificación en [\[ref. OpBas\]](#)
- Expresar las primitivas [\[ref. Primitivas\]](#) en términos de operaciones básicas
- Implementar los posibles cambios según las reglas [\[ref. reglas de cambios\]](#) para que sean aplicados en el diseño.
- Proveer una interfase gráfica que permita aplicar cambios en el esquema fuente, despliegue las dependencias de los atributos y relaciones del esquema del DW. Así como permitir al diseñador en todo momento

- visualizar el orden en que fueron aplicadas las operaciones básicas para obtener a partir de una relación del esquema fuente , las relaciones 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 operaciones básicas [\[ref. OpBas\]](#) o cambios posibles en la base fuente [\[ref. a los SSchange\]](#) .

2. Conceptos y trabajos relacionado

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

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 : adaptacional y de versiones. En el enfoque adaptacional 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 adaptacional 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 cuando son usados. Ambas estrategias buscan llegar al mismo final , llegar a un estado consistente para el nuevo esquema. 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).

Al utilizar el enfoque de versiones , sólo la última versión puede ser modificada. Este mecanismo nos permite tener distintos estados en del 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 supervisión 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 un DW

Data warehouses 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 data warehouse los asista para tomar decisiones ,por ejemplo ,de negocios. La naturaleza de estos requerimientos está continuamente cambiando y son altamente subjetivos. Además en los requerimientos no sólo se exige que una rápida respuestas a las consultas sobre estos , 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. Una estrategia común es construir Data warehouse a partir de data marts locales (por ejemplo uno por departamento de la empresa) . El conocimiento adquirido en esta fase , es usado para la construcción (en paralelo) de un diseño global del DW. Los data marts son más fáciles de implementar que un diseño global de un data warehouse y después es necesario invertir un tiempo menor en análisis para diseñarlo.

En un data warehouse pueden ocurrir cambios o se requiere que ocurran en distintas situaciones (cambios en los requerimientos) .En cualquier caso el data warehouse debe ser adaptado a los cambios que ocurrieron (en la fuente), ejemplo cambios en el esquema (lógicos o conceptuales) , cambios en la propiedades físicas de los datos (en la ubicación de los datos) o cambios en los tiempos para la extracción de los datos de la fuente (performance) .

El problema de la evolución en un data warehouse 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 [tesis de Adriana]) en base a posibles cambios sobre el esquema fuente. En este tipo de perspectiva es necesario versionar las vistas del DW ya que hay aplicaciones que podrían no adaptarse a la nueva. La segunda es el problema de mantener la ¿¿ extent ?? de las vistas de un DW, es el caso de

mantener un cubo multi-dimensional en el caso de que se actualice una dimensión de esta.

Cómo adaptar el data warehouse una vez que se han producido cambios en el esquema fuente para una serie de cambios posibles, es el objetivo de este proyecto. La actualización del DW según el cambio se basa en las especificaciones de [\[tesis de Adriana\]](#) .

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

El proyecto completo se basa en la tesis escrita por Adriana Marotta: "A Transformations Based Approach for Designing the Datawarehouse" Reporte técnico, Facultad de Ingeniería, InCo [1]. Dicha tesis se concentra en proponer soluciones para diseño y evolución de un DW.

El proyecto es la continuación de otro [2] el cual se encarga de implementar la solución propuesta en el capítulo tres de [1] que se enfoca en el diseño de un DW a través de la aplicación de primitivas de transformación de esquemas. En dicho capítulo se hace un análisis de lo que debe hacer cada primitiva, también se describen estrategias de diseño y reglas de consistencia.

En el capítulo cuatro se propone una solución para la evolución del DW. Un DW puede evolucionar como consecuencia de: o bien un cambio en el esquema fuente o bien un cambio en los requerimientos del DW. Ambos casos deben tratarse separadamente ya que ellos involucran diferentes taxonomías de cambios y diferentes procesos para impactar el esquema del 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 una 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 elemento del esquema fuente. Esta traza debe proveer la información sobre la secuencia de primitivas que fueron aplicadas al elemento.

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' para abreviar). 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 un atributo B en el DW es requerido para el cálculo de A. 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.

Este proyecto se continúa para implementar 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

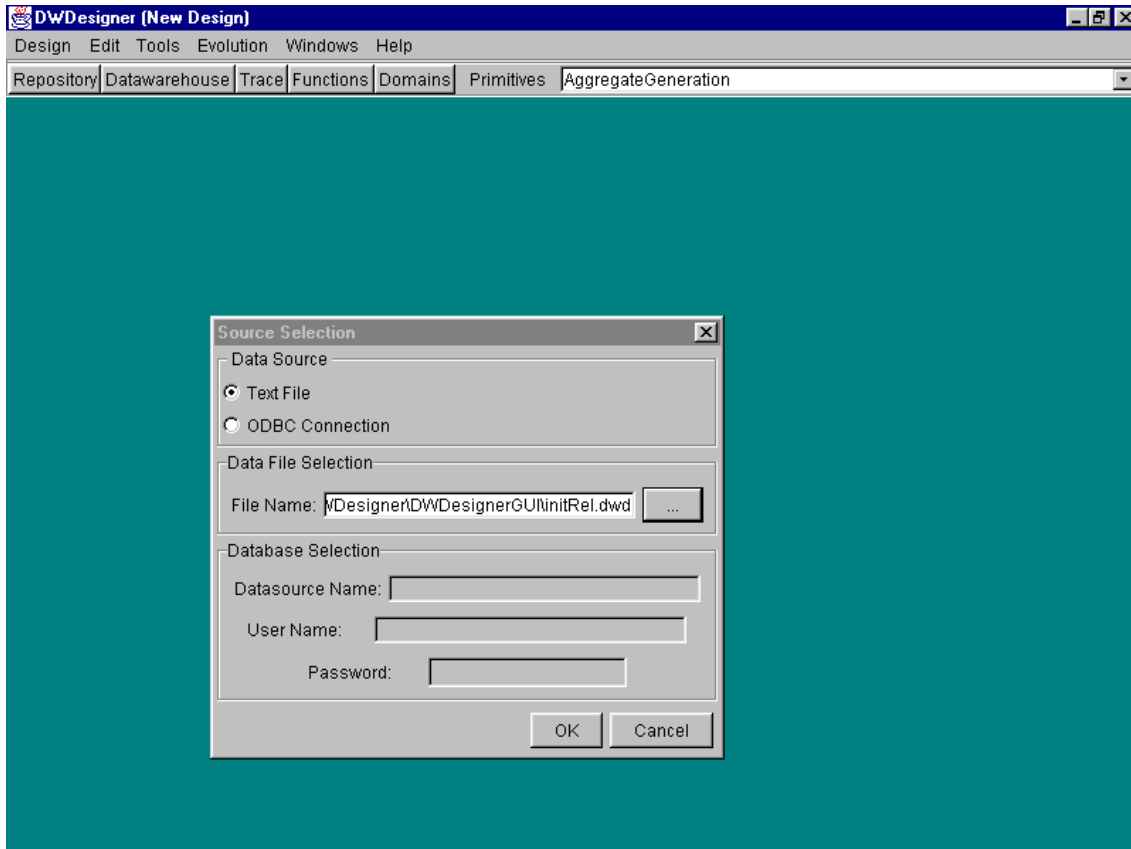
El objetivo de esta sección es explicar que hace el proyecto DWD 99 y en que situación se encuentra el mismo para ser continuado.

2.4.1 Características y funcionalidades.

Como dijimos en la sección anterior, el proyecto llamado DWD diseña e implementa lo que se especifica en el capítulo tres de [1]. Se construye un prototipo de ayuda en el diseño de un DW relacional. La herramienta permite diseñar un DW a partir de un esquema fuente justamente aplicando las primitivas especificadas. Se implementaron las 14 primitivas. 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 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:



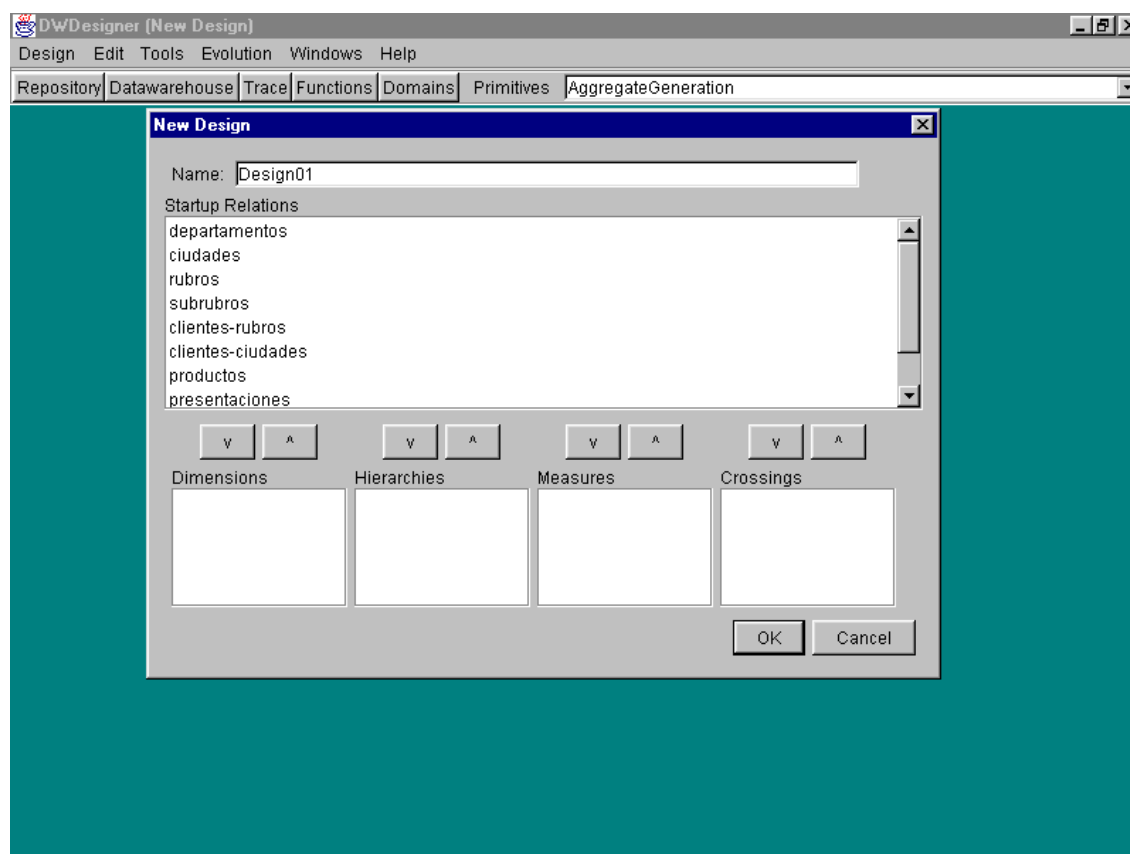
-Se ingresa un nombre para el diseño y se clasifican las relaciones de entrada en uno de los cuatro tipos siguientes: dimensión, cruzamiento, medida y jerarquía. Las relaciones de dimensión son las que representan información descriptiva del mundo real. Las relaciones de cruzamiento son relaciones que 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 a continuación:



-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 a relaciones tanto del DW, como del esquema fuente como 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.

Supongamos que en un principio tenemos una cierta relación R en el esquema fuente y queremos aplicar primitivas a esta relación para formar 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:

$\text{source} = \{ R \}, \text{repository} = \{ R \}, \text{DW} = \{ \}$

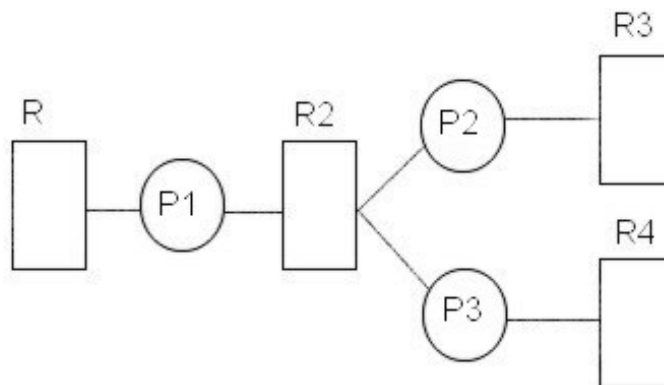
Supongamos que aplicamos la primitiva p1 a R y generamos como relación de salida R2, esta se introduce en el DW y en el repositorio:

$\text{source} = \{ R \}, \text{repository} = \{ R, R2 \}, \text{DW} = \{ R2 \}$

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

$\text{source} = \{ R \}, \text{repository} = \{ R, R2, R3, R4 \}, \text{DW} = \{ R3, R4 \}$

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. Notar que el esquema fuente siempre permanece incambiado. En la figura se muestra como sería el proceso de diseño explicado anteriormente:



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 diferenciados. Por un lado la primera parte (DWD) implementa una herramienta para el diseño de un DW y la segunda parte 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 ejemplo 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. Diseño

Presentaremos la parte de diseño del proyecto. Se presentan las partes más relevantes de la misma. De acuerdo a como encaramos las solución.

3.1. Introducción

Nuestro producto debía ser una extensión a un sistema desarrollado anteriormente al cual en el resto del capítulo llamaremos “Proyecto Anterior.” La etapa de diseño se llevó a cabo al comienzo del proyecto, luego de haber estudiado los temas necesarios para entender los conceptos que se manejaban en los requerimientos. Lo primero que tuvimos que hacer fue estudiar el diseño del Proyecto Anterior, para que nuestro diseño sea compatible. La política que tomamos fue de modificar lo menos posible el diseño anterior, y que nuestro aporte fuera agregando componentes y no modificando los existentes. Aún así hubo que modificar algunas de las clases anteriores para que “avisaran” a las nuestras cuando se modifica el diseño del Datawarehouse. Otro tipo de modificaciones que se hicieron al diseño anterior fue para agregar alguna funcionalidad que nos fuera de utilidad a las clases anteriores.

3.2. Funcionalidades

En esta sección mostramos los requerimientos fundamentales de nuestra aplicación, y los describimos de forma de obtener los conceptos que luego van a estar asociados a las principales clases de nuestro diseño.

Las funcionalidades que nuestro producto debe ofrecer son:

- 1- **Traza Detallada:** Descomponer el diseño de la Datawarehouse en un conjunto de operaciones básicas, que permitan deducir las dependencias a nivel de atributo entre elementos de la base fuente y los elementos de la Datawarehouse. A este conjunto de operaciones básicas y su interrelaciones le llamamos “Traza detallada”.
- 2- **Dependencias:** Dado un diseño de una Datawarehouse, calcular las dependencias entre los elementos de la base de datos fuente y los elementos de la Datawarehouse.
- 3- **Propagación de los cambios:** Al producirse un cambio en el esquema fuente, propagar el cambio determinando que elementos de la Datawarehouse siguen teniendo sentido luego del cambio, cuales no y cuales dejan de tener sentido pero podría cambiarse la forma de obtenerse para que lo tengan.
- 4- **Visualización de la traza detallada:** El producto también incluye una forma de ver gráficamente las dependencias entre tablas (Relaciones) del Datawarehouse y las relaciones de la base de datos fuente.

3.2.1. Traza Detallada.

En el contexto del **Proyecto Anterior** el diseño de una Datawarehouse, estaba dado por una serie de primitivas que se aplicaban a las relaciones del esquema fuente o a salidas de otras primitivas. Según está descrito en la **Tesis**, cada primitiva se puede descomponer en función de operaciones básicas. Nuestra tarea era generar un grafo que se dedujera de la Traza del **Proyecto Anterior** sustituyendo las primitivas por su descomposición en operaciones básicas. Para ello se modificó uno de los métodos de las primitivas para que al aplicarse generaran su descomposición en forma de grafo. La traza detallada completa no se genera en ningún momento, para recorrerla lo que se hace es recorrer la traza **Común**, y para cada nodo, recorrer su “sub Traza Detallada”. Esto se hizo así para poder intercambiar fácilmente entre el modo de diseño de la Datawarehouse y el modo de evolución.

3.2.2. Dependencias

Cuando se aplica una primitiva a una relación o mas relaciones, los atributos de la relación de salida pueden depender de tres formas respecto a los atributos de las relaciones de entrada o no depender.

- 1- No Dependency: No hay dependencia entre el atributo de entrada y el de salida.
- 2- Copied: El atributo de la relación de salida es una copia del atributo de la relación de entrada.
- 3- Calculated: El atributo de la relación de entrada es usado en la función que calcula el atributo de salida.
- 4- Required: El atributo de entrada es requerido para poder evaluar el atributo de salida. Esto pasa cuando el atributo de entrada es usado

como atributo de Join entre tablas y la función de cálculo está definida sobre ese join.

En realidad las dependencias que interesan son entre atributos del esquema fuente y atributos de la Datawarehouse, pero estas se deducen obteniendo todos los caminos que van desde un atributo dado del esquema fuente hasta algún atributo de la Datawarehouse, cada uno de estos caminos desembocará en algún atributo de la Datawarehouse, para cada camino la dependencia entre el origen y el destino del camino será la mas fuerte de todas las dependencias encontradas en el camino, la dependencia mas fuerte es Required, seguida por Calculated, luego por Copied y NoDependency.

3.2.3. Propagación de los cambios

Cada cambio en la Base fuente generará cambios o no en la Datawarehouse dependiendo de si el cambio es importante y de cómo los elementos de la Datawarehouse dependen de los elementos cambiados. Cuando un cambio se produce en un elemento del esquema fuente, el cambio se notifica a la primer operación básica que depende de este elemento, y a su vez cada operación básica notifica a las operaciones básicas que toman a sus salidas como entradas de los cambios que ella debe producir en sus salidas, así los cambios se van sucediendo hasta que llegamos a los elementos de la Datawarehouse.

3.2.4. Visualización de la traza detallada

La traza detallada completa es un grafo que se deduce de sustituir en la traza común, los nodos que representaban aplicaciones de las primitivas, por su descomposición en operaciones básicas, así tenemos un grafo que puede llegar a ser fácilmente mucho mas grande que el de la traza común. En este punto había dos opciones, una era generar una única traza detallada a partir de un diseño en función de operaciones básicas, y luego trabajar en paralelo con las dos trazas, la otra opción, que fue la que tomamos, era que cada nodo de la traza detallada mantuviera su subgrafo, y que cuando se precisara recorrer la traza detallada completa, ya sea para deducir dependencias o para dibujarla, cada nodo era responsable de asociar cada una de sus entradas con algún nodo de entrada de su sub traza detallada, y luego la recorriera hasta salir de ella para entrar a otra primitiva o llegar a una relación del Datawarehouse.

3.3. Arquitectura

3.3.1. Mecánica de Funcionamiento

Al ser nuestro proyecto una continuación de un **proyecto anterior**, nuestra arquitectura tenía que adaptarse a la anterior también. Cómo se desprende de la sección anterior tenemos básicamente 4 subpartes del proyecto que analizar.

3.3.1.1. Traza Detallada.

De la sección Traza Detallada.3.2.1 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 aplicación de una primitiva.

En el diseño del **Proyecto Anterior** 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 obviamente, 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 que describe la acción realizada por ella, así es la primitiva también quien genera la traza detallada, porque ella y solo ella sabe como describirse en función de operaciones básicas.

Para recorrer la traza detallada completa hay que recorrer la traza general pero para cada nodo de esta recorrer la traza detallada que contiene su **TraceEntry**. El grafo que representa esta traza consta de objetos de las siguientes clases DetailedEdge (Aristas detallada) y DetailedNode (Nodo detallado), donde los segundos son los nodos de la traza y los primeros son las aristas que unen los nodos. Los nodos se dividen en tres subclases de DetailedNode, InputNode, OutputNode e InternalNode, InputNode y OutputNode, vinculan la traza detallada con la traza general. 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.

3.3.1.2. Dependencias

Como se explicó en la sección 3.2.2 los atributos de las relaciones del Datawarehouse 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 Datawarehouse, cómo esta información luego va a ser utilizada para preguntarle al usuario que desea hacer en función de las dependencias y también para propagar los cambios se ideó una estructura que permitiera almacenar el camino que llevó a cada dependencia y la operación básica que originó 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 Calculated y se quiere cambiar la función de cálculo, esto se puede sin tener que volver a recorrer la lista. Para esto creamos la clase **DependenciesList**, que contiene la relación y el atributo de entrada, y para cada atributo de la Datawarehouse que depende del atributo solicitado, el camino que lleva de uno al otro en forma de una lista ordenada de dependencias. El primer elemento de esta lista contiene la dependencia más fuerte, y el resto conforman el camino. Los elementos de esta lista son de la clase **Dependency**.

La clase **Dependency** contiene la información de dependencia 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. 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()`.

3.3.1.3. Propagación de los cambios

Los cam

3.3.1.4. Visualización de la traza detallada

3.4. Diseño Conceptual

3.5. Diseño de clases

4. Implementación

Veremos las decisiones tomadas en la implementación , así como las razones que nos llevaron a tomarlas. Sobre el lenguaje de programación escogido , como logramos la extensibilidad , la proyección en el futuro de la herramienta.

4.1 Introducción.

Éste 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.

La tarea de implementación requirió un gran porcentaje de tiempo de dedicación al proyecto. Además de eso, al ser este una continuación, necesitamos tomar decisiones importantes a la hora de implementar. Decisiones tomadas en el proyecto anterior estaban realizadas con el fin de optimizar el diseño de un DW (el objetivo del proyecto anterior). Sin embargo, esas decisiones no nos resultaban 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. El capítulo citado también 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.

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.

4.2 Lenguaje de programación y ambiente de desarrollo.

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

La herramienta de diseño usada también fue la misma: Rational Rose®. Ésta herramienta permite realizar diseños en UML de manera sencilla y además generar parte del código (cabecales 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 poseía 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.

4.3 Operaciones básicas

El primer paso en la implementación de nuestro proyecto fue implementar las operaciones básicas según las especificaciones de la tesis [[ref tesis Adriana](#)], ya que como se especifica en la tesis , 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 la tesis de evolución de esquemas se especifica la equivalencia 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 data warehouse se realiza modificando los parámetros de las operaciones básicas.

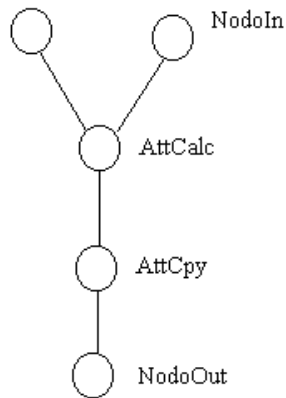
Los objetivos que teníamos al implementarlas es el de que cumplieran las especificaciones según la tesis y que fuera extensible . Es decir , al igual que las primitivas son extensibles , lograr lo mismo con las operaciones básicas , que en el futuro si se desea agregar alguna sea fácil de incorporarlas.

Para lograr estos objetivos se decidió implementar una clase abstracta llamada BasicOperations , la cual tiene los métodos comunes a todas (uno de los más relevantes es el de getDependency , que dado un atributo y una relación devuelve la dependencia de este a nivel de operación básica (es decir si la operación lo usa como copiado , calculado , requerido o no tiene dependencia) , más adelante se explica como funcionan las dependencias con más detalle. Luego se implementa cada operación básica como una clase que hereda de BasicOperations , estas clases tiene atributos propios (además de los comunes a todas) que son los parámetros. El método más relevante de cada una de ellas es el apply. Este método genera a partir de los parámetros de entrada la relación/ones de salida correspondiente. El esquema de cómo quedarían las operaciones básicas y como heredan estas se describe en la parte de diseño.

4.4 Traza detallada

Como se mencionó en la sección anterior , cada primitiva es equivalente a una secuencia de operaciones básicas (se puede expresar una primitiva en término de unidades más chicas llamadas operaciones básicas) . Para lograr los objetivos en la aplicación de los cambios del DW , es necesario tener la primitiva expresada en términos 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 de los mismos respecto al DW , que sólo se puede lograr a nivel de operaciones básicas. El segundo motivo para expresar las primitivas en término de operaciones básicas es que para propagar los cambios una vez que se producen en el esquema fuente , las modificaciones se propagan a nivel de operaciones básicas (modificación de parámetros según las reglas de propagación provistas en la tesis).

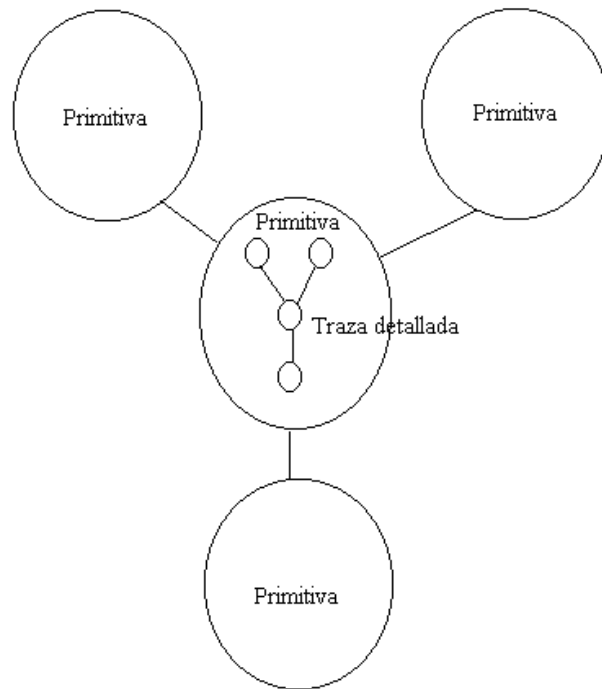
Para expresar las primitivas en términos de operaciones básicas fue necesario hacer que cada primitiva contuviera la expresión de ella en op. Básicas. La estructura usada para representar la secuencia de aplicación de cada primitiva en términos de operaciones básicas fue la de un grafo. Cada grafo de 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 siguiente figura representa un esquema de una traza detallada en una primitiva.



La traza detallada que muestra la figura es la que se incluye dentro de cada primitiva. La idea de la traza es tener nodos (hay 3 tipos de nodos). Los nodosIn , tienen como entrada las relaciones que son de entrada de la primitiva , los NodosInternos , estos contienen operaciones básicas con los parámetros correspondientes y los NodosOut , estos tienen como salida las relaciones que son de salida de las primitivas. La 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 correspondiente al apply de la operación básica del nodo de entrada.

Como se verá más adelante , incluimos la traza detallada dentro de cada primitiva en el momento que esta 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 disponen de todos los parámetros para generarla.

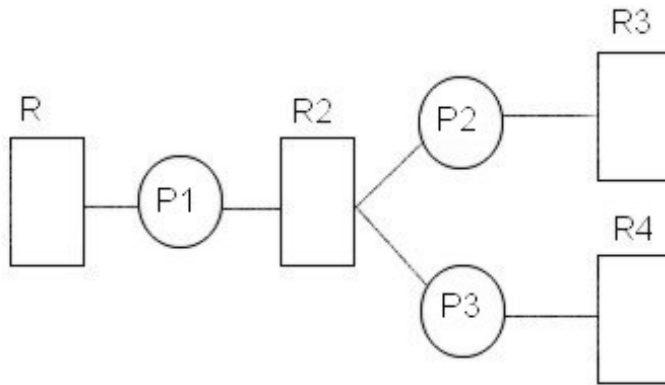
Hasta el momento se cuenta con 2 niveles , uno es el nivel de las primitivas utilizado para el diseño del DW (según la especificación de la tesis para el diseño) , el otro nivel , es un nivel más fino (el detallado) es a nivel de operaciones básicas, es el que se encuentra dentro de cada primitiva (traza 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 se muestran los 2 niveles mencionados.



4.5 Deducción de dependencias

4.5.1 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. Por ejemplo, supongamos que tenemos el siguiente diseño:



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.

4.5.2 Solución

En la figura anterior 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 solo 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.

4.5.3 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` el cual 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)
- Relación de salida
- Atributo de salida
- Un puntero a la operación básica misma

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 }`

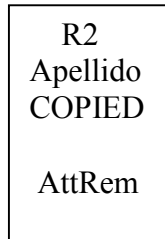
y que a esta relación le aplicamos la operación básica `AttRem` con el siguiente conjunto de atributos a borrar: `X = { dirección, teléfono }`

La relación de salida luego de aplicada la primitiva 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 caso 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. El 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 un ejemplo: Supongamos que tenemos la relación 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 hacer 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

2) Ya hablamos de cómo se deducen las dependencias al aplicar una operación básica, ahora nos podemos preguntar cómo 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.

```
relation relIn;  
Attribute attIn;  
ArrayList[] Dependency;
```

Diagram illustrating the notation for dependencies:

- R1** Relación
- A1** Atributo

Dependencias

The diagram shows a vertical rectangle representing a relation $R1$. It has four rows. The first row is connected to two circles, which are then connected to a third circle. The second row is connected to a single circle. The third row contains vertical dots. The fourth row is connected to three circles in a horizontal chain.

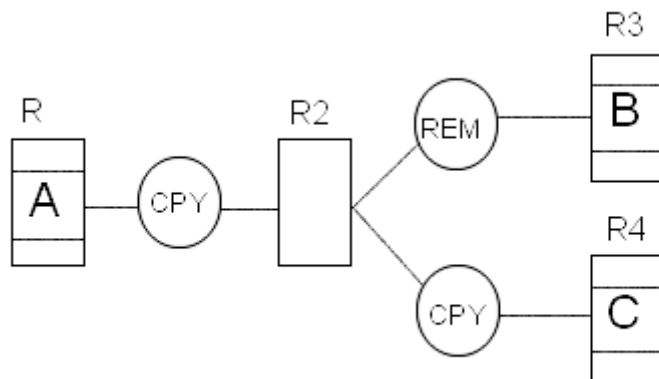
28

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) 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 funcionamiento del método `getDependencies()` se describe con un ejemplo a continuación:



Supongamos que tenemos el diseño que se muestra en la figura anterior (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). Luego 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:

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

2) 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 hacia el DW y para cada primitiva que encuentra en el camino llama al método `getDependencies()`, 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.

3) 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 completo 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.

De ésta manera, al hacer tres métodos diferentes para hallar las dependencias reducimos la cantidad de código, facilitamos el mantenimiento y la

comprensión del mismo y aprovechamos al máximo las características de un lenguaje orientado a objetos haciendo que cada método haga lo que le corresponda.

4.6 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 hay que 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.

Remove 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 éste 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.

Por 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

fuentes 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 posibilidad 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 eliminando como parámetro. Si se elige borrar el atributo el procedimiento es el mismo que en los casos anteriores.

Renombrar atributo.

Este cambio tiene una particularidad la cual diferencia su implementación de los demás cambios. 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 cambiara 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 (en la primer operación básica ni bien comienza la traza detallada) 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 definan la clave primaria y las claves foráneas 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:

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

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

$R2 = \{ \text{Nombre, Apellido, CI, Dirección, Teléfono, Mes} \}$

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

$R3 = \{ \text{CI, Dirección, Teléfono, Mes} \}$

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.

4.7 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 y cada vez que se generaba una primitiva teníamos niveles nuevos en la misma. 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 las 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 permite a partir de una relación saber cuales son las primitivas que la toman como entrada, esta estructura resulta bastante copleja 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 ésta 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.

4.8 Implementación oculta

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 oculto. 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 traza detallada en cada primitiva. Cada primitiva está compuesta de operaciones básicas que forman la traza detallada. Por eso decimos que cada primitiva contiene una 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 oculto. 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 la de diseño de DW y ésta estaba pensada 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 etapa 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.

4.9 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, la herramienta para diseño DWD 99 estaba diseñada de tal forma que agregar nuevas primitivas fuese fácil. La orientación a objetos de Java permitió que esto se implementara en forma sencilla. Nosotros 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.

4.10 Trabajo futuro.

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.

4.10.1 Primitivas

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 75% 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
P12.1 Denormalizad Hierarchy Generation
P14 New Dimension Crossing

Las primitivas que no fueron implementadas son las siguientes:

P10 Data Array Creation
P11.2 Horizontal Partition
P12.2 Totally Normalized Hierarchy Generation (SnowFlake)
P12.3 Free Decomposition
P13 Minidimension Break Off

4.10.2 Sustitución de esquemas

Parte de la especificación es que al agregar una relación o un atributo en el esquema fuente se pueda volver a la etapa de diseño, continuar con el mismo y tener la posibilidad de sustituir un nuevo esquema generado por uno viejo que haya quedado obsoleto. Parte de este requerimiento quedó sin implementar.

La herramienta permite agregar una relación o atributo y luego continuar con el diseño. Lo que no se permite es sustituir un esquema por otro existente.

5. Test

Pueba de lo implementado. En este capítulo comentaremos brevemente el método que utilizamos para testear los distintos módulos y la herramienta.

5.1 Introducción

Tanto el proyecto anterior como el nuestro son prototipos, eso implica entre otras cosas que no se desea una fase especial para testeo. Sin embargo el producto debe ser testado para asegurar la calidad del mismo y lo que se hizo fue hacer un testeo unitario de los módulos a medida que se iban implementando. Esto nos asegura a medida que vamos avanzando con la implementación a aumentar la confiabilidad de lo que ya hemos programado. En la implementación de nuestro proyecto hay varios niveles, por un lado están las operaciones básicas que fué lo primero que implementamos, luego la traza detallada que estaba contenida en cada primitiva la cual contiene varias operaciones básicas. Lo último es implementar los cambios en la base fuente y luego probarlos con un diseño completo que incluyera varias aplicaciones de primitivas y por lo tanto la construcción de varias trazas detalladas y la unión de las mismas. Una vez mas recalcamos los diferentes niveles en el proyecto y el orden de implementación del mismo fue el descrito anteriormente. Por haber implementado de esa manera resulta interesante ir testeando lo que ya fue implementado porque luego sería integrado con las próximas clases; por ejemplo luego de haber implementado las operaciones básicas seguía la traza detallada y lo que hicimos fue implementarla mientras hacíamos un testeo de las operaciones básicas que luego serían usadas en la traza detallada.

5.2 Metodología

Al ser el proyecto un prototipo no queremos una fase tan estricta de testeo. Por tal motivo no profundizamos en implementarla como una fase formal. No desarrollamos un plan de testeo ni casos de test previos. Simplemente a medida que fuimos terminando la implementación de las clases las íbamos testeando. Hicimos un testeo unitario para cada clase implementada utilizando módulos drivers que usaran la clase a testear, no se implementaron stubs. Utilizamos un testeo de caja negra pero también realizamos inspecciones de código. Se hicieron revisiones por pares, una vez que un implementador terminaba la programación de un módulo, él mismo lo probaba y luego lo hacían los otros integrantes del grupo. No hubo roles específicos, cada uno de los integrantes se dedicó tanto a implementar como a testear. No se generaron reportes de errores, simplemente al encontrar un error se informaba al programador o si se estaba haciendo un testeo unitario la misma persona se encargaba de corregir el error.

5.3 Ejecución del testeo

A continuación se describen los pasos que llevó el testeo del producto desde su comienzo con las operaciones básicas hasta el final cuando integramos ambas herramientas. El orden en que se describen los casos es el mismo en el que fueron ejecutados.

5.3.1 Operaciones básicas

Como dijimos anteriormente las primeras clases implementadas fueron las operaciones básicas. Las mismas toman como entrada una o más relaciones y devuelven sólo una como salida. Para probar el correcto funcionamiento de las mismas construimos las relaciones de entrada necesarias para cada operación básica, además de eso nos construimos los otros parámetros de entrada para la aplicación de la misma (por ejemplo en el caso de AttRem un parámetro es el conjunto X de atributos a borrar). Con eso ya estamos en condiciones de generar la relación de salida y verificar que la misma sea correcta, otro método que consideramos de principal importancia y quisimos probar desde un principio fue el método que nos devuelve las dependencias de una relación de entrada a una de salida. O sea, los métodos que mas estamos interesados en probar a este nivel son apply() (el cual genera la relación de salida) y getDependency() (el cual dado un atributo de una de las relaciones de entrada nos dice cual es la dependencia que tenía en la relación de salida).

Se hizo un módulo driver de testeo para cada operación básica, un ejemplo de lo anterior es el siguiente:

Consideremos la operación básica AttRem que dada una relación R1 y un conjunto de atributos X elimina el conjunto de atributos X de R1. R1 es la relación de entrada a la operación básica y como salida tenemos una relación R2 que es una copia de R1 con el conjunto de atributos X eliminados. De esta forma tenemos

como relación de entrada R1, como parámetro el conjunto de atributos a eliminar X y como relación de salida R2. Para testear esto se construyeron dos casos base:

1) Prueba del método apply()

Para probar éste método en la operación básica AttRem lo que hicimos fue crear la relación R1 y el conjunto X, luego llamamos a apply() y verificamos que la relación generada R2 sea correcta.

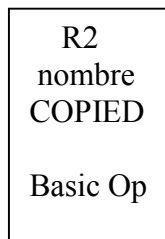
Ejemplo: Tomamos una relación R1 con los atributos: { nombre, apellido, dirección, teléfono, documento } y el conjunto X de atributos: { teléfono, documento }. Luego de llamar al método apply() la relación de salida debe ser idéntica a R1 pero con el conjunto X de atributos eliminados o sea R2 debe contener lo atributos { nombre, apellido, dirección }.

2) Prueba del método getdependency()

Luego de aplicada la operación básica nos interesa saber si las dependencias se generan en forma correcta. Para eso probamos llamar al método getDependency() y ver que devuelva la dependencia correcta. Recordemos que el método getdependency() devuelve objetos de la clase Dependency (en general uno solo) que contiene cuatro campos: relación, atributo, dependencia y operación básica.

Ejemplos: Llamamos a getDependency() con el atributo nombre de R1 como parámetro, el objeto devuelto es: { R2, nombre, COPIED, BasicOperation } siendo BasicOperation la operación básica desde la cual se llama a getDependency() y nos interesa para luego tener el camino desde el esquema fuente hasta el DW pasando por todas las operaciones básicas, por ahora no es relevante. Lo que nos dice la dependencia es que el atributo nombre de R1 está copiado en R2 como nombre.

El objeto Dependency se podría ver de la siguiente forma:



Llamamos a getDependency() con el atributo teléfono de R1 como parámetro, el objeto devuelto es: { R1, teléfono, NODEPENDENCY, BasicOperation }, esto significa que el atributo teléfono no tiene dependencias en la relación de salida. En ese caso la relación y el atributo permanecen intactos y estos se ignoran.

La única operación básica que tiene otro tipo de dependencia es AttCalc la cual además contiene los tipos de dependencias CALCULATED y REQUIRED. Estos se probaron únicamente cuando se probó la operación básica AttCalc.

5.3.2 Trazas detalladas

Al finalizar el testeo de las operaciones básicas debemos seguir con la traza detallada. Ésta está compuesta por operaciones básicas por lo que resultó productivo haber testado antes las mismas. Cada primitiva tiene una traza detallada en su interior por lo que hay 20 tipos de trazas detalladas (contando sub-primitivas) y debíamos testear que el grafo estuviera armado en forma correcta y que la traza detallada devolviera las dependencias en forma correcta. Para testear el correcto armado del grafo hicimos inspecciones de código recorriendo el mismo y comprobando la correctitud del mismo (en un principio no nos quedó otra opción que ésta para verificar la correctitud del grafo, luego que implementamos el método de dibujo del grafo nos fue más sencillo el testeo). Luego de eso hicimos un testeo similar al de las operaciones básicas para verificar que la o las relaciones de salida fueran correctas. El procedimiento fue el siguiente:

- 1) Creamos las relaciones de entrada a la traza detallada que coinciden con las relaciones de entrada a la primitiva que la contiene.
- 2) Llamamos al método apply() para cada operación básica que está incluida en la traza detallada (se debe llamar en orden a éste método ya que la relación de salida de una operación básica coincide con la de entrada a la siguiente).
- 3) Comprobamos que las relaciones de salida de la traza detallada se hayan generado en forma correcta. La relación de salida de la traza detallada será la relación de salida de la última operación básica (si hay dos relaciones de salida habrá dos operaciones básicas).

Ejemplo: Consideremos la operación básica Temporalization con el campo key=true. La misma toma una relación R1 como entrada y está compuesta de las siguientes operaciones básicas:

```
AttCpy( Att(R1), R1, R2 )  
AttAdd( {T}, R2 )  
KeyAdd( XT, R2 ) ( X es la clave de R1 )
```

Para testear que la traza detallada de esta primitiva funcionara en forma correcta primero preparamos el caso con los siguientes parámetros:

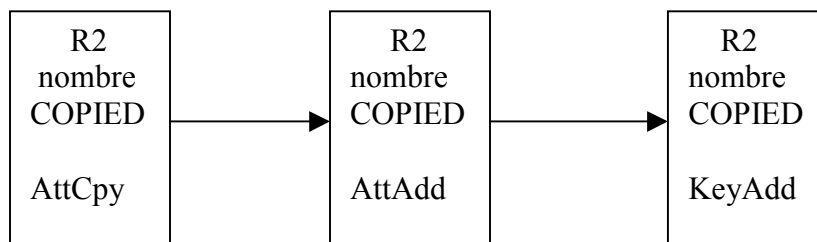
Tomamos una relación R1 con los atributos: { nombre, apellido, dirección, teléfono }
Llamamos tiempo al atributo T
La clave X de R1 está compuesta por los atributos: { nombre, apellido }

Luego llamamos al método `apply()` en `AttCpy`, `AttAdd` y `KeyAdd` respectivamente para que genere las relaciones de salida, la relación de salida de `AttCpy` se toma como entrada para `AttAdd`, la relación de salida de `AttAdd` se toma como entrada para `KeyAdd` y la salida de esta es la que se testea que sea correcta y debe coincidir con la relación de salida de la primitiva (si bien no son el mismo objeto deben tener los mismos atributos, la misma clave y las mismas claves foráneas).

Verificamos que `R2`, la relación de salida de `KeyAdd` contenga los siguientes atributos: { nombre, apellido, dirección, teléfono, tiempo } y la clave de `R2` sea { nombre, apellido, tiempo }.

Luego de saber que la relación generada como salida es correcta debemos asegurarnos que las dependencias se deducen en forma correcta en la traza detallada. Tenemos ya probadas las dependencias en las operaciones básicas. Ahora hay que probar el método `getDependencies()` que devuelve listas de objetos `dependency` concatenando las dependencias que devuelven las operaciones básicas. Para probar esto usamos la traza detallada generada anteriormente y las relaciones que se usaron para probar la misma.

Ejemplo: Llamamos al método `getDependencies()` con el atributo `nombre` de la relación `R1`. La lista de dependencias tendría la siguiente forma (solo hay una lista en este caso):



Esta lista indica que se pasó por tres operaciones básicas y se llamó a `getDependency()` en cada una de ellas. Cada rectángulo representa un objeto de la clase `Dependency` que el método `getDependencies()` se encargó de concatenar y devolver en una lista. Luego de esto en la última dependencia tenemos el atributo y la relación de salida que dependen del original, en este caso el atributo tiene el mismo nombre pero en otro objeto y pertenece a otra relación. La dependencia se deduce como la mas fuerte en todo el camino y en éste caso es `COPIED`. También observamos que en la lista de dependencia tenemos las tres operaciones básicas en el orden que fueron aplicadas. Si hay que actualizar algún camino por ejemplo borrando atributos ya sabemos cuales son las operaciones básicas que debemos recorrer porque las tenemos en la lista.

5.3.3 Diseño completo

Luego de haber testeado cada traza detallada nuestro próximo paso es construir un DW usando DWD y comenzar a probar nuestra herramienta haciendo cambios en el esquema fuente para luego deducir las dependencias y propagando los cambios. Para esto debemos tener integradas ambas herramientas lo cual consideramos costoso a esta altura, por un lado la integración en sí podía introducir errores que se sumaban a los que podía ya tener nuestra herramienta que no había sido testeada totalmente (habíamos llegado solo hasta testear las trazas detalladas). No considerábamos apropiado realizar una integración del tipo Big-Bang donde podían aparecer un montón de errores difíciles de localizar.

Por ese motivo decidimos postergar el testeo de las herramientas integradas y realizar un testeo sólo con nuestra herramienta. Aquí aparece una dificultad ¿cómo testear un diseño completo de un DW sin usar DWD si esta es justo la herramienta que se encarga del diseño? Para eso necesitamos simular el uso del DWD para la construcción de un DW y hacerla manualmente tratando que quede lo más similar posible que si la hubiéramos hecho con DWD. Por supuesto que esto tiene un costo mayor que integrar las herramientas y realizar las pruebas con las herramientas integradas pero consideramos que vale la pena el esfuerzo adicional a cambio de localizar mejor los errores que aparecen ya que pensamos que cuanto más tarde aparecen los errores más costoso resulta repararlos.

Para probar el diseño completo sin usar DWD construimos la traza en forma manual. Comenzamos construyendo las relaciones del esquema fuente. Luego simulamos la aplicación de una primitiva con la siguiente secuencia de pasos:

- 1) Construimos la traza detallada a mano. Para simplificar esta tarea usamos las trazas detalladas que ya habíamos construido al realizar otras pruebas.
- 2) Instanciamos un objeto primitiva (de la clase TraceEntry) y le incluimos la traza detallada construida anteriormente.
- 3) Generamos a mano la relación de salida de la primitiva. Esto lo hicimos asegurándonos que sea correcta, por ejemplo:
Supongamos que queremos aplicar la primitiva DataFilter a una relación R1 conteniendo los atributos nombre, dirección y teléfono queriendo filtrar el atributo teléfono. La relación de salida la construimos a mano pero sabemos que debe ser una cierta relación R2 con los atributos nombre y dirección. De ésta manera nos aseguramos que las relaciones de salida son correctas. Esta es una forma de simular el método apply() de las primitivas.

Este procedimiento lo realizamos hasta que consideramos completo nuestro diseño para realizar pruebas. Las relaciones de salida de las últimas primitivas que aplicamos van a formar nuestro DW. En el ejemplo anterior nuestro esquema fuente estaría compuesto por R1 y el DW por R2. Obviamente para realizar una prueba más completa no nos alcanza con aplicar una sola primitiva, debemos hacer un diseño más grande.

Luego de finalizado el "diseño manual" ya estamos en condiciones de realizar las pruebas. Para realizar las mismas efectuamos los siguientes pasos:

- 1) Hacemos un cambio en alguna relación de nuestro esquema fuente.
- 2) Llamamos al método `getAllDependencies()` que nos devuelve las dependencias desde el esquema fuente hasta el DW. Aquí controlamos que este método nos devuelva las listas de dependencias en forma correcta, este método no había sido testeado aún y es de fundamental importancia cuando operamos en el diseño completo ya que es el que se encarga de juntar los caminos que va devolviendo el método `getDependencies()` en cada primitiva.
- 3) Llamamos al método `propagate` para propagar los cambios y verificamos que los cambios se hayan reflejado en el DW en forma correcta.

Creemos que el hecho de haber usado este procedimiento para testear el diseño antes de integrar las herramientas fue muy productivo ya que se encontraron errores que si no se hubieran localizado a ésta altura hubieran sido mas complicados de encontrar y reparar. Por eso consideramos que fue productivo el trabajo adicional para probar el diseño antes de integrar ambas herramientas. Este es justamente el próximo paso y nos dirigimos a él con un mayor grado de confiabilidad en nuestra herramienta.

5.3.4 Integración con DWD

Luego de haber testeado paso por paso los distintos componentes de nuestra herramienta llegó el momento de hacer la integración de la misma con el DWD. La finalidad de esto es probar ambas herramientas como una sola y ver que todo funcione correctamente.

A esta altura ya estamos en condiciones de probar las herramientas como un producto final. La idea es diseñar un DW ahora sí utilizando DWD y luego comenzar a hacer cambios en la base fuente y verificar que estos se vean reflejados en forma correcta en el DW.

Este testeo se realiza en dos partes:

- 1) Primero que nada queremos asegurarnos que todas las primitivas funcionen correctamente al hacer la evolución. Debemos probar entonces por lo menos un caso para cada primitiva y ver que los cambios se propaguen en forma correcta.
- 2) Luego de probada cada primitiva por separado nos interesa probar un diseño que abarque la mayor cantidad posible de primitivas. En este caso podemos además probar los casos de borde que se pueden presentar.