Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

SISTEMA DE AYUDA EN EL APRENDIZAJE DEL PROCESAMIENTO DE CONSULTAS EN BASES DE DATOS DISTRIBUIDAS

Alumno: Jesús Ruiz Campillo

Tutor: Prof. D. CARLOS PORCEL GALLEGO
Dpto: Informática

Septiembre, 2017

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don CARLOS PORCEL GALLEGO, tutor del Proyecto Fin de Carrera titulado: SISTEMA DE AYUDA EN EL APRENDIZAJE DEL PROCESAMIENTO DE CONSULTAS EN BASES DE DATOS DISTRIBUIDAS, que presenta Jesús Ruiz Campillo, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2017

El alumno: Los tutores:

Agradecimientos

Me gustaría manifestar mi agradecimiento a todas aquellas personas que, en algún modo, han aportado su pequeño grano de arena para que haya podido llegar a la realización de este proyecto.

A mis padres y hermanos, por su apoyo prestado desde que inicié mi formación, sin ellos no hubiera sido posible.

A mi tutor del proyecto, Carlos Porcel Gallego por la formación recibida y el tiempo y esfuerzo dedicado durante la realización de este proyecto.

A la Escuela Politécnica Superior de Jaén, por haber tenido la suerte de formar parte de ella durante estos cuatro años.

Índice

CAPITULO 1: INTRODUCCIÓN

1.1 Introducción al proyecto

La creciente necesidad de almacenar datos de forma masiva hizo que surgiesen las bases de datos distribuidas, a día de hoy siguen evolucionando por distintos factores como son: la globalización que exite hoy en día entre empresas y estas estan a la vez mas descentralizadas, el aumento de la potencia de las computadoras y la comparticion de grandes cantidades de datos. Dicho esto, es necesario conocer estos tipos de sistemas y como funcionan internamente.

Como ya se explicará más adelante la diferencia entre una base de datos distribuida y una centralizada es donde estan almacenadas los datos y como estos están relacionados unos con otros. Para poder relacionar los datos es necesario usar catalogos, que son diccionarios donde se describen las caracteristicas de estos datos. Este paso se hace justo después de elegir como se van a fragmentar los datos, es decir como se van a distribuir fisicamente. La fragmentación que se haga es muy importante, es un factor clave en la eficiencia cuando se acceden a los datos.

Para poder recuperar los datos correctamente se accede primero al diccionario para consultar donde están alojados los datos, este paso depende de si los datos a recuperar tienen alguna condición, por ejemplo ciudad = ‘SF’. Si ningún dato de todos los fragmentos cumplen la condición ya no es neseario intentar acceder, con el elevado coste que esto supondría. En los capitulos posteriores se expondrá con detalle los pasos a seguir internamente para acceder a los datos.

1.2 Propósito general

El proposito de este proyecto no es otro que ayudar a los alumnos durante su aprendizaje mientras estan cursando la asignatura “Base de datos Distribuidas”.

El procesamiento de consultas distribuidas es algo esencial que el alumno debe de aprender muy bien, ya que describe como es el funcionamiento interno de una base de datos distribuidas y por ende expone el potencial de estos sistemas.

1.3 Objetivos

Tenemos tres objetivos principales que se dividen en una serie de subobjetivos que se muestran a continuación.

1. Transformación de una consulta global en una consulta fragmentada en una base de datos distribuida.

   1.1. Transformación de la consulta global (expresada en SQL) en una expresión equivalente del algebra relacional.

   1.2. A partir de la expresión del algebra relacional, obtener el Árbol algebraico correspondiente a la consulta.

   1.3. Transformación de una expresión algebraica en su correspondiente expresión canónica, substituyendo cada relación que aparece como operando por la expresión algebraica que reconstruye esa relación a partir de sus fragmentos.

2. Optimización de la consulta fragmentada utilizando una serie de criterios que se citan a continuación.

   2.1. Realizar las proyecciones sobre los fragmentos en lugar de realizarlos sobre tablas completas, y además obtener solo aquellos atributos que nos sean necesarios para realizar la consulta.

   2.2. Realizar las selecciones sobre los fragmentos correspondientes, en lugar de realizarlas sobre tablas completas.

   2.3. A raíz del objetivo anterior, buscar contradicciones entre las condiciones de las selecciones y las condiciones que definen cada uno de los fragmentos. Esto nos permitirá ignorar aquellos fragmentos que no sean necesarios para realizar una consulta.

   2.4. Obtener también contradicciones en las condiciones de los operandos de un producto natural distribuido.

   2.5. Distribución de los productos naturales que aparezcan en una consulta, para lo cual se deberán hacer antes los productos naturales, y después las uniones.

   2.6. Distribución de las agrupaciones y la evaluación de funciones de conjunto en una consulta global. Para ello, habrá que retrasar la aplicación de las uniones.

3. Una vez que hemos obtenido la consulta distribuida, expresarla de nuevo en SQL para poder acceder a la base de datos.

1.4 Motivación

Cada vez más se están considerando usar applets educativas con el fin de ayudar en el aprendizaje a los alumnos. Estas herramientas suelen ser muy sencillas de utilizar ya que están implementadas bajo una metodología centrada en el usuario. Por ello, resultan muy útiles a la hora de explicar temas complejos como en este caso es el procesamiento de consultas distribuidas.

La interfaz es muy simple tan solo es necesario que el usuario escriba la consulta para que pueda ver todo el proceso interno que conlleva lanzar una consulta distribuida. El usuario puede lanzar todo tipo de consultas desde la más simple a la más compleja

CAPÍTULO 2. DESCRIPCIÓN DEL PROBLEMA

2.1 Introducción

2.1.1 ¿Por qué las bases de datos distribuidas?

Hay varias razones por las cuales se desarrollan las bases de datos distribuidas. A continuación se listan algunas de las motivaciones fundamentales.

Razones económicas y de organización. Muchas organizaciones están descentralizadas, y un enfoque de base de datos distribuida se adapta de forma más natural a la estructura de la organización. Con los recientes desarrollos, las razones económicas para tener grandes centros centralizados están disminuyendo progresivamente. Este tipo de razones son probablemente las más importantes para el desarrollo de bases de datos distribuidas.

Interconexión de las bases de datos existentes. Las bases de datos distribuidas son la solución natural cuando existen varias bases de datos en una organización y surge la necesidad de desarrollar aplicaciones globales. En este caso, la base de datos distribuida se crea siguiendo un enfoque de *'abajo hacia arriba'* usando las bases de datos ya existentes. Este proceso podría requerir un cierto grado de reestructuración local; sin embargo, este esfuerzo es mucho menor que el que se necesita para la creación de una base de datos centralizada completamente nueva.

Crecimiento incremental. Si una organización crece añadiéndole nuevas unidades de organización autónomas, entonces el enfoque de base de datos distribuida soporta un crecimiento incremental con un mínimo grado de impacto sobre las unidades ya existentes. Con un enfoque centralizado, el crecimiento tiene un mayor impacto no solo sobre las nuevas aplicaciones sino también sobre las ya existentes.

Reducción de la sobrecarga en la comunicación. En una base de datos distribuida geográficamente, el hecho de que muchas aplicaciones sean locales reduce la sobrecarga de comunicación con respecto a una base de datos centralizada. Por esto, maximizar la localidad de las aplicaciones es uno de los objetivos principales en el diseño de bases de datos distribuidas.

Consideraciones de rendimiento. La existencia de varios procesadores autónomos influye en el incremento del rendimiento a través de un alto grado de paralelismo. Esta consideración se puede aplicar a cualquier sistema multiprocesador, y no solo a bases de datos distribuidas. Sin embargo, las bases de datos distribuidas tienen la ventaja de que la descomposición de los datos refleja que la aplicación depende de criterios que maximizan la localidad de la aplicación; de esta forma se minimiza la interferencia entre diferentes procesadores. La carga es compartida entre los diferentes procesadores, y se permiten los cuellos de botella críticos, tales como la red de comunicación o los servicios comunes. Este efecto es una consecuencia de los requisitos de capacidad de procesamiento autónoma para las aplicaciones locales en la definición de bases de datos distribuidas.

Fiabilidad y disponibilidad. El enfoque de base de datos distribuida, especialmente con datos redundantes, puede usarse también para obtener un alto grado de fiabilidad y disponibilidad. Sin embargo, conseguir este objetivo requiere del use de técnicas que todavía no se entienden completamente. La capacidad de procesamiento autónomo de los diferentes lugares no garantiza una total fiabilidad del sistema, pero asegura la propiedad de **degradación elegante**; en otras palabras, los fallos en una base de datos distribuida pueden ser más frecuentes que en una centralizada debido al mayor número de componentes, pero el efecto de cada fallo se limita a aquellas aplicaciones que usan los datos del lugar donde se produce el fallo, y es raro que falle el sistema completo.

Las motivaciones anteriores no son nuevas. ¿Por qué el desarrollo de sistemas de bases de datos distribuidas tiene un comienzo determinado? Hay dos razones fundamentales: primero, el reciente desarrollo de pequeños computadores, que proporcionan a bajo costo muchas de las capacidades que antes eran suministradas por grandes estaciones de trabajo, constituye el soporte hardware necesario para el desarrollo de sistemas de información distribuidos; segundo, la tecnología de bases de datos distribuidas está basada en otras dos tecnologías: tecnología de redes de computadores y la tecnología de bases de datos. Es una tarea complicada construir una base de datos distribuida sobre una red y un conjunto de sistemas de gestión de

base de datos en cada lugar; esto supone un gran esfuerzo sin esos bloques de construcción.

2.1.2 Comparación entre bases de datos distribuidas y centralizadas

Las bases de datos distribuidas no son simples implementaciones de bases de datos centralizadas, porque permiten el diseño de sistemas que presentan características distintas de los tradicionales sistemas centralizados. Las características del enfoque tradicional son: control centralizado, independencia de los datos, reducción de la redundancia, estructuras físicas complejas para un acceso eficiente, integridad, recuperación, control de la concurrencia, privacidad, y seguridad.

Control centralizado. La posibilidad de suministrar control centralizado sobre la información de una gran empresa u organización fue considerada como una de las motivaciones más fuertes para introducir las bases de datos; fueron desarrolladas como ficheros. La función fundamental de un **administrador de bases de datos (DBA)** era garantizar la seguridad de los datos.

En las bases de datos distribuidas, la idea de control centralizado tiene un menor énfasis; esto depende también de la arquitectura. En general, en bases de datos distribuidas es posible identificar una estructura de control jerárquica basada en un **administrador global de la base de datos**, que es el responsable central de la base de datos, y en **administradores locales de la base de datos**, que son los responsables de sus correspondientes bases de datos. Sin embargo, se debe enfatizar que los administradores locales de las bases de datos deben tener un alto grado de autonomía, hasta el punto en que el administrador global de la base de datos este totalmente ausente y la coordinación entre localidades se realiza por los administradores locales. Esta característica se llama usualmente **autonomía local**. Las bases de datos distribuidas podrían diferir bastante en el grado de la autonomía local: desde completar la autonomía local sin un administrador centralizado de base de datos hasta casi completamente el control centralizado.

Independencia de datos. Es también una de las principales motivaciones para introducir el enfoque de las bases de datos. Esencialmente, significa que la

organización actual de los datos es transparente al programador de aplicaciones. Los programas se escriben teniendo una vista 'conceptual' de los datos, es lo que se llama esquema conceptual. La principal ventaja de la independencia de los datos es que los programas no se ven afectados por cambios en la organización física de los datos.

En bases de datos distribuidas, la independencia de datos tiene la misma importancia que en las B.D. tradicionales, pero se añade un nuevo aspecto que es la transparencia en la distribución. Por transparencia en la distribución entendemos que los programas se escriben como si la B.D. no esté distribuida. De esta forma, la corrección de los programas no se ve afectada por el movimiento de los datos de un lugar a otro; sin embargo, su velocidad de ejecución si se ve afectada.

La independencia de datos se proporcionaba en B.D. a través de una arquitectura multinivel con diferentes descripciones de los datos y direccionamiento entre ellos; para este propósito se desarrollaron las nociones de esquema conceptual, esquema de almacenamiento y esquema externo. De forma similar, la transparencia en la distribución se obtiene en B.D. distribuidas (B.D.D.) introduciendo nuevos niveles y esquemas.

Reducción de la redundancia. En B.D. tradicionales, la redundancia se reducía en la medida de lo posible por dos razones: primero, las inconsistencias entre varias copias de los mismos datos lógicos se evitan teniendo solo una copia, y segundo, con la eliminación de redundancia se ahorra espacio de almacenamiento. La reducción de la redundancia se obtiene a partir de los datos compartidos, por ejemplo, permitiendo que varias aplicaciones accedan a los mismos ficheros y registros.

Sin embargo, en B.D.D., hay varias razones para considerar la redundancia de los datos como una característica deseable: primero, la localidad de las aplicaciones se ye incrementada si los datos se replican en todos los sitios donde las aplicaciones los necesitan, y segundo, la validez del sistema se puede incrementar, debido a que si los datos están replicados, un fallo en un lugar no interrumpe la ejecución de las aplicaciones en otros lugares. La conveniencia de la replicación de los datos se incrementa debido a que si nosotros tenemos varias copias de un elemento, podemos

realizar la recuperación sobre una copia, mientras que las actualizaciones se deben realizar sobre todas las copias.

Estructuras físicas complejas y accesos eficientes. El soporte para estas estructuras es la parte más importante de los sistemas de gestión de bases de datos (DBMSs). La razón para suministrar estructuras de acceso complejas, es obtener eficiencia en el acceso a los datos.

En B.D.D., las estructuras de acceso complejas no son la herramienta adecuada para el acceso eficiente. No se necesita suministrar accesos eficientes a una B.D.D. a través de estructuras físicas entre los distintos lugares, porque es muy difícil de construir y mantener tales estructuras y porque no es conveniente para navegar' por el nivel de registros en B.D.D.

Un plan de acceso distribuido puede ser escrito por el programador o producido automáticamente por un optimizador. Escribir un plan de acceso distribuido es similar a escribir un programa de navegación en bases de datos centralizadas, en el sentido de que el programador especifica cómo se debe acceder a la base de datos. Sin embargo, la navegación entre lugares se debe desarrollar a nivel de grupos de registros, mientras que se puede desarrollar la navegación de un registro cada vez para el procesamiento local en cada lugar. Por eso, para construir planes de acceso un lenguaje de navegación es menos apropiado que uno no procedural orientado a conjuntos.

Todavía quedan problemas por resolver en el diseño de un optimizador que produce automáticamente un plan de acceso. Es conveniente dividir estos problemas en dos categorías: **optimización global** y **optimización local**. La optimización global consiste en determinar qué datos deben ser accedidos en cada lugar y por consiguiente que ficheros de datos deben ser transmitidos entre los distintos lugares. El principal parámetro para la optimización global es el costo de comunicación, aunque el costo de acceso a las bases de datos locales también se ha de considerar en algunos casos. La relativa importancia de estos factores depende de la relación entre los costos de comunicación y los costos de acceso a disco, que dependen del tipo de red de comunicación.

La optimización local consiste en decidir coma desarrollar los accesos a la base de datos de cada lugar; los problemas de la optimización global son típicos de las bases de datos no distribuidas.

Integridad, recuperación y control de concurrencia. Estos conceptos, aunque se refieren a problemas distintos, están muy relacionados. Para muchos, la solución es proporcionar transacciones. Una **transacción** es una unidad atómica de ejecución; por ejemplo, una secuencia de operaciones en la que cada una se desarrolla por completo o no se desarrolla.

Está claro que las transacciones atómicas son el medio de obtener la integridad de la base de datos, porque aseguran que se realizan todas las acciones que transforman la base de datos de un estado consistente a otro, o el estado consistente inicial no se ye alterado.

Hay dos enemigos peligrosos de la atomicidad de la transacción: fallos y concurrencia. Los fallos podrían causar que el sistema se interrumpa en la mitad de la ejecución de una transacción, violando así el requerimiento de atomicidad. La ejecución concurrente de distintas transacciones podría permitir una inconsistencia.

El control de la concurrencia trata de asegurar la atomicidad de la transacción en presencia de ejecución concurrente de transacciones. Este problema se puede ver como un problema de sincronización, que en bases de datos distribuidas (como en todos los sistemas distribuidos) es más duro que los sistemas centralizados.

Privacidad y seguridad. En bases de datos tradicionales, el administrador de la base de datos, que tiene un control centralizado, puede asegurar que solo se realicen los accesos autorizados. Sin embargo, el enfoque de las bases de datos centralizadas es en sí mismo, sin procedimientos de control especializados, más vulnerable a violaciones de privacidad y seguridad que otros enfoques más antiguos de ficheros separados.

En bases de datos distribuidas, los administradores locales se enfrentan esencialmente con el mismo problema que los administradores de bases de datos en

una base de datos tradicional. Sin embargo hay dos aspectos peculiares en una base de datos distribuida: primero, en una base de datos distribuida con un alto grado de autonomía entre los lugares, los propietarios de los datos locales se sienten más protegidos porque fuerzan sus propias protecciones en lugar de depender de un administrador central de la base de datos; segundo, los problemas de seguridad son intrínsecos a sistemas distribuidos en general, porque las redes de comunicación pueden representar un punto débil con respecto a la protección.

2.2 Etapas en el diseño de bases de datos distribuidas

Cuando diseñamos una base de datos centralizada nos ocupamos de:

1.- Diseñar el esquema conceptual que describe todos los datos y relaciones entre datos.

2- Diseñar la base de datos física.

En una base de datos distribuida estos dos problemas se convierten en diseño del esquema global y el diseño de las bases de datos físicas locales; las técnicas que se pueden aplicar a estos problemas son las mismas que en bases de datos centralizadas. La distribución de bases de datos añade a las etapas anteriores las dos siguientes:

3. Diseño de la fragmentación.
4. Diseño de la asignación de fragmentos.

La distinción entre estos dos problemas es conceptualmente relevante, mientras el primero se ocupa del criterio lógico que motiva la fragmentación de la relación global, el segundo se ocupa de la localización física de los datos en varias localidades. Sin embargo, esta distinción debe ser introducida con cuidado; en general, no es posible determinar la fragmentación óptima y la asignación resolviendo los dos problemas independientemente, puesto que están relacionados.

Aunque el diseño de programas de aplicación se hace después del diseño del esquema, el conocimiento de los requerimientos de la aplicación influye en el diseño

del esquema, puesto que el esquema debe ser capaz de soportar aplicaciones eficientemente. Así, en el diseño de una base de datos, se necesita un conocimiento suficientemente preciso de los requerimientos de la aplicación. En los requerimientos de la aplicación se incluyen:

1.- El lugar origen de la aplicación.
2.- La frecuencia de activación de la aplicación.
3.- El número, tipo, y la distribución estadística de los accesos hechos por cada aplicación a cada dato objeto requerido.

2.2.1 Características fundamentales del diseño de bases de datos distribuidas

En el diseño de datos distribuidos, han de tenerse en cuenta las siguientes características:

Localidad de Procesamiento: Se distribuirán los datos para maximizar el procesamiento local. La forma más simple para caracterizar la localidad de procesamiento es considerar dos tipos de referencias a los datos*: local y remota.* Claramente, una vez que son conocidos los lugares de origen de las aplicaciones, las referencias locales y remotas dependen solo de la distribución de los datos.

Una extensión a este simple criterio de optimización se toma cuando una aplicación tiene localidad completa. Usamos este término para designar las aplicaciones que pueden ser completamente ejecutadas en sus localidades de origen.

Disponibilidad y fiabilidad de los datos distribuidos: Se consigue un alto grado de disponibilidad para aplicaciones de solo lectura mediante el almacenamiento de múltiples copias de la misma información; el sistema debe ser capaz de cambiar a una copia alternativa cuando no está disponible una que debería ser accesible en condiciones normales.

La fiabilidad se consigue también mediante el almacenamiento de múltiples copias de la misma información, puesto que es posible recuperarla ante caídas del sistema o ante perdidas físicas de una de las copias.

Distribución de carga: La distribución de carga se hace para aprovechar las diferentes potencias o recursos de cada localidad, y para maximizar el grado de paralelismo en la ejecución de aplicaciones. Puesto que la distribución de carga puede afectar negativamente a la localidad de procesamiento, será necesario considerar los pros y contras en el diseño.

Costes de almacenamiento y disponibilidad: La distribución de la base de datos debe reflejar el costo y disponibilidad de almacenamiento en los diferentes lugares. Es posible tener lugares especializados en la red para almacenamiento de datos, o por el contrario tener lugares que no soporten almacenamiento masivo. Normalmente, el costo del almacenamiento de datos de aplicaciones no es relevante comparado con los costos de CPU, I/O, y de transmisión de las aplicaciones, pero la limitación de almacenamiento disponible debe ser considerada en cada lugar.

2.2.2 Estrategias de diseño de bases de datos distribuidas

Existen dos alternativas al diseño de datos distribuidos, las estrategias descendente (bottom-down) y ascendente (bottom-up).

En la estrategia descendente, comenzamos por el diseño del esquema global, y procedemos al diseño de la fragmentación de la base de datos, y después continuamos con la asignación de fragmentos a las localidades, creando las imágenes físicas. La estrategia se completa con el desarrollo, en cada lugar, del *diseño físico* de los datos que están asignados a él.

Esta estrategia es la más atractiva para sistemas que son especificados desde un principio, como una base de datos distribuida.

Cuando la base de datos distribuida se desarrolla como la agregación de bases de datos existentes, no es fácil seguir la estrategia descendente. En estos casos el esquema global es producido frecuentemente como un compromiso entre las descripciones de datos existentes.

Cuando las bases de datos existentes son agregadas, puede usarse una estrategia ascendente para el diseño de la distribución de datos. Esta estrategia está basada en la **integración** de los esquemas existentes en un solo esquema global.

Debe resaltarse que la estrategia ascendente se presta menos fácilmente al desarrollo de relaciones fragmentadas horizontalmente. De hecho, los fragmentos horizontales de una misma relación global deben tener el mismo esquema de relación; esta característica se fuerza fácilmente en un diseño descendente, mientras que es difícil descubrirla a posteriori, en bases de datos diseñadas independientemente y que posteriormente son integradas. Puesto que los fragmentos horizontales son relevantes y una característica útil de una base de datos distribuida, el proceso de integración debe intentar modificar las definiciones de las relaciones locales, de manera que se puedan consideran como fragmentos horizontales de una relación global común.

En resumen, el diseño ascendente requiere:

1.- La selección de un modelo común de bases de datos para describir el esquema global.

2.- La traducción de cada esquema local al modelo de datos común.

3.- La integración del esquema local en el esquema global común.

2.2.3 El diseño de la fragmentación de la base de datos

El diseño de la fragmentación es el primer problema que debemos resolver en un diseño descendente de datos distribuidos. El propósito del diseño de la fragmentación es determinar fragmentos no solapados que sean *unidades lógicas de asignación.*

El diseño de fragmentos consiste en la agrupación de tuplas, en el caso de fragmentación horizontal, o atributos, en el caso de la vertical, que tienen las mismas propiedades desde el punto de vista de su asignación.

La descomposición de relaciones globales en fragmentos se puede realizar aplicando dos tipos distintos de fragmentación: **horizontal** y **vertical**. En principio los

veremos de forma separada y entonces consideraremos la fragmentación más compleja que se puede obtener al aplicar una combinación de ambas.

En todos los tipos de fragmentación, un fragmento se puede definir como una expresión de un lenguaje relacional (nosotros usaremos el álgebra relacional) que toma relaciones globales como operandos y produce el fragmento como resultado. Por ejemplo, si una relación global contiene datos sobre empleados, se puede definir (usando la operación de selección) un fragmento que contenga solo datos sobre empleados que trabajan en el departamento D_1.

Sin embargo, hay algunas reglas que se deben seguir a la hora de definir fragmentos:

Condición de completitud: todos los datos de una relación global deben estar en algún fragmento, es decir, no debe suceder que un dato que pertenezca a una relación global, no esté en ninguno de los fragmentos.

Condición de reconstrucción: siempre se debe poder reconstruir una relación global a partir de sus fragmentos. La necesidad de esta condición es obvia, puesto que una base de datos distribuida solo se almacena los fragmentos, y si es necesario, la relación global se ha de construir usando esta operación de reconstrucción.

Condición de fragmentación disjunta: los distintos fragmentos de una relación global deben ser disjuntos. Esta es una regla que la deben cumplir los fragmentos horizontales pues de los contrario se repetirían tuplas en algunos fragmentos, mientras que se puede relajar un poco en los fragmentos verticales.

2.2.3.1 Fragmentación Horizontal

Consiste en dividir las tuplas de una relación global en subconjuntos; esto resulta especialmente útil en bases de datos distribuidas, donde cada subconjunto puede contener datos que tengan propiedades geográficas comunes.

Existen dos tipos de fragmentación llamados primaria y derivada; la fragmentación derivada se define en términos de la fragmentación primaria.

Determinar la fragmentación horizontal de una base de datos equivale a determinar las propiedades lógicas de los datos, así como los predicados de fragmentación, y las propiedades estadísticas de los datos, así como el número de aplicaciones que referencian a los fragmentos; esta coordinación de aspectos lógicos y estadísticos es bastante difícil.

2.2.3.1.1 Fragmentación Primaria

La fragmentación horizontal primaria se define usando selecciones en relaciones globales; la correctitud de la fragmentación primaria requiere que cada tupla de la relación global sea seleccionada en un solo fragmento. Así, determinar la fragmentación primaria de una relación global requiere determinar un conjunto disjunto y completo de predicados de selección. La propiedad que requerimos para cada fragmento es que los elementos de estos deben estar referenciados homogéneamente por todas las aplicaciones. La completitud se garantiza con la típica restricción de integridad para las bases de datos, y es lo que se llama **integridad referencial**.

Se puede definir expresando cada fragmento como una operación de selección sobre la relación global. La correspondiente operación de reconstrucción de la relación global es la unión.

Llamaremos **cualificación** al predicado que se usa en la operación de selección para definir un fragmento. En términos de cualificación, una fragmentación horizontal es completa si el conjunto de cualificaciones es completo respecto a los valores permitidos. Por otro lado una fragmentación horizontal es disjunta cuando el conjunto de cualificaciones sea mutuamente excluyente.

2.2.3.1.2 Fragmentación horizontal derivada

La fragmentación horizontal derivada de una relación global R no está basada en propiedades de sus atributos, sino que deriva de la fragmentación horizontal de otra relación. La fragmentación derivada se usa para facilitar el producto natural entre fragmentos.

La operación de algebra relacional que la realiza es el semi producto natural, mientras que la operación de reconstrucción sigue siendo la unión, puesto que se trata de una fragmentación horizontal.

2.2.3.2 Fragmentación Vertical

Determinar la fragmentación vertical de una relación global R requiere agrupar en conjuntos los atributos que son referenciados de la misma forma por las aplicaciones. Distinguimos en el problema de **particionamiento vertical**, que los conjuntos deben ser disjuntos, y en el problema de **agrupamiento vertical**, que los conjuntos se pueden solapar. Las condiciones de correctitud para el particionamiento vertical requieren que cada atributo de R pertenezca al menos a un conjunto y que cada conjunto incluya una llave de R o un identificador de tupla.

La operación correspondiente del algebra que realiza la fragmentación vertical es la proyección de aquellos atributos que se desea estén en cada uno de los fragmentos. Sera completa cuando todos los atributos estén en algún fragmento. La operación de reconstrucción de la relación global es la del producto natural.

Determinar la fragmentación para una relación global R no es fácil, puesto que el número de posibles particionamientos crece combinatoriamente con el número de atributos de R, y el número de posibles agrupaciones es incluyo mayor. Así, ante la presencia de una relación grande, son necesarias aproximaciones heurísticas que determinen las particiones o las agrupaciones; indicaremos generalmente como deben operar tales métodos.

Hay dos posibles aproximaciones alternativas para fragmentar atributos:

1.- Escindir (partir). La aproximación de división en la cual las relaciones son progresivamente divididas en fragmentos.

2.- Agrupar. La aproximación de agrupación en la cual los atributos son progresivamente agregados para constituir fragmentos.

Ambas aproximaciones pueden ser clasificadas como heurísticas avariciosas (*Greedy*), puesto que realizan en cada iteración la mejor de las posibles elecciones. En ambos casos, se usan fórmulas para indicar cuál es el mejor agrupamiento o división.

El agrupamiento vertical introduce **replicación** de fragmentos, puesto que son replicados los valores de los atributos solapados. Las aplicaciones de solo-lectura se benefician de la replicación, porque pueden referenciar datos localmente. Para aplicaciones de modificación la replicación no es conveniente, puesto que deben modificarse todas las copias, para conservar la consistencia.

2.2.3.3 Fragmentación Mixta

Los fragmentos obtenidos por las operaciones anteriores son también relaciones, por lo que se les puede aplicar operaciones de fragmentación de forma recursiva. La reconstrucción se obtiene aplicando las reglas de reconstrucción en orden inverso al que se han ido aplicando las operaciones de fragmentación.

Para representar la fragmentación mixta se usan los **árboles de fragmentación**. En un árbol de fragmentación, la raíz corresponde a una relación global, los nodos hoja corresponden a los fragmentos, y los nodos intermedios corresponden a los resultados intermedios de las expresiones que definen los fragmentos. El conjunto de nodos que son hijos de un nodo dado representa la descomposición de este nodo por una operación de fragmentación (horizontal o vertical).

La manera más simple de construir una fragmentación mixta consiste en:

1.- Aplicar fragmentación horizontal a fragmentos verticales.

2.- Aplicar fragmentación vertical a fragmentos horizontales.

Aunque estas operaciones pueden ser repetidas recursivamente, generando arboles de fragmentación de cualquier complejidad, parece que no es de interés practico tener más de dos niveles de fragmentación.

2.2.4 Ejemplo general

En este apartado, mostramos un ejemplo de cómo se puede realizar un diseño de la fragmentación para una base de datos distribuida. El esquema general es el siguiente:

Empleado = (Enum, nombre, salario, IRPF, mgrnum, Dnum)
Departamento=(Dnum,nombre,área,mgrnum)
Proveedor=(Pnum,nombre,ciudad)
Suministro = (Pnum, Anum, Dnum, cantidad)

Consideraremos una base de datos distribuida para una compañía en California que tiene tres sucursales en San Francisco (localidad 1), Fresno (localidad 2) y en Los Ángeles (localidad 3); Fresno está situada a la mitad entre San Francisco y Los Ángeles. Hay 30 departamentos, agrupados físicamente de la siguiente forma:

- Los primeros 10 están cerca de San Francisco,
- Los departamentos entre el 11 y 20 están cerca de Fresno, y
- Los departamentos por encima de 20 están cerca de Los Ángeles.

Los proveedores están en San Francisco o Los Ángeles de forma equitativa. Además está la noción de áreas en función de la que se divide la compañía; el área 'Norte' incluye San Francisco , el área 'Sur', incluye Los Ángeles, y Fresno esta exactamente en la frontera, con algunos departamentos cercanos a San Francisco en el área norte, y otros en el área sur.

Comencemos con la relación Proveedor = (Pnum, nombre, ciudad). Ciudad solo puede tomar dos valores, 'SF ' o 'LA'. Una aplicación importante pide el nombre del proveedor que tenga un determinado número. Una consulta SQL para esta aplicación es:

select nombre from Proveedor where Pnum = $X

La aplicación se puede ejecutar en cualquiera de las tres localidades; si se genera en 'SF', se van a referenciar proveedores de 'SF'' con un 80% de probabilidad; si se genera en 'Fresno', se referencian proveedores de 'SF' y de 'Los Ángeles'

con igual probabilidad; si se genera en 'LA', se referencian proveedores de 'LA' con un 80% de probabilidad. Esto se debe a que los departamentos cercanos a una ciudad tienden a usar proveedores cercanos a ella.

Consideramos el conjunto P para fragmentar Proveedor:

P = {p1, p2}

p1: ciudad = 'SF'

p2: ciudad = 'LA'

P es un conjunto completo y minimal.
Aunque el ejemplo es simple, se pueden ver dos características importantes:

1. Los predicados relevantes para realizar la fragmentación (predicados de P) no pueden deducirse del código de la aplicación; en nuestro ejemplo, los valores de ciudad generan los predicados, sin embargo en la consulta no hay referencia a ellos.

2. Las simplificaciones entre los predicados de P pueden reducir el número de fragmentos. En este caso, podríamos considerar como fragmentos, los correspondientes a los siguientes términos de predicado:

y1: (ciudad = ' SF') AND (ciudad = 'LA')

y2: (ciudad = ' SF') AND NOT (ciudad = 'LA')

y3: NOT (ciudad = ' SF') AND (ciudad = 'LA')

y4: NOT (ciudad = ' SF') AND NOT (ciudad = 'LA')

pero sabemos que:

(ciudad = 'LA') = NOT (ciudad = 'SF')

y

(ciudad = 'SF') = NOT (ciudad = 'LA')

por lo que podemos inferir que y1 e y4 son contradicciones, y que y2 e y3 se reducen a p1 y p2.

$Proveedor_1 = \sigma_{ciudad='SF'}(Proveedor)$

$Proveedor_2 = \sigma_{ciudad='LA'}(Proveedor)$

Consideremos ahora la relación global

Departamento (Dnum, nombre, area, mgmum)

Hay dos tipos de aplicaciones importantes:

1. Aplicaciones administrativas; se generan en las localidades 'SF' o 'LA' de forma que las aplicaciones administrativas sobre departamentos del área norte, se generan en 'SF' y las que son sobre departamentos del área sur, se generan en 'LA''.

2. Aplicaciones sobre el trabajo desarrollado por cada departamento; pueden generarse en cualquier localidad, pero referencian a tuplas de los departamentos más cercanos a su localidad de origen con mayor probabilidad que a tuplas de otros departamentos.

Obtenemos el siguiente conjunto de predicados:

p1: Dnum ≤ 10
p2: 10<Dnum ≤20
p3:Dnum>20
p4: area = 'Norte'
p5: area = 'Sur'

Sin embargo hay implicaciones entre ellos (por ejemplo, area='Norte' implica que Dnum > 20 sea falso); por eso, la fragmentación se reduce a cuatro fragmentos:

y1: Dnum≤10
y2: (10 < Dnum≤20) AND (area = 'Norte')
y3: (10 < Dnum ≤ 20) AND (area = 'Sur')

y4: Dnum > 20

Esta fragmentación se representa en la siguiente tabla (Tabla 2.1) donde las filas tienen los predicados p1, p2 y p3, y las columnas tienen los predicados p4 y p5.

	P4: área = 'Norte'	**P5: área = 'Sur'**
P1: dnum ≤ 10	Y1	FALSE
P2: 10 < Dnum ≤ 20	Y2	Y3
P3: Dnum > 20	FALSE	Y4

Tabla 2.1

$Departamento_1 = \sigma_{Dnum \leq 10}$ (Departamento)

$Departamento_2 = \sigma_{10 < Dnum \leq 20}$ (Departamento)

$Departamento3 = \sigma_{Dnum > 20}$ (Departamento)

Como última consideración para este ejemplo, comentar algo sobre la asignación de fragmentos. Claramente, los fragmentos correspondientes a y1 e y4 deben situarse en 'SF' y 'LA' respectivamente. Sin embargo para la asignación de y2 e y3 tenemos dos alternativas:

- Para aplicaciones administrativas, situaríamos y2 en 'SF' e y3 en 'LA'.

- Para aplicaciones sobre el trabajo de los departamentos, situaríamos los fragmentos en Fresno.

Consideremos ahora la relación Suministro (Pnum, Anum, Dnum, cantidad).

- Algunas aplicaciones requieren información sobre los suministros de determinados proveedores; para ello se realiza el producto natural entre Proveedor y Suministro.

$Suministro_1 = Suministro\ JN_{Pnum=Pnum}\ Proveedor_1$

$Suministro_2 = Suministro\ JN_{Pnum=Pnum}\ Proveedor_2$

- Otras aplicaciones necesitan información sobre los suministros de un departamento, para lo que se tendrá que hacer el producto natural entre Departamento y Suministro.

$Suministro_1$ = Suministro JN $_{Pnum=Pnum}$ $Departamento_1$
$Suministro_2$ = Suministro JN $_{Pnum=Pnum}$ $Departamento_2$
$Suministro_3$ = Suministro JN $_{Pnum=Pnum}$ $Departamento_3$

Escogemos una u otra según el número de veces que se ejecute cada aplicación.

Consideramos ahora la relación Empleado (Enum, nombre, salario, IRPF, mgmum, Dnum).

Supongamos que hay dos tipos de aplicaciones que usan esta relación:

1. Aplicaciones administrativas, concentradas en la localidad 3, que requieren nombre, salario e IRPF de los empleados.
2. Aplicaciones sobre el trabajo que se desarrolla en cada departamento, que requieren nombre, mgmum, y Dnum de los empleados; estas aplicaciones se generan en todas las localidades y se referencian con una probabilidad del 80% a tuplas de empleados de un mismo grupo de departamentos.

Dadas esas aplicaciones, hacemos una fragmentación vertical. Incluimos la llave Enum en ambos fragmentos. Si usamos fragmentación vertical, debemos decidir cuál de los dos fragmentos tiene el atributo nombre. En este caso una fragmentación vertical no sería eficiente, ya que cualquiera de las dos aplicaciones podría hacer un producto natural y varias referencias no locales. Debido a que el atributo nombre es relativamente estable (los empleados no cambian a menudo de nombre), sería más óptimo usar agrupación vertical, dando lugar a la siguiente fragmentación:

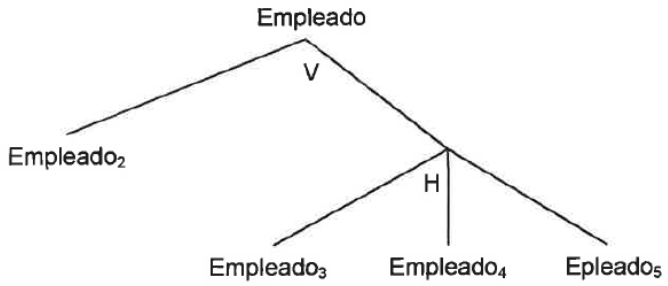
$Empleado_1$ (Enum, nombre, salario, IRPF)
$Empleado_2$ (Enum, nombre, mgmum, Dnum)

Consideremos de nuevo la relación global Empleado, fragmentada verticalmente en $Empleado_1$ y $Empleado_2$.

Consideremos que las aplicaciones que tratan con el trabajo realizado por cada uno de los departamentos, que usan $Empleado_2$, referencian con una probabilidad del 80% a tuplas de los departamentos situados alrededor de la localidad en la que son generados; en este caso $Empleado_2$ puede ser fragmentado horizontalmente por grupos de departamento, obteniendo la siguiente fragmentación:

$Empleado_1 = \sigma_{Dnum \leq 10} (\prod_{Enum,nombre,mgrnum,Dnum} (Empleado))$
$Empleado_2 = \sigma_{10 < Dnum \leq 20} (\prod_{Enum,nombre,mgrnum,Dnum} (Empleado))$
$Empleado3 = \sigma_{Dnum > 20} (\prod_{Enum,nombre,mgrnum,Dnum} (Empleado))$
$Empleado4 = \prod_{Enum,nombre,salario,IRPF} (Empleado)$

Podemos representar esta fragmentación de Empleado en el siguiente árbol:



Por tanto, teniendo en cuenta todas esas consideraciones, el esquema de fragmentación definitivo es el siguiente:

$Empleado_1 = \sigma_{Dnum \leq 10} (\prod_{Enum,nombre,mgrnum,Dnum} (Empleado))$
$Empleado_2 = \sigma_{10 < Dnum \leq 20} (\prod_{Enum,nombre,mgrnum,Dnum} (Empleado))$
$Empleado_3 = \sigma_{Dnum > 20} (\prod_{Enum,nombre,mgrnum,Dnum} (Empleado))$
$Empleado_4 = \prod_{Enum,nombre,salario,IRPF} (Empleado)$
$Departamento_1 = \sigma_{Dnum \leq 10} (Departamento)$

$Departamento_2 = (\sigma_{10<Dnum\leq 20}$ (Departamento)

$Departamento3 = \sigma_{Dnum>20}$ (Departamento)

$Proveedor_1 = \sigma_{ciudad='SF''}$ (Proveedor)

$Proveedor_2 = \sigma_{ciudad='LA'}$, (Proveedor)

$Suministro_1 = SuministroJN_{Pnum=Pnum}\ Proveedor_1$

$Suministro_2 = Suministro\ JN_{Pnum=Pnum}\ Proveedor_2$

2.3 Procesamiento distribuido de consultas

2.3.1 Introducción

El objetivo del procesamiento distribuido de consultas es dividir una consulta en subconsultas que accedan a fragmentos. Una consulta global es una consulta que hace referencia a relaciones globales, mientras que una consulta fragmentada hace referencia a fragmentos. El sistema de BDD es el encargado de transformar una consulta global en consultas fragmentadas.

Para hacer esta Transformación se pueden usar una serie de reglas que se llaman *transformaciones de equivalencia* y son conjuntos de reglas que se pueden aplicar a una consulta global con el objetivo de reescribirla en una expresión equivalente en términos de consultas fragmentadas. Se van a usar para intentar simplificar las consultas globales.

2.3.2 Transformaciones de equivalencia para consultas

Una consulta relacional se puede expresar usando diferentes lenguajes: SQL, algebra relacional, calculo relacional. Para nuestro propósito vamos a usar algebra relacional y SQL. Como sabemos, es posible transformar la mayoría de consultas SQL a expresiones equivalentes del algebra relacional y viceversa; por tanto, cualquiera de los lenguajes anteriores sirve para expresar la semántica de una consulta. Sin embargo, podemos interpretar una expresión del algebra relacional no solo como la especificación de la semántica de una consulta, sino también como la especificación de una secuencia de operaciones. Desde este punto de vista, dos expresiones con la

misma semántica pueden describir dos secuencias distintas de operaciones. Por ejemplo,

$$PJ_{nombre,Dnum}\ SL_{Dnum=15}\ Empleado$$

y

$$SL_{Dnum=15}\ PJ_{nombre,Dnum}\ Empleado$$

son expresiones equivalentes pero definen dos secuencias distintas de operaciones.

A partir de ahora, tal y como se ha representado el ejemplo anterior, la notación que se va a utilizar para representar los distintos operadores del algebra relacional es la siguiente:

SL_F: selección	PJ_A: proyección	JN_F : producto natural
SJ_F: semi join	DF: diferencia	UN: unión
CP: producto cartesiano		

2.3.2.1 Árbol algebraico de una consulta

Un árbol algebraico es una manera de representar una consulta gráficamente. Es un árbol cuyas hojas son relaciones y los nodos son operaciones unarias o binarias. Un árbol algebraico, en definitiva, representa el orden parcial de ejecución de las operaciones de una consulta. El árbol algebraico se puede considerar como el árbol de producción de la expresión en si misma considerando la siguiente gramática:

R → identificador
R→(R)
R → *operador_unario* R
R → R *operador_binario* R
Operador_unario → SL_F | PJ_A
Operador_binario → UN |DF | CP | JN_F | SJ_F

Consideremos por ejemplo la consulta Q1 que requiere el número de proveedor (Pnum) de aquellos proveedores que han suministrado artículos en el área Norte. La consulta corresponde a la siguiente expresión del algebra relacional.

Q1: PJ_{pnum} $SL_{area='Norte'}$ (Suministro JN $_{Dnum=Dnum}$ Departamento)

En la figura 2.1 se muestra el árbol algebraico para la consulta Q1.

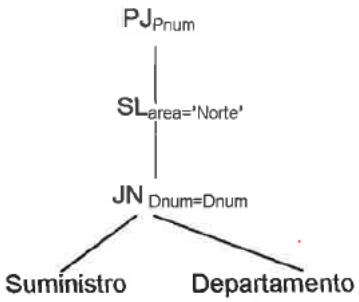


Figura 2.1 Árbol algebraico para la consulta Q1

Si observamos con detenimiento, podemos darnos cuenta que sería más apropiado hacer la operación de selección sobre la relación Departamento puesto que área es un atributo de dicha relación.

2.3.2.2 Transformaciones de equivalencia en el álgebra relacional

Dos relaciones son equivalentes (R⇔S) cuando representan la misma transformación de nombres de atributos en valores, independientemente del orden de esos atributos. Por ejemplo las dos relaciones siguientes son equivalentes.

R	
A	B
A_1	b_1
a_2	b_2
a3	b_3

S	
C	D
b_1	a_1
b_2	a_2
b_3	a_3

Dos expresiones del algebra relacional son equivalentes ($E_1 \Leftrightarrow E_2$) cuando los resultados de cada una de ellas son relaciones equivalentes.

Supongamos $E_1 \rightarrow R_1$ y $E_2 \rightarrow R2$ .Si $R_1 \Leftrightarrow R_2$ entonces $E_1 \Leftrightarrow E_2$

La equivalencia de transformaciones se puede dar sistemáticamente para expresiones pequeñas, por ejemplo, expresiones de dos o tres operandos. Estas transformaciones se clasifican en categorías de acuerdo al tipo de los operadores que se ven involucrados. Vamos a denotar U y B a las operaciones unarias y binarias respectivamente. Tenemos:

• Conmutatividad de operaciones unarias:

$$U_1\ U_2\ R \Leftrightarrow\ U_2\ U_1\ R$$

• Conmutatividad de operandos en operaciones binarias:

$$R\ B\ S \Leftrightarrow S\ B\ R$$

• Asociatividad de operaciones binarias:

$$R\ B(S\ B\ T) \Leftrightarrow (R\ B\ S)B\ T$$

• Idempotencia de operaciones unarias:

$$U\ R \Leftrightarrow U_1\ U_2\ R$$

Ejemplo: PJ_{nombre} (Empleado) $\Leftrightarrow PJ_{nombre}\ PJ_{nombre,Dnum}$ (Empleado)

• Distributividad de operadores unarios con respecto a operadores binarios:

$$U\ (R\ B\ S) \rightarrow U\ (R)\ B\ U\ (S)$$

• Factorización de operaciones unarias:

$$U(R)\ B\ U(S) \rightarrow U(R\ B\ S)$$

Es la inversa que la anterior, sin embargo, no se pone doble flecha porque no siempre se da la doble implicación para la misma pareja de operadores.

Para optimizar el árbol algebraico y por tanto la consulta, existen dos criterios:

Criterio 1. Utilizar la idempotencia de la selección y de la proyección para generar selecciones y proyecciones más adecuadas para cada relación que aparece como operando en una consulta.

Criterio 2. Bajar las selecciones y las proyecciones lo más posible en el árbol de una consulta.

Estos criterios derivan de la consideración de que las operaciones binarias, especialmente los productos naturales, son las operaciones más costosas en sistemas de bases de datos, y por tanto es conveniente reducir los tamaños de los operandos de operaciones binarias antes de realizarlas. En bases de datos distribuidas, estos criterios son incluso más importantes: las operaciones binarias requieren la comparación de operandos que podrían estar en distintas localidades. La transmisión de datos es uno los mayores componentes de los costos y retrasos asociados con la ejecución de consultas.

La figura 2.2 muestra un árbol algebraico modificado para la consulta Q1.

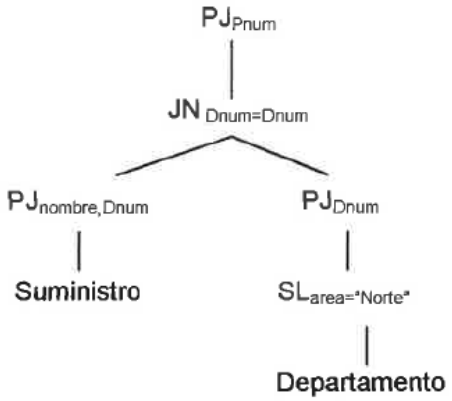


Figura 2.2 Un árbol algebraico modificado para Q1

En dicho árbol se han aplicado las siguientes transformaciones:

1. Distribución de la selección con respecto al producto natural; así, la selección se aplica directamente a la relación **Departamento**.
2. Generación de dos nuevas operaciones de proyección y se distribuyen con respecto al producto natural.

Citar que se ha aplicado el criterio 1 (uso de la idempotencia) a la proyección antes de distribuirla; en cambio, la selección, se ha trasladado directamente a la relación Departamento.

2.3.2.3 Grafo de operadores y determinación de subexpresiones comunes

Una característica importante al aplicar transformaciones a una expresión es determinar las sub expresiones comunes, es decir, subexpresiones que aparecen en una expresión más de una vez. La idea es que dichas subexpresiones comunes sean evaluadas una sola vez. Para ello se transforma el correspondiente árbol algebraico en un grafo de operadores en el que se combinan aquellas hojas que representan la misma relación y a continuación se combinan otros nodos intermedios que representan operadores u operaciones comunes.

Para ver el método, consideremos la siguiente consulta: *"dar los nombres de los empleados que trabajan en un departamento cuyo gestor tiene el número 373, pero que no ganen más de 35000".* Una expresión posible para la consulta podría ser:

Q2: $PJ_{Empleado.nombre}$((EmpleadoJN $_{Dnum=Dnum}$ SL $_{mgrnum=373}$ Departamento)
DF (SL $_{salario>35000}$ Empleado JN $_{Dnum=Dnum}$ SL$_{mgrnum=373}$ Departamento))

El árbol algebraico correspondiente se muestra en la figura 2.3.a. Comenzamos combinando las hojas correspondientes a las relaciones Empleado y Departamento. Entonces se aplica la propiedad de factorización de la selección sobre salario con respecto al producto natural (al hacer esto, la selección se mueve hacia arriba). Ahora, se pueden combinar los nodos correspondientes a la selección sobre mgrnum y por Ultimo el nodo correspondiente al producto natural; el árbol algebraico lo podemos ver en la figura 2.3.b. Podemos identificar la siguiente subexpresión:

Empleado JN $_{Dnum=Dnum}$ SL $_{mgrnum=373}$ Departamento

Una vez que se han identificado las subexpresiones comunes, podemos usar las siguientes propiedades para transformar un grafo de operadores en un árbol:

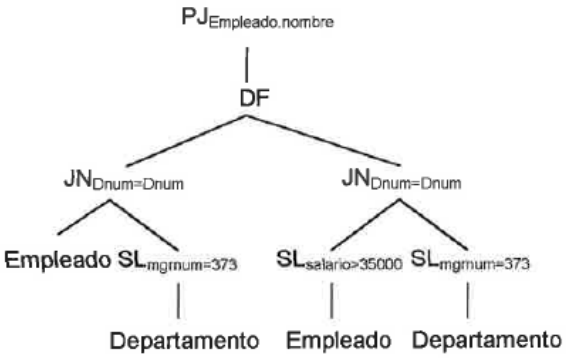
R NJN R ⇔R
R UN R ⇔ R
R DF R ⇔Ø

R NJN $SL_F(R) \Leftrightarrow SL_F(R)$
$SL_F(R)$ UN R ⇔ R
R DF $SL_F(R) \Leftrightarrow$ SL $_{N0T\ F}(R)$

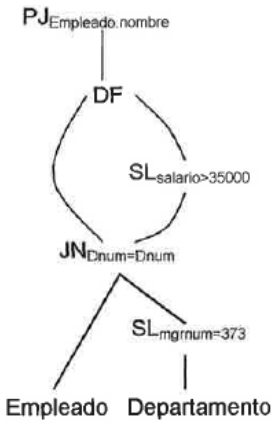
$SL_{F1}(R)$ NJN $SL_{F2}(R) \Leftrightarrow SL_{F1\ AND\ F2}(R)$
$SL_{F1}(R)$ UN $SL_{F2}(R) \Leftrightarrow$ SL $_{F1OR\ F2}$ (R)
$SL_{F1}(R)$ DF $SL_{F2}(R) \Leftrightarrow$ SL $_{F1\ AND\ NOT\ F2}$ (R)

Podemos aplicar la sexta propiedad a nuestro ejemplo, reduciéndose el árbol (figura 2.3.c); por último, podemos aplicar la idempotencia a la proyección y colocar las selecciones y proyecciones cerca de las hojas, obteniendo así el árbol de la figura

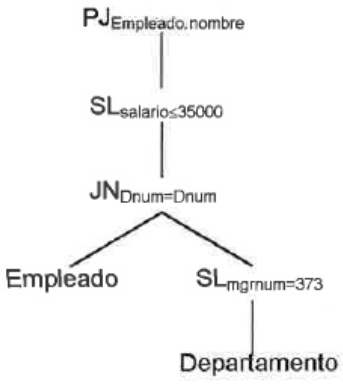
2.3.d. Podemos ver que este nuevo árbol algebraico corresponde a una consulta que podría haber sido dada inicialmente por un programador experto.



(a)



(b)



(c)

(d)

Figura 2.3 Simplificaciones de la consulta Q2

2.3.3 Transformación de consultas globales en consultas fragmentadas

Ahora nos centramos en aspectos propios de bases de datos distribuidas. Anteriormente, hemos indicado que la fragmentación es una característica de las bases de datos distribuidas; a continuación se introduce una transformación estándar que transforma una expresión algebraica sobre el esquema global en una expresión algebraica sobre el esquema de fragmentación.

2.3.3.1 Expresión canónica de una consulta fragmentada

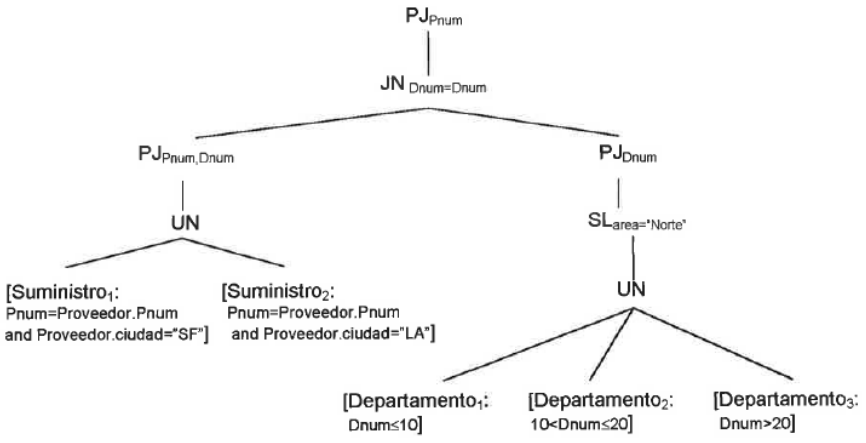
Dada una expresión algebraica sobre el esquema global, la expresión canónica se obtiene sustituyendo cada relación global que aparece como operando por la expresión algebraica que reconstruye esa relación a partir de sus fragmentos. De la misma forma, transformamos un árbol algebraico sobre la relación global en un árbol algebraico sobre el esquema de fragmentación sustituyendo para las hojas del primer árbol las expresiones correspondientes de la inversa del esquema de fragmentación.

Recordar que la inversa del esquema de fragmentación da la regla para reconstruir una relación global a partir de sus fragmentos; la nueva expresión (sobre el esquema de fragmentación) producirá el mismo resultado que la expresión antigua

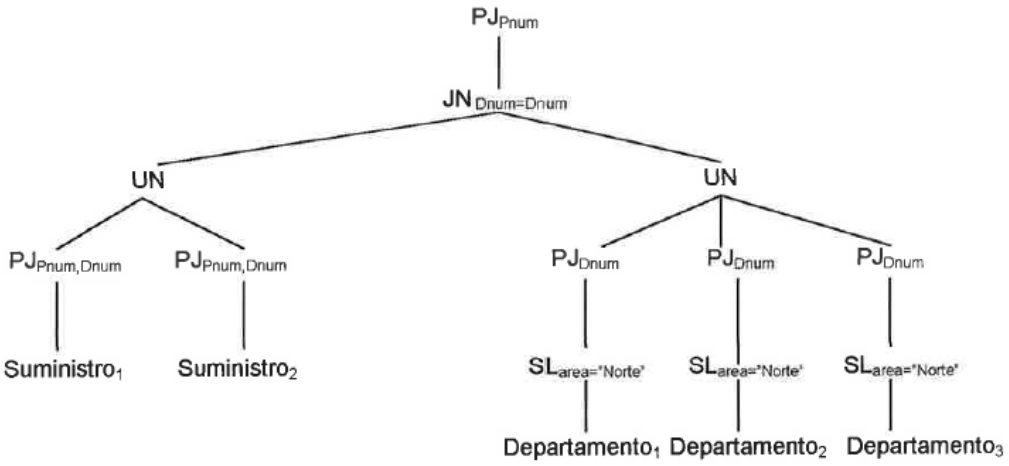
(sobre el esquema global). Un aspecto importante es que las hojas del árbol algebraico de la expresión canónica son ahora fragmentos en lugar de relaciones globales.

La figura 2.4.a muestra la transformación del árbol algebraico de la consulta Q1 de la figura 2.2 en un árbol algebraico de la expresión canónica de Q1.

Una expresión canónica corresponde a una forma conservativa de realizar una consulta, que consiste primero en reunir los fragmentos de todas las relaciones globales en relaciones temporales y entonces aplicarles la consulta global. Esto no es un enfoque eficiente. Así, .pues, la expresión canónica también es la peor expresión para una consulta dada; pero una expresión canónica es una expresión algebraica, y podemos aplicarle cualquier transformación de equivalencia. En concreto, usaremos la distribución de la selección y la proyección con respecto a la unión y al producto natural con objeto de distribuir el procesamiento de los fragmentos; por tanto, los criterios 1 y 2 aumentan su relevancia en el contexto de consultas sobre el esquema de fragmentación. La figura 2.4.b muestra la aplicación de estos dos criterios al árbol algebraico de la figura 2.4.a.



(a) Expresión canónica para la consulta Q1



(b) Bajar las selecciones y proyecciones.

Figura 2.4 Transformaciones sobre el árbol algebraico de la consulta Q1

Anteriormente, cuando hablábamos de fragmentos horizontales, hicimos una introducción a la cualificación, es decir, un predicado que expresa una propiedad de todas las tuplas que pertenecen al fragmento. Ahora hablaremos del use de las cualificaciones para simplificar las consultas distribuidas.

2.3.3.2 Algebra de relaciones cualificadas

Una relación cualificada es una relación extendida con una cualificación; la cualificación se puede ver como una propiedad intencional (propiedad que tiene muy poca probabilidad de cambiar su valor con el tiempo, por ejemplo el nombre de la B.D., los nombres de sus relaciones, de los atributos, etc.) que poseen todas las tuplas de la relación. Se representa con un par [R: q_R], donde R es una relación llamada el *cuerpo* de la relación cualificada y q_R es un predicado que se llama la *cualificación* de la relación cualificada. Los fragmentos horizontales son ejemplos típicos de relaciones cualificadas, en los que la cualificación corresponde al predicado que da lugar a la fragmentación.

Normalmente no se requiere que la cualificación de una relación sea evaluada para las tuplas de la relación; sin embargo, si la cualificación se evalúa, ha de ser verdadera.

El álgebra de relaciones cualificadas es una extensión del algebra relacional que usa como operandos relaciones cualificadas; esta algebra requiere la manipulación tanto de cualificaciones como de relaciones. Las siguientes reglas definen el resultado de aplicar las operaciones del algebra relacional a relaciones cualificadas.

Regla 1 $SL_F\ [R: q_R] \rightarrow [SL_F\ R: F\ AND\ q_R]$

Esta regla manifiesta que la aplicación de una selección SL_F a una relación [R: qR] produce una relación cualificada que tiene la relación SL_F R como cuerpo y el predicado F AND q_R como su cualificación.

Regla 2 $PJ_A\ [R: q_R] \rightarrow [_{PJA}\ R: q_R]$

La cualificación del resultado de una proyección permanece sin cambios, incluso si la proyección elimina algunos de los atributos sobre los que la cualificación se haya evaluado. La razón está en la naturaleza de las cualificaciones; las cualificaciones mantienen información intensional, mas que información extensional. Por tanto, es correcto eliminar atributos usados para expresar la cualificación, sin eliminar la cualificación en sí misma.

Regla 3 $[R: q_R]\ CP\ [S: q_S] \rightarrow [R\ CP\ S: q_R\ AND\ q_S]$

La extensión de la cualificación a q_R AND q_S es más intuitiva.

Regla 4 $[R: q_R]\ DF\ [S: q_S] \rightarrow [R\ DF\ S: q_R]$

La extensión de la diferencia es menos intuitiva; cuando eliminamos las tuplas de una relación S a una relación R, la cualificación del resultado mantiene la misma cualificación de R. No sería correcto cambiarlo por q_R AND NOT q_S, porque podría haber tuplas de R para las que q_R AND NOT q_S no se cumple.

Sin embargo, con la regla 4 se puede producir el siguiente problema: consideremos la operación de intersección del algebra relacional tradicional, definida

como R IN S = R DF (R DF S). Partiendo de la definición, es fácil mostrar que la intersección es conmutativa, es decir,

$$R \text{ IN } S = S \text{ IN } R$$

Si ahora aplicamos la definición del algebra extendida, nos encontraremos con un resultado sorprendente:

$[R: q_R] \text{ IN } [S: q_S] \rightarrow [R: q_R] \text{ DF } ([R: q_R] \text{ DF } [S: q_S]) \rightarrow$
$[R: q_R] \text{ DF } [R \text{ DF } S: q_R] \rightarrow [R \text{ DF } (R \text{ DF } S): q_R]$
$[R \text{ IN } S: q_R]$ (4.a)

$[S: q_S] \text{ IN } [R: q_R] \rightarrow [S: q_S] \text{ DF } ([S: q_S] \text{ DF } [R: q_R])$
$[S: q_S] \text{ DF } [S \text{ DF } R: q_S] \rightarrow [S \text{ DF } (S \text{ DF } R): q_S]$
$[S \text{ IN } R: q_S]$ (4.b)

De hecho, el resultado que nos gustaría encontrar es

$$[R \text{ IN } S: q_R \text{ AND } q_S]$$

porque las tuplas de la intersección son aquellas que pertenecen a ambas relaciones, para las que se cumple tanto q_R como q_S. Observar que q_R AND q_S implica tanto q_R como q_S; de este modo, los resultados (4.a) y (4.b) no son erróneos, pero sí que se pierde alguna información; no se obtiene el predicado más restrictivo que se satisface por todas las tuplas de R IN S. Par tanto, en el álgebra de relaciones cualificadas la intersección no es conmutativa.

Regla 5 $[R: qR] \text{ UN } [S: q_S] \rightarrow [R \text{ UN } S: q_R \text{ OR } q_S]$

La unión se extiende tomando la disyunción de cualificaciones.

Regla 6 $[R: q_R]\ JNF\ [S: q_S] \rightarrow [R\ JNF\ S: q_R\ AND\ q_S\ AND\ F]$

Demostración (recordar que el producto natural deriva del uso de la selección y el producto cartesiano)

$[R: q_R] JN_F\ [S: q_S] \rightarrow$
$SL_F\ ([R: q_R]\ CP\ [S: q_S]) \rightarrow$
$SL_F\ [R\ CP\ S: q_R\ AND\ q_S] \rightarrow$
$[SL_F\ (R\ CP\ S): q_R\ AND\ q_S\ AND\ F] \rightarrow$
$[R\ JN_F\ S: q_R\ AND\ q_S\ AND\ F]$

Regla 7 $[R: q_R]\ SJ_F\ [S: q_S] \rightarrow [R\ SJ_F\ S: q_R\ AND\ q_S\ AND\ F]$

Demostración (recordar que el semi producto natural deriva del uso de la proyección y el producto natural)

$[R: q_R]\ SJ_F\ [S: q_S] \rightarrow$
$PJ_{Attr(R)}\ ([R: q_R]\ JN_F\ [S: q_S]) \rightarrow$
$PJ_{Attr(R)}\ [R\ JN_F\ S: q_R\ AND\ q_S\ AND\ F] \rightarrow$
$[PJ_{Attr(R)}\ (R\ JN_F\ S): q_R\ AND\ q_S\ AND\ F] \rightarrow$
$[R\ SJ_F\ S: q_R\ AND\ q_S\ AND\ F]$

Una vez definida el álgebra de relaciones cualificadas, ahora damos la extensión de las relaciones de equivalencia para ella. Dos relaciones cualificadas son equivalentes si sus cuerpos son relaciones equivalentes y sus cualificaciones representan la misma función de verdad (si aplicamos ambas cualificaciones a la misma tupla, obtenemos el mismo valor de verdad). Todas las transformaciones de equivalencia del algebra relacional son también ciertas en el álgebra de relaciones cualificadas.

Usamos cualificaciones para eliminar fragmentos que no están implicados en una consulta. Cuando una relación cualificada tiene una cualificación contradictoria, esa relación es intrínsecamente vacía, y la correspondiente rama del árbol será ignorada. Consideremos la cualificación que se produce después de una selección o un producto natural; en estos casos, la cualificación incluye una conjunción de predicados.

Un ejemplo de una cualificación contradictoria es DNUM = 1 AND DNUM = 5. Las relaciones cualificadas con cualificaciones contradictorias son intrínsecamente vacías, y la correspondiente rama del árbol será ignorada. Las contradicciones descienden de propiedades intencionales de fragmentos, y se pueden usar durante la compilación de la consulta.

Ejemplos de subexpresiones que reducen a la relación vacía son:

$SL_{ciudad=”LA”}$ [$Proveedor_1$: ciudad = "SF"]

o

[$Departamento_1$: Dnum <= 10] $JN_{Dnum=Dnum}$ [$Empleado_3$: Dnum > 20]

Reconociendo que una de las subexpresiones de una consulta es intrínsecamente vacía podremos realizar simplificaciones sustanciales en el árbol de la consulta; las siguientes transformaciones de equivalencia, en las que R puede ser una relación normal o cualificada, son bastante útiles:

$SL_F(Ø) \Leftrightarrow Ø$

$PJ_A(Ø) \Leftrightarrow Ø$

$R\ CP\ Ø \Leftrightarrow Ø$

$R\ UN\ Ø \Leftrightarrow R$

$R\ DF\ Ø \Leftrightarrow R$

$Ø\ DF\ R \Leftrightarrow Ø$

$R\ JN_F\ Ø \Leftrightarrow Ø$

$R\ SJ_F\ Ø \Leftrightarrow Ø$

$Ø\ SJ_F\ R \Leftrightarrow Ø$

A continuación damos nuevos criterios para simplificar expresiones sobre el esquema de fragmentación; se podrían usar junto con los criterios 1 y 2 ya citados.

Criterio 3. Llevar las selecciones hasta el nivel de las hojas, utilizar el álgebra de relaciones cualificadas y aplicar esas selecciones, a continuación sustituirlas por la relación vacía si la condición de la selección y la cualificación de la relación son contradictorias.

Criterio 4. Aplicar el álgebra de relaciones cualificadas para evaluar la cualificación de los operandos del producto natural, sustituir el subárbol correspondiente incluyendo el producto natural y sus operandos, por la relación vacía, si la cualificación del resultado de ese producto natural es una contradicción.

Estos criterios se usan en las siguientes secciones para simplificar relaciones fragmentadas horizontalmente y para simplificar los productos naturales entre relaciones fragmentadas horizontalmente**.**

2.3.3.3 Simplificaciones de relaciones fragmentadas horizontalmente

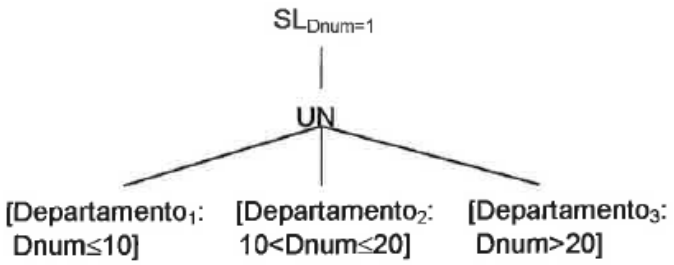
Se muestra este tipo de simplificación con un ejemplo. Consideremos la consulta Q3 sobre la relación Departamento, que está fragmentada horizontalmente:

Q3: SL $_{Dnum=1}$ Departamento

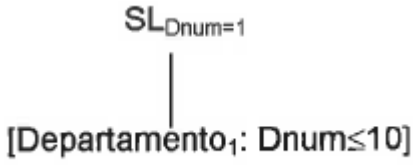
La forma canónica de la consulta se muestra en la figura 2.5.a. Bajamos la selección hacia las hojas distribuyéndolas con respecto a la unión. Entonces, se aplican las selecciones de acuerdo al criterio 3; la cualificación de resultados de las selecciones sobre Departamento$_2$ y Departamento3 es una contradicción:

[Departamento$_1$: Dnum<=10 AND Dnum=1]
[Departamento$_2$: 10<Dnum<=20 AND Dnum=1]
[Departamento3: Dnum>20 AND Dnum=1]

De esta forma, estas dos subexpresiones se substituyen en el árbol por la relación vacía. Por último, simplificamos la operación de unión; el resultado final es el árbol simple de la figura 2.5.b.



(a) Expresión canónica de la consulta Q3



(b) Simplificación de la consulta Q3

Figura 2.5 Simplificación de relaciones fragmentadas horizontalmente

Observar que si las relaciones globales están fragmentadas horizontalmente de forma apropiada, la simplificación anterior se podría aplicar a la mayoría de las consultas que operan sobre ellas.

2.3.3.4 Simplificación de productos naturales entre relaciones fragmentadas horizontalmente

Al hacer una consulta con un producto natural entre dos relaciones R y S, hay dos maneras de hacerlo; la primera requiere traer todos los datos que hacen falta, y teniéndolos todos, realizar el producto natural. La segunda consiste en ir haciendo los productos naturales parciales y luego juntar los datos; a esto último se le suele denominar producto natural distribuido. Ninguna destaca sobre la otra. Se prefiere la

primera si las condiciones sobre los fragmentos son altamente selectivas; se prefiere la segunda si hay pocos fragmentos.

Los grafos de producto natural ayudan al diseño de la fragmentación, pero necesitamos mostrar cómo se pueden construir dichos grafos para consultas especificas; cualquier intento de realizar un buen diseño podría ser inútil si no damos tales reglas de construcción cuando transformamos consultas globales sobre el esquema de fragmentación.

La siguiente transformación de equivalencia permite la transformación de una consulta procesada sin producto natural distribuido a una consulta procesada con producto natural distribuido: los productos naturales que aparecen en la consulta global se deben distribuir con respecto a las uniones. Observar que esto corresponde a subir las uniones en el árbol algebraico, es decir, retrasar la unión de fragmentos. Se resume esta transformación en el criterio 5:

Criterio 5. Con objeto de distribuir los productos naturales que aparecen en una consulta global, las uniones (representan colecciones de fragmentos) deben subirse por encima de los productos naturales.

Construir un grafo de producto natural requiere, pues, aplicar el criterio 5 (para distribuir el producto natural) seguido del criterio 4 (para eliminar los productos naturales entre fragmentos que no contribuyen al resultado). El grafo de producto natural viene dado por todos los pares de fragmentos de R y S que persisten en el árbol algebraico.

Vamos a mostrar un ejemplo de un producto natural distribuido. Supongamos una consulta que requiere el número de todos los proveedores que suministran algún artículo. La expresión algebraica de la consulta es:

$$Q4: PJ_{Pnum} (Suministro\ JN_{Pnum=Pnum}\ Proveedor)$$

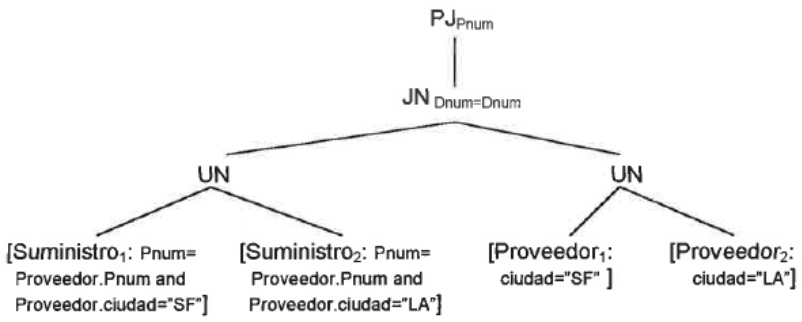
La figura 2.6.a muestra la forma canónica de la consulta. Recordamos que la fragmentación de Suministro deriva de la fragmentación de Proveedor; es decir, cada

tupla de Suministro se almacena en el fragmento $Proveedor_1$, si se refiere a proveedores de San Francisco, o en el fragmento $Proveedor_2$, si se refiere a los proveedores de Los Ángeles.

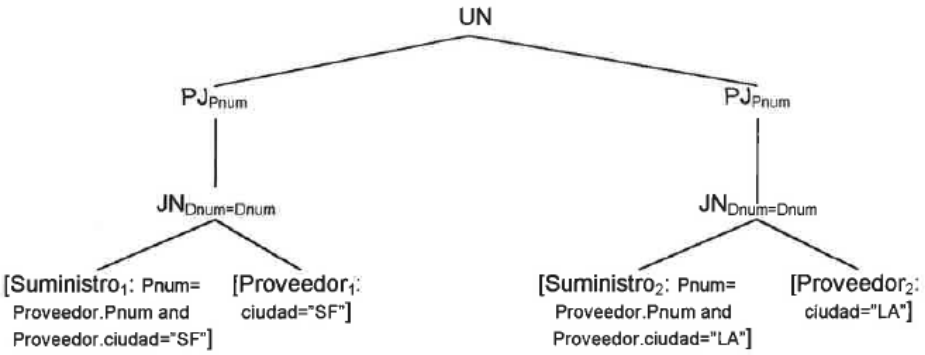
Aplicando el criterio 5, subimos las dos uniones por encima del producto natural; así, se generan cuatro productos naturales entre fragmentos. Después podemos aplicar el criterio 4, y ver que dos de ellos son intrínsecamente vacios porque sus cualificaciones son contradictorias. Los productos naturales vacios son aquellos de $Proveedor_1$, (en "SF") con $Suministro_2$ (de los proveedores de "LA"), y de la misma forma los de $Proveedor_2$ (en "LA") con $Suministro_1$ (de los proveedores de "SF"). De esta forma el árbol se reduce al de la figura 2.6.b. Asumiendo que los fragmentos con el mismo índice están situados en la misma localidad (es decir, que los datos sobre Suministro están almacenados junto con los datos sobre Proveedor), este árbol algebraico corresponde a una forma eficiente de evaluar la consulta, porque cada producto natural es local a cada localidad.

Observar que el producto natural distribuido tiene un grafo de producto natural que está clasificado como "simple"; este grafo es

$Proveedor_1$ ——————— $Suministro_1$
$Proveedor_2$ ——————— $Suministro_2$



(a) Expresión canónica de la consulta Q4.



(b) Producto natural distribuido de la consulta Q4.

Figura 2.6 Simplificación de productos naturales entre relaciones fragmentadas horizontalmente

2.3.3.5 Uso de inferencia para más simplificaciones

Hasta ahora hemos visto que la determinación de contradicciones entre criterios de selección de consultas y cualificaciones de fragmentos es bastante útil. Las contradicciones anteriores han sido fáciles de detectar, descubriendo que los valores requeridos para un atributo o grupo de ellos por una consulta, no son compatibles con respecto a los criterios de fragmentación también expresados sobre el mismo atributo o grupo de atributos. Sin embargo, determinar que una fórmula es una contradicción se puede hacer de forma más sofisticada usando información intencional, y requiere, en general, el use de un demostrador de teoremas.

A continuación mostramos ejemplos de cómo se puede usar información adicional para simplificar consultas. Consideremos de nuevo la consulta Q1 que requiere el número de proveedor (Pnum) de aquellos proveedores que han suministrado artículos en el área Norte, para los que se ha desarrollado el árbol de operados de la figura 2.4. Supongamos que el optimizador de consulta tiene disponible el siguiente conocimiento:

1. El área Norte solo incluye departamentos del 1 al 10.
2. Los pedidos de los departamentos 1 al 10 se dirigen a los proveedores de San Francisco.

Usamos dicho conocimiento para inferir contradicciones que permiten eliminar subexpresiones.

a. Aplicando la primera suposición de antes, podemos escribir las siguientes implicaciones:

Área = "Norte"→NOT (10 < Dnum ≤ 20)

Área = "Norte" → NOT (Dnum > 20)

Usando el criterio 3, podemos aplicar la selección a los fragmentos **Departamento$_1$, Departamento$_2$** y **Departamento$_3$** y evaluar la cualificación de los resultados. Debido a esas implicaciones, dos de ellas son contradictorias. Esto nos permite eliminar las subexpresiones para los fragmentos **Departamento$_2$** y **Departamento$_3$**. De esta forma, el árbol algebraico de la figura 2.4.b se reduce al de la figura 2.7.a.

b. Ahora aplicamos el criterio 5 para la distribución del producto natural; en principio, necesitamos incluir Departamento1 con los subárboles que incluyen Suministro1 y Suministro2. Pero sabemos, usando la suposición 1, que:

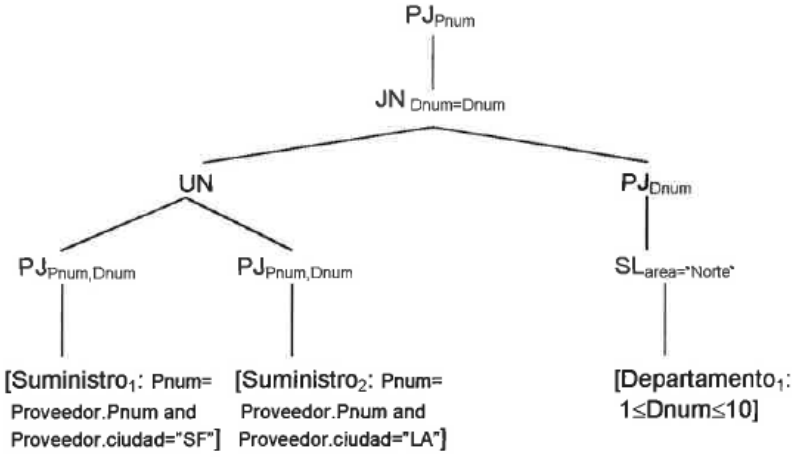
Área = "Norte"→ Dnum≤10

y usando la suposición 2 que:

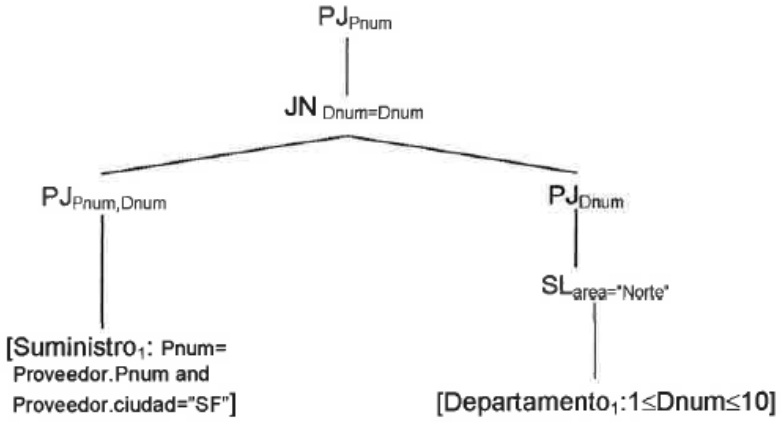
Dnum ≤ 10→

NOT (Pnum = Proveedor.Pnum AND Proveedor.ciudad = "LA")

Aplicando el criterio 4, es posible deducir que el único subárbol necesario para hacer el producto natural es Suministro1. El árbol algebraico final se muestra en la figura 2.7.b.



(a)



(b)

Figura 2.7 Simplificación de un árbol algebraico usando inferencia

2.3.3.6 Simplificación de relaciones fragmentadas verticalmente

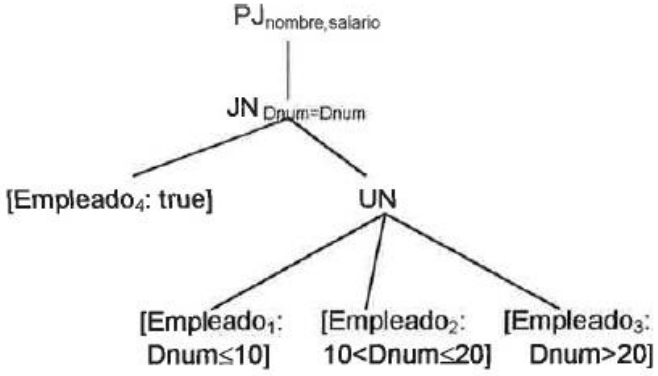
Ahora nos centramos en la simplificación de relaciones fragmentadas verticalmente, que es dual a la simplificación de los fragmentados horizontalmente. Se trata de determinar un subconjunto apropiado de fragmentos que sea suficiente para responder la consulta, y entonces eliminar todos los otros fragmentos de la consulta, así como los productos naturales que sean usados a la inversa del esquema de fragmentación para reconstruir las relaciones globales. En concreto, si los fragmentos que se requieren son uno solo, no hay necesidad de realizar operaciones de producto natural.

Lo ideal es determinar en tiempo de compilación (cuando se forma el árbol) el subconjunto propio de fragmentos verticales que son necesarios y suficientes para realizar la consulta. Para determinar ese subconjunto, se van comparando los atributos de la consulta con los que hay en cada uno de los fragmentos verticales. Si ese conjunto es subconjunto propio de alguno de los fragmentos, el resto de fragmentos se ignoran. Si hay más de un fragmento en que se cumpla, habría que esperar a tiempo de ejecución para determinar a qué fragmentos se acceden.

Mostramos la simplificación con un ejemplo. Consideremos la consulta Q5, que requiere los nombres y salarios de todos los empleados. La consulta sobre el esquema global es simplemente:

Q5: $PJ_{nombre,salario}$ Empleado

El árbol algebraico de la expresión se muestra en la figura 2.8.a. Recordemos que Empleado tiene una fragmentación mixta: primeramente está fragmentada verticalmente en un fragmento $Empleado_4$ (que describe la gestión de los salarios de los empleados) y un segundo fragmento, que está fragmentado horizontalmente en $Empleado_1$, $Empleado_2$, $Empleado_3$. Observamos que los atributos de $Empleado_4$ incluyen nombre y salario, que la consulta requiere. Entonces es posible responder la consulta usando solo el fragmento $Empleado_4$, y el árbol algebraico se puede simplificar despreocupándonos de los otros fragmentos y del producto natural. El árbol resultante para la consulta Q5 se muestra en la figura 2.8.b.



(a) Expresión canónica de Q5

(b) Consulta simplificada

Figura 2.8 Simplificación de relaciones fragmentadas verticalmente

Observar que si se hace de forma apropiada el diseño de la fragmentación vertical, la simplificación anterior se podría realizar para la mayoría de las consultas que se realicen.

2.3.6 Conclusiones

Antes de ejecutar una consulta que opera sobre relaciones globales es necesario transformarla en una consulta que opera sobre fragmentos. La forma más sencilla de realizar esta transformación es substituir la relación global que aparece en la consulta global con la inversa de sus expresiones de fragmentación. La consulta resultante, que opera sobre fragmentos, se llama expresión canónica de la consulta global.

La expresión canónica se podría ejecutar directamente en la base de datos distribuida; sin embargo, en general, esta expresión representa una forma muy poco eficiente de ejecutar una consulta. Es, pues, conveniente modificar la expresión canónica aplicándole transformaciones de equivalencia. Varios criterios heurísticos indican las transformaciones que se podrían aplicar con objeto de obtener expresiones más simples.

También es posible aprovecharse de la información disponible sobre la fragmentación con objeto de eliminar de la expresión de la consulta aquellas expresiones que no tengan ninguna contribución al resultado final. Los conceptos de relaciones cualificadas y de algebra de relaciones cualificadas se han desarrollado para este propósito.

CAPÍTULO 3: ANÁLISIS

3.1 Planificación

La planificación es un conjunto de toma de decisiones mediante las cuales se pretende lograr una serie de objetivos, teniendo en cuenta el entorno que nos rodea así como factores internos o externos que pueden afectar al proyecto (Jiménez, 1982).

Para la planificación, es necesario apreciar una serie de factores que se muestran divididos en tres categorías:

- Recursos humanos
- Planificación temporal
- Estimación de costes

3.1.1 Recursos humanos

Se considera necesario analizar qué recursos humanos se precisan en el desarrollo de un proyecto.

En este caso, será oportuno la participación de los siguientes entes:

Jefe de proyecto:

Persona encargada de coordinar al equipo y garantizar que las tareas se están desarrollando en función de los objetivos previstos. Es el encargado de elaborar la memoria final

Analista:

Trabajará en la fase de análisis, cuya principal función será establecer los requisitos técnicos necesarios en el proyecto para su posterior comunicación al Diseñador

Diseñador:

Rol responsable del diseño de la apariencia de la aplicación, fundamentándose en los datos facilitados anteriormente por el analista.

Programador:

Encargado de implementar el código de la aplicación para su correcto funcionamiento.

Tester:

Este perfil comprobará que la aplicación funciona correctamente, ejecutando pruebas en búsqueda de fallos. Se intentará que sea una persona que tenga características comunes con el usuario final del producto, al cual se le trasmitirá nociones del funcionamiento de la aplicación, para que se pueda proceder a las pruebas finales.

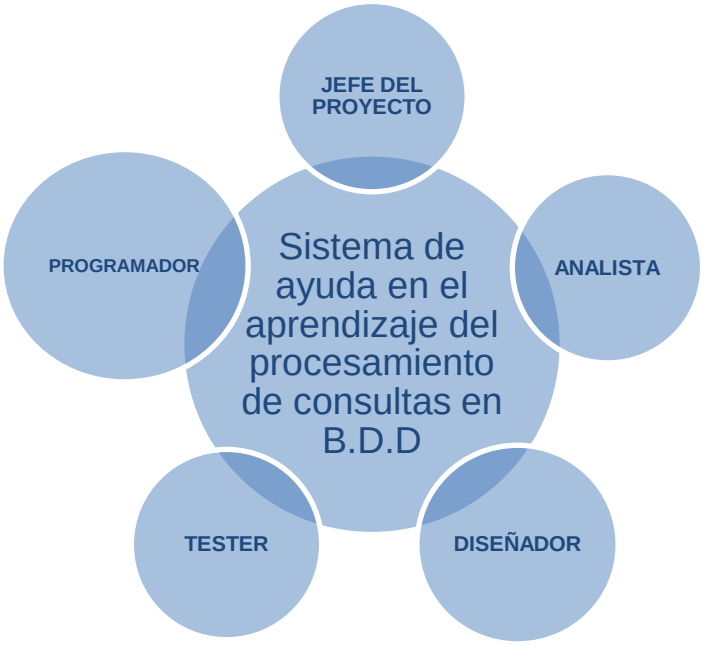


Figura 3.1 Distribución de roles

3.1.2 Planificación temporal

Es preciso, en el comienzo de una actividad, definir cuáles son los tiempos a emplear en cada una de las tareas, para poder llegar así a la consecución de la misma.

En el caso que nos ocupa, se ha procedido de la siguiente forma:

Tarea	Estimación temporal
1. Búsqueda de información	2 semanas
2. Planificación	1 semana
Recursos humanos	
Planificación temporal	
Estimación de costes	
3. Análisis del proyecto	1 semana
4. Diseño del proyecto	2 semanas
Diseño de la interfaz web	
5. Implementación de la aplicación	5 semanas
Obtención del árbol algebraico	
Simplicaciones de selecciones y proyecciones	
Eliminación de subexpresiones comunes	
Bajada de las selecciones	
Fragmentación	
Distribución de productos naturales	
Eliminación de productos naturales inútiles	
Obtener la consulta SQL a partir del árbol de operador	
Aplicación del álgebra de relaciones cualificadas	
6. Tester	1 semana

7. Elaboración de la documentación		2 semanas
	Redacción y revisión de la memoria	
	Manual de usuario	
	Manual de instalación	
TOTAL		14 semanas

Tabla 3.1 Planificación

Gantt

El diagrama de Gantt es una herramienta que hace posible programar las distintas tareas para la realización de un proyecto. Así, se ha estimado conveniente la creación de un diagrama de este tipo para poder visualmente analizar nuestra planificación temporal.

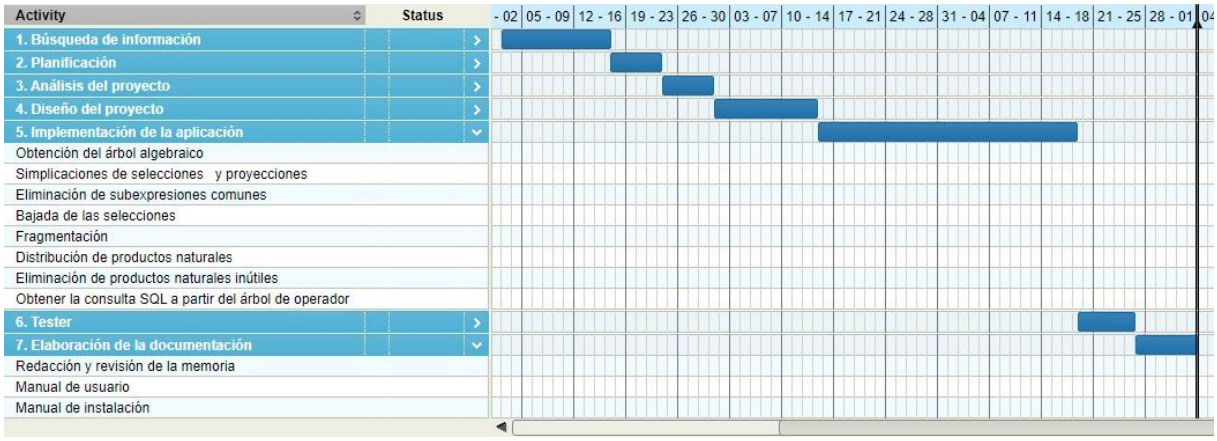


Figura 3.2 Diagrama de Gantt

3.1.3 Estimación de costes

En este apartado vamos a definir los distintos costes que forman parte de este proyecto, teniendo en cuenta los recursos humanos y el hardware y software necesario para el desarrollo del mismo.

DESCRIPCIÓN	COSTE UNITARIO	UNIDAD/ HORAS TOTALES	COSTE TOTAL

1. Hardware			680 €
Ordenador portátil ASUS Intel (R) Core (TM) i7-4510U, 8 GB	650 €	1	650 €
Router	30 €	1	30
2. Software			190 €
Licencia Microsoft Office Profesional 2016	190 €	1	190 €
Paint	0,00 €	1	0,00 €
3. Costes generales			135 €
Internet	45 €/mes	3	135 €
4. Gastos de personal			8930 €
Jefe del proyecto	52 €/hora	22 horas	1 144 €
Diseñador	27 €/hora	75 horas	2 025 €
Programador	20 €/hora	175 horas	3 500 €
Analista	27 €/hora	63 horas	1 701 €
Tester	14 €/hora	40 horas	560 €
COSTE TOTAL			9 935 €

Tabla 3.2 Estimación de costes

3.2 Descripción de la información

Esta sección proporciona una descripción detallada del problema. Pasamos a describir el flujo y la estructura de la información.

El software a construir se encarga de procesar datos, es decir transformar datos de una forma a otra; de esta forma, se acepta una entrada (consulta global), se manipula de manera que se optimice esa consulta y se produce una salida o respuesta del sistema, que sería la subconsulta distribuida que intenta acceder a los fragmentos.

El flujo de información representa la manera en la que los datos y el control cambian conforme el sistema cambia de estado. En los siguientes apartados se analizan estos aspectos.

3.2.1 Flujo de datos

Una vez que hemos analizado las partes fundamentales del sistema, la información puede representarse como un flujo continuo que sufre una serie de transformaciones conforme va de la entrada a la salida. El diagrama de flujo de datos (DFD) se utiliza como herramienta grafica para la descripción del flujo de la información.

En principio, realizamos una revisión del modelo fundamental del sistema, que engloba el DFD de nivel 0 y la información complementaria. El diagrama que obtenemos se llama diagrama de contexto y es el siguiente:

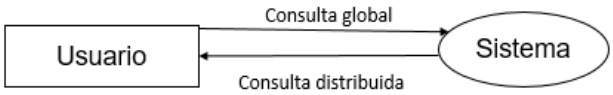


Figura 3.3. DFD 0

Recordemos que la consulta global, consiste en una consulta SQL, que proporcionamos al sistema, que transforma interiormente como veremos a continuación con más detalles, y luego devuelve la consulta distribuida (consulta que accede a los fragmentos) también expresada en SQL.

A continuación realizamos una revisión y un refinamiento del diagrama anterior, obteniendo el siguiente diagrama de nivel 1:

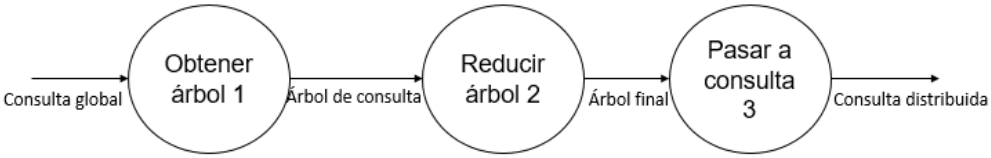


Figura 3.4. DFD 0.1

Podemos seguir profundizando en cuanto a *Reducir árbol* y obtener el siguiente diagrama (DFD 0.1.2).

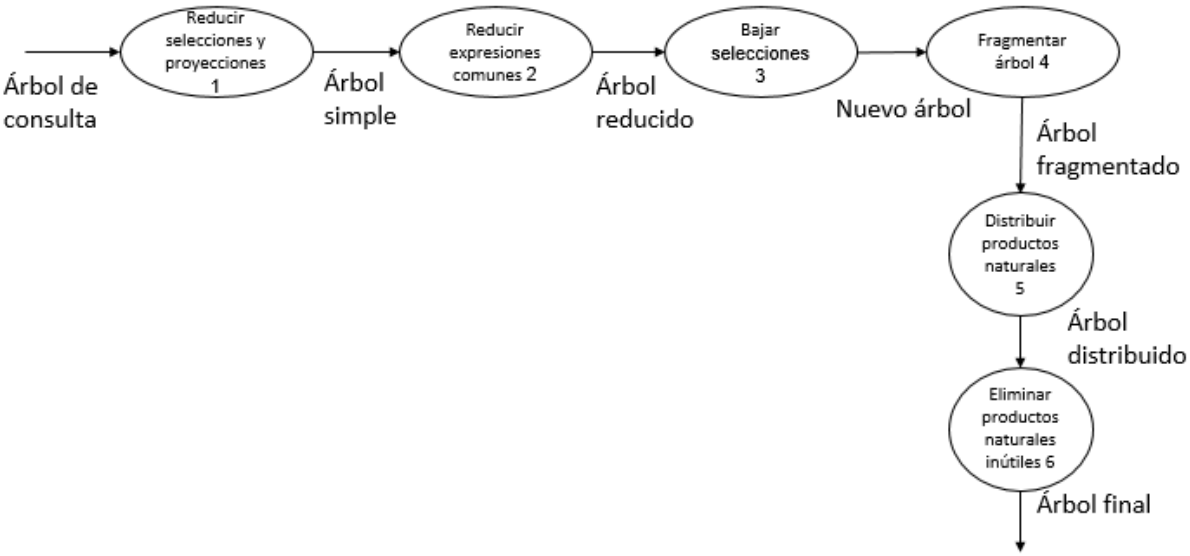


Figura 3.5. DFD 0.1.2

3.2.2 Flujo de control

Como esta es una aplicación que esta conducida por el uso de datos, no por el uso de sucesos, entonces no es necesario crear el Flujo de Control ya que prácticamente no informa de nada nuevo sobre las características del Sistema. El uso de los DFD's que hemos realizado son suficientes como para obtener una idea modelizada del funcionamiento del sistema que queremos informatizar.

3.3 Descripción del comportamiento

En esta sección se examina el funcionamiento del software como consecuencia de los sucesos externos y de las características de control generadas internamente.

Dado que nuestro sistema se basa en una aplicación software, entonces los estados del sistema serán los estados por los que deba de pasar la aplicación.

Según lo anterior, podemos comprobar que son pocos los estados en los que se puede encontrar la aplicación. Concretamente son los siguientes:

- Al ejecutar la aplicación, se entrar en el estado inicial en el que la aplicación no hace nada; simplemente espera a que el usuario introduzca una consulta global. Se mantiene en este estado hasta que el usuario introduzca la consulta.

- El siguiente estado al que se pasa es un estado en el que el sistema está procesando la consulta introducida por el usuario. Dentro de este estado, se realizaran una serie de operaciones (que ya se han comentado anteriormente) sobre los datos introducidos por el usuario para optimizar la consulta, por lo que el sistema irá pasando por una serie de estados secuenciales conforme se vaya procesando la consulta. Estas operaciones son: obtener el árbol para una consulta dada, reducir las selecciones y las proyecciones en un nodo, reducir expresiones comunes, bajar las selecciones en el árbol, fragmentar el árbol, distribuir los productos naturales, eliminar los productos naturales que no nos sean útiles y pasar del árbol final obtenido a la consulta distribuida. En la aplicación tendremos la posibilidad de ir viendo paso a paso estas transformaciones por lo que el sistema se encontrará en un estado u otro según el proceso que se esté llevando a cabo.

- Una vez optimizada la consulta el sistema pasa a un estado en el que lo que hace es devolver el resultado, es decir, la consulta distribuida que accede a los fragmentos y los datos obtenidos por dicha consulta en función de la consulta global introducida por el usuario. Se mantendrá en este estado hasta que el usuario confirme que ha observado los resultados y está dispuesto a salir del sistema o bien a introducir una nueva consulta.

CAPÍTULO 4: DISEÑO

4.1 Diseño de la arquitectura

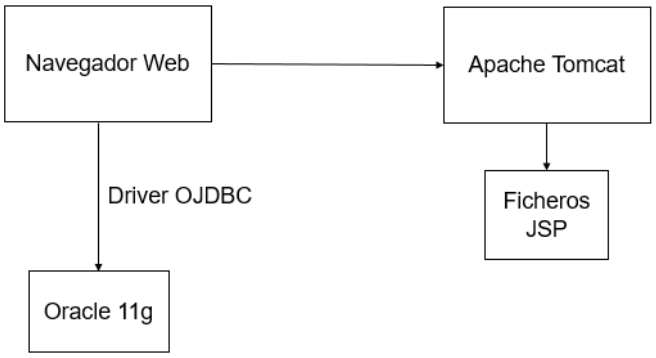


Figura 4.1. Diseño de la arquitectura.

En primer lugar se muestra un gráfico (figura 4.1) que ilustra cómo es la arquitectura que se ha usado para la realización del sistema, y después se explican los puntos más relevantes para comprender el diseño.

Este grafico sirve para ilustrar el diseño de la arquitectura que he seguido para la implementación de la aplicación.

Los elementos software que he usado para la realización del sistema de información son los siguientes (ver figura 4.1):

- Navegador web.
- Servidor de páginas web java (Apache Tomcat)
- Sistema gestor de base de datos Oracle.

1.- Navegador Web: Va a ser la interfaz gráfica que he utilizado para el sistema. Se encarga de mostrar el contenido de las páginas JSP. El navegador que he usado es Mozilla Firefox versión 55.03 aunque sirve cualquier navegador Web que permita ejecutar código java.

2.- Servidor de Páginas Web: En mi caso he utilizado el servidor Apache Tomcat. El servidor se compone de un conjunto de servicios encargados de detectar

peticiones de navegadores Web y de enviar la información (páginas Web) que se le soliciten.

3.- Sistema gestor de base de datos: Como gestor de base de datos se ha usado Oracle 11g. Está compuesto por los datos necesarios para poder relacionar los distintos fragmentos entre sí, junto con meta datos esenciales para que el sistema funcione correctamente. Más adelante se explicará con más detalle cómo se ha estructurado la base de datos. La conexión entre Oracle y java se ha usado mediante el driver OJDBC.

4.2 Diseño de la Metabase de datos

4.2.1 Introducción

Normalmente la administración de una base de datos alude a una gran variedad de actividades para el desarrollo, control, mantenimiento y verificación del software de las aplicaciones para la base de datos. La administración de una base de datos no es solo un problema técnico, debido a que envuelve políticas bajo las que los usuarios acceden a la base de datos, por lo que es también un problema de organización.

Se considera importante dos aspectos técnicos de la administración de una base de datos:

1. El contenido y gestión de los catálogos: información que requiere el sistema
para acceder a la base de datos. En los sistemas distribuidos, los catálogos incluyen la descripción de la fragmentación y asignación de los datos y la traducción a nombre locales; los catálogos llegan a ser en sí mismos una base de datos distribuida, que se debe distribuir y gestionar de una manera eficiente.

2. La extensión de los mecanismos de protección y autorización a sistemas distribuidos.

La característica más importante en la administración de una base de datos es el grado de autonomía local dada en cada localidad. Hay dos soluciones extremas, la ausencia de cualquier tipo de autonomía local y una autonomía local completa.

En el primer caso, las funciones de un administrador global de una base de datos no son diferentes de las que realiza el administrador de una base de datos centralizada; sin embargo, realizar estas funciones es mucho más duro debido a la distribución del sistema. Un sistema que no tenga autonomía local puede ser bastante diferente en el 'grado de distribución' de los algoritmos que implementan las funciones de administración.

En el segundo caso, las funciones de un administrador global de la base de datos están muy limitadas, debido a que cada localidad se administra de forma independiente.

4.2.2 Gestión del catálogo en bases de datos distribuidas

Los catálogos de las bases de datos distribuidas almacenan toda la información que es útil para el sistema para un acceso correcto y eficiente a los datos, y para verificar que los usuarios hagan un acceso adecuado a ellos. Los catálogos se usan para:

1. Aplicaciones de interpretación. Los datos referenciados por las aplicaciones a distintos niveles de transparencia, son traducidos a datos físicos.
2. Aplicaciones de optimización. Para obtener un plan de acceso se requiere la asignación de los datos, los métodos de acceso disponibles en cada localidad, e información estadística (registrada en los catálogos).
3. Aplicaciones de ejecución. La información del catálogo se usa para verificar que los usuarios realizan los accesos de forma adecuada.

4.2.2.1 Contenido de los catálogos

Son posibles varias clasificaciones de la información que normalmente se almacena en los catálogos de una base de datos distribuida. Siguiendo la arquitectura que hemos mencionado anteriormente, obtenemos:

1. Descripción del esquema global. Incluye el nombre de las relaciones globales y de los atributos.
2. Descripción de la fragmentación. En fragmentación horizontal, se incluye la cualificación de los fragmentos; en fragmentación vertical, se incluyen los atributos que pertenecen a cada fragmento; en fragmentación mixta, se incluyen tanto el árbol de fragmentación como la descripción de la fragmentación correspondiente a cada nodo del árbol que no sea hoja.
3. Descripción de la asignación. Nos da la traducción entre fragmentos y las imágenes físicas.
4. Información de consistencia (protección y restricciones de integridad). Incluye información sobre las autorizaciones de acceso, o restricciones de integridad sobre los valores permitidos para los datos. A la hora de crear las tablas y catálogos están se han seguido unas restricciones que más adelante se expondrán.

4.2.2.2 Distribución de los catálogos

Los catálogos constituyen una base de datos distribuida, cuyos fragmentos y asignación deben ser cuidadosamente diseñados siguiendo las pautas indicadas en el apartado 2.

La asignación y gestión de catálogos están íntimamente relacionadas con el grado de autonomía local existente en cada localidad. De hecho, una de las características distintivas de la autonomía local es que cada localidad debe ser capaz de gestionar sus propios datos sin tener en cuenta

otras localidades. Consideremos, por ejemplo, la creación de un nuevo nombre de un objeto. Para preservar la autonomía local, el mecanismo de asignación de nombres debe asegurar que el nuevo nombre será único dentro del sistema distribuido sin acceder a todos los catálogos, y no se necesita que la información del catálogo sobre el nuevo nombre este inmediatamente disponible en todas las localidades.

En una base de datos distribuida los catálogos se pueden asignar de diferentes formas, pero hay tres alternativas:

Catálogos centralizados. El catálogo completo se almacena en una localidad; esta solución tiene limitaciones obvias, tales como la perdida de localidad de aplicaciones que no estén en la localidad central y la perdida de disponibilidad del sistema.

Catálogos completamente replicados. Los catálogos están replicados en cada localidad; esta solución implica la lectura del catálogo local a cada localidad, pero aumente la complejidad de modificación de los catálogos, debido a que se requiere la actualización de los catálogos de todas las localidades.

Catálogos locales. Los catálogos son fragmentados y asignados de forma tal que se almacenan en la misma localidad que los datos que referencian.

Son posibles varias alternativas intermedias; por ejemplo, es posible tener catálogos centralizados en una localidad y catálogos locales en el resto de localidades. Esto es lo habitual en sistemas de bases de datos distribuidas con una localidad central y una red en estrella que conecta la localidad central con las localidades remotas.

Además, es posible distribuir las distintas partes del catálogo de diferentes formas; por ejemplo, los elementos del 1 al 3 podrían estar

completamente replicados, mientras que el elemento 4 podría estar almacenado localmente.

4.2.3 Diseño

Teniendo en cuenta las consideraciones desarrolladas en los puntos anteriores, procedemos al diseño de la metabase de datos que vamos a usar en nuestro proyecto.

Primeramente vamos a mostrar cual es la información que necesitamos para obtener un catálogo adecuado a nuestro sistema. Sería lo siguiente:

- Información sobre las **tablas** de las que consta la base de datos distribuida:

- A cada tabla se le dará un identificador para poder distinguirla de las demás, por lo que será único para cada tabla (id_tabla).
- También se almacenará el nombre de la tabla (nombre_tabla).

- Información referente a los **atributos** que tenga cada una de las tablas que componen la base de datos distribuida. Necesitaremos:

- Identificador del atributo, único para cada atributo, para poder distinguirlos (id_atributo).
- Identificador de la tabla a la que pertenece el atributo en cuestión (id_tabla).
- Nombre del atributo (nombre_atributo).
- Tipo de atributo, es decir, si se trata de un entero, una cadena de caracteres, si es de tipo fecha, etc. (tipo_atributo).

- Información sobre los **fragmentos** en los que se encuentran divididas las tablas en función del tipo de fragmentación que se haga en el diseño. Necesitaremos la siguiente información:

- Identificador del fragmento, para distinguirlo de los demás id_fragmento).
- Nombre del fragmento (nombre fragmento).
- Identificador de la tabla a la que pertenece el fragmento (id_tabla).
- Información sobre el tipo de fragmento, es decir, si se trata de un fragmento horizontal, vertical, o se trata de una fragmentación mixta (tipo_fragmento).
- Si se trata de un fragmento derivado de otro, también habrá que indicar cual ese fragmento del que deriva (fragmento_deriva).
- Condición que define el fragmento, es decir, la cualificación del fragmento (condición).

- También necesitamos saber, en caso de que se trate de fragmentación vertical, los atributos que pertenecen a cada uno de los fragmentos. Se indicará:

- Identificador del fragmento id_fragmento).
- Identificador del atributo (id_atributo).

- Se ha utilizado una tabla auxiliar para indicar que tablas están relacionadas y de esta manera acceder en la aplicación de manera rápida cuando se necesite. Por ejemplo, la tabla departamento está relacionada con empleado y suministro con proveedor.

- Identificador de una de las tablas relacionadas (id_tabla_o)
- Identificador de la otra relacionada tabla (id_tabla_d)

Entre paréntesis se ha indicado el nombre que se ha dado a cada uno de los atributos al crear las tablas.

Teniendo en cuenta toda esa información que necesitamos tener almacenada, proponemos el diagrama E/R (entidad/relación) de la figura 4.2, que posteriormente pasaremos a tablas.

Analizando el diagrama E/R podemos ver que el conjunto de tablas que necesitamos es el siguiente:

tabla (id_tabla, nombre)
atributo (id_atributo, id_tabla, nombre_atributo, tipo)
fragmento (id_fragmento, nombre_fragmento, id_tabla, tipo, fragmento_deriva, condición)
pertenece (id_fragmento, id_atributo)
relacionado(id_tabla_o, id_tabla_d)

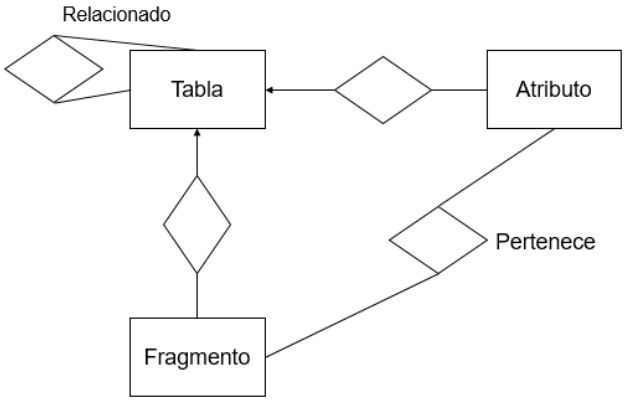


Figura 4.2 Diagrama E/R

Una vez que hemos visto el diagrama E/R y su paso a tablas, nos queda analizar las restricciones a que se tienen que ver sometidos los datos que se almacenen en estas tablas. Son las siguientes:

- Integridad referencial en fragmentos, es decir, que los fragmentos deben referenciar a tablas ya existentes.
- Al insertar un nuevo fragmento, habrá que tener en cuenta que si el fragmento es derivado (el campo tipo_fragmento vale ‘d’), el campo fragmento_deriva no puede tener un valor nub porque debe especificar un fragmento correcto. Al

insertar un fragmento de tipo 'vertical' u 'horizontal', comprobar que no se está indicando que deriva de algún fragmento, es decir, en caso de que no sea derivado (tipo ‘d’), el campo fragmento_deriva debe tener un valor nulo.

- Integridad referencial en atributos, es decir, que los atributos deben referenciar a tablas ya existentes.
- Integridad referencial en la tabla pertenece respecto a las tablas atributo y fragmento, es decir que las tuplas de pertenecer han de hacer referencia a atributos y fragmentos que hayan sido previamente insertados. Además el fragmento que se está introduciendo debe ser de tipo vertical (tipo fragmento='v').
- Los tipos de fragmentos validos son solo tres:
  - 'v: vertical.
  - 'h': horizontal.
  - 'd': derivado.

- Cuando se borre una tabla, se han de borrar los fragmentos que derivan de ella y cuando se borre un fragmento, que también se borren los fragmentos que derivan de él.
- Cuando se borre una tabla, se borran también los atributos de dicha tabla.
- Si se borra un atributo, se han de borrar sus tuplas correspondientes en la tabla pertenece.
- Si se borra un fragmento vertical, se han de borrar sus tuplas correspondientes en la tabla pertenece.

Una última consideración es comentar que para mantener la información referente a los tipos de atributo, hemos usado un solo carácter, lo que quiere decir que esta información va a estar codificada. En efecto, en ese carácter se ha de indicar un número que en realidad codifica el tipo de atributo. Las posibles alternativas, que se han considerado, son las siguientes:

- '1': se usa para indicar que el atributo es de tipo carácter.
- '2': indica que el atributo es de tipo numérico.
- ‘3': usado para indicar el caso en que el atributo sea de tipo fecha.

En principio, solo se ha considerado esas alternativas, pues esos son los tres tipos de datos que habitualmente se van a usar en las aplicaciones con las que nos encontramos, pero es una codificación que permite que se pueda ampliar a otros tipos de una forma sencilla.

4.2.3.1 Definición de las tablas

Para concluir este apartado de diseño de la metabase de datos, vamos a dar un listado de las diferentes tablas del sistema para que se vea como debe de realizarse su implementación. Se ha usado Oracle para la realización de dicha implementación.

- Tabla tabla:

```
create table tabla (
id_tabla int PRIMARY KEY,
nombre varchar(20) NOT NULL UNIQUE
);
```

- Tabla atributo:

```
create table atributo (
id_atributo int PRIMARY KEY,
id_tabla int REFERENCES tabla(id_tabla) ON DELETE CASCADE,
nombre varchar(20) NOT NULL,
tipo varchar(20)
);
```

- Tabla fragmento:

```
create table fragmento (
id_fragmento int PRIMARY KEY,
id_tabla int REFERENCES tabla(id_tabla) ON DELETE CASCADE,
nombre varchar(20) NOT NULL UNIQUE,
tipo char(1) check (tipo IN ('h','v','d')),
fragmento_deriva varchar(20),
```

```
condicion varchar(256)
);
```

- Tabla pertenece:

```
create table pertenece (
id_fragmento  int REFERENCES fragmento(id_fragmento) ON DELETE
CASCADE,
id_atributo int REFERENCES atributo(id_atributo) ON DELETE
CASCADE,
primary key(id_fragmento,id_atributo)
);
```

- Tabla relacionado:

```
create table relacionado (
id_tabla_o int references tabla(id_tabla),
id_tabla_d int references tabla(id_tabla)
);
```

4.3 Diseño de la interfaz

Como se ha comentado anteriormente la interfaz será gráfica y se usará una página web para mostrar la distinta información a los usuarios. La información a mostrar va desde la pantalla principal de la aplicación donde el usuario escribe y lanza la consulta hasta que muestra el resultado de la misma en su forma distribuida.

El hecho de partir de un buen diseño de la interfaz de usuario facilita el buen uso posterior de la aplicación. El vocabulario utilizado deberá ser adecuado y consistente en cuanto a los mensajes que se muestren por pantalla, así como en la ayuda del mismo.

4.4 Diseño detallado

En este apartado, procedemos a analizar con más detalle las funciones que el sistema ha de realizar. Como ya hemos visto anteriormente, el usuario introduce una

consulta global que hay que transformar en una consulta distribuida. También hemos visto que este proceso de transformación se compone de una serie de pasos, que son: obtener el árbol para una consulta dada, reducir las selecciones y las proyecciones en un nodo, reducir expresiones comunes, bajar las selecciones en el árbol, fragmentar el árbol, distribuir los productos naturales, eliminar los productos naturales que no nos sean útiles y pasar del árbol final obtenido a la consulta distribuida.

A continuación comentamos los aspectos fundamentales del diseño de estos pasos.

4.4.1 Obtención del árbol algebraico

El árbol algebraico de una consulta es fundamental. Es el punto de partida del proceso de transformación de una consulta global en consultas fragmentadas, y por tanto, centraremos nuestro interés en su obtención.

El árbol algebraico lo obtendremos a partir de la consulta SQL que se esté considerando. Dado que la sintaxis de una consulta SQL puede ser muy variada, es necesario la construcción de un parser encargado de determinar si la consulta es correcta, y si esta lo es, obtendrá el árbol algebraico de la consulta, junto a la consulta en algebra relacional correspondiente.

Para la construcción del parser es necesaria la construcción de un analizador léxico encargado de la obtención de los tokens, y de un analizador sintáctico que compruebe que la sintaxis de la consulta SQL es correcta y que internamente construya el árbol algebraico mediante la ejecución de acciones semánticas adecuadas conforme se aplican las reglas que constituyen la gramática de una consulta SQL.

El analizador léxico se construirá haciendo uso de una herramienta llamada Jlex. Dada una especificación de un analizador léxico, esta herramienta esta herramienta es capaz de implementar el analizador léxico en Java.

El analizador sintáctico se construirá haciendo uso de una herramienta llamada Java Cup. Dado un analizador léxico que proporcione los tokens de la gramática y una especificación de una gramática, esta herramienta es capaz de implementar el analizador sintáctico en Java y de introducir acciones semánticas en las reglas que forman la gramática.

Los tokens que reconoce el analizador léxico son los siguientes: OPCOMPARACION, WITH, SELECT, WHERE, FROM, ALL, DISTINCT, NOT, START,MINUS,UNION,COMA, PARENIZDA, PARENDCHA, IDENTIFICADOR, CADENA,ASTERISCO,PUNTO,PUNTOYCOMA,AND, ENTERO, REAL.

La gramática (donde las palabras en mayúscula son tokens y las palabras en minúscula son símbolos no terminales (reglas o producciones)) que reconoce las consultas SQL es la siguiente:

```
sentencias ::= sentencia_select
                    |
                    sentencia_select UNION sentencia_select
                    |
                    sentencia_select MINUS sentencia_select
                    ;

sentencia_select ::= SELECT columnas FROM tablas wher{: System.out.println("Sentencia analizada correctamente. Sin errores."); :}
                    | error {: throw new Exception("Error sintactico en la sentencia."); :}
                    ;

columnas ::= ASTERISCO
                    |
                    columnas1
                    ;

columnas1 ::= columna
```

```
                    |
                    columna COMA columnas1
                    ;

columna ::= IDENTIFICADOR
                    | IDENTIFICADOR PUNTO IDENTIFICADOR
                    ;
tablas ::= tabla
                    |
                    tabla COMA tablas
                    ;

tabla ::= IDENTIFICADOR
                    |
                    IDENTIFICADOR IDENTIFICADOR
                    ;

wher ::=
                    |
                    WHERE condiciones
                    ;

condiciones ::= PARENIZDA condiciones PARENDCHA
                    |
                    condicion
                    |
                    NOT condicion
                    |
                    condicion AND condiciones
                    |
                    NOT condicion AND condiciones
                    ;

condicion ::= IDENTIFICADOR OPCOMPARACION IDENTIFICADOR
```

|

IDENTIFICADOR OPCOMPARACION ENTERO

|

IDENTIFICADOR OPCOMPARACION REAL

|

IDENTIFICADOR OPCOMPARACION CADENA

|

ENTERO OPCOMPARACION IDENTIFICADOR

|

REAL OPCOMPARACION IDENTIFICADOR

|

CADENA OPCOMPARACION IDENTIFICADOR

|

IDENTIFICADOR PUNTO IDENTIFICADOR OPCOMPARACION IDENTIFICADOR PUNTO IDENTIFICADOR

|

IDENTIFICADOR PUNTO IDENTIFICADOR OPCOMPARACION ENTERO

|

IDENTIFICADOR PUNTO IDENTIFICADOR OPCOMPARACION REAL

|

IDENTIFICADOR PUNTO IDENTIFICADOR OPCOMPARACION CADENA

|

ENTERO OPCOMPARACION IDENTIFICADOR PUNTO IDENTIFICADOR

|

REAL OPCOMPARACION IDENTIFICADOR PUNTO IDENTIFICADOR

|

CADENA OPCOMPARACION IDENTIFICADOR PUNTO IDENTIFICADOR

4.4.2 Simplificación de selecciones y proyecciones

Una vez que obtenemos el árbol algebraico de la consulta SQL el siguiente paso debe ser eliminar las subexpresiones comunes del árbol. Sin embargo, antes de realizar este paso vamos a explicar una idea que nos puede ayudar mucho a la hora de la implementación de todos estos algoritmos complicados.

Como podemos observar, el árbol algebraico está formado por nodos unarios y binarios. Los nodos unarios pertenecen a las selecciones y a las proyecciones mientras que los binarios son nodos que almacenan productos naturales, diferencias y uniones. Dado que todos los siguientes algoritmos van a trabajar sobre los productos naturales, las diferencias y las uniones (los nodos binarios) debemos encontrar una manera de eliminar los nodos unarios ya que lo único que hacen es entorpecernos la labor. Claro que los debemos eliminar de manera que tengamos arboles equivalentes.

Por tanto, la idea es que tengamos arboles algebraicos de nodos binarios y las selecciones y las proyecciones se guarden de alguna manera que no originen dificultades a la hora de implementar los algoritmos posteriores.

Veamos el siguiente árbol:

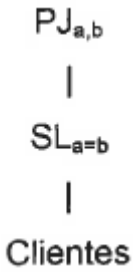


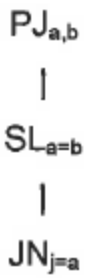
Figura 4.3. Caso base

Este es el caso base, tenemos una serie de selecciones y proyecciones sobre una tabla. Nuestro objetivo es que se quede un árbol en el que el único nodo sea la tabla clientes y las selecciones y las proyecciones se engloben en este nodo. Para realizar esto, lo que vamos a hacer es que en nuestro nuevo árbol algebraico vamos a tener un nuevo par de campos en cada nodo: uno para agrupar las selecciones y

otro para agrupar las proyecciones. Por tanto, en nuestro árbol, el árbol anterior queda así:

Clientes [Sel:a=n][Proj:a,b]

Donde Sel y Proj son las dos variables que almacenan las dos selecciones sobre la tabla. En el caso de un nodo binario, actuamos de la siguiente manera:



Agrupamos las selecciones y las proyecciones en sus variables correspondientes:

$JN_{j=a}$. [Sel:a=n][Proj:a,b]

Y veamos corno transformaríamos un árbol completo:

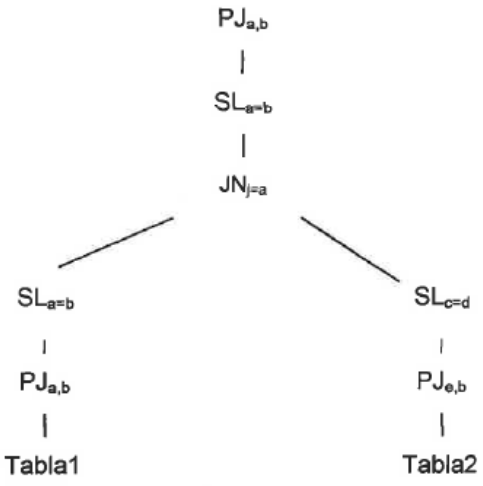
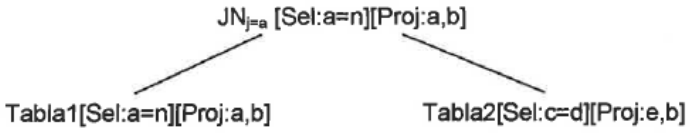


Figura 4.4. Árbol completo

Y el árbol algebraico queda así:



Esta forma de representar los arboles algebraicos nos va a permitir tener arboles muy grandes de manera muy simplificados. A la hora de realizar algoritmos como el de eliminar expresiones comunes y el de distribuir los productos naturales, esta forma de representar los arboles nos va a permitir implementarlos y ejecutarlos de manera rápida y eficiente.

El algoritmo que pasa de un árbol a otro es el siguiente:

```
ArbolN Pasar a_Arbol(Arbol a) {
        Arbol b;
        ArbolN n=new ArbolN(n);
        n.selección=new Selección();
        n.proyección=new Proyección();
        B=a;
        While (b.ope sea una SL o una PJ) {
                n.selección=n-selección+ AND +b.selección;
                n.proyección=n.proyección+ AND +b.proyección;
                b=b.hijo;
         }
n.ope=b.ope;
If (b no es hoja) {
        n.hizda=Pasar A_Arbol(a.hizda);
        n.hdcha=Pasar A_Arbol(a.hdcha);
}
return(n);
```

}

4.4.3 Eliminación de subexpresiones comunes

4.4.3.1 Introducción

En este apartado vamos a comentar como se ha conseguido eliminar las subexpresiones comunes que aparecen en las consultas SQL. El objetivo es detectar el máximo de expresiones repetidas dentro del árbol y simplificarlas mediante reglas que permitirán obtener expresiones más sencillas computacionalmente. Las reglas que vamos a utilizar para realizar la simplificación son las siguientes:

$R \text{ UN } R \Leftrightarrow R$

$R \text{ DF } R \Leftrightarrow \emptyset$

$SL_F(R) \text{ UN } R \Leftrightarrow R$

$R \text{ DF } SLF(R) \Leftrightarrow SL_{\text{Not } F}(R)$

$SL_{F1}(R) \text{ UN } SL_{F2}(R) \Leftrightarrow SL_{F1 \text{ or } F2}(R)$

$SL_{F1}(R) \text{ DF } SL_{F2}(R) \Leftrightarrow SL_{F1 \text{ and not } F2}(R)$

Además, dado que la diferencia puede dar relaciones vacías hemos de tratar también las reglas de simplificación del vacío que son las siguientes:

$R \text{ UN } \emptyset \Leftrightarrow R$

$R \text{ DF } \emptyset \Leftrightarrow R$

$\emptyset \text{ DF } R \Leftrightarrow \emptyset$

La eliminación de expresiones comunes es un algoritmo importante dentro del concepto del programa en general. Aunque realmente no forme parte de la simplificación de consultas distribuidas el hecho de que antes de pasar a simplificar tengamos la consulta lo más correcta posible nos va a permitir mayor eficiencia y rapidez a la hora de tratar los árboles. Además, conseguiremos eliminar partes innecesarias del árbol que lo único que harán serán entorpecernos.

4.4.3.2 Planteamiento general del algoritmo

El algoritmo comentado en este apartado recibe un árbol de una expresión en algebra relacional y elimina de él todas las subexpresiones comunes de acuerdo con las reglas anteriores. El árbol que recibe, recordemos, es un árbol en el que no existen nodos unarios, todos los nodos son binarios.

Para reducir un árbol, lo único que tenemos que mirar es si los dos hijos son iguales. En el caso en el que lo sean aplicaremos la regla correspondiente de las mencionadas más arriba, si no lo son, no hay que reducir. Esto es un algoritmo recursivo, porque, claro, antes de reducir el árbol actual, habrá que reducir primero a los dos hijos.

Por tanto, las dos reglas básicas del algoritmo son las siguientes:

- Reducir los hijos del nodo actual.
- Si el hijo de la izquierda es igual al hijo de la derecha, reducir aplicando la regla necesaria en función del operador del nodo actual.

Un boceto en pseudocódigo que ilustra el funcionamiento del algoritmo es el siguiente:

```
REDUCE(Arbol A) {
        Si A es hoja, return(a)       //No reducirrnos nada.
        Si A no es Hoja {
                Reduce (A-Hizda);
                Reduce (A->Hdcha);
                Si(A->Hizda=A->Hdcha){
                        Case A.Operador of
                                UN :   Actuar Según_UN.
                                DF:    Actuar Según_DF
                        }
```

```
                retum(A modificado);
        }
}
```

Como vemos, no es un algoritmo complicado. Como es un algoritmo recursivo, lo primero que hacemos es ir bajando a lo largo del árbol hasta las hojas y una vez que estamos allí, empezamos a reducir (empezando primero por las hojas por supuesto). Vamos subiendo y viendo si los hijos de los nodos son iguales, en el caso de que lo sea se reducirá en función del operador que tenga el nodo en ese momento.

4.4.3.3 ¿Cómo sabemos si dos árboles son iguales?

Como hemos visto en el algoritmo anterior, es necesario implementar una función que nos diga si dos árboles son iguales. Esta función se implementa muy fácil si recurrimos a la recursividad. Veamos los siguientes detalles:

Dos árboles A y B son iguales si:

- El nodo raíz de A es igual al nodo raíz de B.
- Los hijos de A son iguales a los hijos de B.

Como vemos es un planteamiento puramente recursivo y que nos va a permitir diseñar un algoritmo bastante sencillo. El siguiente pseudocódigo nos explica con detalle el algoritmo implementado.

```
Bool Igual(Arbol A, Arbol B) {
Si (A y B son hojas) {
        Si (A y B son iguales)
        Return(true);
        Else
                Retum(false);
} else {
        Si ( A y B no son hojas) {
                Si (Ay B son iguales) {
                        Valorl=lgual(A.hizda,B.hizda);
                        Valor2=lgual(A.hdcha,B.hdcha);
                         If (Valol=true) and (valor2=true)
```

```
                                        Return(true);
                                    else
                                        return(false);
                                } else
                                    retum(false);
                            } else {
                                return(false)
                            }
                        }
                    }
```

Es un algoritmo recursivo sencillo que sigue las dos reglas mencionadas anteriormente

- Primero, mira si los dos árboles son hojas. Si lo son, si A=B, devuelve true y false si no lo son.
- Si un nodo es hoja y otro no. Se devuelve directamente false y no se entra en la parte de código en la que los dos nodos no son hojas.
- Si ninguno de los dos nodos es hoja y A=B, miramos si los hijos son iguales. Si lo son, los dos árboles son iguales, si no lo son devuelve false.

4.4.3.4 Implementación de las Reglas de Unión:

Cuando se ha detectado que los árboles son iguales, hay que actuar según sea el operador del nodo raíz. En el caso de la Unión:

- Si los dos nodos hijos son tablas vacías (el vacío), el nodo actual se convierte también en el vacío.
- Si uno de los dos hijos es el vacío, en nodo actual pasa a ser el hijo que no es el vacío.
- Si la selección de uno de los hijos es el TODO (todas las tuplas de la tabla), entonces el nodo actual es la tabla de los dos hijos también con el TODO.

- Si los dos hijos tienen selecciones hechas sobre la tabla, el nodo actual es la tabla repetida (uno de los dos hijos) y la selección del nodo actual es el OR de las dos selecciones de los hijos.

4.4.3.5 Implementación de las reglas de la diferencia

Las operaciones que hay que realizar en el caso de tener una diferencia en el nodo raíz son las siguientes:

- Si los dos hijos son la relación vacía, el nodo actual es también el vacío.
- Si es el nodo izquierdo el que es el vacío, entonces el nodo actual también pasa a ser el vacío.
- Si es el nodo derecho el que es el vacío, el nodo actual pasa a ser el hijo izquierdo.
- Si en el nodo derecho la selección es de todas las tuplas de la tabla (TODO), entonces el nodo actual es el vacío.
- Si los dos hijos tienen selecciones hechas sobre la tabla, el nodo actual es la tabla repetida (uno de los dos hijos) y la selección del nodo actual es el AND de las dos selecciones de los hijos con la selección del hijo a la derecha negada (A AND NOT B, A selección del hijo a la izquierda y B selección del hijo a la derecha).

4.4.4 Bajada de las selecciones

El algoritmo de bajada de las selecciones consiste básicamente en seleccionar todas las selecciones que hay en el árbol y bajarlos al nivel de su hoja correspondiente, es decir, de su tabla correspondiente. Este hecho va a servir para conseguir, o por lo menos intentar, que las selecciones se hagan en la localidad de cada fragmento y no se tengan que hacer complicadas selecciones distribuidas a lo largo del desarrollo del árbol algebraico.

Como podemos suponer, la bajada de selecciones es un algoritmo bastante complicado. Se debe sobre todo a que se debe realizar un amplísimo recorrido del

árbol junto con la utilización de la metabase de datos para obtener los diferentes atributos de cada tabla. Es un algoritmo recursivo para bajar inicialmente a las hojas, para después recorrer el camino que va desde la hoja hasta la raíz del árbol.

Las claves del algoritmo se pueden resumir en los siguientes pasos:

- Bajar hasta las hojas.
- Una vez en una hoja, recorremos el camino que va desde la hoja hasta la raíz.
- En cada nodo que esté en este camino, detectar la subexpresión de la selección de ese nodo que pertenece a la tabla de la hoja (la subexpresión es la parte de la selección que está llena de atributos de la tabla).
- Bajar la subexpresión a la hoja y eliminarla, si se puede, del nodo.

En estas claves, bajar hasta las hojas y recorrer el camino hasta la raíz es bastante sencillo. Para bajar simplemente, se utiliza un algoritmo recursivo que pare la recursividad una vez que se ha encontrado una hoja. Para subir hasta la raíz simplemente vamos buscando el padre de cada nodo del camino.

Detectar la subexpresión de la selección que pertenece a la tabla es algo más complicado. Sobre todo, resulta complicado porque puede ser que dentro de una expresión haya más de una subexpresión que pertenezcan a esa tabla. Además, en el caso en el que haya expresiones muy complicadas con muchos AND y muchos OR no resulta sencillo determinar cuándo se puede bajar la subexpresión o no.

Para detectar las subexpresiones se ha implementado un algoritmo que funciona perfectamente con todo tipo de expresiones lógicas complicadas. El algoritmo va detectando todas las subexpresiones que pertenezcan a la tabla que tratamos en cuestión y devuelve una lista de expresiones que son las que pertenecen.

Decidir si se deben o no bajar las selecciones es también un criterio complicado. Si tenemos expresiones lógicas sencillas en las selecciones no hay muchos problemas. El gran problema surge cuando hay muchos OR y muchos AND anidados.

Vamos a explicar esto con un ejemplo. Veamos la siguiente expresión que se ha obtenido en una selección cualquiera de un árbol.

((a=10) AND (b>10)) OR (h<20)

En este caso, a y h son atributos que pertenecen a la tabla que estamos tratando. El algoritmo nos daría una lista con las subexpresiones (a=10 y h<20). Ahora hay que determinar si se pueden bajar al nivel de las hojas o no. Para ello, hay que obtener el árbol de la expresión que es el siguiente.

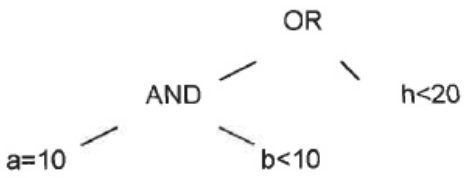


Figura 4.5. Selección cualquiera en un árbol.

La raíz de la expresión a=10 es a=10 y vemos que en el camino de a=10 al OR de la raíz existe un OR por lo que no se puede bajar a la hoja. Lo mismo le pasa a h<20. En este caso no se bajarían ninguna de las dos subexpresiones detectadas.

Si en lugar de un OR, en la raíz de la expresión hubiera un AND entonces ambas subexpresiones si se podrían bajar. Este criterio nos permite bajar las selecciones con seguridad de no perder información en el proceso.

El algoritmo de bajada de selecciones en pseudocódigo se puede ver así:

```
Bajada_De_Select(nodo a) {
        Si a es una hoja
                Bajar Los_Select(a);
                 Else {
```

```
                    Bajada_De_Select(Hijo a la izquierda de a);
                    Bajada_De_Select(Hijo a la derecha de a);
                }
        }

        Bajada_De_Select(nodo a) {
            B=padre(a);
            Mientras B no sea la raiz del arbol {
            Lista_Expresiones=Obtener Expresiones (Select de B, Tabla de A);
        Para cada una de las expresiones de Lista_Expresiones {
                Se puede bajar la expresión?
                Si se puede bajarla y eliminarla del Select de B.
            }
            B=Padre de B;
        }
}
```

Obtener_Expresión es la función que da las subexpresiones de la selección que pertenecen a la tabla de A.

4.4.5 Fragmentación

La fragmentación es el proceso por el que las tablas globales se transforman en fragmentos. Lo que se hace es que una tabla se sustituye en el árbol por su expresión canónica correspondiente. La expresión canónica es la operación de reconstrucción del árbol a partir de los fragmentos, por lo tanto si tenemos una fragmentación horizontal lo que hacemos es sustituir la hoja de la tabla del árbol por la unión de sus fragmentos que lógicamente conseguirá reconstruir la tabla global.

La fragmentación trata, por tanto, de buscar en el árbol las hojas y sustituirlas por la expresión canónica de la tabla en función de sus fragmentos según el tipo de fragmentación que tengamos.

El algoritmo desarrollado es un algoritmo muy sencillo que Baja hasta las hojas y fragmenta según el tipo de fragmentación:

```
Void Fragmentar(Arbol a) {
        If a es Hoja {
                Si es Fragmentacion Horizontal
                        Fragmentar H(a);
                Si es Fragmentacion Verical
                        Fragmentar V(a);
                Si es Fragmentacion Derivada
                        Fragmentar D(a);
        } else {
                        Fragmentar(a.hizda);
                        Fragmentar(a.hdcha);
        }
}
```

Veamos ahora como se hace cada una de las fragmentaciones:

•**Fragmentación Horizontal** La fragmentación horizontal se realiza sustituyendo la hoja de la tabla por la unión de los fragmentos. Sin embargo, hemos de tener en cuenta que debemos eliminar aquellos fragmentos que ya sabemos de los que no vamos a obtener tupla alguna. Esto se sabe por la selección que tiene la tabla en ese momento. Como en este paso ya se han bajado las selecciones, nosotros sabemos por la condición de la selección que tuplas van a ser necesarias a lo largo del desarrollo del árbol y, por tanto, podremos eliminar directamente aquellos fragmentos que no tengan tuplas necesarias.

Para saber si un fragmento tiene tuplas validas o no, usamos el álgebra de relaciones cualificadas. Juntamos la condición que cumplen las tuplas del fragmento junto con la selección de la tabla en ese momento con un and y aplicamos el álgebra de relaciones cualificadas. Si el resultado es la tupla vacía el fragmento se puede eliminar pues no genera tuplas relevantes, en otro caso el fragmento se queda.

• **Fragmentación Vertical**: En la fragmentación vertical se sustituye la hoja de la tabla actual por el producto natural de los fragmentos verticales de la tabla. En este caso, determinar si un fragmento es válido o no, depende de la selección de la tabla.

Depende de si los atributos que contiene el fragmento se van a usar a lo largo de la ejecución del árbol y, por tanto, de la consulta SQL.

Para determinar si un atributo se usa a lo largo del árbol se va explorando el camino que va desde la hoja a la raíz. Si en uno de los nodos se tiene que el atributo se utiliza en una proyección o una selección, entonces el atributo se usa; en caso contrario, no se usa.

En el caso en el que ninguno de los atributos se use a lo largo del árbol entonces el fragmento se puede eliminar; en el caso en el que alguno de los atributos se use, entonces el fragmento debe quedarse.

• **Fragmentación horizontal derivada**: En el caso de una fragmentación derivada, la tabla se sustituye por la unión de los semi productos naturales de cada fragmento con el fragmento del que deriva. Para determinar si un fragmento debe quedarse o ser eliminado actuamos de la misma manera que en la fragmentación horizontal. Lo que hacemos es aplicar el álgebra de relaciones cualificadas a la selección de la tabla y a la condición que cumple el fragmento del que deriva (no el fragmento de la tabla) ya que es este el que indica la condición que cumplen las tuplas del fragmento. En el caso en el que el álgebra de relaciones cualificadas de la tupla vacía el fragmento se puede eliminar.

Puede que esto no haya quedado tan claro como en las fragmentaciones anteriores, por lo que proponemos un ejemplo.

La tabla Empleado se fragmenta en dos fragmentos de forma derivada en función de los fragmentos de la tabla Departamento:

Figura 4.6. Selección en tabla empleado

Pasa a ser:

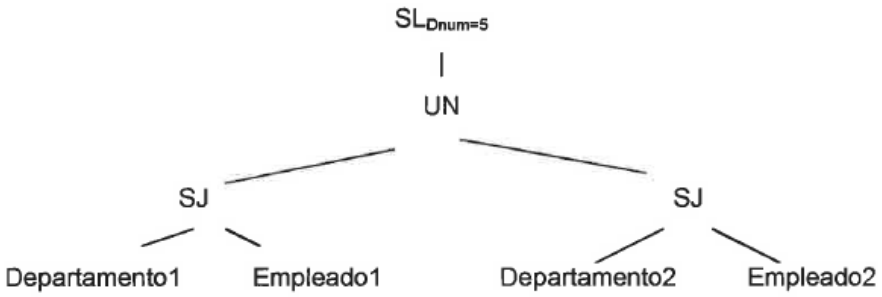


Figura 4.7 Reconstrucción de la tabla empleado

Los fragmentos de Departamento cumplen que:

Departamento1: Dnum<10;
Departamento2: Dnum>=10;

Ahora, para ver que fragmentos se deben de quedar aplicamos el álgebra de relaciones cualificadas:

En Empleado1, hacemos (Dnum<10) AND (Dnum=5) y como no da vacío el fragmento se queda.

En Empleado2 hacemos (Dnum>=10) AND (Dnum=5) y como da vacío el fragmento se elimina.

Como vemos, las condiciones de los fragmentos son las condiciones de las tablas de las que deriva (en este caso de Departamento) y utilizaremos estas condiciones y la condición de la selección de la tabla para ver si el fragmento se debe quedar o se debe eliminar.

La tabla fragmentada realmente queda así:

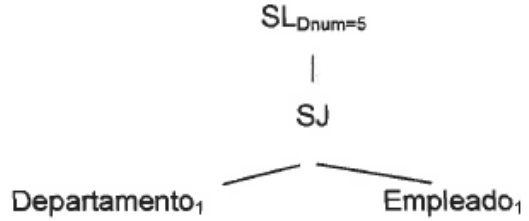


Figura 4.8 Tabla final fragmentada

Para concluir indicaremos que en nuestro árbol no existe el semi producto natural, lo simulamos a través de un producto natural (JN) junto con una proyección de los atributos de la tabla Empleado.

4.4.6 Distribución de productos naturales

La distribución de productos naturales implementa el criterio 5 de la lista de criterios que debemos implementar.

Para distribuir los productos naturales que aparecen en las consultas globales, las uniones (que representan conjuntos de fragmentos) deben de colocarse debajo de los productos naturales que queremos distribuir. Para la distribución de los productos naturales se utiliza la siguiente regla que es la que nos va a permitir tal operación:

$$(R' \text{ UN } R'') \text{ JN}_F (S' \text{ UN } S'') \Leftrightarrow \text{UN}((R' \text{ JN}_F S'), (R' \text{ JN}_F S''), (R'' \text{ JN}_F S'), (R'' \text{ JN}_F S''))$$

La implementación de esta regla se realiza mediante un algoritmo recursivo que se mueve desde la raíz hasta las hojas y que se detiene en la siguiente situación:

- Si no estamos en una hoja y ni los hijos son hojas.
- Si el operador de este nodo es un producto natural y los operadores de los hijos son uniones.

Si nos encontramos un nodo que cumple tales condiciones entonces pasamos a distribuir el producto natural de la siguiente manera:

- Primero hacemos que el nodo actual pase a ser una unión.
- Combinamos los hijos de la siguiente manera.
- Cogemos el hijo a la izquierda del hijo a la izquierda del nodo y lo combinamos mediante un producto natural con el hijo a la izquierda del nodo a la derecha del nodo. Este pasa a ser el primer hijo del nodo actual.

- Cogemos el hijo a la izquierda del hijo a la izquierda del nodo y lo combinamos mediante un producto natural con el hijo a la derecha del nodo a la derecha del nodo. Este pasa a ser el segundo hijo del nodo actual.
- Hacemos lo mismo con el hijo a la derecha del hijo a la izquierda del nodo con el hijo a la izquierda del hijo a la derecha del nodo.
- Y por último con los dos hijos de hijos que faltan por coordinar: el hijo a la derecha del hijo a la izquierda con el hijo a la derecha del hijo a la derecha.

Visto así en palabras, el algoritmo parece complicado, pero si vemos el siguiente pseudocódigo podemos observar que es bastante sencillo:

```
Void Distribuir Join(Arbol a) {
        If (A es una hoja y A.izda es una hoja y A.dcha es una hoja) {
                Distribuir Join(a.hizda);
                Distribuir.Join(a.hdcha);
                //Ante todo, hay que hacer que la recursividad distribuya
                //primero a los hijos.

                Si (A.ope es JN y A.hizda.ope es UN y A.hdcha.ope es UN) {
                        Crear_Arbol(A.hizda.hizda, A.hdcha.hizda);
                        Poner el nuevo arbol como hijo de A.
                        Crear Arbol(A.hizda.hdcha, A.hdcha.hizda);
                        Poner el nuevo arbol como hijo de A.
                        Crear Arbol(A.hizda.hizda, A.hdcha.hdcha);
                        Poner el nuevo arbol como hijo de A.
                        Crear_Arbol(A.hizda.hdcha, A.hdcha.hdcha);
                        Poner el nuevo arbol como hijo de A.
                }
        }
}
```

Esta es la filosofía del algoritmo implementado: se trata de buscar las situaciones en las que se pueda distribuir el producto natural de manera recursiva y distribuirlo. El algoritmo real tiene en cuenta que se pueden dar circunstancias en las

que un nodo unión tenga más de un hijo. Esta situación no aporta mayor gravedad: el algoritmo sigue siendo el mismo, solo que en vez de combinar el hijo a la izquierda y a la derecha se combinan todos los nodos que estos nodos puedan tener como hijos y que darán lugar a n*m combinaciones de productos naturales siendo n el número de nodos a la izquierda y m el número de nodos a la derecha.

Además, el algoritmo real también contempla que alguno de los hijos sea un fragmento. En este caso tendríamos un producto natural junto con un nodo hoja y un nodo unión como hijos. Siguiendo la filosofía mencionada el problema se resuelve fácilmente de la misma manera, solo que ahora combinamos el nodo hoja con los hijos del nodo unión.

4.4.7 Eliminación de productos naturales inútiles

Una vez que se han distribuido los productos naturales, es posible que aparezcan cantidad de ramas de productos naturales que no generen tupla alguna. Esto es debido a que, al distribuir los productos naturales se cruzan todos los fragmentos entre las distintas tablas y lo más probable es que muchos de estos productos naturales no den lugar a ningún tipo de dato que sea necesario devolver en la consulta. En este apartado, lo que vamos a tratar es de detectar estos productos naturales y eliminarlos.

La detección de un producto natural inútil es sencilla aplicando el álgebra de relaciones cualificadas. Lo que hacemos es detectar las cualificaciones de las tablas del producto natural y aplicar el álgebra de relaciones cualificadas. La cualificación de cada fragmento es la selección que lleve ese fragmento en ese momento, junto con la condición que cumplen las tuplas de ese fragmento. Para ver esto es mejor usar un ejemplo.

El siguiente producto natural es un producto natural inútil:

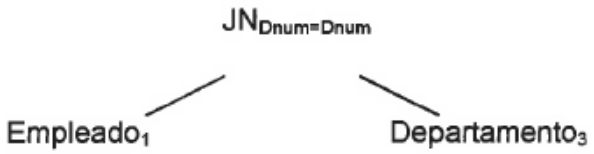


Figura 4.9 Producto natural inútil

Empleado1 cumple que Dnum<10 y Departamento3 cumple que Dnum>20. Obviamente al realizar este producto natural el resultado es la tupla vacía.

La cualificación de Empleado1 es, por tanto, su selección más su condición de fragmentación. Como no tiene selección, su cualificación es Dnum<10. A Departamento3 le pasa lo mismo y su cualificación es Dnum>20.

Ahora, para detectar si el producto natural genera la tupla vacía lo que hacemos es aplicar el álgebra de relaciones cualificadas a la condición (Dnum<10) AND (Dnum>20) y lógicamente obtenemos la relación vacía con lo que hemos detectado el producto natural inútil.

Por tanto, lo que hacemos en definitiva es:

- Buscar a lo largo del árbol hojas unidas mediante un producto natural.
- Si se detecta un caso, construimos la cualificación del producto natural uniendo las cualificaciones de las dos hojas mediante un AND. (Recordemos que la cualificación de cada hoja es la condición que cumplen las tuplas de ese fragmento junto con la selección que tengan hechas en ese momento).
- A esta cualificación le aplicamos el álgebra de relaciones cualificadas y si da la relación vacía se elimina el producto natural; en caso contrario, se mantiene.

El algoritmo que utilizamos es un algoritmo recursivo que Baja hasta las hojas y busca la situación en la que aparezca un producto natural con hojas por hijos.

```
Elimina_Join(Arbol a) {
        Si A no es una hoja {
```

```
                Elimina_Join(A.hizda);
                Elimina_Join(A.hdcha);
                Si (A.Hizda es hoja y A.Hdcha es hoja también) {
                        Si el operador de A es el JN.
                        Cadena=Obtener Cualificación (A.Hizda) + AND +
                                + Obtener Cualificación(A.Hizda);
                        Aplicar Albebra(Cadena);
                        Si da Vacío
                                A pasa a ser el vacío;
                        Else
                                A se mantiene como esta.
                }
        }
}
```

Obtener_cualificación devuelve la condición de las tuplas del fragmento unido con un and a la selección de la tabla en ese momento.

El algoritmo real implementado es un calco del pseudocódigo anterior, solo que la implementación resulta un poco más costosa de to que el algoritmo al principio parece indicar. Sin embargo, la estructura y el esqueleto del mismo son exactamente igual al pseudocódigo. Además, el algoritmo real funciona para el árbol que hemos diseñado que en momentos puede que sea binario o multihijo en función de las necesidades de cada momento.

4.4.8 Obtener la consulta SQL a partir del árbol de operador

Obtener la consulta SQL a partir de nuestro árbol de operador va a resultar bastante sencillo. Sin embargo, antes de proceder al cálculo de la consulta debemos de preparar el árbol para que se adapte a las condiciones especiales que el lenguaje SQL de Oracle impone. Estas son las siguientes:

- Las uniones y las diferencias no pueden tener selecciones o proyecciones sobre ellas. Dado que SQL solo admite uniones y diferencias de sentencias

de selección, es imprescindible que las diferencias y las uniones no tengan ningún tipo de selección ni de proyección. Lo que debemos de hacer es bajar las selecciones y las proyecciones at nivel de los productos naturales.

- No puede haber por encima de una unión ningún producto natural ni ningún producto cartesiano (CP). Es posible que haya alguna unión que no se haya podido distribuir en el proceso de distribución. Esto es normalmente porque tenemos un producto natural de una hoja y de una unión. Esto es normal en el caso de que tengamos fragmentaciones verticales. Dado que Oracle no permite productos naturales ni productos cartesianos de uniones de selecciones, es vital que subamos la unión por encima de los productos naturales o de los productos cartesianos necesarios para generar consultas correctas.

Estas dos características implican situaciones especiales que debemos tratar para que el paso del árbol a consulta no genere consultas SQL no válidas.

En el primer caso, lo que tenemos que hacer es bajar las selecciones y las proyecciones al nivel de las hojas. Dado que en este momento los árboles que vamos a tener van a ser uniones y diferencias de productos naturales.

Esto se realiza de forma fácil realizando un recorrido por el árbol desde la raíz a las hojas; en el que en el caso de encontrarnos una unión o una diferencia bajaremos sus selecciones o proyecciones a sus hijos correspondientes. El algoritmo se muestra a continuación:

```
Void Bajar_Selecciones_Y_Proyecciones(Arbol a) {
        Si (A es una UN o una DF) {
                Hizda.Seleccion=Hizda.Seleccion+ "AND"+a.Seleccion;
                Hdcha.Seleccion=Hdcha.Seleccion+ "AND"+a.Seleccion;
                Hizda.Proyeccion=Hizda.Proyeccion+ "AND"+a.Proyeccion;
                Hdcha.Proyeccion=Hdcha.Proyeccion+ "AND"+a.Proyeccion;
                a.Seleccion="TODO";
                a.Proyeccion="*";
         }
```

}

En el caso en el que tengamos que subir las uniones que se han quedado sin distribuir, tenemos que ver lo siguiente. La situación en la que una unión no se distribuya se debe a que el hermano de la unión no es otra unión sino un producto natural. La situación que queremos explicar se refleja en el siguiente árbol:

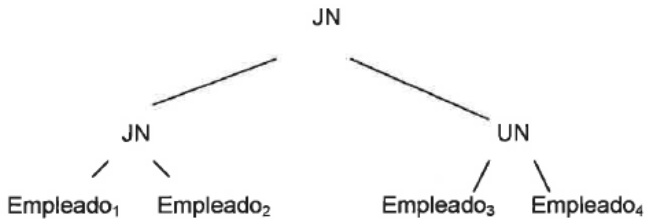


Figura 4.10 Árbol con fragmentación vertical y horizontal

Este árbol resulta de resolver una fragmentación de la tabla Empleado que tiene por una lado una fragmentación vertical entre un fragmento que está fragmentado verticalmente también entre Empleado1, y Empleado3 y otro fragmento que a su vez se fragmenta en una fragmentación horizontal entre Empleado3 y Empleado4.

En este tipo de casos, no podemos distribuir el producto natural haciendo una unión de los productos naturales de Empleado1, con Empleado3 y Empleado4, y de Empleado2 con Empleado3 y Empleado4. Lógicamente esto no se puede hacer porque no dan la misma tabla un árbol que otro.

Para resolver esto lo que hacemos es hacer primero un producto natural de la tabla Empleado3 con el producto natural completo y después otro con la tabla Empleado4 con el producto natural completo. El resultado de esta acción, para que se entienda mejor, se puede ver en el siguiente árbol:

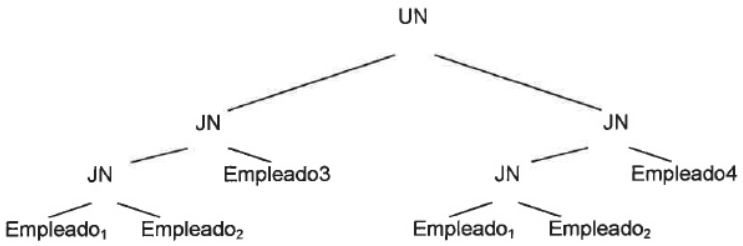


Figura 4.11 Resultado de distribuir el producto natural.

Esto no se hace para obtener una consulta más eficiente, se hace para poder obtener una consulta SQL posible de ejecutar. Dado que este es un caso que se puede dar con mucha posibilidad, debido a las fragmentaciones verticales, está claro que debe ser resuelto antes de tratar de pasar el árbol a consulta. En el caso de que tengamos productos cartesianos en vez de productos naturales, se actúa de la misma manera, no hay ningún tipo de problemas.

Una vez que hemos conseguido bajar las selecciones y las proyecciones y subir las uniones que se hayan quedado por debajo debido a la unión, podemos pasar a conseguir la consulta SQL. En estos momentos tenemos un árbol cuya topología es la siguiente:

- Las hojas son tablas.
- Los padres de las hojas son productos naturales o cartesianos.
- La parte alta del árbol son uniones o diferencias y en ningún caso puede haber un producto natural o un producto cartesiano por encima de una unión o una diferencia.

Este árbol se podría ver como que los nodos internos son uniones y diferencias y las hojas son los grupos de productos naturales y tablas que se encuentran al final de cada rama del árbol.

En efecto, los productos naturales y los productos cartesianos, se deben de agrupar al final de la rama del árbol porque cada grupo de estos dará lugar a una consulta SQL independiente. Veamos el siguiente ejemplo:

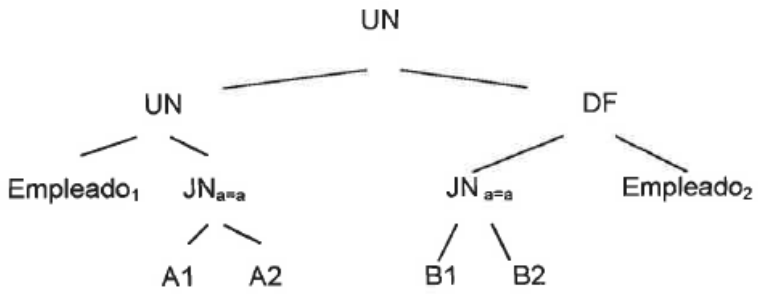


Figura 4.12 Continuación del ejemplo

La consulta SQL que da este árbol es:

((select * from empleado1) union (select * from A1,A2 where a=a)) union ((select * from empleado2) union (select * from B1,B2 where a=a))

Como vemos, en ningún caso puede ser posible que haya uniones o diferencias por debajo de los productos naturales.

El algoritmo que pasa de un árbol a una consulta es un algoritmo recursivo del tipo de los que hemos visto:

```
Void Paso_A_Consulta(Arbol a) {
        If (A es una hoja) {
                //Estamos ante una tabla que debe dar una sentencia select
                // Sencilla
        return("Select+a.proyeccion+"from"+a.tabla+"where"+a.seleccion);
         } else {
                if (A.ope="UN") {
                return(Paso_A_Consulta(a.hizda)+"union"+
Paso_A_Consulta(a.dcha) }
                if (A.ope="UN") {
                return(Paso_A_Consulta(a.hizda)+"minus"+
Paso_A_Consulta(a.dcha) }
                if (A.ope="JN" || A.ope="CP") {
                        p=Obtener Proyecciones(a);
                         s=Obtener Selecciones(a);
```

```
                    t=Obtener Tablas(a);
                    return("Select p from t where s");
                }
            }
        }
```

Cuando se llega a un producto natural, se sabe que ese producto natural da una consulta SQL formada por él y los productos naturales y tablas que tenga por debajo. La operación Obtener_Proyecciones devuelve las proyecciones de los productos naturales y las tablas que hay por debajo del producto natural actual. La operación Obtener_Selecciones, hace lo mismo pero con las selecciones. Obtener_Tablas obtiene todas las tablas que hay por debajo del producto natural.

4.4.9 Aplicación del algebra de relaciones cualificadas

En concreto, en este apartado se describe la comprobación de la consistencia de un predicado que nos será muy útil para la implementación de los diferentes criterios de simplificación de árboles algebraicos. Por ejemplo cuando bajamos los operadores de selección (SL) en el árbol algebraico necesitamos una función que compruebe si la condición del operador SL es consistente con la cualificación del fragmento, en el caso de que no sea consistente eliminamos el fragmento. Lo mismo sucede para la distribución de los operadores de producto natural, también necesitamos comprobar si las cualificaciones de los fragmentos son consistentes.

Para su construcción también se ha utilizado las herramientas Jlex y Java Cup que se ha mencionado anteriormente.

Construiremos un analizador léxico que reconozca los tokens que aparecen en un predicado y una analizador sintáctico que determine si el predicado es correcto y que mediante acciones semánticas detecte inconsistencias en éste.

Los tokens que reconoce el analizador lexico son los siguientes: ENTERO, REAL, CADENA, IDENTIFICADOR, OPBOOLEANO, NOT, PARENIZDA, PARENDCHA, OPCOMPARACION.

La gramática (donde las palabras en mayúscula son tokens y las palabras en minúscula son símbolos no terminales (reglas o producciones)) que reconoce los predicados con los que vamos a trabajar en la implementación de los diferentes criterios de simplificación es la siguiente:

predicado::= literal
|
literal OPBOOLEANO predicado
|
PARENIZDA predicado PARENDCHA
|
NOT predicado
|
PARENIZDA predicado PARENDCHA OPBOOLEANO predicado;
|
literal ::= IDENTIFICADOR OPCOMPARACION IDENTIFICADOR
|
IDENTIFICADOR OPCOMPARACION numero
|
IDENTIFICADOR OPCOMPARACION CADENA
|
numero OPCOMPARACION IDENTIFICADOR
|
CADENA OPCOMPARACION IDENTIFICADOR
|
numero ::= ENTERO
|
REAL

CAPÍTULO 5: PRUEBAS

5.1 Introducción

A continuación se detalla aquellas acciones y pruebas que nos permitirán validar el funcionamiento del sistema y así comprobar los posibles errores que el sistema tenga.

La fase de pruebas corresponderá a comprobar intensivamente todo aquello que concierna tanto a la navegación como a la realización concreta de acciones dentro de nuestra aplicación.

Por tanto, se deberá realizar una verificación exhaustiva de todas y cada una de las acciones que se puedan realizar dentro de cada página concreta. En concreto, se introducirán varias consultas para comprobar que el sistema trabaja correctamente y obtiene unos resultados correctos, tal y como hemos ido viendo a lo largo de los apartados anteriores.

Respecto a la parte de obtención de resultados, debemos comprobar que también esa información sea gestionada de forma adecuada, y para ello debemos realizar una serie de simulaciones que nos permitieran verificar que toda esa información está siendo generada coherentemente. Además, dicha información se debe presentar de forma que sea fácilmente entendible y comprensible, de cara a su correcta interpretación.

5.2 Pruebas

Para comprobar que la aplicación funciona de acuerdo con lo establecido, y se obtienen resultados correctos, según lo que sabemos que debe salir, hemos pensado en la introducción de ciertas consultas para ver cómo se van procesando dichas consultas; aquí analizamos los subárboles que se van obteniendo tras la aplicación de cada uno de los pasos que forman el proceso de transformación de consultas globales en consultas fragmentadas.

Consulta 1

La primera consulta que vamos a considerar es para mostrar que la aplicación realiza correctamente la fragmentación tanto horizontal como vertical de una tabla dada; es la siguiente:

"Mostrar los nombres de los empleados".

En primer lugar vamos a expresar esta consulta en SQL:

```
select nombre from Empleado;
```

Una vez que tenemos la consulta expresada mediante SQL, procedemos a expresarla de una forma equivalente usando el álgebra relacional. Sería la siguiente:

PJ nombre (empleado)

El siguiente paso es obtener el árbol algebraico correspondiente a dicha consulta. Sería el siguiente (figura 5.1):

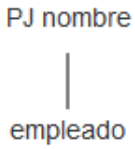


Figura 5.1

Ya que tenemos el árbol algebraico, pasamos a ver el árbol fragmentado. Para ello hay que ver coma está fragmentada la tabla Empleado. Como eso ya lo vimos en el Ejemplo General en el apartado 2.2.4, nos limitamos a mostrar directamente el árbol en la figura 5.2.

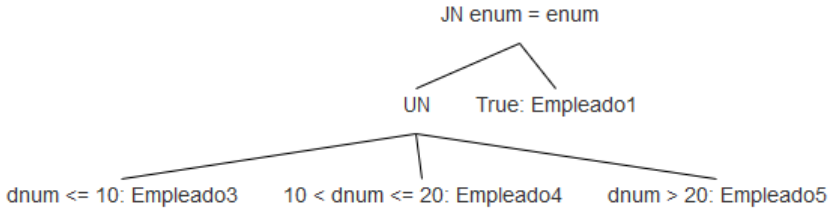


Figura 5.2

El siguiente paso en el procesamiento de la consulta, es distribuir el producto natural (criterio 5) con respecto a la unión, por lo que habrá que subir la unión por encima del producto natural. El árbol que se obtiene se muestra en la figura 5.3.

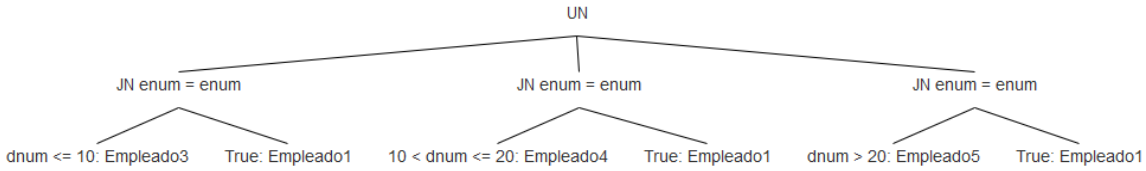


Figura 5.3

Una vez que hemos obtenido el árbol final de la consulta, pasamos dicho árbol a consulta SQL distribuida. Por tanto, la consulta resultante sería la siguiente:

(select nombre from Empleado1,Empleado5 where Empleado5.enum = Empleado1.enum) union (select nombre from Empleado1,Empleado4 where Empleado4.enum = Empleado1.enum) union (select nombre from Empleado1,Empleado3 where Empleado3.enum = Empleado1.enum)

Consulta 2

La segunda consulta que vamos a considerar, es para mostrar cómo se eliminan los fragmentos verticales que no sean necesarios para la consulta. En este caso, la consulta es:

"Mostrar los salarios de todos los empleados"

La consulta expresada mediante SQL seria la siguiente:

select salario from Empleado;

El siguiente paso es expresar la consulta usando el álgebra relacional.

PJ salario (empleado)

El árbol algebraico correspondiente se muestra en la figura 5.4.

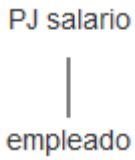


Figura 5.4

El árbol fragmentado se obtiene a partir de la fragmentación de la tabla Empleado y es el mostrado en la figura 5.5.

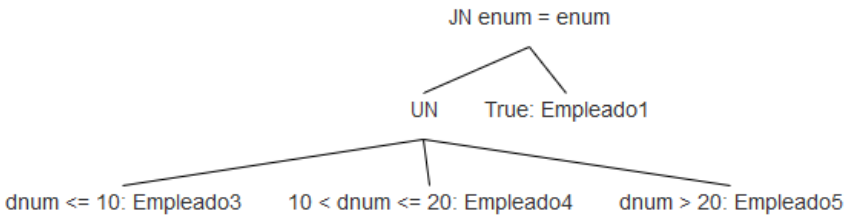


Figura 5.5

En este caso, el atributo salario solo se encuentra en el fragmento vertical Empleado1 por lo que podremos ignorar el resto de fragmentos que contendrán otros atributos que no nos hacen falta para realizar esta consulta. El árbol resultante de esta consulta se muestra en la figura 5.6.

Figura 5.6

El último paso es obtener la consulta a partir del árbol final de dicha consulta; en este caso, obtenemos una sencilla consulta en SQL:

select salario from Empleado1

Consulta 3

La siguiente consulta que vamos a ver se usa para comprobar el funcionamiento de la aplicación ante un producto natural entre dos tablas. La consulta es la siguiente:

"Obtener el nombre y salario de los empleados que trabajan en cada departamento".

La consulta SQL correspondiente seria la siguiente:

select nombre,salario from Empleado,Departamento where Empleado.Dnum = Departamento.Dnum;

El siguiente paso a realizar es expresar la consulta en álgebra relacional:

PJ nombre, salario (empleado JN dnum=dnum departamento)

El árbol algebraico para dicha consulta es el que se muestra en la figura 5.7.

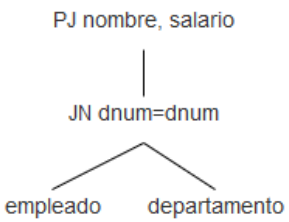


Figura 5.7

A continuación se procede a la obtención del árbol fragmentado. Recordemos que la tabla Empleado es una fragmentación mixta, mientras que Departamento, es simplemente una fragmentación horizontal. El árbol fragmentado es el que se muestra en la figura 5.8.

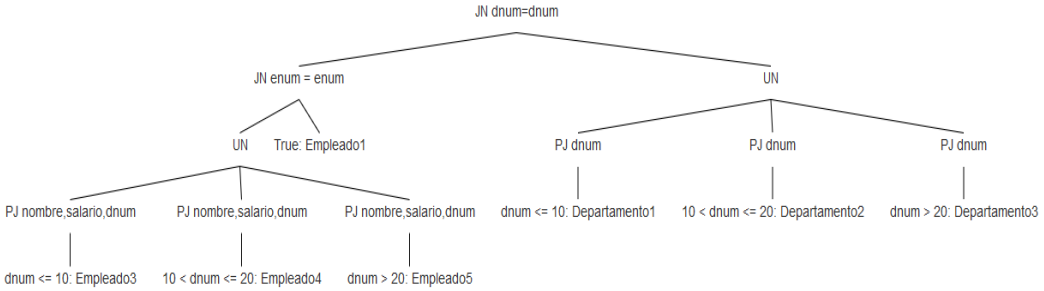


Figura 5.8

A continuación se procede a distribuir los productos naturales; para ello se sube la unión por encima de los productos naturales. El árbol que resulta es muy extenso, pero muchos de los fragmentos que aparecen se eliminan pues nos encontramos con ciertas contradicciones (debidas al atributo Dnum), por lo que por cuestión de espacio, se ha optado por mostrar el árbol definitivo (excepto la última rama pero se sabe que vendría la rama con departamento3 y empleado2), en el que ya se han eliminado las ramas en las que existía alguna contradicción; este árbol se muestra en la figura 5.9.

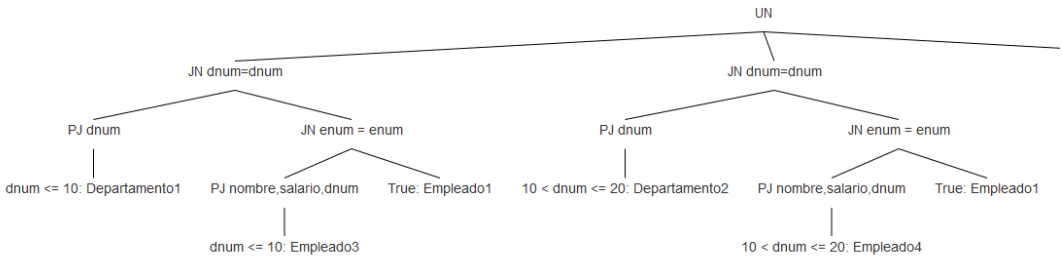


Figura 5.9

El último paso es obtener la consulta a partir de dicho árbol; es la siguiente:

(select nombre, salario from Departamento3, Empleado5, Empleado1 where Departamento3.dnum = Empleado5.dnum and Empleado5.enum = Empleado1.enum) union (select nombre, salario from Departamento2, Empleado4, Empleado1 where Departamento2.dnum = Empleado4.dnum and Empleado4.enum = Empleado1.enum

) union (select nombre, salario from Departamento1, Empleado3, Empleado1 where Departamento1.dnum = Empleado3.dnum and Empleado3.enum = Empleado1.enum)

Consulta 4

Esta consulta es similar a la anterior, pero añadiéndole una condición para eliminar aquellos fragmentos que presenten algún tipo de contradicción con respecto a la condición. La consulta es la siguiente:

"Obtener el nombre y salario de los empleados que trabajan para el departamento número 5".

La consulta SQL correspondiente seria la siguiente:

Select nombre,salario from Empleado,Departamento where Empleado.Dnum = Departamento.Dnum and Departamento.Dnum = 5;

A continuación se expresa la consulta usando el álgebra relacional:

PJ nombre, salario SL dnum = 5 (empleado JN dnum=dnum departamento)

El árbol algebraico correspondiente a dicha consulta es el que se muestra en la figura 5.10.

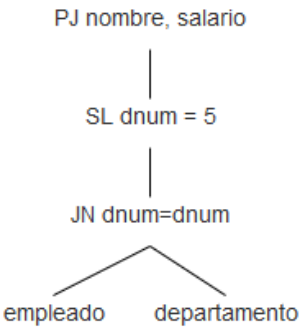


Figura 5.10

A continuación se procede a la obtención del árbol fragmentado, que sigue la misma idea que en la consulta anterior: una fragmentación mixta y otra horizontal. El árbol fragmentado se muestra en la figura 5.11.

Antes de proceder a distribuir el producto natural, bajamos la condición de la selección al nivel de las hojas, para eliminar aquellos fragmentos en los que haya una contradicción. Tal es el caso de los fragmentos Empleado4 y Empleado5 de la tabla Empleado, y Departamento2 y Departamento3 de la tabla Departamento. Como ya hemos visto con anterioridad, estas contradicciones se deben a que la consulta requiere aquellos datos cuyo Dnum sea 5, y esto da lugar a contradicción con las condiciones de fragmentación de los fragmentos anteriores. El árbol correspondiente a este paso se muestra en la figura 5.12.

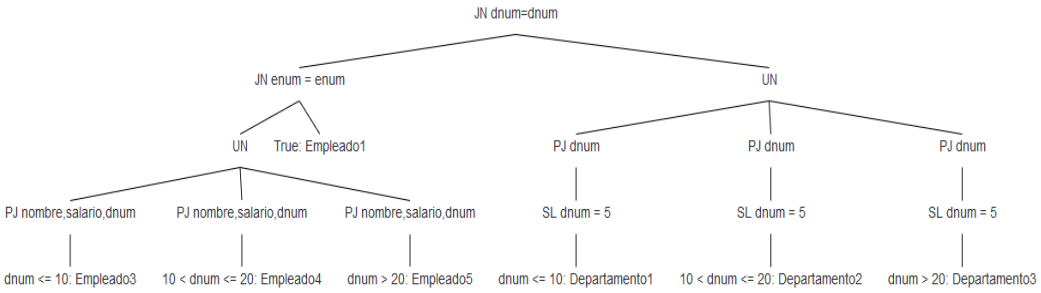


Figura 5.11

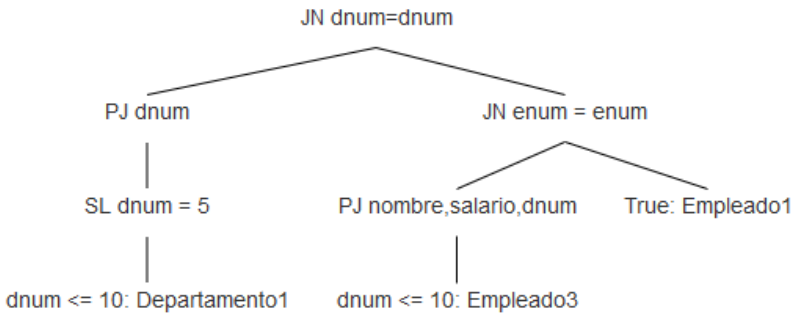


Figura 5.12

El árbol representado en la figura 5.12 es el árbol final para esta consulta. El último paso es obtener la consulta a partir de dicho árbol:

select nombre, salario from Empleado3,Departamento1 where Empleado3.dnum = Departamento1.dnum and Empleado3.enum = Empleado1.enum and Departamento1.dnum = 5

Consulta 5

En la consulta que vamos a probar ahora, se ha probado la introducción del operador de diferencia (minus):

“Obtener el identificador de departamento de aquellos con nombre igual a 001 y el identificador sea menor o igual a 5”

La consulta se puede expresar en SQL de la siguiente forma:

select dnum from departamento where nombre = '001' minus select dnum from departamento where dnum > 5

Una vez que tenemos la consulta expresada mediante SQL, procedemos a expresarla de una forma equivalente usando el álgebra relacional

PJ dnum ((SL nombre = '001' departamento) DF (SL dnum > 5 departamento))

A continuación obtenemos el árbol algebraico correspondiente a dicha consulta. Sería el que se muestra en la figura 5.13.

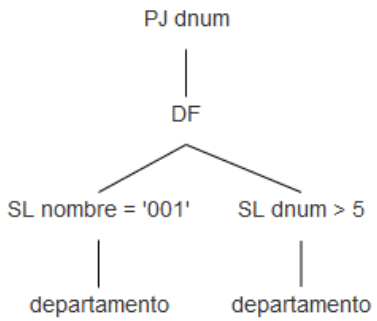


Figura 5.13

Observando el árbol algebraico podemos ver que se le puede aplicar la regla vista anteriormente (apartado 2.3.2.3):

$$SL_{F1}(R)\ DF\ SL_{F2}(R) \Leftrightarrow SL_{F1\ AND\ NOT\ F2}(R)$$

Al aplicar la regla se obtiene el siguiente árbol (Figura 5.14)

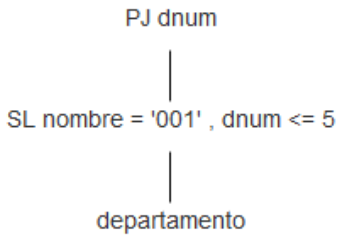


Figura 5.14

El siguiente paso es usar el criterio 3, que nos dice que bajemos las selecciones hasta el nivel de las hojas, usar el álgebra de relaciones cualificadas, aplicar esas selecciones y sustituirlas por la relación vacía si la condición de la selección y la cualificación de la relación sea contradictoria. Al aplicar ese criterio vemos que hay una contradicción en los fragmentos departamento2 y departamento3, estos se eliminan del árbol quedando como en la figura 5.15.

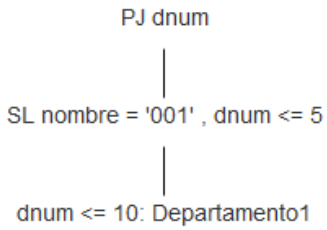


Figura 5.15

Una vez obtenido el árbol final, ya podemos obtener la consulta correspondiente.

select dnum from Departamento1 where Departamento1.nombre = '001' and Departamento1.dnum <= 5

Consulta 6

La última consulta que se va a probar será una contradicción, para ver el sistema cómo reacciona ante este tipo de situaciones.

“Obtener el id de los proveedores de la ciudad de San Antonio (SA) que hayan suministrado algún producto”

La expresión en SQL corresponde a la siguiente:

select s.pnum from suministro s,proveedor p where s.pnum = p.pnum and ciudad = 'SA'

La consulta usando álgebra relacional quedaría:

PJ pnum SL ciudad = 'sa' (suministro JN pnum=pnum proveedor)

El árbol algebraico para dicha consulta es el que se muestra en la figura 5.16

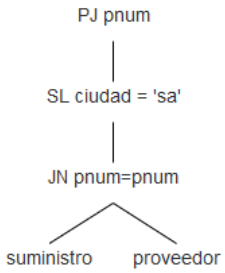


Figura 5.16

Lo siguiente es aplicar el criterio 3 y bajar las selecciones al nivel de las hojas. Al aplicarlas vemos existen contradicciones en los dos fragmentos (Figura 5.17)

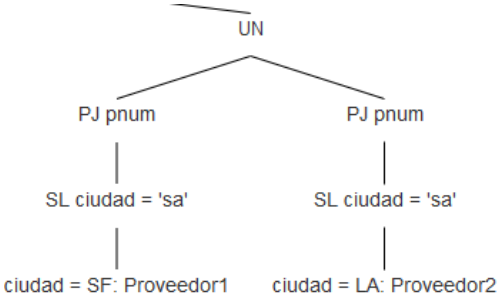


Figura 5.17

Como hemos visto en el árbol anterior existe una contradicción en los dos fragmentos, por lo que no se puede seleccionar ningún dato y esto equivale al conjunto vacío. Véase la Figura 5.18

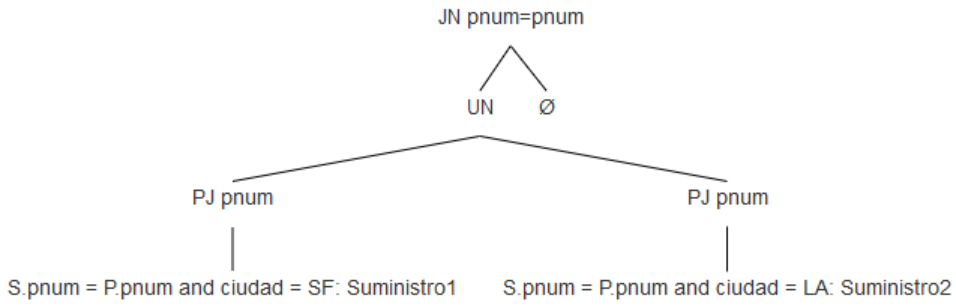


Figura 5.18

Si aplicamos la regla:

R JN_f Ø ⇔ Ø

obtenemos el conjunto vacío para el resultado de esta consulta.

CAPÍTULO 6: CONCLUSIONES

6.1 Conclusiones

Como se comentó al principio de esta memoria la motivación de este proyecto fue la posibilidad de que el alumno pudiede aprender el procesamiento distribuido de consultas de manera gráfica y como complemento de los apuntes seguidos en clase.

Cuando se eligio una interfaz web se pensó en la flexibilidad y cobertura que tienen estos tipos de aplicaciones. En un primer momento se pensó en hacer una aplicación java y luego cargarlo en el navegador en un applet. Aunque los applet son muy potentes pueden llegar a tener serios problemas de compatibilidad sobre todo relacionados con la versión de java instalada. Por esto último una aplicación web es más conveniente siempre y cuando sea posible resolver el problema planteado.

También hay que añadir que este tipo de aplicación concretamente aún hoy en día no existen o son “experimentales” como bien puede ser este proyecto. Una razón puede ser la complejidad que puede llegar a tener una consulta y la dificultad que conlleva procesarla, con todos los operandos que pueden intervenir, predicados, subconsultas etc. Además de que cada gestor de base de datos puede procesar una consulta distribuida de manera diferente por eso en este proyecto se ha seguido una manera general de como se procesaría.

Por último, destacar la ayuda que he recibido por parte de mi tutor tanto de las clases recibidas durante el periodo lectivo como durante el desarollo de este proyecto, ya que ha sido una propuesta suya que me parecio interesante.

6.2 Áreas de mejora y trabajos futuros

La aplicación cumple todos los objetivos que se especificaron al inicio de esta memoria, pero aun así no significa que no se pueda añadir más funcionalidad. La aplicación es útil y potente para consultas simples sin embargo no está diseñada para procesar consultas con operación de conjunto ni tampoco consultas paramétricas. Dicho esto podemos decir que las siguientes funcionalidades harían a la aplicación mucho más interesante:

- Capacidad de procesar consultas con operación de conjunto.
- Capacidad de procesar consultas paramétricas.
- Que la aplicación sea capaz de calcular el coste total desde que se lanzó la consulta hasta que presenta los datos por pantalla.

Aparte de estos tres puntos, otra utilidad que haría a la aplicación mucho más cercana al usuario sería la posibilidad de que el usuario pueda elegir qué criterios aplicar según el estado actual del usuario, alertando al usuario de si su elección ha sido acertada o no.

CAPÍTULO 7: BIBLIOGRAFÍA

BIBLIOGRAFÍA

Acosta, Laura. **“Metodología de procesamiento de consultas distribuidas”.** https://goo.gl/T8CSgn

Cisneros González, José Luis. **“Panorama sobre base de datos. Un enfoque práctico”**. goo.gl/PqjZce

Cervartes, Humberto**. “Arquitectura Software”**.https://sg.com.mx/revista/27/arquitectura-software

Chinchilla Arley, Ricardo**. “Fragmentación de datos en bases de datos distribuidas”**

Hobbs, Ashton**. "Aprendiendo programación para bases de datos con JDBC en 21 días"**. Prentice Hall, 1998.

Korth H.F., Silberschatz A. **"Fundamentos de bases de datos"** (5ª Edición). McGraw-Hill, (2006).

M. Kroenke, David. **“Procesamiento de bases de datos: fundamentos, diseño e implementación”.** https://goo.gl/QGAxzv

Oswaldo Achury, Rodrigo. “**Introducción a La Programación Algoritmos y Dfd**”.https://es.scribd.com/doc/12390775/IntroducciOn-a-La-Programacion-Algoritmos-y-Dfd

Kevin Loney. **“Oracle Database 11g: The complete reference”**. McGraw-Hill, (2009)

Roger S. Pressman. **"Ingeniería del Software. Un enfoque práctico"** (7ª Edición). McGraw-Hill, 2014.

CAPÍTULO 8: ANEXOS

Manual de Usuario

En las siguientes páginas pretendemos establecer las líneas generales de utilización de la aplicación realizada como proyecto fin de carrera. Como sabemos, la aplicación consiste en transformar una consulta global en una consulta distribuida que acceda a los fragmentos. Por tanto, el usuario debe introducir una consulta en formato SQL que será procesada y la aplicación devolverá la consulta SQL en algebra relacional junto con su árbol algebraico. Junto al árbol aparecen dos botones que su función será mostrar el resultado del árbol al aplicarle el criterio correspondiente según el estado del árbol en el momento del pulsar el botón, los botones son “Paso Anterior” y “Siguiente Paso”, que devolverá el árbol a un estado anterior o le aplicará el criterio siguiente, respectivamente.

Introducción

Para ejecutar la aplicación, hay que cargar la página Web principal de la misma, en la que se muestra lo siguiente:

Figura 8.1 Ventana principal

En la interfaz principal podemos distinguir dos elementos significativos: un menú de navegación y un campo de texto, a continuación se detallarán ambos elementos:

Menú de navegación

El menú de navegación está compuesto por tres enlaces, estos son:

- Principal: Este enlace te dirige a la página principal de la aplicación, la misma que aparece en la imagen anterior.

- Apuntes: Al pulsar este enlace se desglosará un pequeño menú que contienen 3 enlaces más, cada uno corresponde a un PDF visto en la asignatura “Base de datos distribuidas”. Si se pulsa uno de estos enlaces se abrirá el PDF en otra pestaña del navegador.

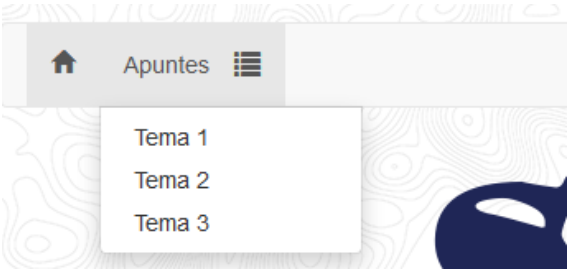


Figura 8.2 Menú principal

- Ayuda: Al igual que el enlace anterior, al pulsarlo se desglosará otro menú con dos enlaces, un enlace hacia el manual de usuario de la aplicación y otro hacia un manual de instalación de la aplicación.

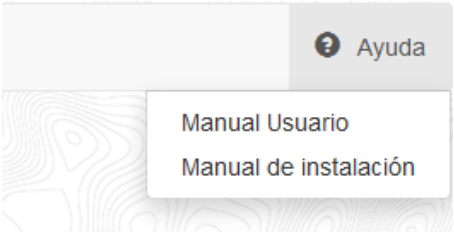


Figura 8.3 Menú ayuda

Campo de texto

Es el elemento en el que el usuario escribe la consulta SQL que quiere lanzar al sistema para que este la procese y le muestre el resultado correspondiente. Por ejemplo, escribimos la consulta “select s.pnum from suministro s,proveedor p where s.pnum = p.pnum” en el campo de texto y la lanzamos pulsando el botón “Enviar”.

Figura 8.4 Campo de texto

Si todo ha ido correctamente, la aplicación nos llevará a otra página donde aparecerá un mensaje que la consulta se ha podido procesar correctamente. En caso contrario, nos dirigirá a la página principal con un mensaje de error detallando cual ha sido la causa del problema.

Figura 8.5 Consulta realizada correctamente

Figura 8.6 Consulta incorrecta

Suponiendo que todo ha ido bien, podemos apreciar que debajo del campo de texto aparece la consulta anteriormente introducida en álgebra relacional. Para el ejemplo expuesto obtenemos la siguiente.

Figura 8.7

Para mostrar el árbol algebraico de la consulta tenemos que pulsar el botón “Empezar”. Al pulsarlo obtenemos el siguiente árbol.

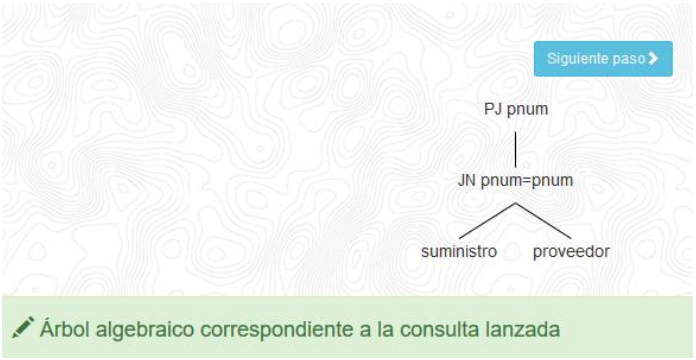


Figura 8.8 Árbol algebraico correspondiente a la consulta lanzada

Como se puede ver, aparecen tres elementos nuevos: Un botón “Siguiente paso”, el árbol algebraico y un mensaje informativo. Al pulsar el botón el árbol algebraico cambiará a un nuevo árbol, resultado de haberle aplicado el criterio correspondiente según la estructura que presente el árbol. El mensaje informativo está asociado al estado en el que se encuentra el árbol y cambiara a la vez que cambia este.

Este es el estado del sistema al pulsar el botón “Siguiente Paso”. Aparece un nuevo botón “Paso anterior” que al pulsarlo devolverá el árbol al estado en el que se encontraba anteriormente. Abajo el mensaje informativo ha cambiado y en él se muestra el criterio que se ha usado para transformar el árbol.

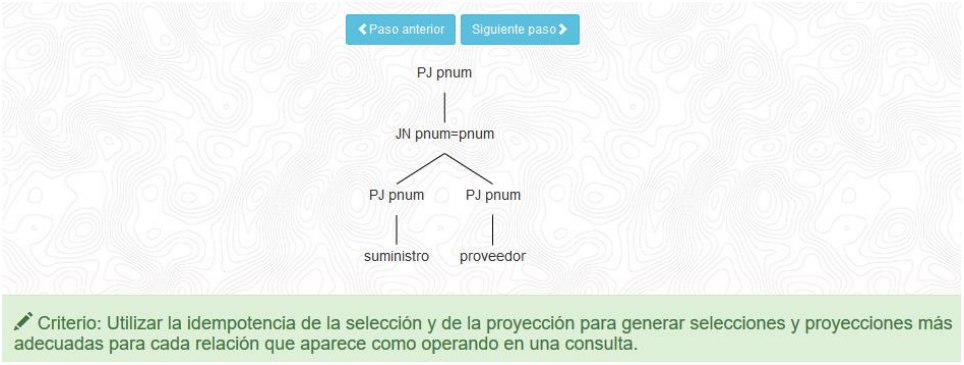


Figura 8.9 Funcionamiento botón “Siguiente paso” y “Paso anterior”

El botón “Siguiente paso” aparecerá hasta que se obtenga el árbol final. Sabemos que hemos llegado a ese estado cuando vemos un nuevo elemento con el contenido “Árbol final” junto al surge un nuevo botón “Mostrar consulta distribuida”.

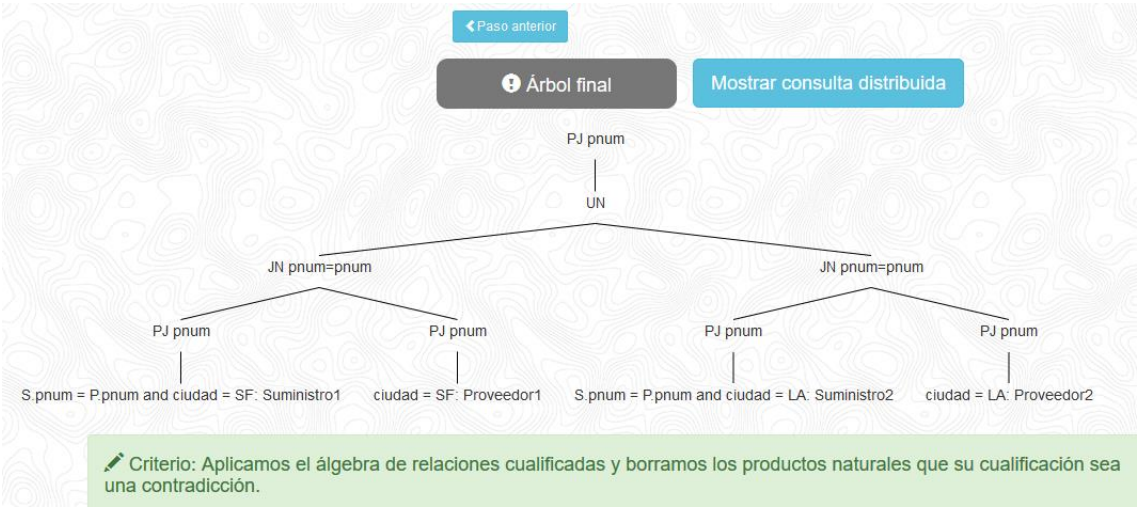


Figura 8.10 Funcionamiento botón “Siguiente paso” y “Paso anterior”

Si pulsamos en el nuevo botón llegaremos al estado final del sistema, en el que nos muestra la consulta SQL lanzada en su equivalencia distribuida. Para la consulta lanzada obtenemos la siguiente distribuida.

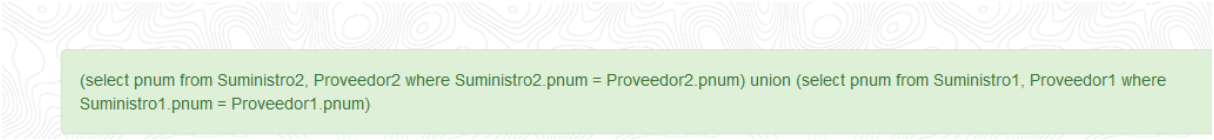


Figura 8.11 Consulta SQL lanzada en su equivalencia distribuida

Manual de Instalación

Introducción

Para poder ejecutar el proyecto “Procesamiento de consultas distribuidas” es necesario disponer del siguiente software:

- ✓ El entorno *NetBeans 8.0x* o cualquier otro que compile proyectos en java y sea compatible con apache tomcat. En esta guía se expondrá como instalar apache con NetBeans.
- ✓ *Apache Tomcat 7* o superior (incorporado en NetBeans).
- ✓ *Gestor de base de datos Oracle.*

Instalando NetBeans IDE

Como primer paso nos dirigimos a la página oficial de Netbeans para descargar el instalador. Una vez en esa ventana, seleccionamos el idioma que queremos que tenga nuestro Entorno de Desarrollo. Para este caso escogemos "Español".

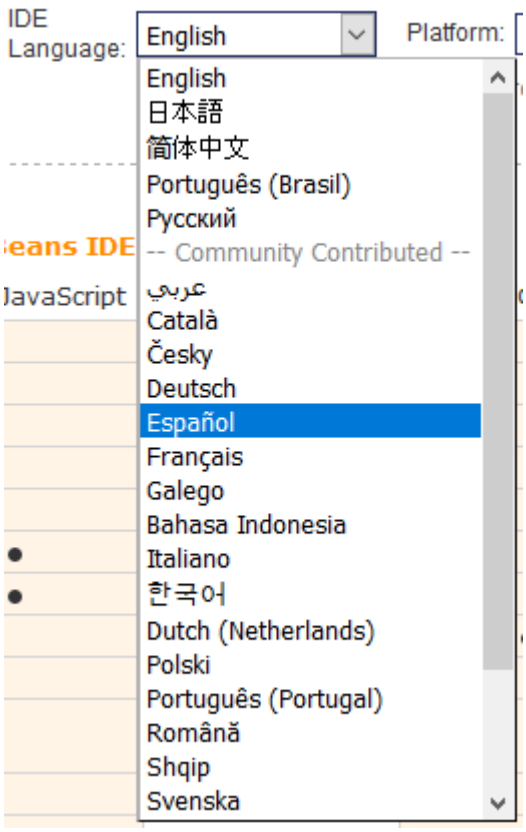


Figura 8.12 Descarga *NetBeans IDE* Paso 1

Seguidamente, seleccionamos la plataforma y/o sistema operativo con el cual cuenta el ordenador donde instalaremos *NetBeans IDE*, en esta ocasión "Windows".

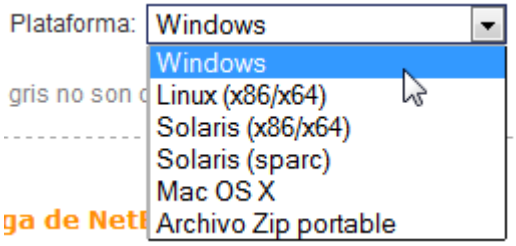


Figura 8.13 Descarga *NetBeans IDE* Paso 2

Ahora, una parte muy importante es elegir el paquete que vamos a descargar, y esto dependerá sobre qué lenguajes trabajamos o qué tipo de proyectos realizamos, ya que algunos paquetes no contienen ciertas tecnologías. En nuestro caso, solo vamos a trabajar con Java así que seleccionamos un paquete que lo contenga.

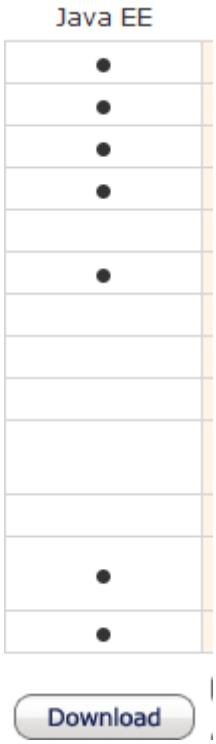


Figura 8.14 Descarga *NetBeans IDE* Paso 3

Después de haber descargado el archivo ejecutable, procederemos a descargar *Java Development Kit (JDK)* ya que, sin este último, no podemos instalar *NetBeans*. Cuando nos encontremos en la página solicitada, nos dirigimos a la sección "Java SE 8" y damos clic en el botón "Download".

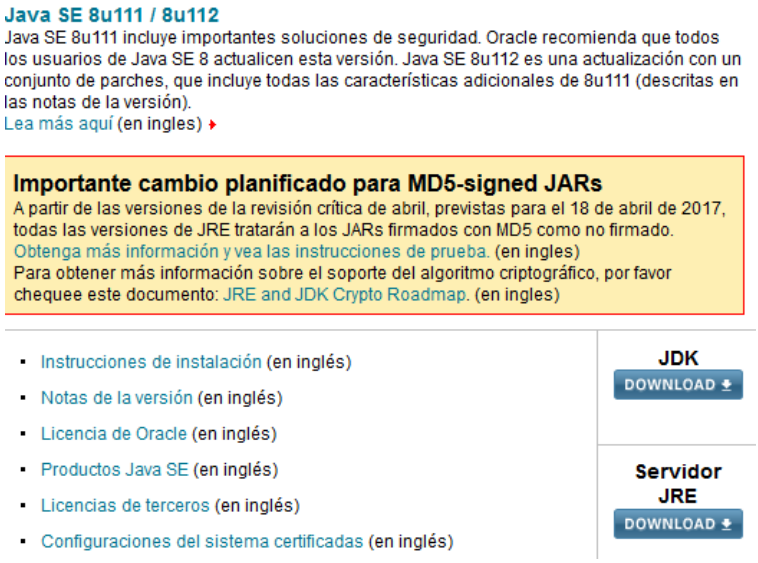


Figura 8.15 Descarga *Java Development Kit (JDK*)

Una vez que tenemos los dos archivos, nos disponemos a instalarlos. Primero, empezamos por la instalación del *JDK*, lo cual es muy simple ya que solamente seleccionamos la ruta de instalación y pulsamos en “siguiente”.

Ahora, ya podemos instalar *NetBeans*, ejecutamos el instalador y se abrirá una ventana en la cual se informa que el instalador se está configurando; una vez se termine dicho procedimiento, podremos ver la ventana inicial para llevar a cabo la instalación.

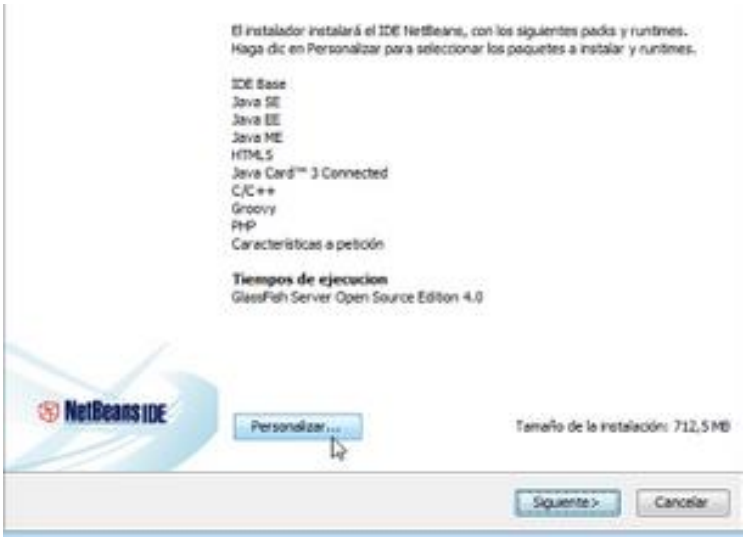


Figura 8.16 Instalando NetBeans IDE Paso 1

En este apartado se nos preguntará si deseamos instalar "JUnit", una herramienta para hacer pruebas controladas a nuestras aplicaciones Java. En esta ocasión, escogeremos la opción "No instalar JUnit" y hacemos clic en "Siguiente".

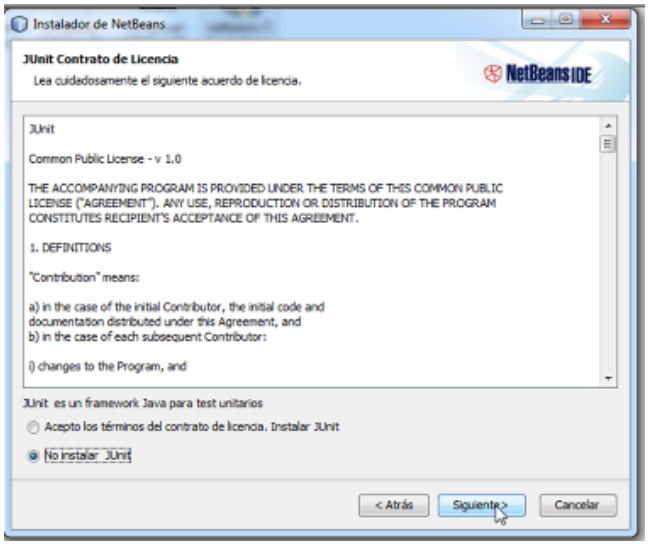


Figura 8.17 Instalando *NetBeans IDE* Paso 2

En la ventana siguiente, se mostrará la ruta de instalación *NetBeans*, la cual podremos modificar dando clic en el botón "Examinar..." y escogiendo la carpeta que deseamos. En este caso, hay que tener en cuenta que dejaremos la ruta que está por defecto.

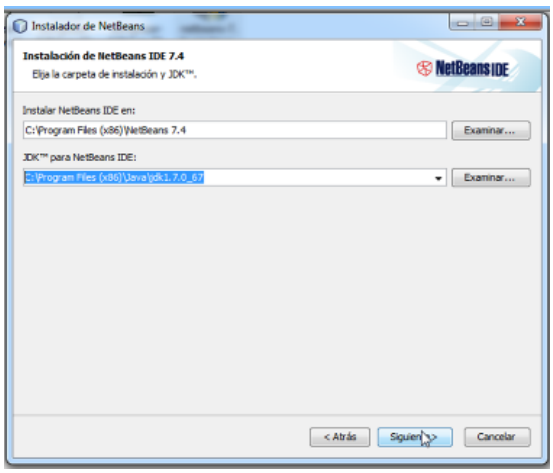


Figura 8.18 Instalando *NetBeans* IDE Paso 3

En la siguiente pantalla, nos preguntará dónde queremos que se instale el servidor de aplicaciones “GlassFish”.

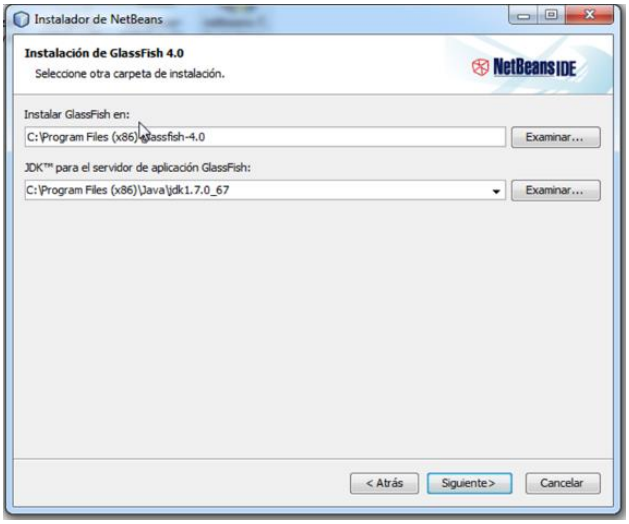


Figura 8.19 Instalando *NetBeans IDE* Paso 4

Como último paso y muy importante, elegimos donde instalar *Apache Tomcat*, necesario para ejecutar nuestro proyecto web de manera local. Este paso solo aparecerá si se ha seleccionado el paquete de instalación de *NetBeans* “all”.

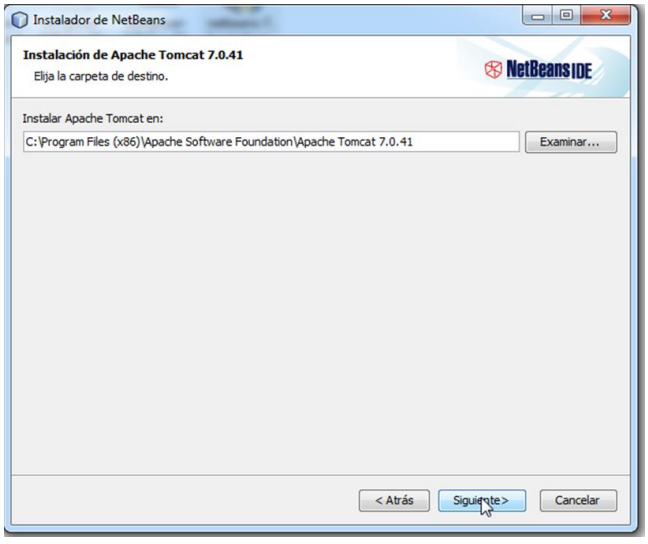


Figura 8.20 Instalación *Apache Tomcat*

En la ventana final aparecerá un resumen de los componentes a instalar junto con su tamaño, daremos a instalar para completar la instalación.

Añadiendo Apache Tomcat a NetBeans

Si no se ha elegido el paquete de instalación que contenga *Apache Tomcat*, por tanto no disponemos del servidor web y tenemos que instalarlo. De este modo, recurrimos a la página de instalación y elegimos el archivo según nuestra plataforma.

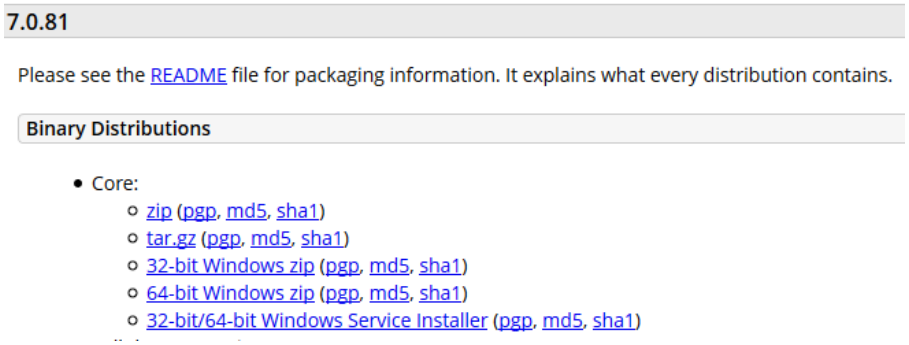


Figura 8.21 Descarga *Apache Tomcat*

La instalación solo requiere elegir el directorio donde se descomprimirá los archivos. Una vez hecho esto, nos dirigimos a *NetBeans*. En la sección “prestaciones”, pulsamos el botón derecho y elegimos “agregar servidor” para añadir uno nuevo.

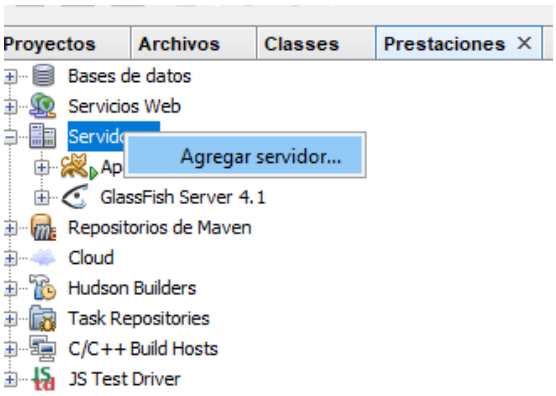


Figura 8.22 Agregar servidor en *NetBeans*

Elegimos *Apache Tomcat* y pulsamos en siguiente.

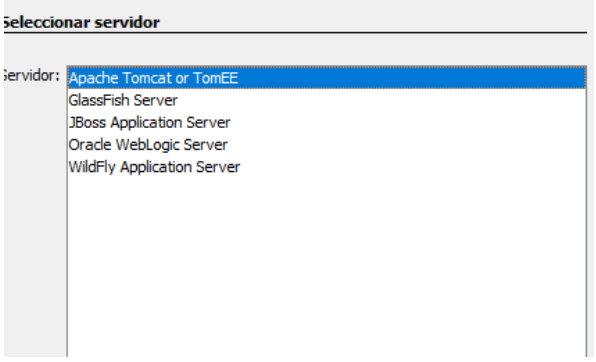


Figura 8.23 Agregar servidor *Apache Tomcat* en *NetBeans*

Especificamos la ruta donde se encuentra *Apache Tomcat*, y tenemos además la opción de crear un usuario para administrar este último. Por defecto, dejamos que la carpeta de configuración esté dentro de *NetBeans*.

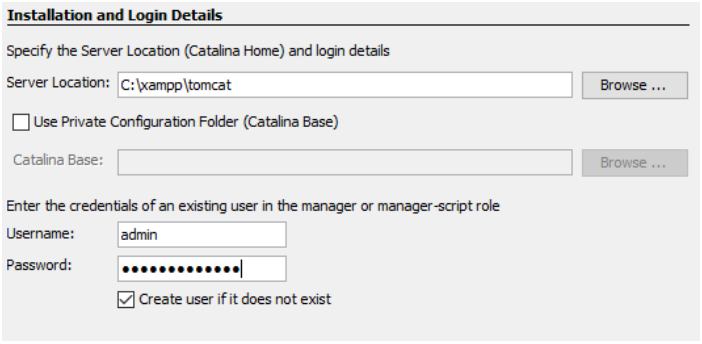


Figura 8.24 Agregar servidor *Apache Tomcat* en *NetBeans*

Una vez hecho esto, dispondremos del servidor *Apache Tomcat* en nuestro *NetBeans*.

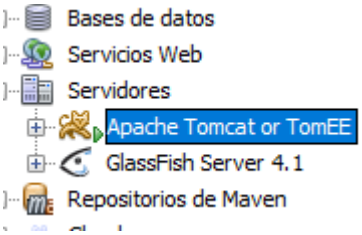


Figura 8.25 *Apache Tomcat* en *NetBeans*

Instalando un Gestor de base de datos.

Para el almacenamiento de los datos se ha usado el Gestor de base de datos *Oracle*. Nos vamos a la página oficial, elegimos la versión deseada y bajamos el instalador.

Ejecutamos el instalador y obtenemos la siguiente ventana, pulsamos en siguiente para continuar.

Figura 8.26 Instalación *Oracle* Paso 1

En esta ventana, el instalador nos muestra la carpeta de destino para la instalación por defecto así como el espacio necesario y el disponible. Puede ser cambiado pulsando en “examinar”.

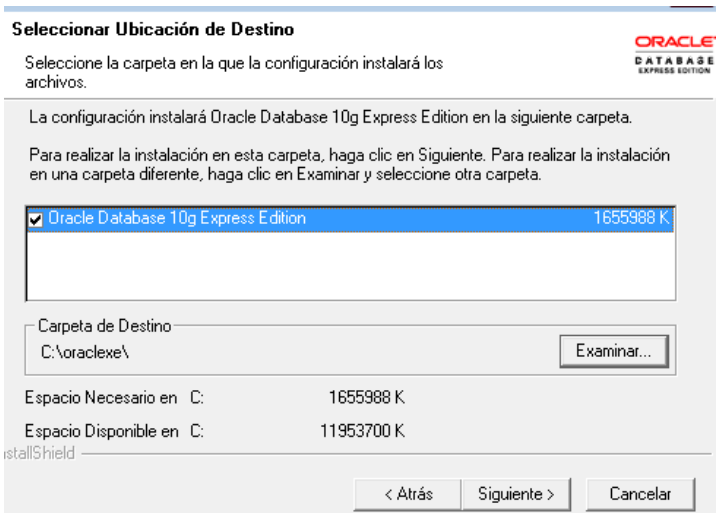


Figura 8.27 Instalación *Oracle* Paso 2

En la siguiente pestaña, elegimos el puerto por defecto; en este caso pondremos “2031”, y elegimos siguiente. También podemos añadir una contraseña al usuario SYS para poder conectarnos con privilegios de administrador.

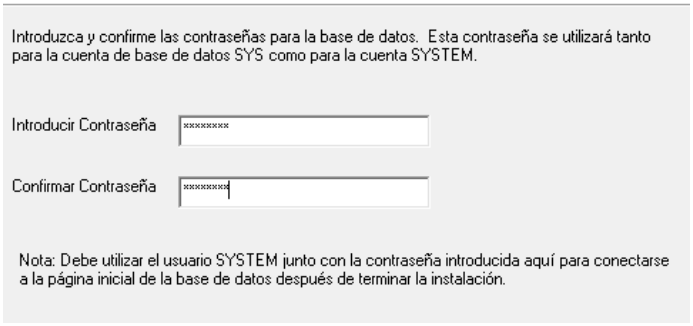


Figura 8.28 Instalación *Oracle* Paso 3

Como último paso, obtendremos un resumen de lo configurado en el instalador. Pulsamos en “instalar” para terminar con la instalación.

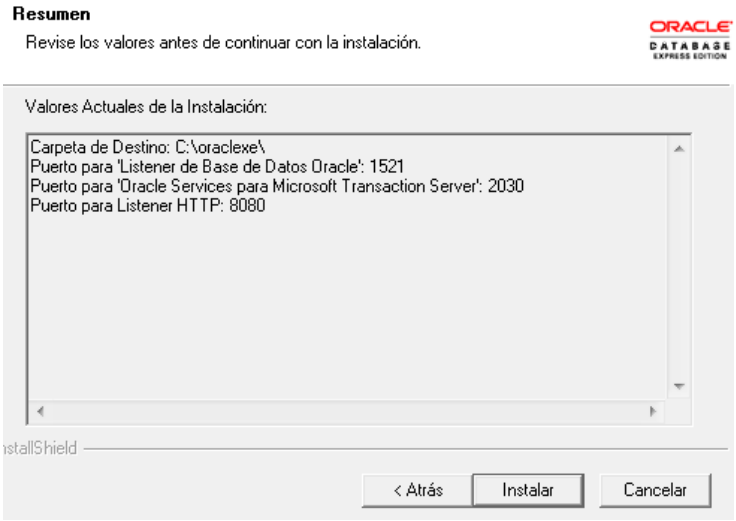


Figura 8.29 Instalación *Oracle* Paso 4

Una vez finalizado, ya podremos ser capaces de crear una base de datos nueva, y añadir nuestras tablas.