Universidad del Bío-Bío, Chile
Facultad de Ciencias Empresariales
Departamento de Sistemas de Información

Aplicación web para fragmentación horizontal y vertical de bases de datos relacionales utilizando algoritmos COMMIN y BEA

Proyecto presentado en conformidad con los requisitos para optar al título de
Ingeniería Civil en Informática

Por:
Ernesto Santander Anabalón
Profesor Guía:
Dra. Mónica Caniupán Marileo

Marzo de 2019
Concepción - Chile

Resumen

Este proyecto se presenta para dar conformidad a los requisitos exigidos por la Universidad de Bío-Bío en el proceso de titulación para a la carrera de Ingeniería Civil en Informática. El proyecto titulado "Aplicación WEB para fragmentación horizontal y vertical de bases de datos relacionales utilizando algoritmos COMMIN y BEA".

Una fragmentación en una base de datos, consiste en dividir una relación en múltiples partes, cada parte es llamada fragmento y éstos pueden ser almacenados en distintos sitios de una red de computadores, los fragmentos generados deben posibilitar la reconstrucción de la relación original. El por qué fragmentar es posible entenderlo de la siguiente manera: generalmente las aplicaciones trabajan con vistas y no con la relación completa, por lo que es natural trabajar con subrelaciones, además, como dichas aplicaciones pueden encontrarse en diferentes sitios y acceder solo a cierta porción de los datos, la solución óptima es fragmentar y distribuir la información, la información al estar cerca de donde es más frecuentemente usada permite aumentar la eficiencia en el acceso de datos.

Este proyecto desarrolla una aplicación WEB basada en lenguaje de programación JAVA que permite realizar fragmentación horizontal aplicando el algoritmo COMMIN y fragmentación vertical utilizando el algoritmo BEA. En una fragmentación horizontal se generan varios fragmentos de una relación, basado en una condición de selección sobre la relación, manteniendo el esquema original de la relación. La fragmentación vertical consiste en dividir una relación en un conjunto de relaciones más pequeñas mediante el uso de los atributos de esa relación.

Agradecimientos

Antes de nada, agradecer a mi profesora guía, Dra. Mónica Caniupán por su paciencia y ayuda en el desarrollo de este proyecto, además de permitirme desarrollar mi proyecto de título con un tema propuesto por ella.

En segundo lugar, agradecer a las personas que me apoyaron en el desarrollo de este proyecto durante un año difícil personalmente y de grandes cambios, dedicado especialmente a mi familia y amigos.

Índice de General

Índice de Figuras

Índice de Tablas

Índice de Algoritmos

Capítulo 1

Introducción

Las empresas e instituciones han tenido la necesidad de almacenar datos tales como datos de los clientes, productos, ventas, etc. Esta información es almacenada generalmente en bases de datos. Una *base de datos* se define como un conjunto de datos organizados y relacionados entre sí, que pueden ser consultados por usuarios para obtener información relevante. En una *base de datos relacional* los datos están organizados en un conjunto de relaciones formalmente definidas mediante el esquema relacional (Ramakrishnan, 2007).

El aumento del tráfico de datos y la necesidad de las empresas de obtener datos desde distintos puntos geográficos dificulta la posibilidad de tener una base datos centralizada. Esto impulsó el surgimiento de los *Sistemas de Bases de Datos Distribuidas* (SBDD) (Özsu y Valduriez, 2011).

Un sistema de bases de datos distribuidos es la combinación de dos conceptos, en primer lugar, los sistemas de bases de datos que buscan integrar las operaciones de las empresas y tener un control centralizado de la información y las redes de comunicación que buscan descentralizar el funcionamiento de las operaciones. Por lo tanto, el objetivo de un SBDD es integrar ambas partes, en términos simples es mantener un único esquema de base de datos, pero que se encuentra distribuida físicamente en distintos sitios y están conectadas entre sí (Özsu y Valduriez, 2011).

El modelo fragmentado, es aquel donde las relaciones se fragmentan (dividen) en varios fragmentos y cada fragmento se almacena en un sitio diferente (aunque se pueden mantener copias de fragmentos). Para realizar una fragmentación existen tres métodos: (i) fragmentación horizontal, donde cada fragmento es un subconjunto de la relación (tuplas), y se lleva a cabo mediante la operación de selección del algebra relacional, (ii) fragmentación vertical que consiste en dividir la relación en un conjunto de relaciones más pequeñas, para así optimizar los tiempos de ejecución de las aplicaciones, (iii) fragmentación híbrida, que corresponde a la combinación de ambos tipos de fragmentaciones previamente mencionadas.

Entre las ventajas de fragmentar una base de datos se distinguen: evita un alto volumen de acceso remoto innecesario, pues cada aplicación situada en sitios diferentes accede a sus

fragmentos asignados, además la fragmentación de relaciones produce accesos paralelos al dividir una consulta en subconsultas dirigidas a diferentes fragmentos (Özsu y Valduriez, 2011).

El sistema desarrollado en este proyecto permite realizar fragmentaciones horizontales y verticales de bases de datos relacionales. Para ello se implementan tanto la fragmentación horizontal a través del uso e implementación del Algoritmo COMMIN y la fragmentación vertical utilizando el Algoritmo BEA, donde el usuario establece las condiciones para realizar las fragmentaciones (Özsu y Valduriez, 2011).

El proyecto esta documentado en una serie de capítulos organizados de la siguiente manera:

En el Capítulo 2 se describe el objetivo general, los objetivos específicos, los alcances y limites de este proyecto, metodología de desarrollo, así como siglas y abreviaciones utilizadas a lo largo de este informe.

El Capítulo 3 se describe los conceptos preliminares necesarios para entender el funcionamiento del sistema desarrollado, en el se describen conceptos como bases de datos relacionales, fragmentaciones y tipos de fragmentación.

El Capítulo 4 se describe los algoritmos utilizados, tanto el algoritmo COMMIN como el algoritmo BEA, así como los pasos previos necesarios para utilizar estos algoritmos.

En el Capítulo 5 se presenta el sistema WEB, su funcionamiento mediante ejemplos y la definición de la interfaz del sistema.

En el Capítulo 6 se detalla el diseño de las pruebas que realizadas al sistema y el resultado esperado de cada función a probar, con la finalidad de demostrar su correcto funcionamiento.

Para finalizar el Capítulo 7 presenta las conclusiones del proyecto y posibles trabajos futuros.

Capítulo 2

Objetivos y Definición del proyecto

En este capítulo se presentan los objetivos generales y específicos asociados al proyecto, además de los alcances y límites del mismo.

2.1. Objetivo General

Implementar una aplicación WEB que permita fragmentar una base de datos relacional de manera horizontal utilizando el algoritmo COMMIN y de manera vertical a través del algoritmo BEA.

2.2. Objetivos Específicos

Los objetivos específicos del proyecto son los siguientes:

1. Analizar los algoritmos COMMIN y BEA para comprender su funcionamiento.
2. Implementar los algoritmos COMMIN y BEA en lenguaje JAVA.
3. Diseñar la estructura e interfaz de la aplicación WEB.
4. Desarrollar aplicación WEB basada en JAVA.
5. Generar bases de datos relacionales para prueba y análisis.
6. Realizar pruebas de aplicación.

2.3. Ambiente de Ingeniería de Software

2.3.1. Metodología de Desarrollo

La metodología a utilizar es el Modelo Incremental, debido a que la aplicación a desarrollar contiene la implementación de dos algoritmos independientes, es posible dividir en

módulos su desarrollo y proporciona las ventajas de los modelos evolutivos al poder obtener una retroalimentación. El Modelo Incremental es un proceso de desarrollo de software donde los requisitos se dividen en múltiples módulos de ciclo del desarrollo de software (Pressman, 2010).

Cada iteración pasa por las fases de análisis, diseño, desarrollo y prueba. Cada versión posterior del sistema agrega funciones a la versión anterior hasta que se haya implementado toda la funcionalidad diseñada. En la Figura 2.1 se visualiza el proceso de desarrollo incremental.

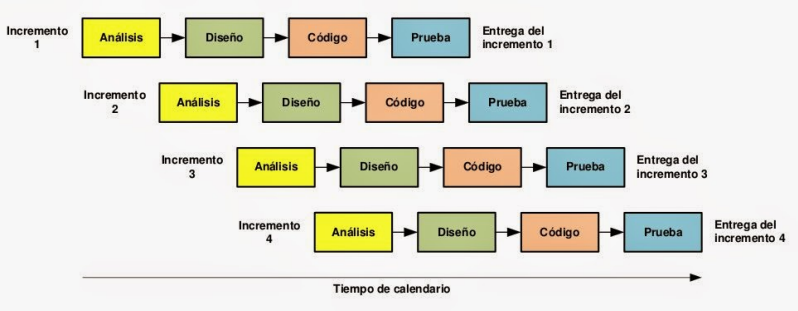


Figura 2.1: Representación visual del modelo incremental

2.3.2. Técnicas y Notaciones

Técnicas

- Modelo Vista Controlador (MVC).
  El patrón MVC es un paradigma o patrón de desarrollo que divide las partes que conforman una aplicación en el modelo, las vistas y los controladores, permitiendo la implementación por separado de cada elemento, garantizando así la actualización y mantenimiento del software (Romero y Díaz, 2012).

  El modelo es el objeto que representa los datos del programa, es el núcleo del sistema y contiene los datos de este. La vista maneja la presentación visual de los datos representados por el modelo y genera una representación visual del modelo y muestra los datos al usuario, y por lo general, interactúa con el controlador. El controlador proporciona un significado a las órdenes del usuario, actuando sobre los datos proporcionados por el Modelo, centra toda la interacción entre la vista y el modelo (Romero y Díaz, 2012). En la Figura 2.2 se visualiza los procesos en un Modelo Vista Controlador.

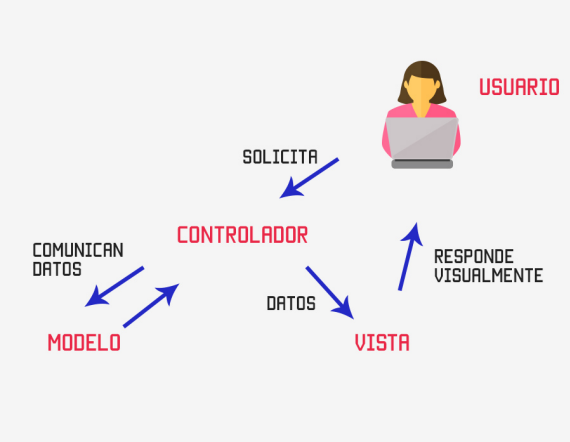


Figura 2.2: Representación del modelo vista controlador

Notaciones

- BEA : Bond Energy Algorithm.
- COMMIN : Conjunto de predicados simples y minimales.
- SGBD : Sistema gestor de bases de datos.
- IDE : Entorno de desarrollo integrado.
- BDR : Base de datos relacional.
- AA : Matriz de afinidad.
- FAC : Frecuencia de acceso.
- CA : Matriz de afinidad agrupada.
- JSP : JavaServer Pages.
- API : Application Programming Interface.
- SQL : Structured Query Language.

2.4. Alcances y Límites

2.4.1. Alcances

- Este proyecto contempla bases de datos relaciones, específicamente bases de datos implementadas en el sistema de gestión de bases de datos PostgreSQL[1].
- El sistema genera automáticamente la fragmentación de una relación seleccionada por el usuario quien, además, define las condiciones de dicha fragmentación.

2.4.2. Límites

- El sistema es posible extenderlo a bases de datos basados en otros gestores de bases de datos.
- Solo se realiza la fragmentación de una relación a la vez y de dicha fragmentación es posible que se fragmenten las relaciones relacionadas a la relación original fragmentada.
- Solo se realiza la fragmentación horizontal y vertical, pero no la fragmentación híbrida.

[1] https://www.postgresql.org/

Capítulo 3

Conceptos Preliminares

En este capítulo se explican los conceptos básicos sobre las bases de datos relacionales, tipos de fragmentaciones y los algoritmos que utilizados en este proyecto. Primero, en la Sección 3.1 se explican las bases de las bases de datos. Luego la Sección 3.4 explica los tipos de fragmentación existentes.

3.1. Base de Datos Relacional

Una base de datos es un conjunto de datos lógicamente coherentes, que describen situaciones del mundo real (Ramakrishnan, 2007). Los datos se refieren a hechos conocidos que se pueden registrar y tienen un significado implícito, por ejemplo, nombres, números telefónicos, precios, etc.

Una base de datos relacional es un conjunto de relaciones que se basa en el modelo de datos relacional donde la estructura principal es la *relación*. Cada relación posee un esquema y una instancia de relación. El esquema especifica el nombre de la relación, sus atributos y el dominio de cada atributo. Cada instancia de una relación es un conjunto de tuplas, también denominadas registros, en el que cada tupla tiene el mismo número de campos definidos por el esquema de la relación (Ramakrishnan, 2007).

Para entender el esquema de una base de datos relacional, considere la relación *productos* (Tabla 3.1) del Ejemplo 3.1. El esquema considera los siguientes atributos: *id_producto, nombre_producto, precio_producto*, para cada atributo existe un conjunto de valores permitidos llamados dominios. El atributo *nombre_producto* tiene como dominio todos los nombres de los productos existentes.

Los valores de los dominios están definidos por el tipo de dato al cuál pertenece el dominio, entre los tipos básicos de dominios definidos en SQL son:

- **char**(n): Una cadena de caracteres de longitud fija, con una longitud n especificada por el usuario.
- **varchar**(n): Una cadena de caracteres de longitud variable con una longitud máxima n especificada por el usuario.

- **integer**: Un entero (un subconjunto finito de los enteros dependiente de la máquina).
- **smallint**: Un entero pequeño.
- **numeric**(p, d): Un número de coma fija, cuya precisión la especifica el usuario. El número está formado por p dígitos (más el signo), y de esos p dígitos, d pertenecen a la parte decimal.
- **real, double precision**: Números de coma flotante y números de coma flotante de doble precisión, con precisión dependiente de la máquina.
- **float**(n): Un número de coma flotante cuya precisión es, al menos, de n dígitos.

Ejemplo 3.1 En la Tabla 3.1 los atributos *id_producto y nombre_producto* son dominios del tipo *varchar*, mientras que el atributo *precio_producto* es un tipo de dominio *integer*.

id_producto	nombre_producto	precio_producto
P01	Jabón	1000
P02	Jugo	150
P03	Manzana	100
P04	Té	2000

Tabla 3.1: Relación productos.

3.2. Sistema de Gestión de Base de Datos (SGBD)

Un Sistemas de Gestión de Base de Datos es un software diseñado para almacenar y administrar grandes volúmenes de datos (Pressman, 2010). Las ventajas de un SGBD son:

- Independencia con respecto a los datos: ofrece una vista abstracta de los datos y oculta los detalles de su almacenamiento.
- Acceso eficiente a los datos: Los SGBD emplean gran variedad de técnicas para almacenar y recuperar los datos de manera eficiente.
- Integridad y seguridad de los datos: Cumple las restricciones de integridad necesarias, además de que se cumplan los controles de acceso que determinan los datos que son visibles para las diferentes clases de usuarios.
- Administración de los datos: Los SGBD permiten una administración centralizada de los datos sin importar la cantidad de usuarios que utilicen esos datos.

- Acceso concurrente y recuperación en caso de fallo: Los SGBD programan los accesos concurrentes a los datos de tal manera que los usuarios puedan creer que solo tiene acceso a los datos un usuario a la vez y también protegen a los usuarios de fallos del sistema.
- Reducción del tiempo de desarrollo de las aplicaciones: Soportan funciones comunes con un gran numero aplicaciones que tienen acceso a los datos del SGBD.

Entre las principales SGBD y más populares, se encuentran las siguientes:

- MySql: Base de datos basada en un servidor, la principal ventaja es su rapidez, pero no se recomienda para grandes volúmenes de datos.
- Oracle: Es una base de datos poderosa, desarrollada por la Corporación Oracle, al ser un software considerado caro, es mayormente usado por grandes empresas, destacan su seguridad y estabilidad, además de confiable para el manejo de grandes cantidades de datos.
- PostgreSQL: Es un potente sistema de base de datos relacional de objetos de código abierto. Al igual que la base de datos Oracle, es recomendada para el manejo de grandes cantidades de información.
- Microsoft SQL Server: es una base de datos potente desarrollada por Microsoft. Se utiliza para manejar grandes volúmenes de informaciones.

3.3. Sistemas de Bases de Datos Distribuidas DDBS

Una base de datos distribuidas se define como una colección de múltiples bases de datos lógicamente interrelacionados distribuidos sobre una red de computadores. Un sistema de base de datos distribuidas es un software que permite la gestión de bases de datos distribuidas y realiza la distribución transparente a los usuarios (Özsu y Valduriez, 2011).

Un DDBS no es una colección de archivos que pueden ser accedidos desde cada nodo de una red de computadores, sino que cada sitio es un sistema de bases de datos en si mismo, pero los sitios están relacionados entre si con el fin de que los usuarios obtengan acceso a los datos almacenados en cualquier punto de la red de tal manera como si los datos estuvieran almacenados en el sitio del propio usuario. Estas bases de datos deben estar bajo una única estructura lógica global y deben ser accedidos mediante una interfaz común.

Un DDBS tampoco debe confundirse con un sistema donde, a pesar de la existencia de una red, la base de datos reside solo en un nodo de la red (Figura 3.1). Pues este caso no es diferente a acceder a una base de datos centralizada. La base de datos es gestionada por solo por un sistema de computador (Sitio 2 en Figura 3.1) y todas las solicitudes son dirigidas a él. Un DDBS es un entorno donde los datos estan distribuidos entre un numero de sitios de la red (Figura 3.2).

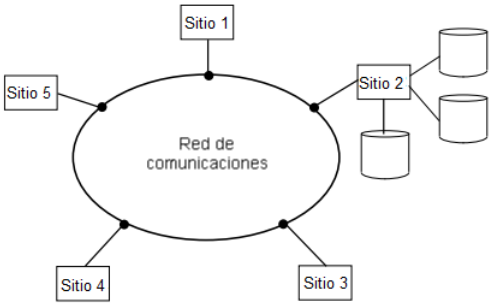


Figura 3.1: Base de datos almacenada en un solo sitio.

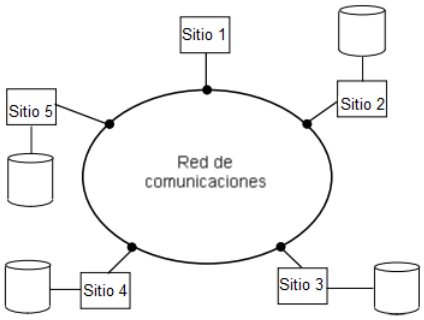


Figura 3.2: Sistema de base de datos distribuida correcta.

3.4. Tipos de Fragmentación

Una fragmentación es el resultado de una proceso de partición de una relación, en el cuál la relación se divide en fragmentos menores, tiene como objetivo buscar alternativas para dividir una relación en otras relaciones más pequeñas.

Existen tres tipos de fragmentación:

Fragmentación horizontal Se realiza sobre las tuplas de la relación, es decir que cada fragmento será un subconjunto de las tuplas de la relación. Los fragmentos se definen mediante una operación de selección. Su reconstrucción se realizará mediante la unión de los fragmentos componentes (Özsu y Valduriez, 2011).

Ejemplo 3.2 *Considere la relación* personas *como muestra la Tabla 3.2, donde se realiza una fragmentación horizontal basada en el genero de las personas, la aplicación* AP1 *utiliza a personas de sexo masculino y la aplicación* AP2 *selecciona a las personas de sexo femenino, se obtienen los fragmentos* personas1 *como se muestra en la Tabla 3.3 y* personas2 *en la Tabla 3.4 respectivamente.*

rut	nombre	apellido	edad	sexo
11111111-1	Francisco	Laguna	56	M
22222222-2	Paola	Sierralta	41	F
33333333-3	Felipe	Casas	30	M
44444444-4	Claudia	Vergara	28	F

Tabla 3.2: Relación *personas* utilizada en fragmentación horizontal.

rut	nombre	apellido	edad	sexo
11111111-1	Francisco	Laguna	56	M
33333333-3	Felipe	Casas	30	M

Tabla 3.3: Relación *personas*1 para aplicación AP1.

rut	nombre	apellido	edad	sexo
22222222-2	Paola	Sierralta	41	F
44444444-4	Claudia	Vergara	28	F

Tabla 3.4: Relación *personas*2 para aplicación AP2.

Fragmentación Vertical Consiste en dividir la relación en un conjunto de relaciones más pequeñas tal que algunas de las aplicaciones de usuario sólo hagan uso de un fragmento. Sobre este marco, una fragmentación óptima es aquella que produce un esquema de división que minimiza el tiempo de ejecución de las aplicaciones que emplean esos fragmentos. La fragmentación vertical se basa en los atributos de la relación para realizar la división, es decir: la subdivisión de atributos en grupos (Özsu y Valduriez, 2011).

Ejemplo 3.3 *Considere la relación personas de la Tabla 3.5. Debido a que existen dos aplicaciones* AP1 *y* AP2*, donde la aplicación* AP1 *solo utiliza el* nombre *y* apellido *de las personas, mientras que la aplicación* AP2 *solo necesita los datos de* edad *y* sexo*. Por lo tanto, se divide la relación* personas *en dos fragmentos personas*1 *representada en la Tabla 3.6 y personas*2 *en la Tabla 3.7,*

Es posible observar como los atributos de la relación original están repartidos entre las fragmentaciones obtenidas, también es posible notar que el atributo rut se encuentran en ambos fragmentos, esto se debe a que rut es la clave primaria *de la relación por lo que debe estar siempre presente.*

rut	nombre	apellido	edad	sexo
11111111-1	Francisco	Laguna	56	M
22222222-2	Paola	Sierralta	41	F
33333333-3	Felipe	Casas	30	M
44444444-4	Claudia	Vergara	28	F

Tabla 3.5: Relación *personas* utilizada para una fragmentación vertical.

rut	nombre	apellido
11111111-1	Francisco	Laguna
22222222-2	Paola	Sierralta
33333333-3	Felipe	Casas
44444444-4	Claudia	Vergara

Tabla 3.6: Relación *personas*1 para la aplicación *AP1*.

rut	edad	sexo
11111111-1	56	M
22222222-2	41	F
33333333-3	30	M
44444444-4	28	F

Tabla 3.7: Relación *personas*2 para la aplicación *AP2*.

Fragmentación Hibrida Esta fragmentación es una combinación de las dos anteriores, para generar este tipo de fragmentación, se debe generar tanto la fragmentación horizontal como la fragmentación vertical, una después de la otra.

Para que la fragmentación sea correcta, se deben cumplir los siguiente criterios (Özsu y Valduriez, 2011):

1. **Completitud** : La descomposición de una relación R en los fragmentos $R_1, R_2, ..., R_n$ es completa si y solamente si cada tupla en R se encuentra en algún fragmento $R_1, R_2, ..., R_n$.

2. **Reconstrucción** : Si la relación R se descompone en los fragmentos $R_1, R_2, ..., R_n$, entonces debe existir algún operador algebraico que permita reconstruir la relación original R.

3. **Fragmentos Disjuntos** : Si la relación R se descompone en los fragmentos $R_1, R_2, ..., R_n$, y la tupla se encuentra en R_j, entonces, no debe estar en ningún otro fragmento R_i donde $R_i <> R_j$.

Ejemplo 3.4 *Utilizando la relación* empleados *(Tabla 3.8) que muestra una lista de empleados, su nombre y sueldo. Se desea generar una lista de empleados con sueldos menores a* 400000 *que sera utilizada por una aplicación* AP1*, mientras que los empleados con sueldos mayores o iguales a* 400000 *serán utilizados por la aplicación* AP2*. Para ello se realiza una fragmentación horizontal a la relación empleados generando dos fragmentos:*

$empleados_1$ *, resultado de la selección* $\sigma_{salario\_empleado<400000}(empleados)$ *(Tabla 3.9).*

$empleados_2$ *, resultado de la selección* $\sigma_{salario\_empleado>=400000}(empleados)$ *(Tabla 3.10).*

id_empleado	nombre_empleado	salario_empleado
001	Pablo	350000
002	Ignacio	500000
003	Fernanda	400000
004	Valeria	365000
005	Rayen	435000

Tabla 3.8: Relación *empleados*.

La fragmentación horizontal obtenida es la siguiente:

id_empleado	nombre_empleado	salario_empleado
001	Pablo	350000
004	Valeria	365000

Tabla 3.9: Relación $empleados_1$ para aplicación AP1.

id_empleado	nombre_empleado	salario_empleado
002	Ignacio	500000
003	Fernanda	400000
005	Rayen	435000

Tabla 3.10: Relación $empleados_2$ para aplicación AP2.

Se comprueba que la fragmentación cumple con el criterio de Completitud, pues no se han perdidos tuplas de la relación empleados, también es posible la Reconstrucción

de la relación original empleados utilizando la unión entre los fragmentos $empleado_1$ *y* $empleado_2$*, por ultimo se comprueba que no existen Fragmentos Disjuntos pues no existen tuplas repetidas entre los fragmentos.*

Ejemplo 3.5 *La relación productos (Tabla 3.11) muestra el detalle de productos. La relación se fragmenta verticalmente para obtener el listado de stock del producto y la marca que sera utilizada por la aplicación* AP1 *y una segunda aplicación* AP2 *solo utiliza los nombres y precios de los productos. Esto genera dos fragmentaciones las cuales son:* $producto1$ *la cual contiene el id_producto, id_marca y id_stock, mientras el fragmento* $producto2$ *contiene los atributos restantes, se comprueba que se cumplen los 3 criterios para una fragmentación correcta, La Completitud se cumple ya que no hay atributos perdidos, la Recostrucción se puede realizar utilizando join entre los fragmentos y el criterio de Fragmentos disjuntos es valido ya que no existen atributos repetidos entre fragmentaciones, excepto por la clave primaria.*

id_producto	nombre_producto	precio_producto	marca_producto	stock_producto
P01	Jabón	1300	Nivea	30
P02	Té	2000	Supremo	16
P03	Leche	1200	COLUN	22
P04	Fideos	2000	Carozzi	8

Tabla 3.11: Relación *productos*.

id_producto	marca_producto	stock_producto
P01	Nivea	30
P02	Supremo	16
P03	COLUN	22
P02	2000	Carozzi

Tabla 3.12: Relación fragmento $productos1$ para la aplicación AP1.

id_producto	nombre_producto	precio_producto
P01	Jabón	1300
P02	Té	2000
P03	Leche	1200
P04	Fideos	2000

Tabla 3.13: Relación fragmento $productos2$ para la aplicación AP2.

En el siguiente Capítulo 4 se describe la definición y ejemplificación de los algoritmos COMMIN y BEA.

Capítulo 4

Algoritmos COMMIN y BEA

En este capítulo presenta los algoritmos utilizados para realizar las fragmentaciones. Para la fragmentación horizontal se implementó el Algoritmo COMMIN que se describe en la Sección 4.1, y para la fragmentación vertical se describe el Algoritmo BEA en la Sección 4.2.

4.1. Algoritmo COMMIN para la Fragmentación Horizontal

El Algoritmo COMMIN es un algoritmo iterativo que genera un conjunto de predicados *completo* y *minimal Pr'* dado un conjunto de predicados simples *Pr*.

Definición 4.1 *Dada una relación $R(A_1, A_2, ...A_n)$ donde A_i es un atributo definido sobre el dominio D_i. Un predicado simple p_j definido sobre R se define como:*

$$p_j : A_i \ \theta \ Valor \ \in \ D_i$$

Donde $\theta \ \in \ \{=, >, <, >=, <=, \neq\}$ y $Valor \in D_i$(Valor es un dominio de A_i).

Un conjunto de predicados simple es *completo* sí y solo sí dos tuplas del mismo fragmento tienen la misma probabilidad de ser accedidos. Se considera que un conjunto de predicados es *mínimal* si todos sus predicados son relevantes, los predicados son relevante cuando influye en la fragmentación. En términos simples el conjunto de predicados debe incluir solo atributos y condiciones usadas en las aplicaciones.

Ejemplo 4.1 *Considere la relación personas indicada en la Tabla 4.1 y las siguientes aplicaciones:*

1. *AP1: Busca personas mayores a 40 años.*
2. *AP2: Utiliza personas de nacionalidad chilena.*

ID	nombre	edad	nacionalidad
001	Jon	56	Chile
002	Leo	41	Argentina
003	Lili	30	Chile

Tabla 4.1: Relación *personas*.

*Utilizando las aplicaciones AP*1 *y AP*2*, un conjunto de predicados simples completo y minimal es:*

$$Pr : \{edad > 40, edad <= 40, nacionalidad = \text{“}Chile\text{”}\}$$

Pero si a ese conjunto se añade el predicado simple:

$$p_i : nombre = \text{“}Leo''$$

*Pr ya no es minimal por que el predicado añadido es irrelevante para las aplicaciones AP*1 *y AP*2*.*

Algoritmo 1: Algoritmo COMMIN

```
   Input: R: relación; Pr: conjunto de predicados simples completo y minimal
   Output: Pr': conjunto de predicados simples completo y minimal
1  begin
2      encontrar p_i ∈ Pr tal que p_i particione R, siempre que fragmente una relación o fragmento y sea
         accedida por al menos una aplicación;
3      Pr' ← p_i
4      Pr ← Pr - p_i
5      F ← f_i                    /*f_i es el fragmento generado a partir de p_i*/
6      repeat
7          encontrar un p_j ∈ Pr tal qué p_j particiona un f_k de Pr' siempre que fragmente una relación o
             fragmento y sea accedida por al menos una aplicación;
8          Pr' ← Pr' ∪ p_j;
9          Pr ← Pr − p_j;
10         F ← F ∪ f_j;
11         if ∃p_k ∈ Pr' que no es relevante then
12             Pr' ← Pr' - p_k;
13             F ← F - f_k;
14     until Pr' es completo y minimal;
```

El algoritmo COMMIN obtiene un conjunto de predicados simples y minimal. El Algoritmo COMMIN inicia en la línea 1 buscando un predicado p_i que fragmente la relación R verificando que particione la relación o y sea accedido por al menos una aplicación (línea 1). En la línea 2 agrega el predicado simple p_i encontrado al conjunto Pr', luego, el predicado p_i es eliminado del conjunto de predicados simple Pr (línea 3). La linea 4 indica como el fragmento f_i que fragmenta R al utilizar p_i es asignado a al conjunto de fragmentos F.

El ciclo **repeat - until**, agrega iterativamente predicados simples Pr al conjunto de predicados simples Pr' , el proceso es detallado a continuación:

El bucle repeat en la línea 5, busca un predicado (línea 6) p_j perteneciente a Pr tal que p_j particiona alguno de los fragmentos f_k de P'_r. En las líneas 7 y 8 al conjunto Pr' se agrega el predicado p_j seleccionado anteriormente y se elimina del conjunto Pr, en lalínea 9 se añade el fragmento f_i al conjunto F. Entre las líneas 10, 11 y 12, se realiza un proceso de descarte donde se eliminan los predicados simples que no sean relevantes y así generar un conjunto de predicados *minimal*, cuando es encontrado p_k a descartar, es eliminando tanto el predicado p_k y el fragmento mintermino f_k correspondiente a p_k de los conjuntos Pr' y F respectivamente.

Finalmente una vez terminado el algoritmo el conjunto de predicados simples Pr' es *completo* y *minimal* (Özsu y Valduriez, 2011).

Una vez generado el conjunto Pr', es necesario generar los predicados minterm a partir de Pr'. Un predicado minterm es una expresión lógica generada a partir de los predicados simples de un conjunto de predicados simples Pr.

Ejemplo 4.2 *Considere el siguiente predicado simple:*

$$Pr = \{edad > 40, nacionalidad = \text{“}Chile\text{”}\}$$

Es posible obtener los siguientes predicados minterm:

$m_1 :\ edad > 40 \wedge nacionalidad = \text{“}Chile\text{”}$
$m_2 :\ \neg(edad > 40) \wedge nacionalidad = \text{“}Chile\text{”}$
$m_3 :\ edad > 40 \wedge \neg(nacionalidad = \text{“}Chile\text{”})$
$m_4 :\ \neg(edad > 40) \wedge \neg(nacionalidad = \text{“}Chile\text{”})$

Estos predicados minterm determinan los fragmentos de la relación R generada. Una vez determinados los predicados minterm, es realizada un descarte de los predicados contradictorios que puedan existir.

Considerando $Pr' = \{nacionalidad = \text{“}Chile\text{”}, nacionalidad = \text{“}Argentina\text{”}\}$*, es posible generar los predicados minterm siguientes:*

$m_1 :\ nacionalidad = \text{“}Chile\text{”} \wedge nacionalidad = \text{“}Argentina\text{”}$
$m_2 :\ \neg nacionalidad = \text{“}Chile\text{”} \wedge nacionalidad = \text{“}Argentina\text{”}$
$m_3 :\ nacionalidad = \text{“}Chile\text{”} \wedge \neg(nacionalidad = \text{“}Argentina\text{”})$
$m_4 :\ \neg(nacionalidad = \text{“}Chile\text{”}) \wedge \neg(nacionalidad = \text{“}Argentina\text{”})$

Se observa que el predicado minterm $m1$ *es contradictorio pues según la relación personas (Tabla 4.1) el atributo nacionalidad no puede ser Chile y Argentina a la vez (la relación persona no considera la doble nacionalidad para este ejemplo).*

Algo a tener en cuenta, es que si existe una relación que contiene una clave foránea perteneciente a la relación fragmentada, dicha relación también debe ser fragmentada pues

genera conflicto entre las restricciones compartidas. Esto puede llegar a generar una cadena de fragmentaciones en orden jerárquico a la primera relación fragmentada.

Ejemplo 4.3 *Considerando la relación* Pais(id_pais, nombre_pais, idioma,poblacion_pais, PIB)*, representada en el esquema de una base de datos llamada* bdpais *indicada en la Figura 4.1. Donde* País *puede tener una o varias regiones y* Región *a su vez, puede tener una o varias* Comunas.

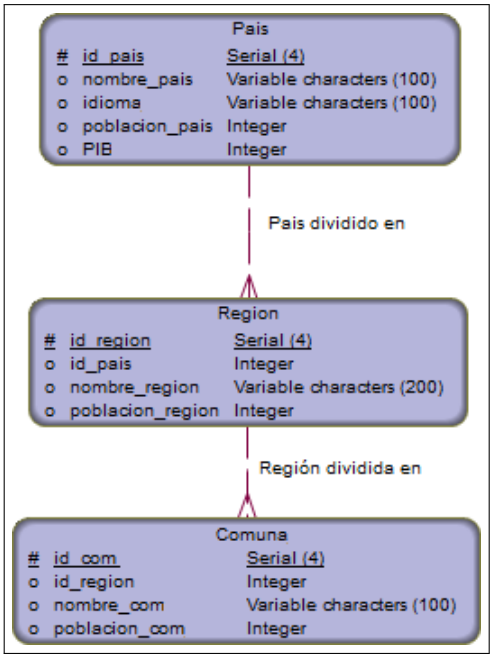


Figura 4.1: Modelo de la bases de datos *bdpais*.

Existen dos aplicaciones :

- *AP1: Determina países con un PIB mayor a 2000000 (millones de dolares) y de idioma español para otorgar mejores garantías de préstamos a los gobiernos.*
- *AP2: Necesita los países de idioma distinto al español y con PIB menores o iguales a 2000000.*

Por lo que se requiere fragmentar la relación País.

El conjunto de predicado simples Pr se define como:

Pr' : { $PIB > 2000000$, $PIB <= 200000$, $idioma = $ "*Español*", $idioma != $ "*Español*"}

Inicialización
$p_i : PIB > 2000000$*; ¿Fragmenta la relación País? Si.*

$Pr' : \{PIB > 2000000\}$
$Pr : Pr - p_i$
$F : F_1 = \sigma_{PIB>200000}(PAIS),\ F_2 = \sigma_{PIB<=200000}(PAIS)$

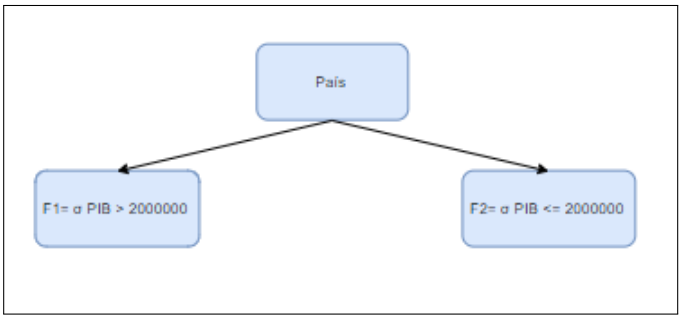


Figura 4.2: Inicialización del algoritmo COMMIN.

Primera Iteración:
$p_i : PIB <= 2000000$*; ¿Particiona alguno de los fragmentos* F_1 *o* F_2 *? No.*
$Pr' : \{PIB > 2000000\}$
$Pr : Pr - p_i$
$F : F_1 = \sigma_{PIB>200000}(PAIS),\ F_2 = \sigma_{PIB<=200000}(PAIS)$

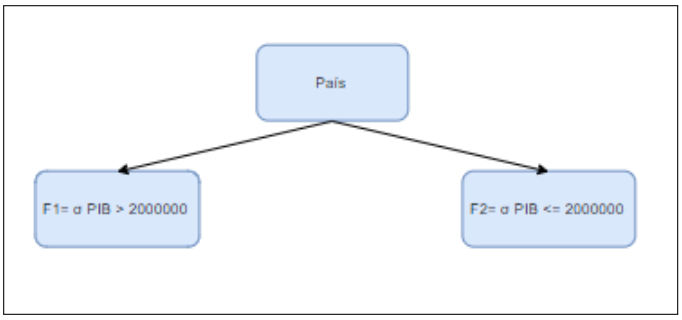


Figura 4.3: Iteración 1 del algoritmo COMMIN

Segunda Iteración:
p_i *: idioma* $=$ *"Español"; ¿Particiona alguno de los fragmentos* F_1 *o* F_2 *? Si, fragmenta* F_1
$Pr' : \{PIB > 2000000,\ idioma = \text{"Español"}\}$
$Pr : Pr - p_i$
$F : F_3 = \sigma_{idioma\ =\ \text{"Español"}}(F_1),\ F_4 = \sigma_{idioma\ !=\ \text{"Español"}}(F_1)$

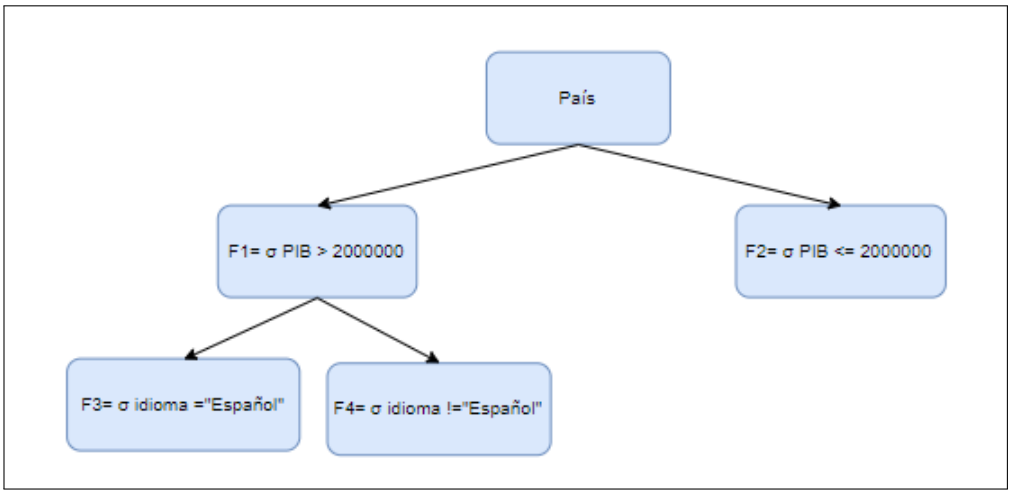


Figura 4.4: Iteración 2 del algoritmo COMMIN

Tercera Iteración:
p_i : *idioma* ! = "*Español*"; *¿Particiona alguno de los fragmentos? Si, fragmenta* F_2.
Pr' : $\{PIB > 2000000$, *idioma* = "*Español*" , *idioma* ! = "*Español*"$\}$
$Pr : Pr - p_i$
$F : F_5 = \sigma_{idioma\ !=\ \text{“}Español\text{”}}(F_2)$, $F_6 = \sigma_{idioma\ =\ \text{“}Español\text{”}}(F_2)$

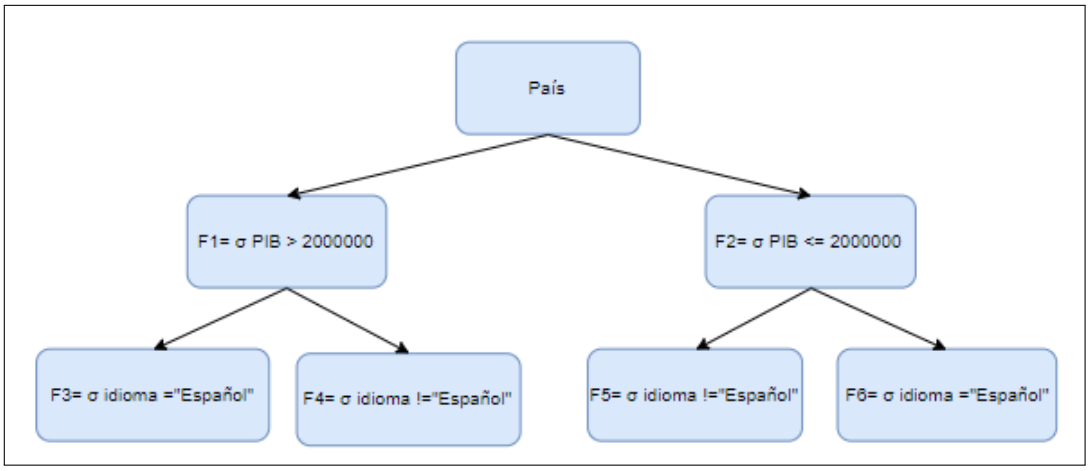


Figura 4.5: Iteración 3 del algoritmo COMMIN

Terminan las iteraciones
$Pr = \{\emptyset\}$ *El algoritmo finaliza al* Pr *ser un conjunto vació y genera a* Pr' *un conjunto de*

predicados simples completo y minimal.
$Pr' = \{PIB > 2000000,\ idioma = \text{“}Español\text{”},\ idioma != \text{“}Español\text{”}\}$

Es decir, la relación País, se fragmenta de la siguiente manera:

- *Pais1:* $\sigma_{PIB > 200000 \wedge idioma = \text{“}Español\text{”}}(País)$
- *Pais2:* $\sigma_{PIB > 200000 \wedge idioma != \text{“}Español\text{”}}(País)$
- *Pais3:* $\sigma_{PIB <= 200000 \wedge idioma = \text{“}Español\text{”}}(País)$
- *Pais4:* $\sigma_{PIB <= 200000 \wedge idioma != \text{“}Español\text{”}}(País)$

Además, al fragmentar la relación Pais es necesario fragmentar la relación Region pues se generan conflictos con la clave foránea id_pais *de Pais que es usada en Region y, a su vez, se debe fragmentar la relación Comuna porque utiiliza la clave foránea* id_región *de Region.*

Para la relación Region la fragmentación derivada es:

- *Region1:* $\sigma\ Region \ltimes_{id=id} (Pais1)$
- *Region2:* $\sigma\ Region \ltimes_{id=id} (Pais2)$
- *Region3:* $\sigma\ Region \ltimes_{id=id} (Pais3)$
- *Region4:* $\sigma\ Region \ltimes_{id=id} (Pais4)$

Y finalmente la fragmentación derivada de la relación Comuna_Ciudad se define de la siguiente forma:

- *Comuna1:* $\sigma\ Comuna\_Ciudad \ltimes_{id=id} (Region1)$
- *Comuna2:* $\sigma\ Comuna\_Ciudad \ltimes_{id=id} (Region2)$
- *Comuna3:* $\sigma\ Comuna\_Ciudad \ltimes_{id=id} (Region3)$
- *Comuna4:* $\sigma\ Comuna\_Ciudad \ltimes_{id=id} (Region4)$

4.2. Algoritmo BEA para la Fragmentación Vertical

A continuación, se presenta el algoritmo BEA. Este tipo de fragmentación busca agrupar atributos que usualmente son consultados simultáneamente, esta relación entre atributos es llamada afinidad y es usada como una medida para saber qué tan relacionados

están dos atributos. Esta fragmentación se basa en un elemento previo, la *Matriz de Afinidad*, para calcular la *Matriz de Afinidad* se debe obtener la *Matriz de Uso* y la *Matriz de frecuencia de accesos*, a continuación se describe cada una de ellas y como calcularlas.

Para obtener la *Matriz de Afinidad* se hace uso de la *Matriz de Uso*, la cual indica los atributos a los que acceden las aplicaciones. Se asigna a $Q = (q_1, q_2, ...q_q)$ como el conjunto de consultas o aplicaciones del usuario que acceden a la relación $R(A_1, A_2, ...A_n)$. Para cada consulta q_i y cada atributo A_j se asocia un valor de uso denotado como $use(q_i, A_j)$ y definido como:

$$\begin{array}{c} \\ q_1 \\ q_2 \\ \vdots \\ q_q \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & \dots & A_n \end{array} \\ \begin{pmatrix} use(q_1, A_1) & use(q_1, A_2) & \dots & use(q_1, A_n) \\ use(q_2, A_1) & use(q_2, A_2) & \dots & use(q_2, A_n) \\ \vdots & \vdots & \ddots & \vdots \\ use(q_q, A_1) & use(q_q, A_2) & \dots & use(q_q, A_n) \end{pmatrix} \end{array}$$

Figura 4.6: Matriz de uso

Ejemplo 4.4 *Considere la relación personas indicada en la Tabla 4.2.*

ID	nombre	edad	nacionalidad
001	Jon	56	Chile
002	Leo	41	Argentina
003	Lili	30	Chile

Tabla 4.2: Relación *personas*.

Y las siguientes consultas:

- q_1:
  SELECT ID,nombre,nacionalidad
  FROM personas
  WHERE nacionalidad = "Chile".

- q_2:
  SELECT ID,edad
  FROM personas
  WHERE edad>40

Definidos A_1 = ID, A_2 = nombre, A_3 = edad y A_4 = nacionalidad, se genera la Matriz de Uso siguiente:

	A_1	A_2	A_3	A_4
q_1	1	1	0	1
q_2	1	0	1	0

Tabla 4.3: Matriz de uso para consulta q_1 y q_2 que acceden relación *personas*.

El siguiente paso es calcular la afinidad de los atributos (A_i, A_j) como indica la Figura 4.7 de la relación $R(A_1, A_2, ...A_n)$ en relación a al conjunto de aplicaciones $Q = \{q_1, ...q_n\}$ para así generar la $Matriz\ de\ Afinidad$, pare ello se utiliza la siguiente formula:

$$aff(A_i, A_j) = \sum_{\text{toda consulta a la que accede } A_i \text{ y } A_j} (FAC)$$

Donde la FAC se calcula como:

$$FAC = \sum_{\text{todos las apliciones}} \text{Frecuencia de la consulta} \times \left(\frac{\text{acceso}}{\text{ejecución}} \right)$$

*Por simplicidad se asumirá que cada consulta accede una vez a el atributo en cada ejecución.

$$\begin{array}{c} \\ A_1 \\ A_2 \\ \vdots \\ A_n \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & \dots & A_n \end{array} \\ \begin{pmatrix} aff(A_1, A_1) & aff(A_1, A_2) & \dots & aff(A_1, A_n) \\ aff(A_2, A_1) & aff(A_2, A_2) & \dots & aff(A_2, A_n) \\ \vdots & \vdots & \ddots & \vdots \\ aff(A_n, A_1) & aff(A_n, A_2) & \dots & aff(A_n, A_n) \end{pmatrix} \end{array}$$

Figura 4.7: Matriz de afinidad

Una vez calculada la afinidad para cada atributo aff(A_i, A_j) se obtiene la *Matriz de Afinidad.*

Ejemplo 4.5 *Consideremos la* Matriz de Uso *del ejemplo anterior de la Tabla 4.3 y la siguiente matriz de frecuencia FAC indicada en la Tabla 4.4 en la cual existen dos sitios S_1 y S_2 que ejecutan las consultas.*

	S_1	S_2
q_1	10	5
q_2	4	0

Tabla 4.4: Matriz FAC.

Se calcula la siguiente Matriz de Afinidad indicada en la Tabla 4.5:

- $Aff(A_1, A_1)$*: Las consultas donde se acceden a el atributo* A_1 *son* q_1 *y* q_2*, por lo que* $Aff(A_1, A_1) = 10 * 1 + 5 * 1 + 4 * 1 = 19$.
- $Aff(A_1, A_2)$*: La consulta donde se accede a los atributos* A_1 *y* A_2 *es solo* q_1*, por lo que* $Aff(A_1, A_1) = 10 * 1 + 5 * 1 = 15$.
- *Utilizando el mismo procedimiento se calcula el resto de las afinidades y se obtiene:*

	A_1	A_2	A_3	A_4
A_1	19	15	4	15
A_2	15	15	0	15
A_3	4	0	4	0
A_4	15	15	0	15

Tabla 4.5: Matriz de afinidad calculada.

Algoritmo 2: Algoritmo BEA

```
   Input: AA: matriz de afinidad
   Output: CA: matriz de afinidad agrupada
 1 begin
 2     {inicializar; recordar que AA es una matriz de dimensión n × n }
 3     CA[•, 1] ← AA[•, 1];
 4     CA[•, 2] ← AA[•, 2];
 5     indice ← 3;
 6     while indice ≤ n do
 7         /*escoger la "mejor"posición para el atributo AA_indice */
 8         for i desde 1 hasta indice − 1 iterandoen 1 do
 9             Calcular cont(A_{i−1}, A_indice, A_i);
10             Calcular cont(A_{indice−1}, A_indice, A_{indice+1});      /*Condición limite*/
11             loc ← ubicación dada por el máximo valor de cont;
12         for j desde indice hasta loc iterando en − 1 do
13             CA[•, j] ← CA[•, j − 1];
14         CA[•, loc] ← AA[•, indice];
15         indice ← indice + 1
16     Ordenar las filas utilizando el mismo orden de los atributos.
```

El Algoritmo BEA utiliza la *Matriz de Afinidad* y re-ordena los atributos, es decir, agrupa los atributos afines entre ellos. Luego re-ordena las filas siguiendo el orden establecido con las columnas.

Este algoritmo inicia en la línea 2 asignando a la una matriz vacía de mismas dimensiones que la *Matriz de Afinidad* AA entrante la cual es llamada *matriz de afinidad agrupada* (CA), las líneas 2 y 3 se asignan las dos primera columnas de la matriz AA a la las dos primeras posiciones de la matriz CA, en la línea 4 se aumenta el indice de la matriz CA a 3, pues ya se añadieron las dos primeras.

Luego en un ciclo **while** (línea 5) cuya iteración inicial es el valor *index* (inicialmente 3) hasta n, donde n es el tamaño la matriz de afinidad (matriz cuya dimensión es n x n). Entre las líneas 7 y 10 se posiciona la columna (la posición de la columna está definida por el valor actual del indice) de la matriz AA en la matriz CA. La posición de la columna en la matriz CA es obtenida al obtener el máximo valor *cont* de dicha columna respecto a la matriz CA, es decir, se busca la posición que maximiza la afinidad de los atributos en la matriz de afinidad agrupada CA.

El valor de *cont*, se obtiene de la siguiente manera:

$$cont(A_i, A_k, A_j) = 2bond(A_i, A_k) + 2bond(A_k, A_j) - 2bond(A_i, A_j)$$

y el valor de *bond* es calculado como:

$$bond = \sum_{z=1}^{n} aff(A_z, A_k,) \times aff(A_z, A_y)$$

Entre las líneas 11 y 14 se reposicionan las columnas de CA para dar espacio a la nueva columna insertada desde AA.Posicionadas todas las columnas de la *Matriz de Afinidad* AA en la *Matriz de Afinidad Agrupada CA*, el Algoritmo BEA reordena las filas del mismo modo en que están ordenadas las columnas (línea **??**).

Una vez el Algoritmo BEA ha generado la matriz CA, se deben encontrar conjuntos de atributos a los que se accede únicamente, o en su mayor parte, por distintos conjuntos de aplicaciones. Para esto se utiliza el siguiente método:

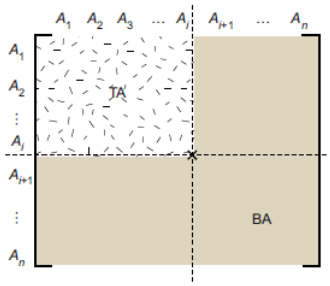


Figura 4.8: Localización de un punto de división durante agrupamiento

Como podemos ver en la Figura 4.8, existen dos conjuntos de atributos, el primer grupo ubicado en la parte superior izquierda que contiene los atributos $A_1, A_2, ...A_i$ que se denomina TA y el segundo grupo en la zona inferior derecha es BA. Utilizando el conjunto de aplicaciones $Q = (q_1, q_2, ...q_q)$ se definen un conjunto de aplicaciones que solo acceden a TA, solo a BA y otro conjunto que acceda a ambos. Estos conjuntos se definen como:

$AQ(q_i) = \{A_j \mid use(q_i, A_j) = 1\}$
$TQ = \{q_i \mid AQ(q_i) \subseteq TA\}$
$BQ = \{q_i \mid AQ(q_i) \subseteq BA\}$
$OQ = Q - \{TQ \cup BQ\}$

La primera ecuación, define los atributos a los que accede la aplicación q_i; TQ es el conjunto de aplicaciones que solo acceden a TA, BQ es el conjunto de aplicaciones que solo acceden a BA y por último, OQ es el conjunto de aplicaciones a las que acceden a ambos (TA *y* BA). Notar que al existir n atributos, existen $n-1$ posibles posiciones para el punto de división.

El mejor punto de división es aquel que permite maximizar los accesos a TQ y BQ en cada uno de los fragmentos, minimizando los accesos a ambos. Para calcular ese punto se utilizan las siguientes ecuaciones:

$$CQ = \sum_{q_i \in Q} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$CTQ = \sum_{q_i \in TQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$CBQ = \sum_{q_i \in BQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$COQ = \sum_{q_i \in OQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

El punto de división ideal se obtiene al encontrar el valor que maximice la siguiente expresión:

$$CTQ \times CBQ - COQ^2$$

Por lo tanto, el punto ideal de división estará entre la posición 1 y la posición $n-1$. Una vez identificado se procede a fragmentar la relación R.

Ejemplo 4.6 *Considere la misma base de datos utilizada en el ejemplo para el Algoritmo COMMIN (véase Figura 4.1), se desea realizar una fragmentación vertical de la relación comuna. La relación comuna tiene cuatro atributos:*

- A_1 : *id_com* .
- A_2 : *id_region* .
- A_3 : *nombre_com* .
- A_4 : *poblacion_com* .

Y cuatro consultas $Q : \{ q_1, q_2, q_3, q_4\}$ *definidas como:*

- q_1*: SELECT id_com, id_region, nombre_com FROM comuna.*

- q_2: *SELECT id_com, id_region FROM comuna.*
- q_3: *SELECT id_region, poblacion_com FROM comuna.*
- q_4: *SELECT nombre_com, poblacion_com FROM comuna.*

La Matriz de Uso *obtenida está representada por la Figura 4.9 , mientras la Figura 4.10 representa la matriz FAC con una cantidad de dos sitios que acceden a las consultas :*

$$\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figura 4.9: Matriz de uso para relación comuna.

$$\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{array} \begin{array}{c} \begin{array}{cc} \textit{Sitio } 1 & \textit{Sitio } 2 \end{array} \\ \begin{pmatrix} 60 & 40 \\ 30 & 26 \\ 0 & 8 \\ 0 & 14 \end{pmatrix} \end{array}$$

Figura 4.10: Matriz FAC para calculo de afinidades.

El siguiente paso es calcular la Matriz de Afinidad, donde la Figura 4.11 muestra el resultado. El cálculo es el siguiente:

$$\begin{aligned}
aff(A_1, A_1) &= 1 \times 60 + 1 \times 40 + 1 \times 30 + 1 \times 26 = 156 \\
aff(A_1, A_2) &= 1 \times 60 + 1 \times 40 + 1 \times 30 + 1 \times 26 = 156 \\
aff(A_1, A_3) &= 1 \times 60 + 1 \times 40 = 100 \\
aff(A_1, A_4) &= 1 \times 30 + 1 \times 26 = 56 \\
aff(A_2, A_1) &= 1 \times 60 + 1 \times 40 + 1 \times 30 + 1 \times 26 = 156 \\
aff(A_2, A_2) &= 1 \times 60 + 1 \times 40 + 1 \times 30 + 1 \times 26 + 1 \times 8 = 164 \\
aff(A_2, A_3) &= 1 \times 60 + 1 \times 40 = 100 \\
aff(A_2, A_4) &= 1 \times 30 + 1 \times 26 + 1 \times 8 = 64 \\
aff(A_3, A_1) &= 1 \times 60 + 1 \times 40 = 100 \\
aff(A_3, A_2) &= 1 \times 60 + 1 \times 40 = 100 \\
aff(A_3, A_3) &= 1 \times 60 + 1 \times 40 + 1 \times 14 = 114 \\
aff(A_3, A_4) &= 1 \times 14 = 14 \\
aff(A_4, A_1) &= 1 \times 30 + 1 \times 26 = 56 \\
aff(A_4, A_2) &= 1 \times 30 + 1 \times 26 + 1 \times 8 = 64 \\
aff(A_4, A_3) &= 1 \times 14 + 1 = 14 \\
aff(A_4, A_4) &= 1 \times 30 + 1 \times 26 + 1 \times 8 + 1 \times 14 = 78
\end{aligned}$$

Por lo tanto, la Matriz de Afinidad *queda de la siguiente manera:*

$$\begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{pmatrix} 156 & 156 & 100 & 56 \\ 156 & 164 & 100 & 64 \\ 100 & 100 & 114 & 14 \\ 56 & 64 & 14 & 78 \end{pmatrix} \end{array}$$

Figura 4.11: Matriz de afinidad obtenida.

El Algoritmo BEA debe reordenar las columnas de la Matriz de Afinidad*, agrupando los atributos afines, para así aumentar la afinidad global de la relación. Para ello, se agregan las dos primeras columnas como muestra la Figura 4.12, luego se calcula mejor posición para maximizar la afinidad para los atributos A_3 y A_4.*

$$\begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cc} A_1 & A_2 \end{array} \\ \begin{pmatrix} 156 & 156 \\ 156 & 164 \\ 100 & 100 \\ 56 & 64 \end{pmatrix} \end{array}$$

Figura 4.12: Matriz de afinidad ordenada, insertados atributos A_1 y A_2.

La posición para A_3 y A_4 se determina con el siguiente cálculo. Para A_3 el calculo es el siguiente:(Se considera a A_f como una columna de valores 0, que se encuentran siempre en los extremos de la matriz)*

$cont(A_f, A_3, A_1) = 2bond(A_f, A_3) + 2bond(A_3, A_1) - 2bond(A_f, A_1) = 87104$
$cont(A_1, A_3, A_2) = 2bond(A_1, A_3) + 2bond(A_3, A_2) - 2bond(A_1, A_2) = 48688$
$cont(A_2, A_3, A_f) = 2bond(A_2, A_3) + 2bond(A_3, A_f) - 2bond(A_2, A_f) = 88592$

El orden (A_2, A_3, A_f) tiene el mayor valor cont por lo que A_3 se posiciona a la derecha de la columna A_2, por lo tanto la $Matriz\ Agrupada\ CA$ es:

$$\begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{ccc} A_1 & A_2 & A_3 \end{array} \\ \begin{pmatrix} 156 & 156 & 100 \\ 156 & 164 & 100 \\ 100 & 100 & 114 \\ 56 & 64 & 14 \end{pmatrix} \end{array}$$

Figura 4.13: Matriz de afinidad ordenada, insertados A_1 , A_2 y A_3.

Finalmente se posiciona el atributo A_4*:*
$cont(A_f, A_4, A_1) = 2bond(A_f, A_4) + 2bond(A_4, A_1) - 2bond(A_f, A_1) = 48976$
$cont(A_1, A_4, A_2) = 2bond(A_1, A_4) + 2bond(A_4, A_2) - 2bond(A_1, A_2) = -26748$
$cont(A_2, A_4, A_3) = 2bond(A_2, A_4) + 2bond(A_4, A_3) - 2bond(A_2, A_3) = -7932$
$cont(A_3, A_4, A_f) = 2bond(A_3, A_4) + 2bond(A_4, A_f) - 2bond(A_3, A_f) = 29376$

Al ser $cont(A_f, A_4, A_1) = 48796$ *la mayor afinidad obtenida, el atributo* A_4 *se posiciona al inicio de la Matriz de Afinidad Agrupada, al posicionar la última columna, se obtiene la matriz de afinidad ordenada por afinidad como se muestra en la Figura 4.14 . Luego se reordenan las filas de la matriz de acuerdo al mismo orden que las columnas y finalmente se obtiene la Matriz de Afinidad Agrupada definitiva 4.15.*

	A_4	A_1	A_2	A_3
A_1	56	156	156	100
A_2	64	156	164	100
A_3	14	100	100	114
A_4	78	56	64	14

Figura 4.14: Matriz de afinidad ordenada.

	A_4	A_1	A_2	A_3
A_4	78	56	64	14
A_1	56	156	156	100
A_2	64	156	164	100
A_3	14	100	100	114

Figura 4.15: Matriz de afinidad agrupada.

El paso final para realizar la fragmentación es obtener el agrupamiento que permita encontrar el conjunto de atributos al cual acceda la mayor parte de las aplicaciones, para el caso de prueba 6.2 se calcula utilizando la Matriz de uso (Figura 4.11), la matriz FAC 4.10 y la Matriz de Afinidad Agrupada 4.15, a continuación el desarrollo:

Para $TA = A_4$ *y* $BA = A_1, A_2, A_3$*:*
$CTQ = 0$
$CBQ = 100$
$COQ = 78$
$CQT \times CBQ - COQ^2 = -6084$

Para $TA = A_4, A_1$ *y* $BA = A_2, A_3$*:*
$CTQ = 0$
$CBQ = 0$
$COQ = 178$
$CQT \times CBQ - COQ^2 = -31684$

Para $TA = A_4, A_1, A_2$ *y* $BA = A_3$*:*
$CTQ = 64$
$CBQ = 0$

$COQ = 114$
$CQT \times CBQ - COQ^2 = -12996$

La mejor distribución para una fragmentación vertical utilizando el Algoritmo BEA para la relación comuna_ciudad, son los subconjuntos de atributos $TA = A_4$ *y* $BA = A_1, A_2, A_3$*, por lo que la fragmentación generada es la siguiente:*

- *Fragmentación 1 :* $TA = A_4$
  $comuna1$: { *id_com* , *poblacion_com* }
- *Fragmentación 2 :* $BA = A_1, A_2, A_3$
  $comuna2$: { *id_com* , *id_region* , *nombre_com* }

A continuación, el Capítulo 5, donde se presenta la interfaz del sistema y su funcionalidad.

Capítulo 5

Desarrollo e Interfaz de los sistemas WEB y Móvil

Este capítulo describe las herramientas utilizadas en el desarrollo del sistema WEB para las fragmentaciones horizontal y vertical. Además, se detalla el diseño de la aplicación WEB desde el punto de vista visual, referente al diseño del sistema, la organización y el funcionamiento de éste.

5.1. Software y Herramientas Utilizadas para el desarrollo

En esta sección se describen el software y las herramientas utilizadas para el desarrollo del sistema WEB para la fragmentación de bases de datos.

5.1.1. Lenguaje de Programación JAVA

JAVA es un lenguaje de programación creado por SUN Microsystems, (empresa que posteriormente fue comprada por ORACLE) para poder funcionar en distintos tipos de procesadores. Su sintaxis es muy parecida a la de C o C++, e incorpora como propias algunas características que en otros lenguajes son extensiones: gestión de hilos, ejecución remota, etc. El paradigma de programación del lenguaje JAVA está basada en la programación orientada a objetos.

JSP y Servlets

Los Servlets y JSP son programas JAVA que se ejecutan en un servidor de aplicaciones JAVA y amplían las prestaciones del servidor WEB. Un servlet es un objeto de JAVA diseñadas para responder a solicitudes HTTP en el contexto de una aplicación WEB que pertenece a una clase que extiende de *javax.servlet.http.HttpServlet*, esto permite crear aplicaciones WEB dinámicas, lo que posibilita realizar consultas, insertar y eliminar datos.

Una página JSP (*JavaServer Page*) es una página HTML a la que se le incrusta código JAVA. Estos fragmentos de código JAVA generan contenido dinámico. Una JSP se convierte

a un Servlet JAVA y se ejecuta en el servidor. Las sentencias JSP incluidas en la JSP pasan a formar parte del Servlet generado a partir de la JSP. El Servlet resultante se ejecuta en el servidor (del Carmen Gómez y Cervantes).

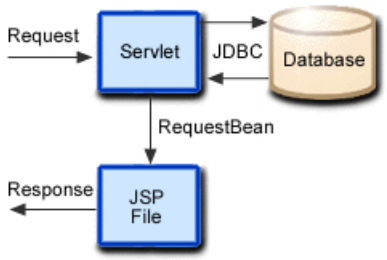


Figura 5.1: Proceso servlet - jsp.

5.1.2. Entorno de desarrollo NetBeans

NetBeans es un IDE de código abierto, que fue desarrollado principalmente para la programación en lenguaje *JAVA*, aunque también soporta lenguajes como *JAVAscript, CSS, C/C++*. NetBeans permite extender su funcionalidad a través de módulos, cada módulo contiene clases *JAVA* que permiten interactuar con las APIs de NetBeans. El IDE de NetBeans soporta el desarrollo de todos los tipos de aplicación JAVA (J2SE, WEB, EJB y aplicaciones móviles).

5.1.3. PgAdmin 4

*PgAdmin 4*es una herramienta para la administración y gestión de *PostgreSQL* y otras bases de datos relacionales derivadas tal como *PostGIS* o *EnterpriseDB*. Es una nueva versión que supuso una re-escritura completa de su predecesora *pgAdmin 3*. Esta nueva versión está diseñada para funcionar tanto en escritorio como en un servidor WEB.

5.2. Diseño de Interfaz y Navegación

A continuación, se describe la interfaz y navegación desde la perspectiva del usuario.

5.2.1. Esquema de Navegación

La Figura 5.2 representa el flujo de navegación que él usuario puede utilizar en el sistema.

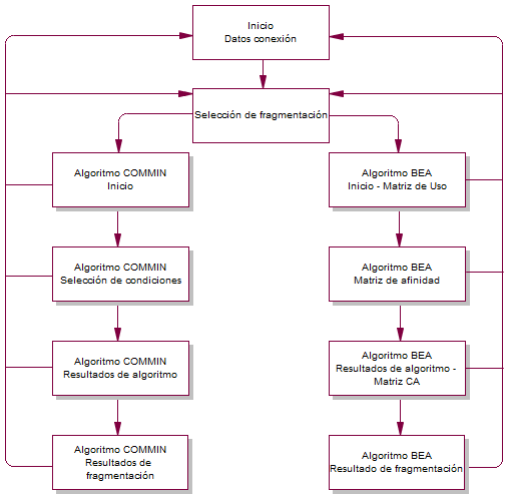


Figura 5.2: Esquema de navegación.

5.2.2. Diseño de la Interfaz

A continuación, se presenta el diseño de las interfaces gráficas del sistema.

Pagina de inicio

La Figura 5.3 muestra el inicio del sistema, donde se encuentra un formulario para que el usuario ingrese la información necesaria para realizar la conexión a la base de datos con la que se trabaja.

Figura 5.3: Interfaz de pantalla de inicio.

Una vez el usuario, realiza la conexión a la base de datos PostgreSQL, el sistema lo redirecciona a la Sección de selección de fragmentación a realizar.

Selección de fragmentación

En la Figura 5.4 se visualiza la interfaz donde el usuario selecciona el tipo de fragmentación.

Figura 5.4: Interfaz para la selección de fragmentación.

El usuario selecciona la fragmentación a realizar y el sistema lo redirecciona a la interfaz para el Algoritmo COMMIN o a la interfaz del Algoritmo BEA.

Interfaces gráficas Algoritmo COMMIN

A continuación, se describe las interfaces dedicadas al Algoritmo COMMIN.

La Figura 5.5 muestra la interfaz de inicio para el Algoritmo COMMIN del sistema, en este ventana el usuario debe seleccionar la relación de la base de datos ingresada que desea fragmentar.

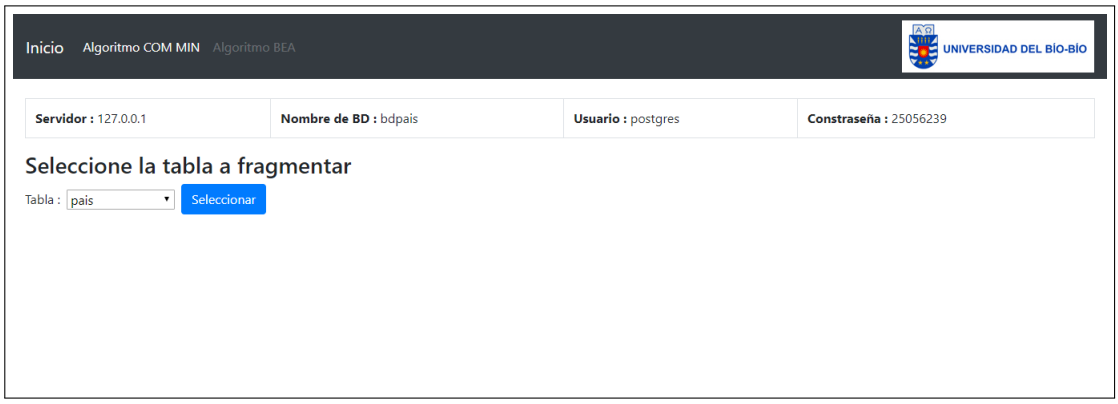


Figura 5.5: Interfaz para algoritmo COMMIN, pestaña de inicio.

La Figura 5.6 presenta la interfaz donde el usuario, una vez seleccionada la relación, debe indicar las condiciones para generar una fragmentación horizontal a través del Algoritmo COMMIN.

Figura 5.6: Interfaz para algoritmo COMMIN, pestaña de condiciones.

Luego de haber ingresado las condiciones, se debe presionar el botón **Iniciar Fragmentación** como muestra la Figura 5.6, este permite que el sistema genere la fragmentación horizontal. En la Figura 5.7 se visualiza la interfaz de resultado del Algoritmo COMMIN una vez ejecutado, presentando las fragmentaciones generadas junto a las condiciones que la generaron.

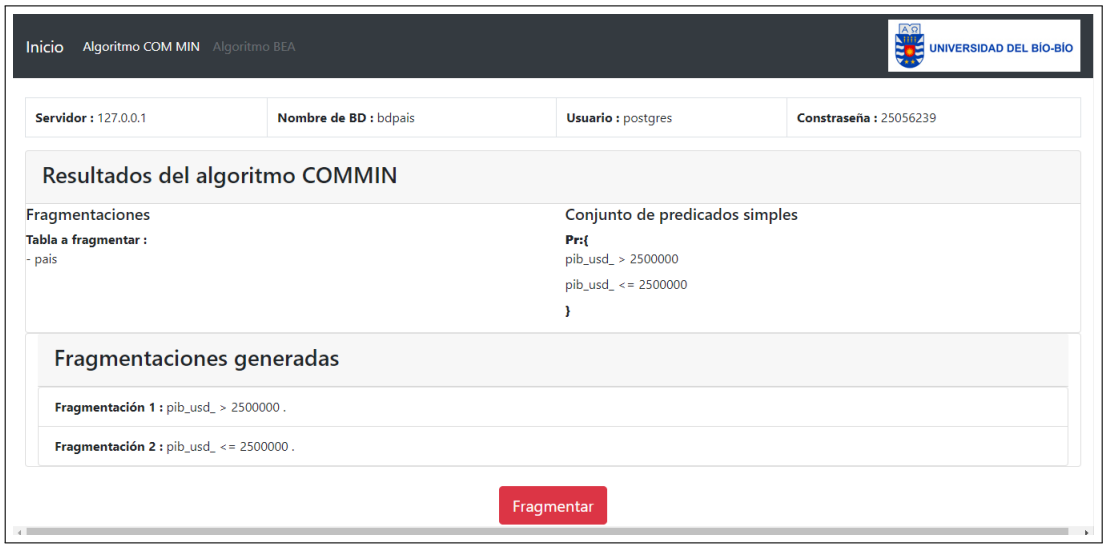


Figura 5.7: Interfaz para algoritmo COMMIN, resultados de la fragmentación.

Una vez ejecutado el Algoritmo COMMIN y el sistema muestra las fragmentaciones obtenidas, el usuario debe confirmar la fragmentación en la base de datos presionando el boton **Fragmentar** (5.7), esto permite que el sistema modifique la base de datos y genere de forma real la fragmentación, finalmente el sistema, como indican las Figuras 5.8 y 5.9, muestra la interfaz final del Algoritmo COMMIN una vez realizada la fragmentación en la base de datos. Contiene paneles colapsables que muestran el detalle de la relación fragmentada antes y después de la fragmentación.

Figura 5.8: Interfaz para algoritmo COMMIN, fragmentación completada parte 1.

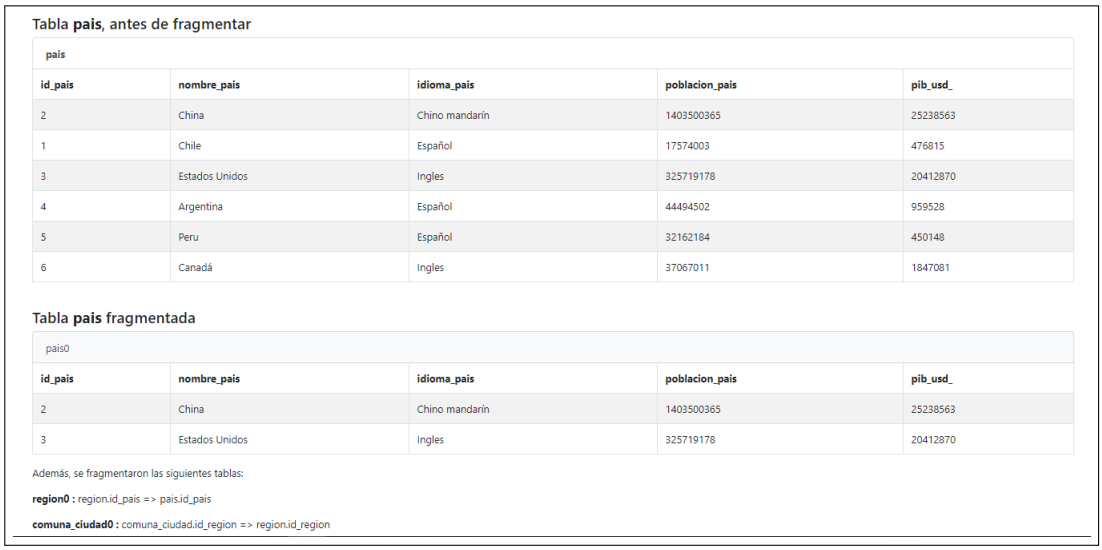


Figura 5.9: Interfaz para algoritmo COMMIN, fragmentación completada parte 2.

Interfaces gráficas Algoritmo BEA

Las siguientes figuras muestran las interfaces del sistema relacionadas al Algoritmo BEA.

La Figura 5.10 presenta la interfaz de inicio para el Algoritmo BEA, donde el usuario debe seleccionar la relación a fragmentar y el número de consultas , luego ingresar los valores **1** en caso de que la aplicación acceda a el atributo correspondiente y **0** si no es utilizado para cada aplicación o consulta generada, para así generar la *Matriz de Uso*.

Figura 5.10: Interfaz para algoritmo BEA, pestaña de inicio y creación de matriz de uso.

Una vez la *Matriz de Uso* es generada y el usuario presiona el botón **Enviar Matriz de Uso**, el sistema redirecciona a la interfaz donde se genera la *Matriz FAC*.

En la Figura 5.11 se muestra la vista para la creación de la *Matriz FAC*, el usuario debe ingresar la cantidad de sitios que acceden a las aplicaciones (consultas), para luego ingresar los valores de frecuencia de acceso en su matriz correspondiente.

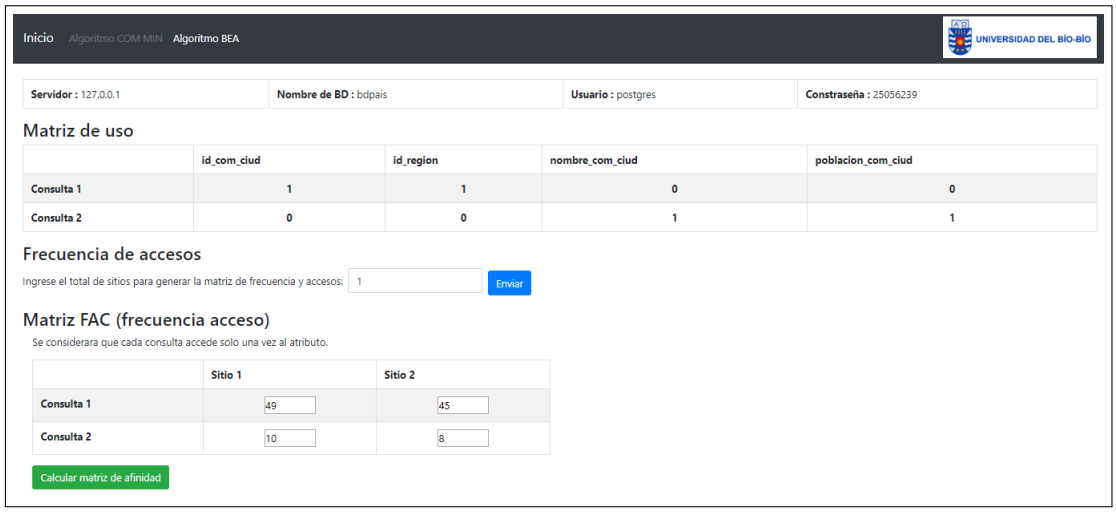


Figura 5.11: Interfaz para algoritmo BEA, pestaña para creación de matriz FAC.

El botón **Calcular Matriz de Afinidad** da la orden de redireccionar y el sistema calcula dicha matriz. La Figura 5.12 muestra la *Matriz de Afinidad* calculada utilizando la *Matriz de Uso* y la *Matriz FAC*.

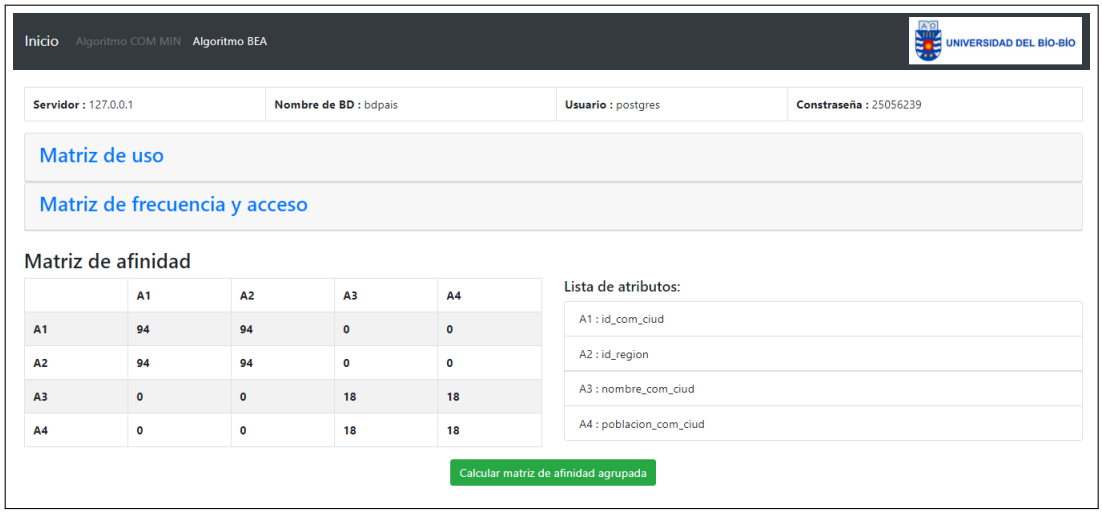
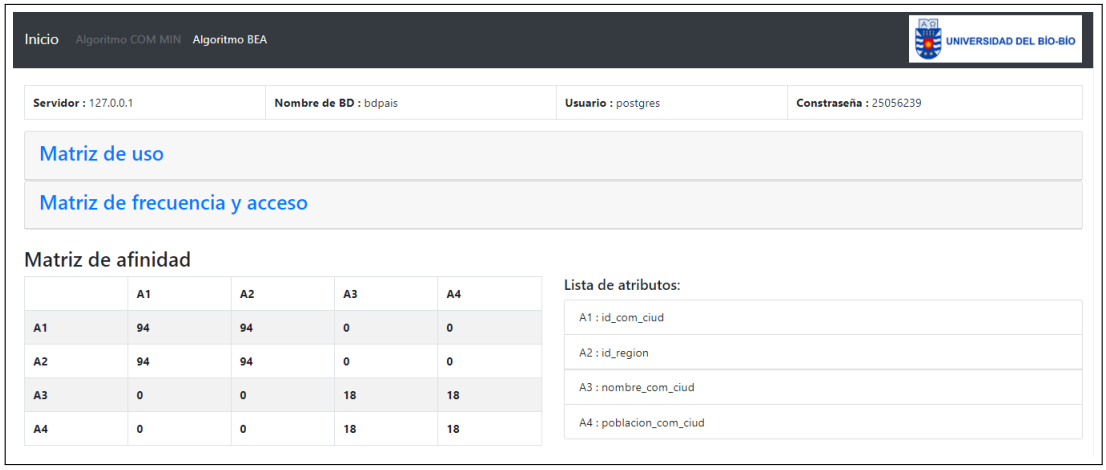


Figura 5.12: Interfaz para algoritmo BEA, pestaña con matriz de afinidad calculada.

Una vez calculada la *Matriz de afinidad*, el sistema debe calcular la *Matriz CA*, para ello se debe presionar el botón **Calcular Matriz de Afinidad Agrupada**. Las Figuras 5.13 y 5.14 muestran la interfaz donde se visualiza la *Matriz CA* calculada a partir de la *Matriz de Afinidad*. Además, se incluye paneles colapsables que muestran las matrices de uso y frecuencia.



	A1	A2	A3	A4
A1	94	94	0	0
A2	94	94	0	0
A3	0	0	18	18
A4	0	0	18	18

Figura 5.13: Interfaz para algoritmo BEA,pestaña con matriz agrupada CA calculada parte 1.

	A1	A2	A1	A4
A1	94	94	0	0
A2	94	94	0	0
A3	0	0	0	18
A4	0	0	0	18

	A1	A2	A1	A4
A1	94	94	94	0
A2	94	94	94	0
A1	94	94	94	0
A4	0	0	0	18

Figura 5.14: Interfaz para algoritmo BEA,pestaña con matriz agrupada CA calculada parte 2.

Finalmente el usuario debe presionar el boton **Fragmentar** para generar la fragmentación en la base de datos, una vez generada el sistema redirecciona a la interfaz final como lo muestra la Figura 5.15, donde se visualiza a través de paneles colapsables el antes y

después de la fragmentación vertical.

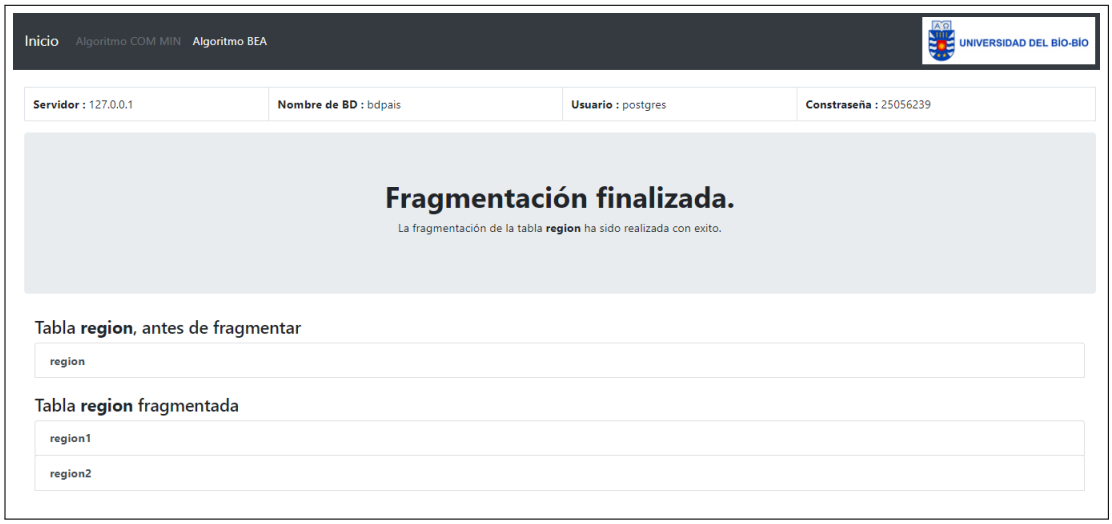


Figura 5.15: Interfaz para algoritmo BEA, fragmentación finalizada

El Capítulo 6 a continuación, describe una serie de pruebas al sistema desarrollado, donde se muestra su correcto funcionamiento.

Capítulo 6

Pruebas

En este capítulo se describe las pruebas realizadas, con el fin de medir la funcionalidad del sistema verificando el correcto funcionamiento de las fragmentaciones realizadas tanto por el algoritmo COMMIN y BEA, comprobando que los fragmentos generados sean correctos a través de un seguimiento del proceso de fragmentación.

6.1. Hardware Utilizado

El hardware especifico utilizado para la ejecución de las pruebas fue la siguiente: Se utilizó un notebook modelo HP Pavilion 14-al120la, cuyo procesador es Intel® Core™ i5-7200U (2,5 GHz, hasta 3,1 GHz, 3 MB de caché, 2 núcleos), una memoria RAM de 12 GB de SDRAM DDR4-2133, disco duro SATA de 1 TB y 5400 rpm y un sistema operativo Windows 10 Home 64bits.

6.1.1. Pruebas de Conexión e Interfaz

En este apartado, se detalla las pruebas realizadas sobre puntos críticos del sistema tanto a la conexión como a la interfaz, La conexión a la base de datos está relacionada para los casos de prueba 6.1 y 6.2, el caso de prueba 6.3 se genera una prueba sobre las condiciones ingresadas para la fragmentación utilizando el Algoritmo COMMIN, luego el caso de prueba 6.4 se refiere a los valores ingresados a la *Matriz de Uso* y sus valores, el caso de prueba 6.5 verifica los valores ingresados a la Matriz de Frecuencia / Acceso FAC.

Prueba de conexión a base de datos

Definición del Caso de prueba	
ID	CP01
Descripción	Conexión a base de datos.
Prerrequisito	Ninguno.
Datos de prueba	Datos de conexión a base de datos: ip del servidor el cual aloja a la base de datos, nombre de la base de datos, usuario y contraseña.
Resultados esperados	Se conecta a la base de datos, el sistema redirecciona a la interfaz de selección de fragmentación.
Resultados obtenidos	Conexión exitosa, los datos ingresados permiten la conexión a la base de datos y el sistema redirecciona a la selección de fragmentación a realizar.
Evaluación de la prueba	Satisfactoria, conexión lograda. Al evaluar la conexión, el sistema logra conectarse a la base de datos, permitiendo trabajar con las relaciones y su posterior fragmentación

Tabla 6.1: Caso de prueba CP01 para conexión a base de datos *bdpais*.

En la Figura 6.1, se pueden ver los datos ingresado.

Figura 6.1: Datos de Conexión a base de datos *bdpais*.

La Figura 6.2 verifica que la conexión fue exitosa, por lo tanto el sistema redirecciona a la interfaz de selección de fragmentación. En la zona demarcada es posible ver los datos de conexión que utiliza el sistema.

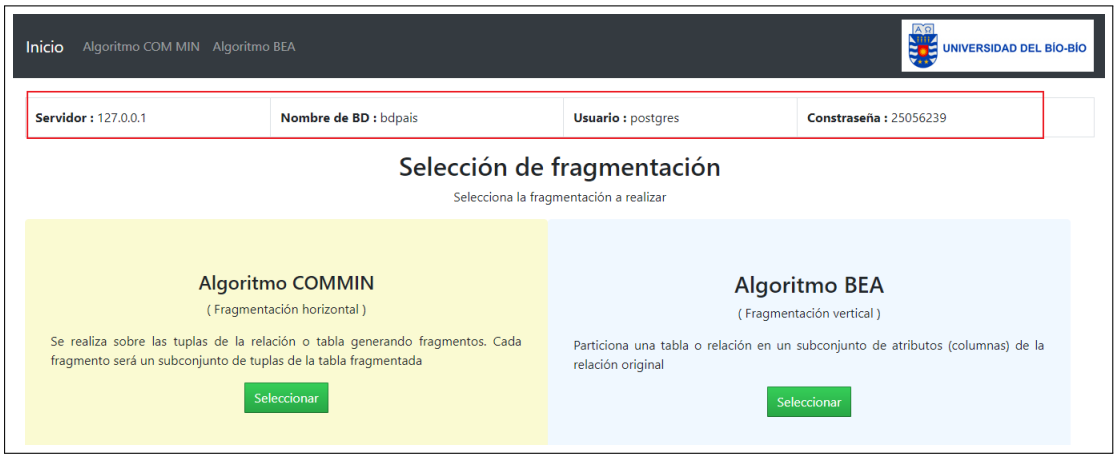


Figura 6.2: Conexión exitosa a base de datos *bdpais*.

Datos de conexión incorrecta

Definición del caso de prueba	
ID	CP02
Descripción	Ingreso de datos de conexión errónea.
Prerrequisito	Ninguno.
Datos de prueba	Datos de conexión correctos a excepción del usuario, que en este caso es ingresado como $ErrorDeUsuario$.
Resultados esperados	Sistema envía aviso de error de conexión.
Resultados obtenidos	Alerta de error al conectar.
Evaluación de la prueba	Satisfactoria, el sistema genera una alerta al intentar conectar una base de datos con datos erróneos

Tabla 6.2: Caso de prueba CP02 para datos de conexión erróneos.

En la Figura 6.3, Se puede verificar que el sistema genera una alerta de error al conectar.

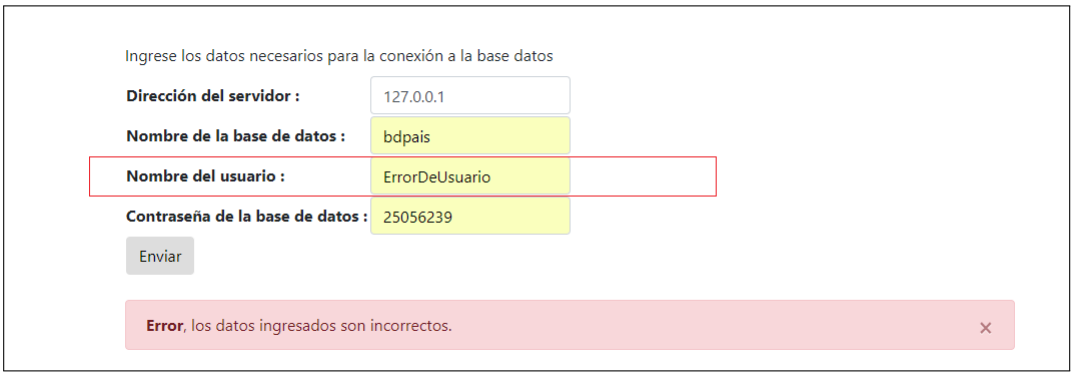


Figura 6.3: Datos de conexión erróneos.

Verificación de condiciones correctas en fragmentación horizontal COMMIN

Definición del caso de prueba	
ID	CP03
Descripción	El sistema verifica que la condición es acorde al atributo seleccionado durante la definición de las condiciones de fragmentación para el Algoritmo COMMIN.
Prerrequisito	Conexión a la base de datos, seleccionar la relación *pais* como objetivo a fragmentar.
Datos de prueba	Añadir como condición que la columna *poblacion_pais* ="Muchos".
Resultados esperados	El sistema indica que la condición es incorrecta. Notar que *poblacion_pais* es una columna de tipo *Integer* y se intenta condicionar con un valor de tipo *String*.
Resultados obtenidos	Alerta de error. Los datos son incompatibles con la columna seleccionada.
Evaluación de la prueba	Satisfactoria, el sistema genera logra evaluar que las condiciones sean correctas.

Tabla 6.3: Caso de prueba CP03 para verificación de condiciones.

En la Figura 6.4, Se verifica que el sistema genera una alerta de error.

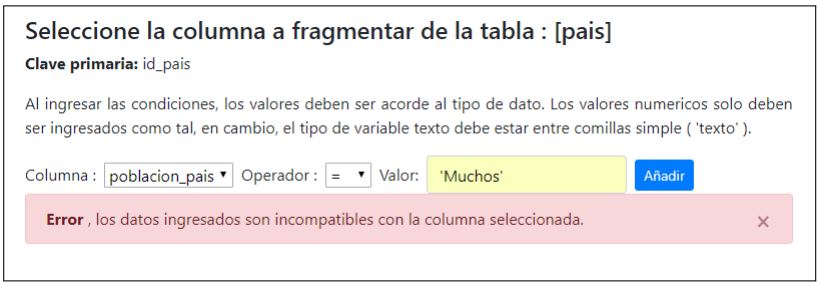


Figura 6.4: Verificación de condiciones.

Verificar valores de la Matriz de Uso en fragmentación con Algoritmo BEA

Definición del caso de prueba	
ID	CP04
Descripción	Verificar que la Matriz de Uso, en cada fila no contenga solo valores 0.
Prerrequisito	Conexión a la base de datos, haber seleccionado la relación a fragmentar, ingresar valores en Matriz de Uso.
Datos de prueba	La Matriz de Uso es llenada con valores 0
Resultados esperados	El sistema debe mostrar una alerta indicando que la matriz no debe estar totalmente ingresada con valores 0.
Resultados obtenidos	Alerta de error. El sistema detecto que la matriz solo tiene 0.
Evaluación de la prueba	Satisfactoria, el sistema logra verificar los valores de la Matriz de Uso.

Tabla 6.4: Caso de prueba CP04 para verificación de valores de matriz de uso.

La Figura 6.5 muestra la alerta al ingresar una Matriz de Uso llenada de ceros.

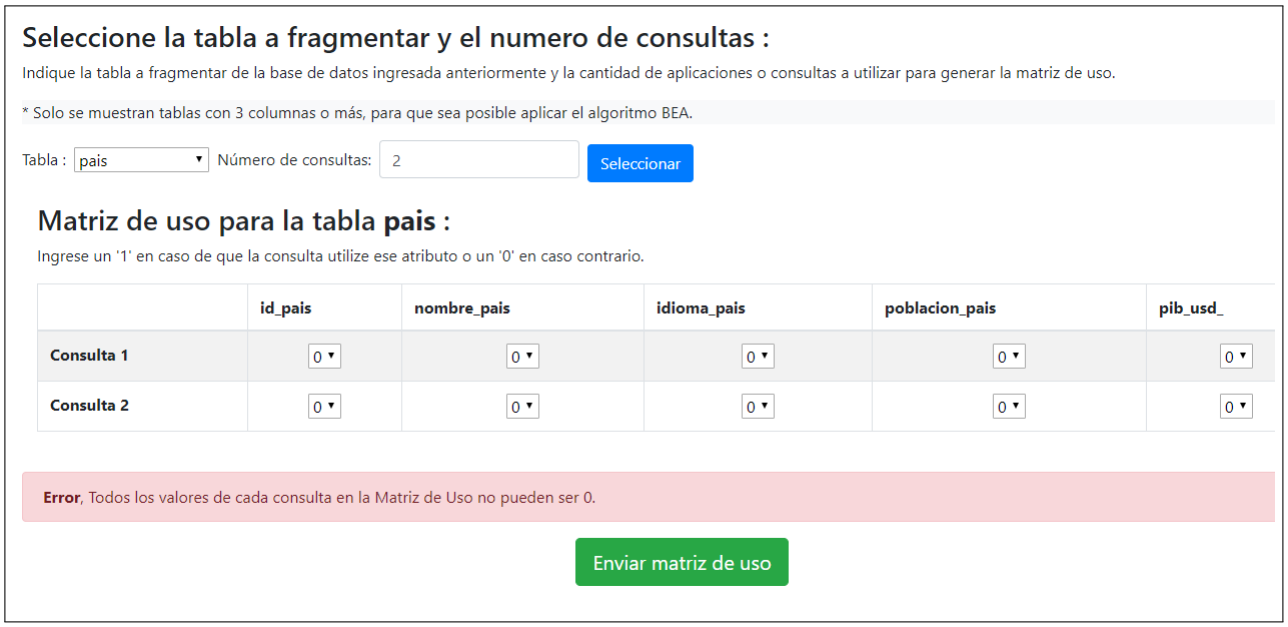


Figura 6.5: Verificar valores de matriz de uso.

Verificar valores de la Matriz FAC en fragmentación con Algoritmo BEA

Definición del caso de prueba	
ID	CP05
Descripción	Verificar que la Matriz FAC, en cada fila no contenga solo valores 0.
Prerrequisito	Conexión a la base de datos, haber seleccionado la relación a fragmentar, haber ingresado la Matriz de Uso, ingresar valores en Matriz FAC.
Datos de prueba	La Matriz FAC es llenada con valores 0
Resultados esperados	El sistema debe mostrar una alerta indicando que la matriz de frecuencia de acceso no debe estar totalmente conformado por valores iguales a 0.
Resultados obtenidos	Alerta de error. El sistema detecto que la matriz solo contiene 0.
Evaluación de la prueba	Satisfactoria, el sistema logra verificar los valores de la Matriz de frecuencia de acceso.

Tabla 6.5: Caso de prueba CP05 para verificación de valores de matriz de uso.

La Figura 6.5 muestra la alerta al ingresar una Matriz FAC donde cada fila contiene solo 0, el sistema genera una alerta avisando sobre el problema.

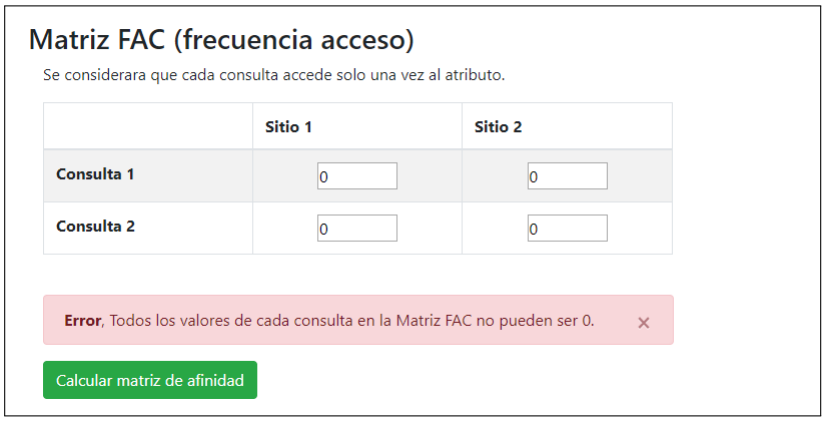


Figura 6.6: Verificar valores de matriz FAC.

6.2. Escenario para Pruebas de Fragmentación

Las pruebas consisten en la fragmentación de relaciones, usando el algoritmo COMMIN y BEA, describiendo el detalle de estas fragmentaciones.

Para la ejecución de estas pruebas se utiliza una base de datos *PostgreSQL* de nombre *bdpais* como se muestra en la Figura 6.7. Las relaciones contienen información verídica sobre los países, regiones y ciudades o comunas (según el modelo administrativo del país).

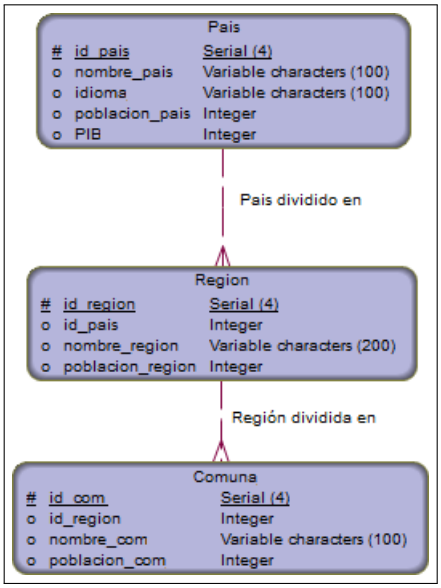


Figura 6.7: Modelo de la bases de datos *bdpais*.

El resultado de esta pruebas, pueden ser comparados con los ejercicios resueltos presentados en el Capítulo 4, de esta manera podemos verificar que los resultados obtenidos

por el sistema son correctos.

6.2.1. Pruebas de Fragmentación

En esta sección se realizan pruebas realizando el proceso completo de fragmentación utilizando el sistema.

Prueba para fragmentación horizontal, Algoritmo COMMIN

Se presenta a continuación la prueba para la fragmentación horizontal, en la Tabla 6.6 se detalla esta prueba. Además se detalla las respuestas del sistema.

Definición del caso de prueba	
ID	CP06
Descripción	Fragmentación horizontal utilizando Algoritmo COMMIN de la relación *pais* perteneciente a la base de datos *bdpais*.
Prerrequisito	Conexión a la base de datos establecida.
Datos de prueba	Condiciones de fragmentación para la relación *pais* : - $pib\_usd\_ > 2000000$. - $idioma\_pais = "Español"$.
Resultados esperados	El sistema genera la fragmentación de la relación y las relaciones derivadas (en caso que corresponda) correctamente y muestra al usuario los resultados de dicha fragmentación.
Resultados obtenidos	Se ingresa al sistema las condiciones para la fragmentación y este genera las fragmentaciones descartando las los minterminos contradictorios. Luego se presiona el botón fragmentar y el sistema realiza la fragmentación en la base de datos mostrando los resultados.
Evaluación de la prueba	Satisfactoria, el sistema genera la fragmentación y muestra resultados. Ajusta las claves foráneas de cada relación obtenida de la fragmentación derivada.

Tabla 6.6: Caso de prueba CP06 para fragmentación COMMIN.

A continuación, se realiza un seguimiento a esta fragmentación y se compara con los resultados del sistema.

En la Figura 6.8 se puede visualizar que el sistema entrega los resultados.

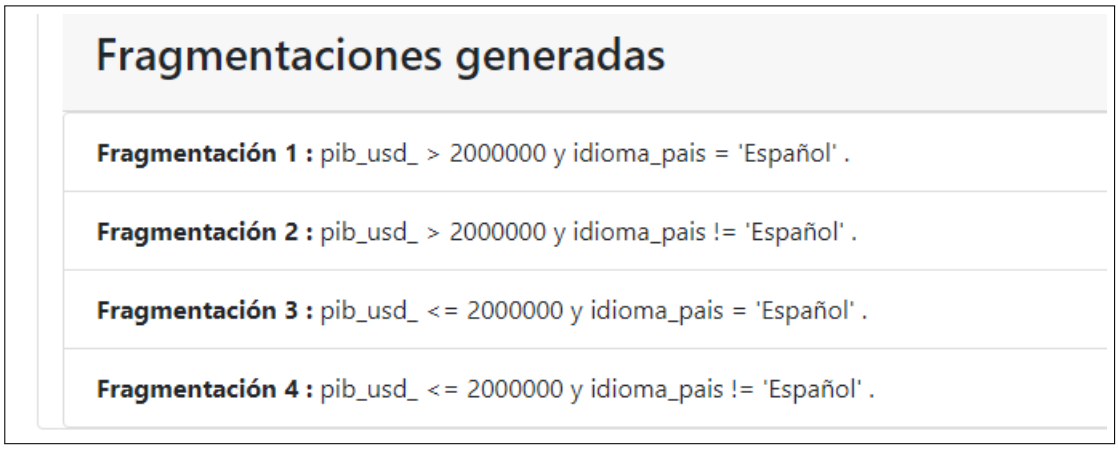


Figura 6.8: Minterminos generados en CP06 (6.6) por el sistema.

Como se indica en la Figura 6.9 , se generan las cuatro fragmentaciones correspondientes en el sistema; la Figura 6.10 muestra una de las relaciones fragmentadas y su contenido, indicando las relaciones relacionadas fragmentadas y que apuntan a ella.

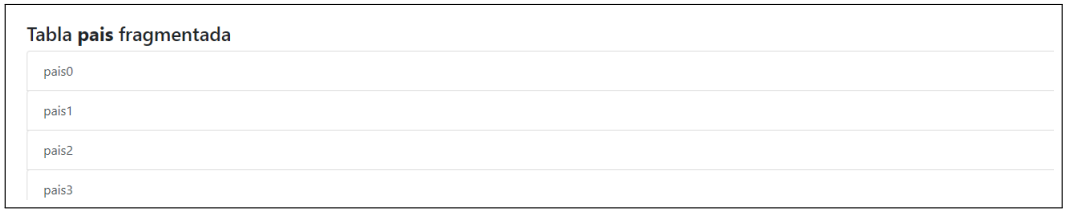


Figura 6.9: Relaciones creadas tras la fragmentación COMMIN.

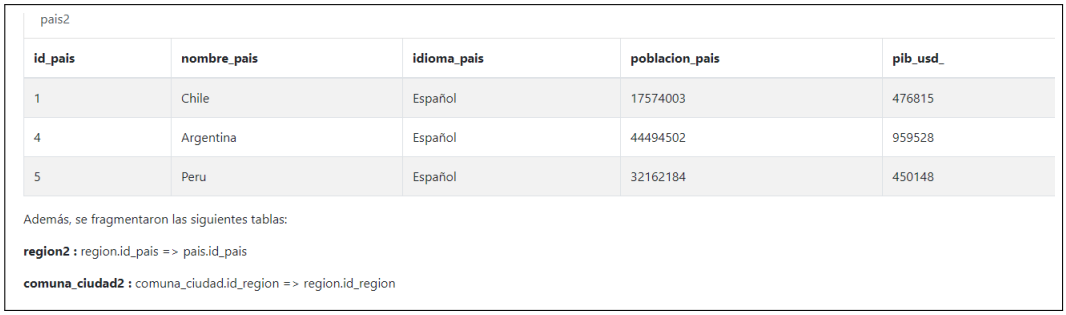


Figura 6.10: Muestra de relación expandida para fragmentación COMMIN .

La Figura 6.11 muestra una representación gráfica de la fragmentación resultante, donde se aprecia las fragmentaciones derivadas obtenidas al fragmentar la relación *pais*. Es posible comparar los resultados obtenidos anteriormente en el Ejemplo 4.3 en el Capítulo 4.

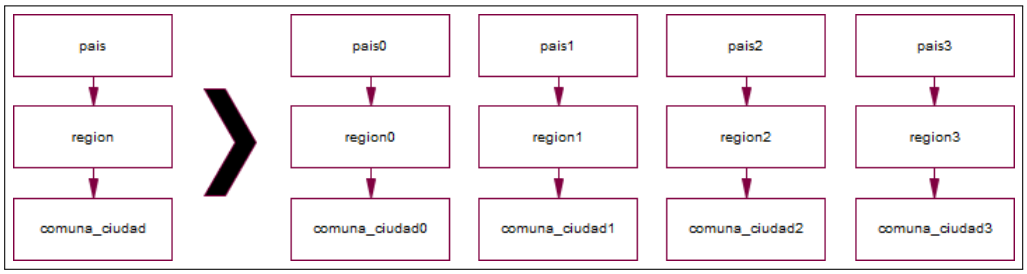


Figura 6.11: Representación gráfica de la fragmentación horizontal COMMIN.

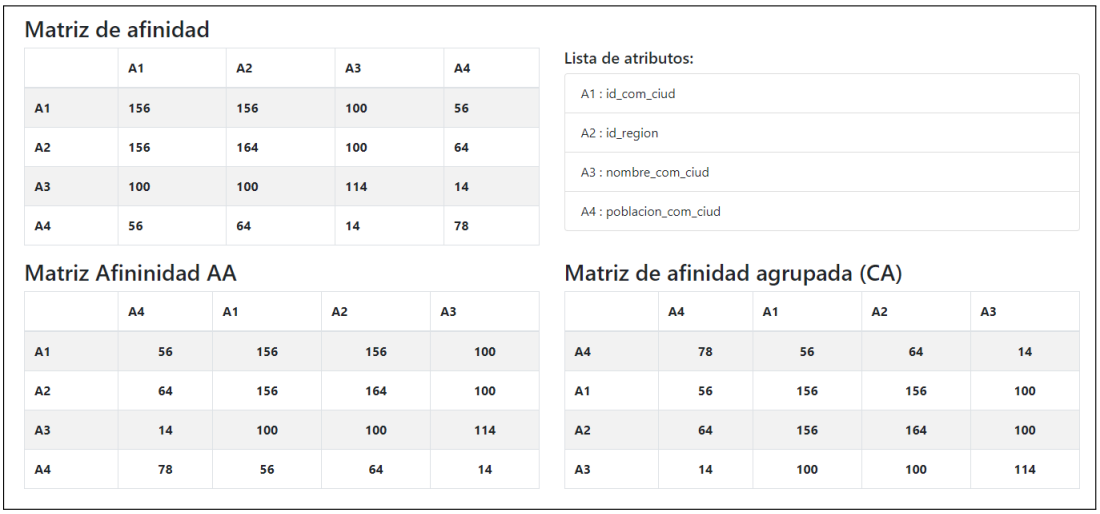
Prueba para fragmentación vertical, Algoritmo BEA

En la siguiente prueba se detalla el caso de uso para la fragmentación vertical, la Tabla 6.7 define el caso de uso, al igual que la prueba anterior se realiza un seguimiento del algoritmo.

Definición del caso de prueba	
ID	CP07
Descripción	Fragmentación vertical a través del Algoritmo BEA de la relación *comuna_ciudad* perteneciente a la base de datos *bdpais*.
Prerrequisito	Conexión a la base de datos establecida.
Datos de prueba	Tabla *comuna_ciudad* seleccionada por el usuario. Usuario ingresa la cantidad de consultas en 4. Ingresa valores para Matrices de Uso y Matriz FAC
Resultados esperados	El sistema recibe correctamente los datos ingresados por el usuario para la relación seleccionada, la Matriz de uso y la Matriz FAC. Una vez ingresados el sistema genera la Matriz de Afinidad, Matriz de Afinidad Agrupada y genera la fragmentación
Resultados obtenidos	Se ingresa correctamente la información necesaria para la poder realizar la fragmentación, el sistema genera las matrices necesarias y finalmente realiza la fragmentación.
Evaluación de la prueba	Satisfactoria, el sistema genera la fragmentación vertical y muestra resultados.

Tabla 6.7: Caso de prueba CP07 para fragmentación BEA.

En la Figura 6.12 se visualiza la vista del sistema con la Matriz Agrupada.



Matriz de afinidad

	A1	A2	A3	A4
A1	156	156	100	56
A2	156	164	100	64
A3	100	100	114	14
A4	56	64	14	78

Lista de atributos:

- A1 : id_com_ciud
- A2 : id_region
- A3 : nombre_com_ciud
- A4 : poblacion_com_ciud

Matriz Afininidad AA

	A4	A1	A2	A3
A1	56	156	156	100
A2	64	156	164	100
A3	14	100	100	114
A4	78	56	64	14

Matriz de afinidad agrupada (CA)

	A4	A1	A2	A3
A4	78	56	64	14
A1	56	156	156	100
A2	64	156	164	100
A3	14	100	100	114

Figura 6.12: Relaciones creadas tras la fragmentación COMMIN.

El sistema genera las mismas fragmentaciones calculadas en el Ejemplo 4.6 durante el Capítulo 4, se visualiza en las Figuras 6.13 y 6.14, donde se puede visualizar los atributos de cada relación como resultado de la fragmentación. Con esto se concluye que el sistema fragmenta correctamente.

Fragmentación finalizada.

La fragmentación de la tabla **comuna_ciudad** ha sido realizada con exito.

Tabla **comuna_ciudad**, antes de fragmentar

comuna_ciudad

Tabla **comuna_ciudad** fragmentada

comuna_ciudad1

id_com_ciud	poblacion_com_ciud
1	5250565
2	299830
3	185966

Figura 6.13: Fragmentación 1 generada por sistema.

Figura 6.14: Fragmentación 2 generada por sistema.

Capítulo 7

Conclusiones y Trabajos Futuros

Durante el desarrollo de este proyecto se cumple con la implementación de un sistema que permite las fragmentaciones horizontales y verticales. Fue un proceso de prueba y error, donde poco a poco fueron tomando forma los algoritmos, los descartes de predicados contradictorios en el algoritmo *COMMIN* o los distintos cálculos de matrices para el algoritmo *BEA*. Otro parte importante en la implementación de los ya citados algoritmos, fue el manejo de los *constraint* de las relaciones, especialmente la correcta creación de claves foráneas acordes a las fragmentaciones, esto termina formando una cadena de eventos que culmina en la fragmentación hasta la última de las relaciones que se vea afecta por el algoritmo.

Para el desarrollo WEB del sistema, JAVA fue el lenguaje indicado para la realización de funciones para el desarrollo de los algoritmos, la posibilidad de crear *servlets* donde se ejecutaban los métodos para la ejecución de los algoritmos y archivos *jsp* que implementa las vistas del sistema junto con la ventaja de poder incluir lenguaje JAVA dentro de el, facilito de gran manera el desarrollo de la aplicación; Del mismo modo, el uso del IDE NetBeans fue una excelente herramienta de desarrollo por su gran cantidad de funciones y asistentes. Esto significo ampliar los conocimientos enormemente en este tipo de proyectos, específicamente el desarrollo WEB con JAVA, del cual no se estaba totalmente interiorizado. En cuanto a conocimientos en cuanto al manejo de bases de datos, este proyecto permitió reforzar y aumentar el entendimiento de las fragmentaciones, permitiendo comprender toda la lógica de estas.

Para trabajos futuros, es una buena posibilidad el de implementar estos algoritmos de fragmentación bases de datos basados en otros motores tales como MySQL de Oracle o Microsoft SQL Server. Permitiendo expandir el alcance del software. Otro punto interesante es crear un sistema que permita crear las fragmentaciones y generar una base de datos nueva con las nuevas relaciones generadas, es decir una base de datos distribuida. Esto está especialmente dirigido a la fragmentación horizontal a través del algoritmo COMMIN, el cual genera fragmentaciones que pueden ser incluidas en otra base de datos específica para los requisitos dados a esa fragmentación, un ejemplo es una base de datos que divida

las relaciones dependiendo de la locación o sucursal donde se encuentre, la cual pueda no necesitar toda la información disponible y solo un segmento de esta, esta es una de las principales razones por la que se generan bases de datos distribuidas.

Referencias

M. del Carmen Gómez y Jorge Cervantes. *Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces*. Servicios Editoriales S.C. http://www.cua.uam.mx/pdfs/revistas_electronicas/libros-electronicos/2017/java/Java.pdf.

Roger S. Pressman. *Ingeniería del software, un enfoque practico*. McGraw-Hill, 2010.

Raghu Ramakrishnan. *Sistemas de gestión de bases de datos*. McGraw-Hill, 2007.

Yenisleidy Romero y Yanette Díaz. *Patrón Modelo-Vista-Controlador*. Revista Telem@tica, 2012.

M Tamer Özsu y Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

Capítulo 8

Anexo - Algoritmos, Desarrollo y Explicación

8.1. Conexión

La conexión a la base de datos, se realiza con ayuda de la librería *PostgreSQL JDBC Driver*, la cual trae una serie de utilidades para la conexión y manipulación de datos sobre bases de datos PostgreSQL, para obtener sus utilidades la librería se debe importar al proyecto, como muestra la Figura 8.1.

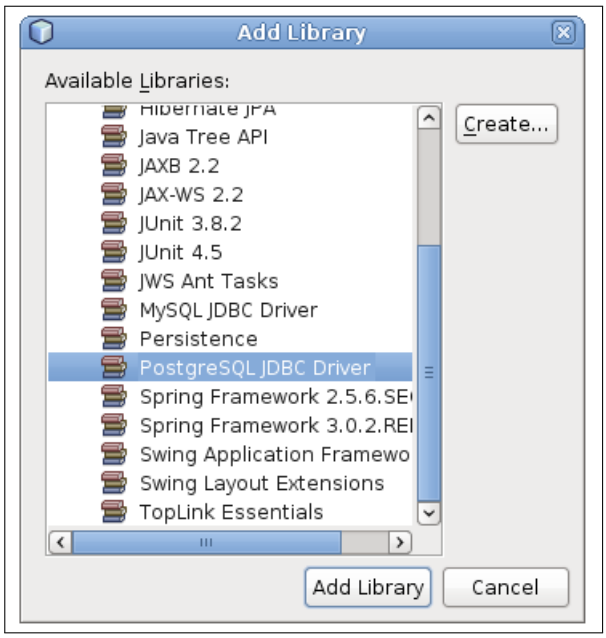


Figura 8.1: Importando librería *PostgreSQL JDBC Driver*

Una vez importada la librería, se genera la función *Conexión*, con la cual el sistema

conecta con el servidor, Postgres y la base de datos. La Figura 8.2 muestra el código con el cuál se ejecuta la conexión.

```
public Conexion(String server, String bd, String user, String pass){

    this.server = server;
    this.bd = bd;
    this.user = user;
    this.pass = pass;
}

public boolean getConexion(){
    try{
        Class.forName("org.postgresql.Driver");
        //String DB_URL = "jdbc:postgresql://"+this.server+"/"+this.bd;
        String DB_URL = "jdbc:postgresql://"+this.server+"/"+this.bd;
        this.con= DriverManager.getConnection(DB_URL, this.user, this.pass);

        return true;
    } catch(ClassNotFoundException e){
        return false;
    } catch(SQLException e){
        return false;
    }
}
```

Figura 8.2: Función *conexión*

La función es llamada cada vez que es necesario trabajar con la base de datos.

Una vez conectado a la base de datos, el sistema redirecciona a la selección de fragmentación como ya fue explicado en el Capítulo 5. A continuación, se entra a detallar el desarrollo e implementación de los puntos críticos relacionados a los algoritmos de fragmentación.

8.2. Algoritmo COMMIN para Fragmentación Horizontal

Para el desarrollo de la fragmentación horizontal a través del Algoritmo COMMIN, se sorteó una serie de dificultades, en los siguientes puntos se explican a través del proceso realizado por el sistema a la hora de realizar la fragmentación horizontal como fueron solucionados.

Como primer punto importante una vez el usuario a seleccionado la relación a fragmentar, se debe definir las condiciones con las cuales se ejecuta el Algoritmo COMMIN, pero el problema encontrado es que dichas condiciones deben ser acordes al tipo de dato sobre la que se está definiendo la condición. Para ello la solución implementada la indica la Figura 8.3:

```
public boolean getSelect(String cond, String tabla) throws SQLException {
    try {
        this.conexion();
        this.consulta = this.con.prepareStatement("Select * from " + tabla + " Where " + cond + "");
        this.datos = this.consulta.executeQuery();
        return true;
    }catch(SQLException e){
        e.printStackTrace();
        return false;
    }

}
```

Figura 8.3: Función *getSelect*

La función *getSelect* simplemente toma la relación, la columna, el operador y la condición que el usuario a ingresado y realiza una consulta $SELECT$ sobre la relación a fragmentar. La función retorna un valor *true* si la consulta se realizó (la condición es acorde a la columna) o un valor $false$ lo que significa hubo un error al intentar ejecutar la consulta, por lo tanto, la condición es errónea. Si la condición es correcta, esta es agregada junto a su versión contraria, es decir, si el operador de la condición es $>$, entonces también se genera una condición con el operador $<=$.

Una vez agregada las condiciones, el sistema pasa a simular el Algoritmo COMMIN. El siguiente paso es generar los predicados, que en este caso el sistema los crea a través de una simulación. La Figura 8.4 muestra el como el sistema genera dichos predicados.

```
ArrayList<String[]> pred = new ArrayList<String[]>();
ArrayList<ArrayList<String[]>> arrayPred = new ArrayList<ArrayList<String[]>>();
float totalAux = total / 2;
int cont = 0, ind = 0;
String aux;
boolean flag = true;
while (!mintermAux2.isEmpty()) {
    pred.clear();
    for (int i = 0; i < total; i++) {
        if (i == 0) {
            aux = mintermPrueba.get(i);
            mintermPrueba.set(i, aux + mintermAux2.get(0) + " AND ");
            pred.add(new String[]{minPred.get(0)[0], minPred.get(0)[1], minPred.get(0)[2]});
        } else {
            if (flag) {
                aux = mintermPrueba.get(i);
                mintermPrueba.set(i, aux + mintermAux2.get(0) + " AND ");
                pred.add(new String[]{minPred.get(0)[0], minPred.get(0)[1], minPred.get(0)[2]});
            } else {
                aux = mintermPrueba.get(i);
                mintermPrueba.set(i, aux + mintermAux2.get(1) + " AND ");
                pred.add(new String[]{minPred.get(1)[0], minPred.get(1)[1], minPred.get(1)[2]});
            }
        }
        cont++;
        if (cont == totalAux) {
            flag = !flag;
            cont = 0;
           }
        }
        arrayPred.add(new ArrayList<String[]>(pred));
        ind = ind + 1;
        totalAux = totalAux / 2;
        mintermAux2.remove(0);
        mintermAux2.remove(0);
        minPred.remove(0);
        minPred.remove(0);
    }
```

Figura 8.4: Generación de predicados para Algoritmo COMMIN

El código puede resultar difícil de entender, pero la mejor forma de entenderlo es de la siguiente manera: En el paso anterior, se generaron un conjunto de condiciones para fragmentar, lo que hace este código es generar los 2^n números de predicados simples y minimales. Donde a todos los predicados generados se agregan una condición a la vez. En cada iteración del código, se añaden condiciones a los predicados, y cada predicado es asignado a un *ArrayList*. Un ejemplo para entender es la siguiente donde solo se crean dos predicados para términos prácticos:

Ejemplo 8.1 *Para la generación de predicados:*
Iteración 1

- *ArrayList 1: sueldo >30000.*
- *ArrayList 2: sueldo <= 30000.*

Iteración 2

- *ArrayList 1: sueldo >30000, ciudad = "Concepción".*
- *ArrayList 2: sueldo <= 30000, ciudad != "Concepción".*

Iteración 3

- *ArrayList 1: sueldo >30000, ciudad = "Concepción", edad <40.*
- *ArrayList 2: sueldo <= 30000, ciudad != "Concepción", edad >= 40*

Pero no solo se crea una lista con las condiciones, sino que al mismo tiempo, se generan un *ArrayList* que contiene variables de tipo *String*, el cual contiene las condiciones para ser usadas directamente en script SQL y permitir generar las fragmentaciones directamente con las listas generadas, es decir:

Ejemplo 8.2 *Genera los script SQL:*
Iteración 1

- *String 1: sueldo >30000 .*
- *String 2: sueldo <= 30000 .*

Iteración 2

- *String 1: sueldo >30000 AND ciudad = "Concepción".*
- *String 2: sueldo <= 30000 AND ciudad <>"Concepción".*

Iteración 3

- *String 1: sueldo >30000 AND ciudad = "Concepción" AND edad <40 .*
- *String 2: sueldo <= 30000 AND ciudad <>"Concepción" AND edad >= 40 .*

Esto permite tener 2 Listas donde cada posición comparte el mismo predicado objetivo, pues el siguiente paso es descartar los predicados redundantes y contradictorios, eliminando de las dos listas el elemento asignado a la posición al cual pertenece el predicado descartado.

El descarte se realiza dentro un ciclo en el cuál cada condición es comparada con cada uno de los otras condiciones pertenecientes a esa lista. Verificando la columna, el operador y el valor de cada condición. Una vez realizadas los descartes y obtenidos los predicados definitivos, el sistema entra en el proceso de fragmentar directamente a la base de datos. Para ello se crean relaciones idénticas a la original 8.5.

```
public void fragmentarNuevaTabla(String tablaFragmentada, String sentenciaSelect,
        String tablas, String condiciones, String sentenciaRelacion) throws SQLException {
    Statement st = con.createStatement();
    st.execute("CREATE TABLE " + tablaFragmentada + "_copia \n"
            + "(LIKE " + tablaFragmentada + " INCLUDING ALL);\n"
            + "INSERT INTO " + tablaFragmentada + "_copia \n"
            + "SELECT DISTINCT " + sentenciaSelect + " \n"
            + "FROM " + tablas + " \n"
            + "WHERE " + condiciones + " " + sentenciaRelacion + "");
    st.close();
}
```

Figura 8.5: Función para generar copias de la relación original

En la Figura 8.5 anterior, la función *FragmentarNuevaTabla* crea una copia exacta de la relación original e inserta los atributos de la lista que contenía las condiciones compatible con SQL.

El segundo paso al crear las relaciones fragmentadas, es agregar los *contraints* de la relación, esto se refiere entre otros a las claves foráneas de la relación, pues al crear la copia de las relaciones utilizando la cláusula *INCLUDE ALL* genera una relación idéntica, pero sin incluir los *constraint*. Para solucionar este problema se crearon dos funciones, una para obtener todos los *constraint* de una relación (Figura 8.6) y una segunda función para agregarlos a la relación fragmentada nueva (Figura 8.7).

```
public ResultSet getConstraintsFK(String tabla) throws SQLException {
    this.conexion();
    this.consulta = this.con.prepareStatement("SELECT tc.table_name, kcu.column_name, \n"
            + "    ccu.table_name AS foreign_table_name,\n"
            + "    ccu.column_name AS foreign_column_name \n"
            + "FROM \n"
            + "    information_schema.table_constraints AS tc \n"
            + "    JOIN information_schema.key_column_usage AS kcu\n"
            + "      ON tc.constraint_name = kcu.constraint_name\n"
            + "    JOIN information_schema.constraint_column_usage AS ccu\n"
            + "      ON ccu.constraint_name = tc.constraint_name\n"
            + "WHERE constraint_type = 'FOREIGN KEY' AND tc.table_name='"+tabla+"';");
    this.datos = this.consulta.executeQuery();
    return this.datos;
}
```

Figura 8.6: Función *getConstraintsFK*

```
public void agregarConstraint(String tablaHijo, String columName,
                              String tablaPadre, String foreignColumn,
                              int a) throws SQLException {
    Statement st = con.createStatement();
    st.execute("ALTER TABLE "+tablaHijo+" \n"
            + "   ADD CONSTRAINT "+tablaHijo+tablaPadre+a+"\n"
            + "   FOREIGN KEY ("+columName+") \n"
            + "   REFERENCES "+tablaPadre+"("+foreignColumn+");");
    st.close();
}
```

Figura 8.7: Función *agregarConstraint*

Luego, en caso de existir una fragmentación derivada, se utiliza la función *getTablasRelacionadas* (Figura 8.8) que a través del *schema*, que utilizan todas las bases de datos PostgreSQL se obtienen las relaciones asociadas obteniendo las claves primarias y foráneas. El proceso de fragmentación es similar al ya descrito anteriormente, insertando en cada relación fragmentada las tuplas relacionadas a la clave foránea de la misma.

```
public ResultSet getTablasRelacionadas(String clavePrimaria) throws SQLException {
    this.conexion();
    this.consulta = this.con.prepareStatement("SELECT tc.table_name as tabla_referencia "
            + "FROM information_schema.table_constraints AS tc "
            + "JOIN information_schema.key_column_usage AS kcu ON tc.constraint_name = kcu.constraint_name "
            + "JOIN information_schema.constraint_column_usage AS ccu "
            + "ON ccu.constraint_name = tc.constraint_name "
            + "WHERE constraint_type = 'FOREIGN KEY' and ccu.column_name='"+clavePrimaria+"'");
    this.datos = this.consulta.executeQuery();
    return this.datos;
}
```

Figura 8.8: Función *getTablasRelacionadas*

8.3. Algoritmo BEA para Fragmentación Vertical

El desarrollo del Algoritmo COMMIN, es principalmente un cálculo de distintas matrices. En primer lugar, el usuario debe seleccionar una relación, la cual debe tener tres o más atributos, sino, la fragmentación no se podría realizar. Para solucionar esto, se utilizó el código de la Figura (8.9).

```
while(rstablas.next()){
    tablas.add(rstablas.getString("nom_tabla"));
}

for(String tabla : tablas){
    rsSelectAll = c.selectAll(tabla);
    rsmd = rsSelectAll.getMetaData();
    numColumnas = rsmd.getColumnCount();
    if(numColumnas >= 3){
        tablasBEA.add(tabla);
    }
}
```

Figura 8.9: Selección de relaciones con más de tres columnas

Una vez se selecciona la relación en el sistema, se ingresa la *Matriz de Uso*, a través de un formulario que utiliza como dimensiones el número de atributos de la relación seleccionada y la cantidad de consultas que el usuario desee. La Figura 8.10 muestra como son asignadas los valores ingresados mediante un formulario a la variable $matrizUso$.

```
int matrizUso[][]= new int[numeroConsultas][numeroColumnas];
String coor;
for(int i = 0; i<numeroConsultas; i++){
    for(int j = 0; j<numeroColumnas; j++){
    matrizUso[i][j]=Integer.parseInt(request.getParameter(String.valueOf(i)+String.valueOf(j)));
    coor = String.valueOf(i)+String.valueOf(j);
    session.setAttribute("coor"+coor, Integer.parseInt(request.getParameter(String.valueOf(i)+String.valueOf(j))));
    }
}
```

Figura 8.10: Asignación de valores $matrizUso$

Otro punto importante al implementar la *Matriz de Uso* es verificar que en cada una de sus filas no sean todas valores cero. Esto se soluciona con un simple recorrido fila por fila sumando los valores de cada elemento, si ese total calculado es cero, significa que toda la fila solo contiene números 0. La solución implementada la indica la Figura 8.11, donde se utiliza una variable de tipo *boolean* para saber el estado en que se encuentra la matriz.

```
int total;
boolean MUvacio=true;
for(int i = 0; i<numeroConsultas; i++){
    total = 0;
    for(int j = 0; j<numeroColumnas; j++){
    total = total + matrizUso[i][j];
    }
    if(total == 0){
        MUvacio=false;
    }
}
```

Figura 8.11: Verificar valores de Matriz de Uso - implementación

Una vez procesada la *Matriz de Uso*, el sistema necesita obtener la *Matriz FAC*, utilizando un identico al anterior descrito. Las Figuras 8.12 y 8.13 obtienen los valores ingresados y verifican que no todos los valores sean ceros respectivamente.

```
int matrizFAC[][] = new int[numeroConsultas][totalSitios];
for (int i = 0; i < numeroConsultas; i++) {
    for (int j = 0; j < totalSitios; j++) {
        session.setAttribute("mfac"+String.valueOf(i)+String.valueOf(j),
                Integer.parseInt(request.getParameter(String.valueOf(i) + String.valueOf(j))));
        matrizFAC[i][j] = Integer.parseInt(request.getParameter(String.valueOf(i) + String.valueOf(j)));
    }
    out.println("<br>");
}
```

Figura 8.12: Asignación de valores Matriz FAC - implementación

```
boolean FACvacio = false;
int total;
for (int i = 0; i < numeroConsultas; i++) {
    total = 0;
    for (int j = 0; j < totalSitios; j++) {
        total = total + matrizFAC[i][j];
    }
    if(total == 0){
        FACvacio = true;
    }
}
```

Figura 8.13: Verificar valores de matriz FAC

El siguiente paso para el sistema generar la fragmentación vertical, es calcular la *Matriz de Afinidad*, para calcularla el sistema utiliza las variable $matrizUso$ y $matrizFAC$. La *Matriz de Afinidad* tiene una dimensión de nxn donde n es el número de columnas. La Figura **??** muestra cómo fue implementado el cálculo.

```
int matrizAfinidad[][] = new int[numeroColumnas][numeroColumnas];
for (int fil = 0; fil < numeroColumnas; fil++) {
    for (int col = 0; col < numeroColumnas; col++) {
        for (int i = 0; i < numeroConsultas; i++) {
            if (matrizUso[i][fil] == 1 && matrizUso[i][col] == 1) {
                matrizAfinidad[fil][col] = matrizAfinidad[fil][col]+ calcularPosFAC(i,1,totalSitios,matrizFAC);
            }
        }

    }
}
```

Figura 8.14: Cálculo de matriz de afinidad

La Figura 8.14 contiene 3 ciclos iterativos, uno dentro de otro, los dos primeros son para recorrer cada elemento de la *Matriz de Afinidad*, mientras que el ultimo es para recorrer la *Matriz FAC*. Antes de calcular el valor se debe verificar que la posición de la matriz *AA* con la cual se realiza el cálculo es válida, por lo tanto se verifica que la [fila] y [columna] de la *Matriz de Uso* sea igual a 1. El cálculo se realiza como ya fue mencionado en la sección 4.1. Para este cálculo se utiliza una función auxiliar llamada *calcularPosFAC* (Figura 8.15) que calcula el valor total de cada fila de la *Matriz FAC* multiplicada por la variable *numAcc* que indica el número de accesos de esa aplicación (columna), por simplicidad el valor de *nuAcc* se asume como 1, es decir cada aplicación solo accede una vez a cada atributo.

```
public int calcularPosFAC(int fila, int numAcc,int totalSitios, int matrizFAC[][]) {
    int result = 0;
    for (int i = 0; i < totalSitios; i++) {
        result = result + matrizFAC[fila][i]*numAcc;
    }
    return result;
}
```

Figura 8.15: Función *calcularPosFAC*

Antes de continuar es necesario aclarar que la última parte de la implementación para el **Algoritmo BEA** (desde el cálculo de la **Matriz Agrupada CA** hasta realizar el agrupamiento) contiene el código más complejo y largo de todo el sistema. Por lo tanto, en su mayoría se realiza una explicación general de lo desarrollado y solo se ahonda en pequeños trozos importantes del código desarrollado.

El sistema necesita generar la *Matriz de Afinidad Agrupada*. Para eso se crea una matriz auxiliar *CA*, la peculiaridad de esta matriz es que contiene dos columnas extras sobre el número de atributos originales y una fila más, el sentido de estas columnas extras es tener çolumnas exteriores"que faciliten el cálculo de la variable *cont*, esta matriz es inicialiada con valores 0. y se agregan las 2 primeras filas de la *Matriz de Afinidad*, pero estas columnas son insertadas desde la posición [1] dejando la primera columna vacía.

```
for (int i = 2; i < numeroColumnas; i++) {
    cont = 0;
    for (int j = 0; j < numeroColumnas; j++) {
        contAux = 0;
        contAux = calcularCont(i, j, matrizAfinidad, matrizCA, numeroColumnas);
        if (cont <= contAux) {
            orden[0] = j;
            orden[1] = i;
            orden[2] = j + 1;
            cont = contAux;
        }
    }
    insertarColumna(orden, matrizAfinidad, matrizCA, numeroColumnas);

}
```

Figura 8.16: Cálculo matriz agrupada

La Figura 8.16 muestra el trozo de código más significativo a la hora de calcular la matriz CA. En ella se aprecia el cálculo del valor *cont* utilizando el método *calcularCont* (Figura 8.17). También es posible ver el llamado al método *insertarColumna* (Figura 8.18) que se encarga de reorganizar las matrices según el orden obtenido del valor más alto de la variable *cont*.

```
public int calcularCont(int indexMA, int indexMAG, int matrizAfinidad[][], int matrizCA[][], int numeroColumnas) {
    int bondIzq = 0;
    int bondDer = 0;
    int bondExtremos = 0;
    int cont = 0;

    for (int i = 0; i < numeroColumnas; i++) {
        bondIzq += matrizAfinidad[i][indexMA] * matrizCA[i][indexMAG];
    }
    for (int i = 0; i < numeroColumnas; i++) {
        bondDer += matrizAfinidad[i][indexMA] * matrizCA[i][indexMAG + 1];
    }
    for (int i = 0; i < numeroColumnas; i++) {
        bondExtremos += matrizCA[i][indexMAG] * matrizCA[i][indexMAG + 1];
    }

    cont = 2 * bondIzq + 2 * bondDer - 2 * bondExtremos;
    return cont;
}
```

Figura 8.17: Método *calcularCont*

```
public void insertarColumna(int orden[], int matrizAfinidad[][], int matrizCA[][], int numeroColumnas)
    for (int j = numeroColumnas; j >= orden[0]; j--) {
        if (j == orden[0]) {
            for (int a = 0; a < numeroColumnas; a++) {
                matrizCA[a][j + 1] = matrizAfinidad[a][orden[1]];
            }
            matrizCA[numeroColumnas][j + 1] = orden[1];
        } else {
            for (int a = 0; a < numeroColumnas; a++) {
                matrizCA[a][j + 1] = matrizCA[a][j];
            }
            matrizCA[numeroColumnas][j + 1] = matrizCA[numeroColumnas][j];
        }

    }
}
```

Figura 8.18: Método *insertarColumna*

Una vez calculada la **Matriz Agrupada CA**, el sistema realiza un re-ordenamiento de la matriz CA. La muestra de código para esta implementación se obvia debido al tamaño y complejidad del mismo. Pero su objetivo es el mismo, calcular las variables CTQ, CBQ, COQ y con ellos calcular el mejor agrupamiento posible como fue descrito en la sección 4.2.

Una vez el sistema realizo el agrupamiento y generados los conjuntos TA y BA los cuales son almacenados en *ArrayList*, el sistema verifica que los conjuntos contengan la *clave primaria*. Como se muestra en la Figura 8.19.

```
//Añade el PK al array que no lo contenga
Boolean pk = false;
for (int i = 0; i < PKList.size(); i++) {
    if (TAList.contains(PKList.get(i))) {
        //TA ya tiene pk
        pk = true;
    }
}
if (!pk) {
    for (int i = 0; i < PKList.size(); i++) {
        TAList.add(0, PKList.get(i));
    }
}

pk = false;
for (int i = 0; i < PKList.size(); i++) {
    if (BAList.contains(PKList.get(i))) {
        //BA ya tiene pk
        pk = true;
    }
}
if (!pk) {
    for (int i = 0; i < PKList.size(); i++) {
        BAList.add(0, PKList.get(i));
    }
}
```

Figura 8.19: Agregar claves primarias a conjuntos TA y BA

Finalmente el último paso para fragmentar a través el **Algoritmo COMMIN** del sistema es realizar la fragmentación en la base de datos. Para ello genera copias y agrega los *constraint* respectivos al igual que en el **Algoritmo COMMIN** tanto para la relación que contiene asignada al conjunto TA como para el conjunto BA, pero la diferencia es

que esta vez el sistema elimina los atributos de las relaciones que no pertenezcan a los conjuntos TA y BA respectivos. Con esta implementación, el algoritmo es finalizado.