UNIVERSIDAD SIMÓN BOLÍVAR

Ingeniería de la Computación

Extensión de *PostgreSQL* con Mecanismos de Optimización de Consultas Basadas en Preferencias

Por

Fabiola R. Di Bartolo L., Francelice J. Sardá S.

Proyecto de Grado

Presentado ante la Ilustre Universidad Simón Bolívar

como Requerimiento Parcial para Optar al Título de

Ingeniero de la Computación

Sartenejas, Septiembre de 2007

UNIVERSIDAD SIMÓN BOLÍVAR

DECANATO DE ESTUDIOS PROFESIONALES

COORDINACIÓN DE INGENIERÍA DE COMPUTACIÓN

ACTA FINAL DEL PROYECTO DE GRADO

EXTENSIÓN DE *POSTGRESQL* CON MECANISMOS DE OPTIMIZACIÓN DE CONSULTAS BASADAS EN PREFERENCIAS

Presentado Por:

FABIOLA R. DI BARTOLO L., FRANCELICE J. SARDÁ S.

Este proyecto de Grado ha sido aprobado por el siguiente jurado examinador:

Prof. Soraya Carrasquel

Prof. Víctor Fuentes

Prof. Marlene Goncalves Da Silva (Tutor Académico)

SARTENEJAS, 23 de octubre de 2007

Extensión de *PostgreSQL* con Mecanismos de Optimización de Consultas Basadas en Preferencias

Por

Fabiola R. Di Bartolo L., Francelice J. Sardá S.

RESUMEN

El paradigma de consultas basadas en preferencias ha surgido como una respuesta a la necesidad de sistemas de programas que soporten la toma de decisiones. Cada vez más, se ha mostrado cómo el acoplamiento directo de este tipo de consultas a la capa física de las aplicaciones se traduce en manejadores de bases de datos que pueden aproximar eficientemente los deseos de usuarios, en base a los datos que en ellos existen. Un caso particular de las consultas basadas en preferencia es el *Skyline*. El *Skyline* resuelve las preferencias estableciendo un orden parcial sobre los datos, inducido por las condiciones que el usuario quisiera que se cumplieran. Ésta es una solución bastante popular a las consultas preferenciales, por lo que existe una amplia gama de trabajos relacionados con la inclusión del *Skyline* como un operador agregado dentro de los manejadores de base de datos.

Sin embargo, al ser un operador que halla una respuesta aproximada de lo que el usuario desea, supone un procesamiento complejo de los datos existentes y un conjunto de datos respuesta mucho menor que los existentes en la base de datos. Esto ha motivado el desarrollo de trabajos que se orienten a la optimización especializada de las consultas *Skyline* donde se considere construir planes que hagan un *push-down* del operador en el plan de evaluación, con el objetivo de disminuir el tamaño de los resultados intermedios y poder obtener el *Skyline* de manera eficiente.

En este trabajo se detalla la extensión realizada a *PostgreSQL* para permitir la optimización de consultas *Skyline*. Se proponen dos algoritmos de optimización, *dPeaQock* y *ePeaQock*, y un modelo de costo de mayor generalidad que los propuestos en trabajos anteriores. Durante el desarrollo de este proyecto se logró la extensión del planificador y del modelo de costo de *PostgreSQL*, con resultados notablemente favorables a las hipótesis aquí propuestas. Los algoritmos y el modelo de costo aquí propuestos mostraron funcionar bastante bien para datos uniformes y reales.

Finalmente, como resultado de este trabajo de investigación se concluye que la optimización especializada de consultas *Skyline* se traduce en mejores tiempos de evaluación, por ende en la obtención rápida y eficiente de resultados de interés para el usuario.

Agradecimientos

Agradecemos a Dios por darnos fortaleza y perseverancia en todo momento.

Agradecemos a nuestros padres Solange Lara, Beatriz Silva, Gaetano Di Bartolo y Oscar Sardá por apoyarnos y comprendernos siempre. A Astrid quien con sólo 5 años de edad supo comprender las ausencias de su mamá. A mamá Celina por su Fé en nosotras. A Andreina Sardá por su apoyo incondicional.

Agradecemos a todos los que nos ayudaron y nos alegraron la vida durante el desarrollo de este trabajo: Diego Guerrero, Pedro Piñango, Marynel Vázquez, Celso Gorrín, Tomás Lampo, Astrid Paravisini, Jesús Graterol, Eduardo Ruíz, Carolina Chang, Carolina Martínez, Carlos Gómez y Yubiley Chang. A todos nuestros amigos que siempre estuvieron pendiente.

Agradecemos a nuestra tutora Marlene Goncalves quien no restringió nuestra creatividad en este trabajo, siempre nos guió acertadamente y con mucha paciencia.

Agradecemos a Carmen Brando, Vanessa González y Maria Esther Vidal por apartar un poquito de su tiempo para asesorarnos en los momentos requeridos.

Índice general

Índice de cuadros

Índice de figuras

Glosario de Términos

Álgebra relacional Uno de los dos lenguajes formales para consultas asociados al modelo relacional. El Álgebra relacional se encuentra definida por un conjunto de operadores básicos (selección, proyección, producto cartesiano, unión y diferencia) y un conjunto de reglas de equivalencia para la composición de estos operadores.

Análisis léxico Primera etapa del análisis de una consulta, en la cual se reconocen las palabras reservadas que se encuentran en ella.

Análisis semántico Tercera y última etapa del *parse* de una consulta, en la cual se validan los identificadores de tablas y atributos utilizados en las consultas.

Análisis sintáctico Segunda etapa del *parse* de una consulta, en la cual se reconoce la estructura de la misma y se verifica el orden de colocación de las cláusulas.

BNL Algoritmo *Block Nested loop*. Utilizado para ejecutar un operador lógico *Skyline* que consiste en ir almacenando las tuplas no dominadas en una ventana de dominantes .

Conjunto Skyline Es el conjunto de tuplas que satisfacen la ejecución de un operador lógico *Skyline*. Matemáticamente, son el conjunto de maximales del orden parcial establecido por las dimensiones de un *Skyline*.

Criterio Skyline Es el par compuesto por una dimensión y una directiva de maximización, minimización o agrupación.

Demonio Proceso que se encuentra en ejecución constantemente.

Dominancia Se dice que una tupla a domina a una tupla b si $a \succ b$ en el orden parcial establecido por las dimensiones de un *Skyline*.

Directiva Partícula atómica de una cláusula *SKYLINE OF* que acompaña a un atributo para indicar información de lo que se desea sobre los valores de ese atributo.

Dimensión Partícula de una cláusula *SKYLINE OF* correspondiente a un atributo.

Espacio de búsqueda Conjunto de soluciones que existen para un mismo problema.

Enfoque Skyline Enfoque que selecciona las tuplas que no son dominadas por ninguna otra tupla.

Evaluación	Es el proceso de construir la respuesta de una consulta mediante la ejecución de un plan de evaluación. Este proceso es llevado a cabo en el evaluador o ejecutor de un SMBD.
Evaluador	Es el componente del sistema manejador de bases de datos que evalua el plan de ejecución proporcionado por el optimizador con el objeto de obtener los resultados de una consulta dada.
Factor de bloqueo	Es la relación existente entre el tamaño de una tupla y el tamaño de un bloque de disco.
Función multicriterio	Es una función formada por múltiples criterios. Aplicado al *SKYLINE OF* comprende el conjunto de dimensiones definidas para esta cláusula.
Nodo o operador *top-level*	Operadores que son colocados en la raíz del plan generado por *PostgreSQL*.
Nodo o operador *down-level*	Operadores que son considerados en la primera etapa de optimización de *PostgreSQL*.
Optimizador	Es el componente del sistema manejador de bases de datos encargado de obtener un plan de ejecución para una consulta dada.
Operador Agregado	Operador relacional que no pertenece al álgebra relacional tradicional. Se consideran operadores tope, es decir, colocados en la raíz del árbol que representa el plan de evaluación.
Operador Físico	Es el algoritmo de ejecución con el que el manejador va a ejecutar un operador lógico.
Operador Relacional o Lógico	Es la representación lógica de la funcionalidad de un bloque SQL. Representa lo que hace la cláusula, no cómo lo hace.
Parser	Es un componente del sistema manejador de base de datos encargado de verificar la correctitud léxico-sintáctica y semántica de una consulta.
PeaQock	Extension de PostgreSQL con mecanismos de evaluación y optimización para consultas Skyline.
Pipeline	Método de transferencia de datos entre operadores que consiste en ir enviando las tuplas que pertenecen a la relación respuesta de aplicar un operador, para que el siguiente la vaya procesando

Plan de evaluación	Estructura de datos que le indica al módulo evaluador qué operadores físicos ejecutar y en qué orden.
Plan lógico	Estructura de datos que almacena el orden en el que se van a realizar los operadores lógicos.
PostgreSQL	Manejador de base de datos relacional disponible en http://www.postgresql.org.
Push-down de un operador	Realizar el *push-down* de un operador es adelantar su evaluación en el plan. A nivel de estructura de representación de un plan, es colocar el operador más cerca de la hojas.
Reescritor	Es un componente del sistema manejador de base de datos responsable de reescritura de vistas.
Relación	Unidad de información en el modelo relacional. Puede ser visto como un conjunto de tuplas que tienen los mismos atributos.
SFS	Algoritmo *Sort Filter Skyline*. Es un algoritmo que introduce una mejora a BNL mediante un ordenamiento previo dado por el ordenamiento en función de las dimensiones, y así en la ventana se introducen solamente tuplas que conforman el conjunto Skyline.
Tupla	En una relación, una tupla es la unidad mínima de información. En el modelo Entidad-Relación pudiera aproximarse al concepto de instancia de una entidad.

Capítulo 1

Introducción

Actualmente, existe una creciente demanda de sistemas capaces de soportar consultas que respondan a las preferencias de los usuarios. Por ello, se requiere contar con aplicaciones capaces de interpretar las preferencias de un usuario y en base a eso ofrecerle un rango de opciones que pudieran satisfacerlos. En este marco se han desarrollado las consultas basadas en preferencias. Éstas pretenden ampliar el conjunto de consultas que se pueden realizar a un manejador, con el objeto de introducir restricciones flexibles. Suponen un mayor procesamiento sobre los datos porque su objetivo es aproximarlos a los deseos que el usuario ha provisto, es decir, ofrecerle lo que mejor satisface sus deseos de los datos existentes.

Una de las soluciones propuestas para calcular las preferencias de los usuarios es el *Skyline* [13]. Este paradigma se ha orientado, entre otras cosas, al soporte de decisiones. Tal es el caso de un usuario que accede a un sitio *Web* para realizar una compra de una computadora de escritorio. Éste aspira obtener una computadora con una buena cantidad de memoria RAM y un buen procesador, al menor costo posible. Todas estas condiciones son deseables para el usuario, pero es posible que mientras mejor sea el procesador y mayor la cantidad de memoria, mayor sea el precio. En una consulta *Skyline*, el usuario definiría que quiere maximizar la cantidad de megas de la memoria RAM, maximizar la velocidad y generación del procesador, y a su vez desea minimizar el costo de la computadora.

El *Skyline* soluciona el cálculo de este tipo de consultas mediante el establecimiento de un orden parcial entre las tuplas de las relaciones correspondientes. Este ordenamiento es inducido por la maximización o minimización simultánea de atributos de dicha relación. Las tuplas que formarán parte de la respuesta son las que pertenezcan al conjunto maximal de ese orden parcial. Calcular el *Skyline* es un problema clásico de computación: la maximización de funciones multicriterio o problema del vector máximo [28, 48]; y la respuesta obtenida generalmente es llamada conjunto *Skyline*.

Se ha reconocido que la implementación eficiente del *Skyline* requiere la introducción de un nuevo operador dentro de los manejadores de base de datos [16]. Para esto se han propuesto diversos algoritmos de implementación[13, 21, 43, 48, 42, 7] que permiten la integración del *Skyline* a los manejadores de base de datos, mediante la adición de un operador agregado. Sin embargo, estudios recientes han demostrado que no basta con incluir al *Skyline* como un operador agregado, sino que se debe estudiar su inclusión en el proceso de optimización para obtener mejores resultados en la evaluación de este tipo de consultas.

El *Skyline* tiene un comportamiento particular. El conjunto *Skyline* generalmente tiene una cardinalidad mucho menor que el conjunto de datos sobre el cual se calculó, de manera similar al operador de selección[16]. Por otro lado, a diferencia del operador de selección, el cálculo del *Skyline* consume altos recursos de procesamiento, a la par que aumenta el tamaño del conjunto respuesta al agregar una condición de preferencia [32]. Asimismo, este comportamiento pudiera indicar que, a diferencia del operador de selección, no siempre será bueno ejecutar el operador de preferencias antes que el resto.

Volviendo al ejemplo, nótese que intuitivamente resulta más práctico y eficiente observar por separado las mejores opciones para cada marca existente de memoria y procesador y luego construir configuraciones de computadoras con esas especificaciones. Obsérvese que esto no es más que dividir las preferencias del usuario en subconjuntos que se apliquen de manera separada a cada condición de sus deseos, para después volver a aplicar las preferencias, pero sobre el conjunto más selecto de opciones.

Con este ejemplo sencillo, se introduce al concepto de optimización de consultas *Skyline*. Se muestra que el considerar aplicar primero un subconjunto de preferencias, puede reducir el volumen de datos a ser procesados posteriormente y así obtener respuestas de una manera más eficiente. Sin embargo, cuando al adelantar la aplicación de esas preferencias se obtiene la misma cantidad de opciones que sin adelantarlas, se está realizando un trabajo costoso sin necesidad.

Trabajos anteriores han mostrado, de manera conceptual o práctica, el beneficio de realizar un proceso de optimización que incluya al *Skyline*, así como han establecido el marco teórico y conceptual referente a este procedimiento [20]. Sin embargo, estos trabajos han llevado orientaciones muy distintas en cuánto al marco experimental considerado y a los resultados obtenidos. En [16], se implementó una extensión de *Microsoft SQL server 2005* para optimizar consultas *Skyline*, pero se consideraron consultas que tuvieran solamente una relación involucrada o un solo *Join*. Por otro lado en [32, 37] se muestra una implementación a nivel de capa lógica y no directamente en el manejador, de un mecanismo que soporta las consultas *Skyline* definidas con varios *Joins*. Si se observa, no existe una integración acoplada a la capa de base de datos que realmente procese el *Skyline* de la manera conveniente.

En base a lo anterior se define la problemática de este trabajo: ¿Es realmente beneficioso, en una concepción amplia del problema, integrar la optimización de consultas *Skyline* dentro de un Sistema manejador de Bases de Datos? Más aún, ¿El modelo algebraico y de costo propuesto en trabajos anteriores para la optimización del *Skyline*, se ajusta de manera adecuada a lo que es necesario?.

Con este planteamiento se motiva este trabajo de grado donde se propone una extensión del manejador *PostgreSQL* para que permita realizar la optimización de consultas *Skyline*. El objetivo general contemplado durante el desarrollo de este proyecto fue extender el manejador relacional de base de datos *PostgreSQL* para que admitiera mecanismos de optimización de consultas basadas en preferencia. Específicamente, los objetivos que se manejaron fueron: extender el algoritmo de optimización de *PostgreSQL* para que incluya la optimización de consultas *Skyline*, estudiar la adaptación de una implementación real del modelo de costo y algebraico propuestos en trabajos anteriores e integrarlos de manera adecuada y, por último, implementar el modelo de costo para que el optimizador pudiera realizar su trabajo de estimación del costo de los planes de una manera eficiente.

Durante el desarrollo de este proyecto se trabajó adicionalmente en la propuesta de un algoritmo evolutivo para la optimización de consultas *Skyline*, desarrollado previamente en un prototipo que se decidió implementar en *PostgreSQL* posteriormente. Esto añadió dos objetivos: observar el comportamiento del algortimo en la implementación en un manejador relacional y determinar el punto en el que éste comienza a comportarse

mejor que el algortimo clásico de optimización de *PostgreSQL*.

El trabajo consta de seis capítulos. El capítulo 2 comprende el marco teórico necesario para comprender el problema de optimización de consultas *Skyline*. El capítulo 3 contiene la explicación detallada del proceso de optimización realizado en *PostgreSQL*. El capítulo 4 comprende el marco metodológico de este proyecto donde se describen los procedimientos de análisis, diseño e implementación llevados a cabo. En el capítulo 5 se presentan los experimentos realizados y los resultados obtenidos para cada uno. Por último, en el capítulo 6, se presentan las conclusiones del trabajo elaborado, las recomendaciones y propuestas para trabajos futuros a realizar.

Capítulo 2

Marco Teórico

Un Sistema Manejador de Base de Datos (SMBD) es un componente de software que sirve como interfaz entre un conjunto de datos que se tienen almacenados y las aplicaciones que quieren acceder a ellos [45]. Un SMBD convencional cuenta con distintos módulos encargados de una tarea específica dentro de la gestión de los datos, estos son: manejador de usuarios y seguridad, manejador de transacciones, manejador de recuperación, manejador de almacenamiento y manejador de consultas [8].

Desde el momento en que un usuario realiza una consulta a un SMBD, ésta se procesa por el manejador de consultas a través de varias etapas para poder retornar su resultado. En primer lugar, una consulta es analizada para verificar que está escrita correctamente y que existen las tablas y atributos a los que hace referencia. Luego, se escoge una "buena" estrategia para evaluar la consulta. Esta estrategia, llamada "plan de evaluación de la consulta", es un procedimiento definido paso a paso cuya aplicación tiene como resultado la respuesta de la consulta.

Toda consulta puede tener varias estrategias o planes de evaluación y cada estrategia puede diferir respecto a las otras en cuanto al tiempo que toma en realizarse. Unido a esto, es deseable que la respuesta de una consulta se obtenga de forma rápida. Por ello surge la necesidad de la optimización de consultas. Optimizar una consulta es estudiar ese conjunto de planes o estrategias que existen para obtener su respuesta y hallar la que permita obtener la respuesta a la consulta en menor tiempo.

Este trabajo de grado consiste precisamente en implementar un proceso de optimización para consultas *Skyline*, que son una solución aceptada al paradigma de consultas basadas en preferencias [42].

Se presenta este capítulo con el objetivo de sentar las bases teóricas necesarias para comprender el proceso de optimización de consultas, más aún, la optimización de consultas *Skyline*. Se definirá el problema de optimización de consultas orientado a las consultas *Skyline*. Adicionalmente, se expondrán conceptos referentes al paradigma de consultas basadas en preferencia, que permiten establecer un marco conceptual acerca de lo que significa optimizar consultas *Skyline*.

2.1. Optimización de consultas

La optimización es la etapa del procesamiento de consultas cuyo objetivo es hallar un "buen" plan para obtener la respuesta a la consulta. Como se mencionó anteriormente y se detallará más adelante, para cada consulta existen varios planes o estrategias de evaluación con costos de ejecución distintos y se quiere reducir en lo posible el tiempo en que el usuario obtiene la respuesta a una consulta. De esta forma, el problema de optimización de consultas consiste en hallar un buen plan de evaluación para una consulta. Por los momentos, se entenderá un plan de la siguiente manera:

Definición 1 *Un plan de evaluación para una consulta es un procedimiento definido paso a paso cuya apli-*

cación tiene como resultado la respuesta de dicha consulta. La aplicación de este procedimiento supone que en cada paso se procesan datos existentes en la base de datos para ir construyendo la respuesta a la consulta. Un "buen" plan es aquel que permite obtener de manera eficiente la respuesta a una consulta.

La eficiencia en la obtención del resultado de una consulta puede ser medida en base a la cantidad de datos que se procesan en cada etapa. Mientras menos datos haya que manejar, menos tiempo se invertirá en la obtención del resultado.

Esta sección se enfocará en presentar el problema de optimización de consultas y su importancia. Se refinará el concepto de plan y "buen" plan y se profundizará en cómo se realiza este proceso en un SMBD.

2.1.1. Álgebra Relacional y su relación con el problema de optimización

En el modelo relacional, la respuesta de toda consulta viene dada en forma de relación. Ésta, está compuesta por aquellos datos, organizados en tuplas, que satisfagan las restricciones expresadas en dicha consulta [45]. Naturalmente, el concepto central de un Sistema Manejador de Base de Datos Relacional (SMBDR) es la relación.

Definición 2 *Una relación es un conjunto de tuplas similares. Dos tuplas son similares si tienen los mismos atributos. Una tupla es la unidad mínima de información compuesta por un conjunto de atributos y sus valores respectivos. A nivel conceptual, en el modelo Entidad Relación, una tupla es una instancia de una entidad.*

En el modelo relacional, existen dos lenguajes formales para representar una consulta: cálculo relacional y álgebra relacional. El cálculo relacional se enfoca en la representación de consultas como una expresión lógica y precisa [45]. Se enfoca en definir la consulta en función de la respuesta que se quiere obtener, expresando lo que se quiere, sin que importe la manera cómo se obtiene la respuesta. De este lenguaje de representación se deriva el *Standard Query Language* (SQL). SQL es un lenguaje declarativo [15] que se usa generalmente en los SMBD relacionales para realizar consultas sobre una base de datos. Este lenguaje permite definir de manera sencilla las consultas en base a lo que se quiere y el manejador es el que se encargará de decidir cómo obtiene la respuesta.

Por otro lado, el álgebra relacional es un lenguaje de representación que, a diferencia del cálculo relacional, se enfoca en la definición de consultas de una manera procedural[1], como una composición de expresiones que al aplicarse construyen el resultado [45]. El álgebra relacional está definida en base a un conjunto de *operadores relacionales o lógicos*: selección (σ) , proyección(π), producto cartesiano ($\times$), unión ($\cup$) y diferencia(-) [45], donde cada uno realiza un procesamiento distinto sobre el o los operandos.

Definición 3 *Un operador lógico o relacional es una representación de una operación relacional. Indica qué tipo de procesamiento se le está realizando a los datos, pero no indica de qué manera se realiza. Formalmente puede ser visto como una función unaria $f : R \to R$ o binaria $f : R \times R \to R$, donde R es el conjunto de relaciones existentes en una base de datos.*

[1]Se especifica un procedimiento de ejecución paso a paso para obtener la respuesta de la consulta.

La Selección es un operador unario que permite definir una condición sobre uno o varios atributos de una relación y su aplicación retorna una nueva relación con las tuplas que satisfacen la condición proporcionada. La Proyección es también un operador unario que extrae las columnas correspondientes a un conjunto de atributos especificados en la condición de proyección y construye una nueva relación en base a esas columnas.

El Producto Cartesiano es un operador binario que construye una nueva relación formada por tuplas resultantes de aparear cada tupla de una relación entrante con todas las tuplas de la otra. La Unión y Diferencia son los operadores usuales de conjuntos.

El *Join* es un operador lógico que se define en base a un Producto Cartesiano y una Selección de la siguiente manera: $A \bowtie B = \sigma_c(A \times B)$, donde c es una condición que involucra a un atributo de A y a un atributo de B. El *Join* es uno de los operadores más usados en los manejadores relacionales.

Una expresión del álgebra relacional es una composición de estos operadores de manera conveniente para que el resultado de esa expresión sea la respuesta a la consulta que se tiene. Formalmente se define una expresión relacional de la siguiente manera [45]:

Definición 4 *Una expresión relacional representa a una consulta y se define recursivamente como: una relación, un operador unario aplicado a una expresión relacional, o un operador binario aplicado a dos expresiones relacionales.*

Adicionalmente el resultado de aplicar una expresión relacional es siempre una relación.

Nótese que se puede representar una expresión relacional como un árbol de operadores por su caracter recursivo. Para ilustrar un poco todo lo dicho anteriormente, supóngase el siguiente ejemplo:

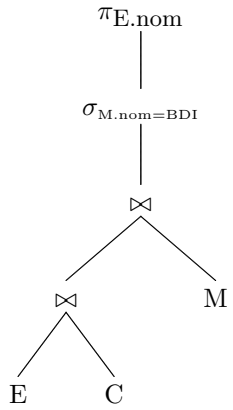


Figura 1: Representación en estructura de árbol del ejemplo 1

Ejemplo 1 *Sean E, C y M las relaciones estudiante, cursa_materia y materias, respectivamente. Los atributos de E son: nombre y carnet, mientras que los de M son: código y nombre, y los de C: carnet y código. Se desea conocer los nombres de los estudiantes que cursan la materia con nombre "BDI".*

Para esta consulta en lenguaje natural, se puede construir la siguiente expresión algebraica $\pi_{\text{E.nom}}(\ \sigma_{\text{M.nom=BDI}}\ (\text{E} \bowtie \text{C} \bowtie \text{M}))$ que realiza un *natural Join* [2] de las tres relaciones, luego aplica una selección y una proyección con las condiciones adecuadas. Observe en la Figura 1 la representación de la expresión como un árbol de operadores.

En el álgebra relacional se encuentran definidas reglas que permiten decidir si dos expresiones son equivalentes o, más aun, permiten construir nuevas expresiones partiendo de una expresión inicial [45]. Es así como a la expresión mostrada en la Figura 1, se pueden aplicar ciertas reglas algebraicas[3] para transformarla en lo que se observa en la Figura 2. Cuando se aplica una regla algebraica se obtiene una expresión algebraica diferente y equivalente.

Definición 5 *Dos expresiones son equivalentes si la relación resultante de aplicar ambas expresiones es igual.*

En consecuencia, una consulta puede ser vista como una interrogante cuya respuesta puede ser obtenida de distintas maneras, tal vez algunas mejores que otras. Así se introduce al problema de optimización. Para mayor detalle acerca de las reglas algebraicas más comunes observar el Apéndice D.

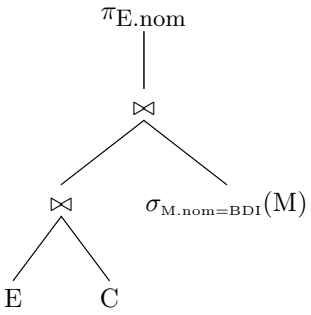


Figura 2: Representación algebraica y en estructura de árbol del ejemplo aplicando una regla algebraica de transformación a la Figura 1

2.1.2. El problema de optimización

Como se dijo anteriormente, la optimización de consultas consiste en la escogencia de una estrategia o plan que permita obtener la respuesta de una consulta dada en el menor tiempo posible. Haciendo uso de la abstracción y de la definición anterior de plan, se puede decir que cada estrategia o plan se corresponde a una expresión algebraica [45]. De esta manera, para ejecutar una consulta el manejador escoge una expresión algebraica, dentro del espacio de expresiones algebraicas equivalentes para dicha consulta.

El objetivo de la optimización es reducir el conjunto de tuplas retornado luego de aplicar cada operador para construir la respuesta, por lo tanto el problema se basa en encontrar un orden de estos operadores que reduzca el procesamiento de datos y en consecuencia el tiempo consumido en obtener la respuesta.

[2] *Join* cuya condición es igualar los atributos de igual nombre en ambas relaciones
[3] Se aplica el bajar la selección $\sigma_c(R \bowtie S) = R \bowtie \sigma_c(S)$

En el contexto físico [4], cada operador relacional puede estar implementado de varias maneras en un SMBD. A cada implementación distinta de un operador relacional se le llama operador físico.

Definición 6 *Un operador físico es una implementación de un operador lógico en un SMBD. Su función es obtener la o las relaciones de entrada (leyendo desde el disco las tablas correspondientes o recibiendo tuplas por pipeline), procesarlas de la manera correspondiente y construir una nueva relación con los datos procesados que se corresponda con el resultado del operador relacional que implementa.*

Se observa entonces que el manejador no debe escoger solamente una expresión algebraica sino la implementación con que deben evaluarse los operadores de esa expresión. En este marco, dado que una expresión algebraica puede ser expresada como un árbol y cada operador relacional se aplica con una implementación física, se proporciona una definición más completa de plan de evaluación:

Definición 7 *Un plan es un árbol de operadores físicos cuya ejecución resulta en una relación. Ésta es la respuesta a la consulta para la cual se definió dicho plan. Un "buen" plan es un árbol cuyo costo de evaluación no es alto o al menos no es el peor.*

Obsérvese que para una consulta dada, la cantidad de planes existentes puede ser alta, dependiendo de su complejidad[5]. Nótese también que cuando un manejador construye un plan de evaluación para una consulta, no hay que decidir solamente sobre el orden de los operadores lógicos sino cuál operador físico va a ser aplicado en representación de cada operador lógico.

Dada la cantidad de planes distintos que pueden existir para evaluar una consulta, resulta beneficioso escoger alguno que aminore los costos de obtener la respuesta. El costo de evaluación de una consulta o plan es la suma total del costo de ejecutar cada operador del plan y es medido, generalmente, en unidades de tiempo o accesos a disco. En este trabajo se definirá costo de evaluación de un plan y costo de evaluación de una consulta de la siguiente manera:

Definición 8 *Dada una consulta y p un plan para evaluar la consulta, sea o_i el operador i del plan, el costo de evaluación C_p se define como:*

$$C_p = \sum_i C_{o_i} \tag{1}$$

Donde C_{o_i} es una función que calcula el costo de ejecutar el operador o_i. El costo de ejecutar un operador puede ser medido en función del tiempo que transcurre entre la obtención de los datos de entrada por parte del operador (leyendo del disco duro o por pipeline) y terminar de procesar los datos. También puede ser expresado como el número de operaciones entrada-salida que el operador realiza.

*Así, El **costo de evaluación de una consulta** es el costo de ejecutar el plan que se escogió para obtener su respuesta.*

[4] En la implementación del manejador de base de datos

[5] Consultas complejas son consideradas aquellas que suponen mucho procesamiento, por ejemplo, consultas que involucran muchas tablas

Por supuesto, el costo de obtener la respuesta de una consulta va a depender de la cantidad de datos que el evaluador tenga que procesar en la ejecución de cada operador. A mayor volumen de datos que procesar, mayor número de operaciones entrada-salida, mayor número de operaciones en memoria principal y por ende más tiempo de procesamiento total. Aunque el costo puede ser medido en distintas unidades, lo que se quiere como consecuencia de reducir el costo de evaluación de una consulta es minimizar el tiempo de obtención de la respuesta.

El problema de optimización de consultas es un problema combinatorio [10] que se define de la siguiente manera:

Definición 9 *Dada una consulta C y el conjunto de planes equivalentes o espacio de búsqueda*[6] *P para obtener su respuesta, optimizar C es hallar un plan de evaluación $p \in P$ que aminore los costo de evaluación dentro de lo posible.*

Los tipos de optimización más comunmente usados son optimización heurística y optimización basada en costo. La **optimización heurística** consiste en el uso de heurísticas que buscan reducir el costo de evaluar de una consulta. Las heurísticas se basan en la aplicación de reglas algebraicas para transformar un plan en uno equivalente pero de menor costo estimado.

Según [25], la heurística para la optimización algebraica consiste principalmente en operaciones que reduzcan el tamaño de los resultados intermedios aplicando selecciones (σ) y proyecciones (π) antes que los *Join*($\bowtie$) u otras operaciones binarias, para disminuir el número de tuplas y el número de sus atributos que se tomarían en cuenta en la ejecución, esto se hace desplazando estos operadores lo más hacia abajo que se pueda en el árbol[7].

Además, aplicar las operaciones de selección y proyección que mejor restrinjan el número y tamaño de tuplas antes que otras operaciones más complejas. Se realiza cambiando los nodos hoja de lugar en el árbol. Lo que se quiere es adelantar la reducción de las relaciones que van a ser la entrada de otros operadores. Si se reduce el número de tuplas que un operador tiene que procesar, entonces se reducirá el tiempo que tarda en ejecutarse ese operador.

La **optimización basada en costo** se enfoca en el uso de fórmulas para estimar cuánto va a costar la evaluación de un operador físico. Esto se realiza utilizando estadísticas que se encuentran almacenadas en el manejador: el número de tuplas de cada relación R, el número de páginas de cada relación e información acerca de los índices asociados a una tabla[8]. Estas estadísticas ayudan a estimar el costo de ejecutar un operador físico mediante la fórmula correspondiente. La mayoría de las veces se supone uniformidad e independencia de los datos que se encuentran almacenados en SMBD cuando se realizan estimaciones del costo de un plan.

Definición 10 *Dada una consulta y p un plan para evaluar la consulta, sea o_i el operador i del plan, el costo*

[6] En los problemas combinatorios, se le llama espacio de búsqueda al conjunto de soluciones equivalentes existentes para ese problema

[7] El "*push down*" de los operadores es ejecutar un plan donde los operadores que reducen el tamaño de las relaciones se ejecuten primero y así se reduzca la cantidad de datos que tienen que procesar los siguientes operadores

[8] Una tabla es la representación física en disco duro de una relación

estimado $\hat{C}_p$ *se define como:*

$$\hat{C}_p = \sum_i \hat{C}_{o_i} \tag{2}$$

Donde $\hat{C}_{o_i}$ *es una función de costo estimado para el operador* o_i*.*

Al conjunto de fórmulas usadas para estimar el costo de los operadores físicos implementados en un manejador se le llama modelo de costo.

Definición 11 *El modelo de costo es el conjunto de fórmulas que permiten estimar el costo de los operadores físicos implementados en un manejador.*

En el Apéndice D se encuentra el modelo de costo propuesto en [45] para el operador *Join* que es considerado de gran relevancia en este trabajo.

Este tipo de optimización calcula el costo estimado de evaluación para un conjunto de planes y escoge para evaluar el que menor costo estimado tenga.

2.2. El optimizador de consultas

La optimización de las consultas se lleva a cabo en el módulo optimizador de un SMBD. El objetivo del optimizador es identificar un plan eficiente para la ejecución de una consulta dada.

El optimizador genera un conjunto de planes equivalentes y escoge para ejecutar el que menor costo estimado tenga [45]. El optimizador de consultas dentro de un manejador se encarga de producir un plan de evaluación para una consulta [25].

Cuando una consulta es analizada por el módulo *parser* [9] del manejador, se construye una estructura de datos (generalmente un árbol) que contiene la información acerca de cuáles operadores relacionales se encuentran asociados a la consulta, por supuesto con toda la información correspondiente a las condiciones asociadas a esos operadores y las tablas que se encuentran involucradas en la consulta. Este árbol producido por la primera etapa de procesamiento de consulta es llamado árbol canónico y es la estructura de entrada para que el optimizador realice su trabajo.

2.2.1. Arquitectura

La arquitectura de un optimizador puede variar según el manejador. A continuación se presenta la estructura generalizada de un optimizador de consultas y su interacción con los demás componentes del procesador de consultas.

- **El planificador:** Es el módulo principal del optimizador. Estudia los planes posibles para ejecutar una consulta dada y selecciona el de menor costo. Implementa una estrategia de búsqueda, que examina el espacio de planes de evaluación. Los planes de evaluación generados son comparados en base a sus costos estimados, que se derivan de otros dos módulos del planificador: el modelo de costo y el catálogo.

[9] Módulo que se encarga de revisar la correctitud sintáctica, léxica y semántica de una consulta

- **Modelo de costo:** Este módulo contiene las fórmulas que calculan el costo asociado a un plan de evaluación. Por cada acción del plan de evaluación, existe una fórmula para calcular su costo. La mayoría de estas fórmulas son estimaciones o aproximaciones.
- **Catálogo:** Este módulo almacena las estadísticas para cada relación. Mantiene la información que va a ser usada por el modelo de costo para realizar sus estimaciones.

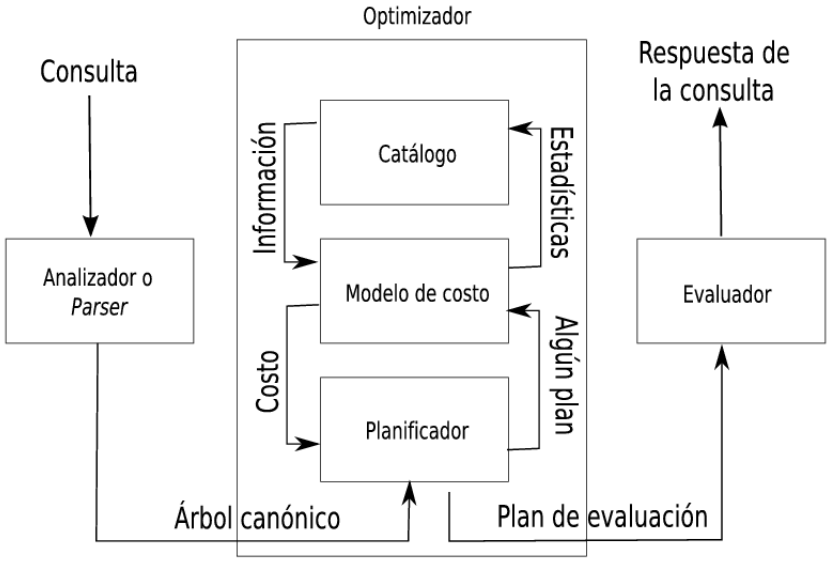


Figura 3: Arquitectura usual de un optimizador

Obsérvese en la Figura 3 que los tres módulos interactúan de manera adecuada para generar un conjunto de planes con costo estimados. Estos planes son estudiados y construídos por el planificador usando información del modelo de costo y del catálogo. Como se mostrará a continuación, el proceso de búsqueda puede variar según como se haya implementado el planificador pero en general son procedimientos que buscan resolver problemas de tipo combinatorio.

2.2.2. Búsqueda de planes

El optimizador de consultas debe contar con un mecanismo de exploración del espacio de búsqueda que sea capaz de construir planes que representen una buena opción para obtener la respuesta de una consulta. El criterio de "buena opción" puede ser tan subjetivo como se quiera, pero en general hay ciertos criterios que predominan al implementar un módulo de optimización de un manejador.

En primer lugar, el tiempo que se tarda el optimizador en hallar un plan para ejecutar no debe ser mayor que ejecutar el plan canónico originado desde el *parser*. Por esta razón el proceso de búsqueda debe implementarse de manera eficiente. Otra práctica muy usada es no explorar el espacio de búsqueda completo, sino un subconjunto de éste para disminuir su tamaño. Los algoritmos utilizados para construir los planes

pueden variar en paradigma y en forma, pero básicamente lo que llevan a cabo es la elaboración de un conjunto de planes equivalentes con un costo estimado asociado y es retornado el plan que menor costo tenga.

Aunque no necesariamente es así, generalmente la búsqueda o construcción de planes se reduce a estudiar y encontrar el orden en el que se ejecutan los *Joins* de una consulta. Esto es porque el *Join* puede ser bastante costoso al ejecutarse dependiendo del orden en que se realice y el operador físico que se utilice para su aplicación. El costo de la evaluación del resto de los operadores físicos pudiera ser despreciable en consideración a un mal plan para ejecutar los *Joins*.

Una de las técnicas para realizar la búsqueda de planes es el uso de la programación dinámica. Los algoritmos de programación dinámica construyen todos los planes no equivalentes [10] dentro de un espacio de búsqueda definido. La búsqueda que realiza es semi-exhaustiva porque explora en profundidad solamente aquellos planes que considera mejores. Se basan en el principio de optimalidad, donde se supone que una buena solución para un subproblema permite construir paso a paso una buena solución para el problema. En los manejadores convencionales, el planificador cuenta con un algoritmo de este tipo para construir el orden en el que los *Joins* deben realizarse. El uso de la programación dinámica se popularizó en el área de optimización de consultas con el surgimiento de System-R [6].

Definición 12 *Un árbol lineal es aquél donde al menos uno de los hijos de cada nodo no hoja es una relación base. Una relación base es aquélla que se lee directamente desde el disco duro y no ha sido procesada por ningún operador.*

El tiempo de búsqueda en los algoritmos basados en programación dinámica puede verse realmente afectado cuando el espacio de búsqueda es muy grande. Por esta razón, cuando se tiene una consulta que realiza *Joins* entre muchas tablas, el número de planes posibles se hace demasiado grande como para realizar una búsqueda semi-exhaustiva. Para solucionar este incoveniente, se ha recurrido al uso de algoritmos que realizan una búsqueda semi-aleatoria dentro del espacio de posibles planes [10, 35], posiblemente mediante el uso heurísticas que permitan acercarse más rápido a un "buen" plan. Entre las soluciones propuestas se encuentran el uso de algoritmos genéticos, algoritmos evolutivos, *tabu search* y otros que se adaptan a la resolución de problemas combinatorios [9, 34].

2.3. Optimización de consultas *Skyline*

Las consultas basadas en preferencia tienen como resultado un conjunto de tuplas de una relación que pueden cumplir en mayor o menor grado con los deseos del usuario. En todo caso el objetivo de las consultas basadas en preferencia es generar un conjunto de tuplas "interesantes" para el usuario, dadas las condiciones que él desearía que esas tuplas cumplieran simultáneamente. Estos deseos del usuario serán nombrados formalmente **restricciones flexibles**.

Como se quiere que se cumplan simultáneamentre una serie de deseos, es probable que estos no puedan

[10] La equivalencia entre planes depende del manejador. Dos planes son equivalentes en PostgreSQL si el conjunto de tablas en árbol es el mismo.

ser satisfechos de manera óptima. Entre varios deseos simultáneos, se tiene que estar dispuesto a ceder en alguna condición para que se pueda cumplir el conjunto completo de condiciones de una manera optimal, es decir, generar opciones que traten de cumplir de la mejor manera posible todas las preferencias del usuario. Por ello la noción de restricciones flexibles.

Existen diversos paradigmas acerca de cómo deben ser implementadas las consultas basadas en preferencia. El paradigma basado en *score* consiste en asignar una puntuación a las tuplas de acuerdo a una función que indica qué tanto le interesa esa tupla al usuario. Para esto se ha propuesto una solución llamada *Top-K* [17] que devuelve las k tuplas que mejor *score* tienen.

Por otro lado existe el paradigma basado en orden, el estudiado en este trabajo, que consiste en establecer un orden parcial sobre las tuplas y escoger las que pertenezcan al conjunto maximal [13] [11]. Para este paradigma surge también un nuevo operador relacional llamado *Skyline*, que es el que le da nombre a las consultas *Skyline*.

Las consultas *Skyline* suponen procesamientos complejos y costosos sobre los datos, ya que calcular el *Skyline*, se corresponde con el problema del vector máximo [11, 12, 28].

La mayoría de los SMBD que soportan este tipo de consultas, realizan una traducción a una consulta anidada equivalente en lenguaje SQL básico que es engorrosa sintácticamente y costosa de ejecutar, por lo que el trabajo del optimizador sigue siendo encontrar un buen plan para una consulta compleja de SQL básico.

Se ha reconocido que la óptimización y evaluación especializada del *Skyline*, resulta beneficiosa al ser un operador que puede reducir significativamente el tamaño de la relación que está procesando[12] [16, 19, 20, 21, 32]. Conviene entonces, que un operador *Skyline* se encuentre al inicio del plan de ejecución, para que reduzca el tamaño de la respuesta desde el principio, se ahorren operaciones entrada/salida; y por ende, se reduzca el tiempo de respuesta de una consulta.

Este argumento unido al problema de optimización planteado anteriormente, permite precisar que es beneficioso incluir en el proceso de optimización el *Skyline*. En pocas palabras, integrar al manejador de base de datos un operador relacional *Skyline* y sus distintas implementaciones como operador físico, se puede traducir en la disminución de costos de evaluación de una Consulta *Skyline*. Aun más, vale la pena considerar su optimización especializada[13] durante el proceso de creación del plan de evaluación. Se tiene entonces una extensión al problema de optimización.

Existen varios estudios acerca de la optimización de consultas *Skyline* que coinciden en la necesidad de adelantar la ejecución del *Skyline* para reducir los costos de evaluación de este tipo de consultas. Los primeros trabajos estudiaron y propusieron los modelos teóricos alrededor del problema de optimización de consultas *Skyline*, estableciendo una definición formal del operador *Skyline* y sus propiedades.

Luego del año 2002, J. Chomicki propone el operador *winnow*[14] [18, 19], estudia y establece sus propie-

[11] También llamado conjunto Pareto-optimal

[12] Un operador *Skyline* puede, dependiendo de las dimensiones que posea, reducir el número de tuplas que son retornados en contraste con la relación de entrada

[13] Un proceso de optimización que incluya al operador *Skyline* dentro de las posibilidades al generar un plan.

[14] Una generalización del *skyline*. Su definición es un operador que devuelve las tuplas que mejor satisfacen la preferencia del usuario

dades algebraicas generales que permiten la transformación de planes que involucren este operador. Un año después, Hafenrichter y Kießling particularizan la definición del operador *winnow* a las preferencias basadas en orden (*Skyline*) [32] y describen las propiedades algebraicas para este tipo de operador: conmutatividad y distributividad sujeta a ciertas condiciones. Estos dos trabajos permitieron sentar las bases en cuanto a cómo establecer la equivalencia entre planes que contienen el operador *Skyline*.

Luego en el 2006, fueron propuestas en [16] un conjunto de fórmulas que permiten estimar el costo de evaluación del operador *Skyline* dados el número de tuplas que va a procesar y el número de dimensiones definidas. Estas fórmulas de costo se basaron en una recursión propuesta anteriormente [27] para calcular el tamaño estimado de aplicar un *skyline* de d dimensiones sobre una relación de n tuplas. En el 2007, nuevamente J. Chomicki propone un conjunto de técnicas para la optimización semántica de consultas *Skyline* basada en restricciones definidas en la base de datos [20]. Aunque este último trabajo resulta bastante interesante, no tiene particular relación con la implementación aquí propuesta por ser un tipo de optimización dependiente del diseño lógico del esquema relacional.

Después de definir formalmente el operador *Skyline*, se detallará en las reglas algebraicas que involucran al *Skyline* y al $Join$ en [32] y se mostrará el modelo de costo propuesto para dos implementaciones u operadores físicos del *Skyline* [16]. El detalle de estos dos últimos aspectos conforman parte del fundamento del trabajo aquí propuesto y permiten comprender los algoritmos implementados para la optimización de consultas *Skyline*.

2.3.1. Definición formal del operador relacional *Skyline*

El operador *Skyline* fue propuesto como extensión de la sintaxis de SQL para manejar consultas que tratan de responder a definiciones de valores deseables, mas no exactos de los datos [13]. Por ejemplo una consulta en lenguaje natural a la cual se busca responder con este operador es: "Quiero hoteles que sean lo menos costosos posible, pero que se encuentren cercanos al mar". Nótese que no existen valores determinados o exactos para lo que se quiere, simplemente existen unas condiciones que determinan lo que se desea de la respuesta.

Obsérvese la sintaxis del operador *Skyline* en una consulta cualquiera:

```
select *
from tabla_1, tabla_2, ... tabla_n
[where ... ]
skyline of att1 [max|min|diff] [, att2 [max|min|diff], ...] ...
```

Nótese que lo que hace la cláusula *SKYLINE OF* es definir sobre un conjunto de atributos pertenecientes a las tablas del $from$ un comportamiento deseable para el valor de cada atributo. El `select *` indica que se quieren proyectar todos los atributos de la relación resultado, el `from` indica qué relaciones están involucradas en la consulta y `where` permite especificar condiciones de *Joins* y selección. Por ejemplo, si se especifica:

```
... skyline of $a_1$ max , $a_2$ min, $a_3$ diff
```

Lo que se quiere es maximizar el valor de a_1, minimizar el valor de a_2 y agrupar los resultados por cada valor distinto de a_3. De esta manera se observa que existen tres directivas (max, min y $diff$) para definir lo que se desea del conjunto de tuplas resultado de la consulta. La claúsula max maximiza el valor del atributo, min lo minimiza y $diff$ agrupa por valores distintos del atributo y calcula el *skyline* con el resto de las dimensiones en cada grupo de valores distintos.

A cada atributo involucrado en la cláusula *SKYLINE OF* se le denominará dimensión del skyline. De esta manera, en el ejemplo anterior, tenemos tres dimensiones. Se denominará criterio del *Skyline* al par comprendido por una dimensión y una directiva.

Definición 13 *Formalmente, el operador Skyline es una extensión de SQL que comprende una lista de atributos y sus directivas correspondientes, cuya respuesta es un conjunto de tuplas interesantes. Una tupla es interesante si forma parte de las opciones que mejor se ajustan a las preferencias del usuario. El conjunto respuesta de aplicar un operador Skyline es llamado conjunto Skyline y comprende todas las tuplas no dominadas de una relación.*

Se introduce el concepto de tuplas interesantes y relación de dominancia entre tuplas, para poder comprender qué datos pertenecen al *skyline*:

Definición 14 *Se definen como tuplas interesantes a aquéllas que no son dominadas por ninguna otra tupla. Una tupla domina a otra cuando es mejor o igual en todas las dimensiones definidas y es estrictamente mejor en al menos una dimensión.*

De manera formal, sean t y p dos tuplas compuestas por los atributos a_i con $i = 1, 2, ..., n$ y sea a_1 **min,** *a_2* **min,...,** *a_k* **min,** *a_{k+1}* **max,** *a_{k+2}* **max, ...,** *a_q* **max,** *a_{q+1}* **diff,** *a_{q+2}* **diff, ...,** *a_n* **diff** *un conjunto de n dimensiones definidas por un operador Skyline. Sea $t(a_i)$ y $p(a_i)$ el valor correspondiente del atributo a_i de la tupla t o p, respectivamente. Entonces t domina a p si se cumplen las siguientes condiciones simultáneamente:*

- $t(a_i) \preceq p(a_i)$ *con i=1,2,...,k*
- $p(a_i) \preceq t(a_i)$ *con i=k+1,...,q*
- $t(a_i) = p(a_i)$ *con i=q+1,...,n*
- *Al menos para una dimensión se cumple que $t_i \prec p_i$ con $i \in [1, k]$ o $t_i \succ p_i$ con $i \in [k + 1, q]$.*

Si no se puede determinar dominancia entre dos tuplas, se dice que esas dos tuplas son incomparables.

Existen varios algoritmos propuestos para la implementación del *skyline* en un SMBD. Los de interés en este trabajo son el *Skyline Block Nested Loop* (BNL) y el *Sort Filter Skyline* (SFS) [13, 43, 28, 48, 7], porque se encuentran integrados al manejador *PostgreSQL*, como proyecto de grado de Brando y González [14] y porque son los que poseen un marco conceptual robusto en cuanto a su modelo de costo.

El algoritmo BNL mantiene una ventana de tuplas incomparables en memoria. En cada iteración se leen tuplas para ser procesadas. Una tupla p es comparada con cada una de las tuplas que se encuentran en la ventana. Como resultado de esta comparación puede suceder que:

- p es eliminada si se consigue que es dominada por alguna de las tuplas de la ventana
- p es agregada a la ventana si domina algunas tuplas en la ventana y se eliminan dichas tuplas, ó si es incomparable con todas las tuplas de la ventana
- p es colocada en un archivo temporal si debe ser agregada y no cabe en la ventana.

En cada iteración, se recrea la ventana de tuplas con las que se van leyendo tanto del disco como del archivo temporal. Al final de todas las iteraciones la ventana resultante es el conjunto *Skyline* de la consulta. En el peor de los casos, el algoritmo tiene que hacer más de una iteración para poder hacer la comparación contra las tuplas que fueron almacenadas en el archivo temporal. Nótese que este algoritmo tiene una estrategia de comparación de todos contra todos lo que lo puede hacer muy costoso en evaluación.

Por otro lado, el SFS es una variación del BNL [21] que realiza un preordenamiento de la relación a ser procesada en base a las dimensiones y directivas de la cláusula *SKYLINE OF*. Con el preordenamiento se asegura que después no se producirán reemplazos o eliminaciones en la ventana. Este algoritmo reduce significativamente el número de comparaciones que se tienen que realizar, pero el costo adicional de preordenamiento puede ser alto. Por ende, si una tupla es agregada a la ventana es porque ya forma parte del conjunto *Skyline*.

Para mayor detalle sobre los algoritmos obsérvese el Apéndice F.

2.3.2. Reglas algebraicas entre el Skyline y *Join* utilizadas por el optimizador

Para el operador *Skyline* existen una serie de reglas algebraicas que permiten establecer una relación de equivalencia entre planes que contengan el operador y los operadores tradicionales de SQL. A continuación se presentan algunas de las reglas algebraicas definidas en [32] que son de interés para este trabajo y que involucran a los operadores *skyline* y *Join*. Estas reglas permiten construir planes equivalentes donde se realiza un *push down* del *Skyline* sobre el *Join*.

Se define la lista de preferencias como un conjunto de criterios definidos sobre una serie de atributos de una o más relaciones. La notación $\text{Skyline}_{[\text{p}]}(R)$ se entiende como un *Skyline* sobre la relación R con el conjunto de condiciones de preferencia P.

Sean R y S relaciones, P_r y P_s el conjunto de preferencias sobre R y S, respectivamente y C la condición $R.x = S.x$ del *Join*. La condición C_o es verdadera si todas las tuplas de R hacen *Join* con al menos una tupla de S [15], se tienen entonces las siguientes reglas algebraicas:

Nótese que cualquier expresión algebraica que representa una consulta *Skyline*, posee un operador *Skyline* en la raíz. Cuando una expresión algebraica posee más de un operador *Skyline*, entonces el *Skyline* se encuentra

[15] Esto es llamado *full Join*

$\text{Skyline}_{[P_r]}\,(\text{R} \bowtie_c \text{S}) \equiv \text{Skyline}_{[P_r]}\,(\text{R}) \bowtie_c \text{S} \qquad \textbf{si } C_0$

$\text{Skyline}_{[P_r]}(\text{R} \bowtie_c \text{S}) \equiv \text{Skyline}_{[P_r]}(\text{Skyline}_{[P_r \cup \{\text{R.x diff}\}]}(\text{R}) \bowtie_c \text{S})$

$\text{Skyline}_{[P_r \cup P_s]}(\text{R} \bowtie_c \text{S}) \equiv \text{Skyline}_{[P_r \cup P_s]}(\text{Skyline}_{[P_r \cup \{\text{R.x diff}\}]}(\text{R}) \bowtie_c \text{S})$

Tabla 1: Reglas algebraicas entre *Skyline* y *Join*

distribuido. Existen otras reglas que relacionan al *Skyline* con otros operadores relacionales tradicionales y están explicadas con mayor detalle en [32].

2.3.3. Modelo de costo para el *Skyline*

Para el operador *Skyline* se han realizado investigaciones con el objeto de definir un modelo de costo que permita estimar la cardinalidad resultante y el costo de aplicarlo. Aun cuando, a nivel físico, existen muchas implementaciones del operador lógico *Skyline* descritas por Borzsonyi, Kossman y Stocker en [13], la estimación del costo de aplicar estos algoritmos solo ha sido aproximada solamente para el BNL y el SFS. Como el operador *Skyline* es exigente en cuanto a recursos de procesamiento [16], el cálculo del costo está orientado a estimar cuántas comparaciones tiene que realizar el algoritmo antes de obtener la respuesta.

2.3.4. Estimación de la cardinalidad del *Skyline*

En un trabajo realizado por Parke Godfrey [27], se propone un modelo de estimación de la cardinalidad de la respuesta del operador *Skyline* basado en la cantidad de tuplas entrantes, las dimensiones definidas en el operador y la probabilidad de que una tupla pertenezca al conjunto de respuesta.

En ese trabajo, se plantea la siguiente fórmula recursiva para calcular el número de tuplas que resulta de aplicar un operador *Skyline* definido con d dimensiones, con directivas min o max, sobre una tabla de cardinalidad n, bajo la suposición de independencia de los valores y la no existencia de valores duplicados. Formalmente:

Definición 15 *La cardinalidad estimada del conjunto Skyline ($\hat{S}_{d,n}$) es el número de tuplas que contiene la relación respuesta al aplicar el Skyline a una relación R. Sea n el número de tuplas de R y d el número de dimensiones definidas en el Skyline, entonces $\hat{S}_{d,n}$ es calculado con la recurrencia en la fórmula 3, bajo la suposición de independencia y unicidad de los valores de los atributos.*

$$\hat{S}_{n,d} = \tfrac{1}{n}\hat{S}_{n,d-1} + \hat{S}_{n-1,d} \tag{3}$$

La recurrencia anterior está definida en base a la probabilidad de que una tupla se encuentre en el conjunto *Skyline*. El primer sumando indica la probabilidad de que una tupla que es dominada en la primera dimensión, pertenezca al conjunto *Skyline*, mientras que el segundo término calcula la cardinalidad del conjunto *Skyline* sin contar la tupla que ya se está considerando.

Existen trabajos que permiten aproximar esta recurrencia ya que no existe fórmula cerrada para calcularla. En [27] se plantea una relación entre la cardinalidad estimada del conjunto *Skyline* y los números armónicos, la cual se presenta con las fórmulas 4 y 5.

$$\hat{S}_{n,d} = \mathrm{H}_{n,d-1} \tag{4}$$

donde

$$\mathrm{H}_{n,k} = \sum_{i=1}^{n} \binom{n}{i} (-1)^{i-1} i^{-k} \tag{5}$$

Esta aproximación difiere bastante respecto a los valores reales de $\hat{S}_{n,d}$ pero su ajuste es aceptable [27]. Existe la necesidad de aproximar la recurrencia porque el tiempo de cómputo para valores grandes puede ser alto.

2.3.5. Costo de los algoritmos BNL y SFS

El algoritmo *Block Nested Loop* (BNL) en el peor de los casos realiza $O(n^2)$ comparaciones. Es por esto que el BNL tiene un costo considerable de CPU en función del número de comparaciones de tuplas que se tienen que realizar. En [16] se propuso la fórmula mostrada a continuación para calcular la cantidad de comparaciones al evaluar un *Skyline*, bajo la suposición de independencia de los datos. Sea d la cardinalidad del conjunto de los criterios del *Skyline*, entonces el número de comparaciones al calcular el *Skyline* en una relación de n es $C_{BNL}(n, d)$ en la fórmula 6.

$$C_{BNL}(n, d) \approx \sum_{j=2}^{n} \frac{\hat{S}_{j-1,d}}{j-1} \hat{S}_{j-1,d+1} \tag{6}$$

En el caso del *Sort Filter Skyline*(SFS), se realiza un preordenamiento en base a una función monotona de ordenamiento, inducida por los criterios *Skyline* definidos. Esto garantiza que las tuplas que entran a la ventana pertenecen al Skyline y no son dominadas por ninguna otra tupla que se procese posteriormente. El cálculo del costo es similar al del algoritmo BNL y se espera que el preordenamiento reduzca considerablemente el número de comparaciones. Se resta una dimensión porque se considera que al menos una dimensión ya posee un orden total estricto [16], lo que restaría realizar las comparaciones para ese caso. El número de comparaciones para SFS se plantea en la fórmula 7.

$$C_{SFS}(n, d) \approx \sum_{j=2}^{n} \frac{\hat{S}_{j-1,d-1}}{j-1} \hat{S}_{j-1,d} \tag{7}$$

Las fórmulas aquí mostradas son aproximaciones propuestas en [16] que suponen independencia y unicidad de los valores de los atributos. En ese mismo trabajo se estudian expresiones que calculan el valor real de la cardinalidad del Conjunto *Skyline*. Sin embargo, los mecanismos allí trabajados suponen la capacidad para poder realizar muestreo sobre los datos e histogramas, herramientas con las que no siempre se cuenta.

[16] Un orden total estricto puede definirse como una relacion sobre un conjunto de elementos donde se cumple $a < b \equiv a \leq b \vee a \neq b$ si a y b pertenecen al conjunto sobre el cual se define el orden

2.4. Algoritmos Evolutivos

Los algoritmos evolutivos son un subconjunto de las técnicas de computación evolutiva. Fueron desarrolladas por Lawrence Fogel [39] en Estados Unidos en los años sesenta. Están basados en la teoría de evolución *Darwiniana* de la selección natural, donde la idea principal es que sobreviven los más aptos para el ambiente donde se desenvuelven.

Los algoritmos evolutivos han representado una solución sencilla y efectiva para los problemas combinatorios [24]. Esta sección es de interés para comprender el algoritmo evolutivo propuesto en este trabajo para la optimización de consultas *skyline*.

El concepto central de este tipo de algoritmos es mantener una población de individuos, donde cada individuo es una posible solución al problema de interés. La población de individuos va cambiando por generaciones (iteraciones del algoritmo), mediante cualquiera de los mecanismos naturales conocidos: reproducción y mutación. Como se observa en la Figura 4, es un algoritmo sencillo que simula la evolución de una población a través del tiempo.

Algoritmo 2.4.1: ALGORITMO EVOLUTIVO(*poblacion*: ∅)

```
poblacion = INICIALIZARPOBLACION()
CALCULARFITNESS(poblacion)
while not CONDICIONPARADASATISFECHA()
   SELECCION(poblacion)
   VARIARPOBLACION(poblacion)
   CALCULARFITNESS(poblacion)
return BEST(poblacion)
```

Figura 4: Estructura general de un algoritmo evolutivo

En el algoritmo se distinguen varios procedimientos: cálculo del *fitness* y la inicialización, selección y variación de la población.

El mecanismo para inicializar la población generalmente es la generación aleatoria de soluciones canónicas al problema que se está trabajando, aunque existen problemas donde resulta mejor realizar una inicialización más orientada a la solución haciendo un trabajo extra para originar mejores individuos iniciales [33]. Cada individuo que conforma la población es una representación abstracta de una de las varias soluciones que pueden existir. En esta etapa de inicialización de la población lo que se hace es crear una serie de individuos, repetidos o no, construyendo esa representación para diferentes posibles soluciones iniciales.

Cada individuo cuenta además con un costo asociado o función de *fitness*. Esta función permite cuantificar qué tan buena solución es la representada por el individuo, para el problema tratado. La función de *fitness* puede variar de problema a problema, pero generalmente tiene que ver con un costo de ejecutar la solución. Por ejemplo, en el clásico problema *Travelling Salesman Problem* TSP [31], se tiene que la función de *fitness* es la suma total de kilómetros que recorre el vendedor es su *tour* por distintas ciudades y un *tour* es mejor

que el otro si el viajero recorre menos kilómetros, es decir, si el *fitness* es menor.

En el pseudo-código proporcionado en el algoritmo 2.4.1, se observa una iteración que se corresponde con el concepto evolutivo de generaciones. En cada iteración se selecciona una parte de la población que va a sobrevivir para la siguiente generación [30]. Seguidamente se completa la población con individuos generados producto de un proceso de variación y luego se calcula para los nuevos individuos su *fitness*.

Este tipo de algoritmos funcionan porque se intenta mantener a los mejores individuos en la población de modo que en cada generación el *fitness* promedio de la población aumente. Cuando la condición de parada impuesta ha sido satisfecha, entonces la solución al problema será el mejor individuo de la población de esa generación.

Formalmente se definen todos los conceptos mencionados anteriormente:

Definición 16 *Un **individuo** o cromosoma es una representación abstracta de una solución al problema tratado. A cada individuo está asociado un valor de **fitness**, que es una puntuación a la calidad de la solución que el individuo representa.*

*La **población** es la serie de individuos existentes en una generación. Una **generación** es cada iteración del algoritmo evolutivo.*

*Los **operadores de variación** son aquéllos que permiten variar la población mediante la transformación de uno o dos individuos, en uno nuevo.*

*El operador de **selección** es el que permite seleccionar la parte de la población que sobrevivirá para la siguiente generación.*

Los operadores de variación pueden ser de mutación o de reproducción. Los operadores de mutación toman un individuo y generan uno nuevo a partir de éste. Los operadores de reproducción parten de dos individuos y generan uno nuevo a partir de ellos. Son llamados operadores de variación precisamente porque son los que permiten diversificar la población y pueden ser considerados funciones que toman como argumento uno o dos individuos existentes y crean uno nuevo en base a ellos.

El operador de selección toma un porcentaje de la población de una iteración y construye una nueva población para la siguiente generación en base a eso.

Otros aspectos importantes a considerar en la implementación de algoritmos evolutivos es la parametrización del algoritmo. En general, existen parámetros de configuración que pueden afectar el desempeño del mismo: tasa de selección, que es el porcentaje de la población que va a sobrevivir; tasa de mutación y reproducción, que son la cantidad de individuos nuevos generados a partir de estos operadores respectivamente; y tamaño de la población. Todos estos parámetros son usualmente entonados porque los valores para el mejor desempeño del algoritmo varían con cada problema tratado.

La utilidad de este tipo de algoritmos viene dada por el tamaño del espacio de búsqueda. Si el espacio es tan grande que se hace costoso explorarlo completamente, es una buena opción implementar un algoritmo evolutivo para hallar una solución optimal al problema. Por su caracter semi-aleatorio es posible que la

solución hallada no sea la mejor, pero lo suficientemente buena como para resolver el problema tratado eficientemente sin haber realizado demasiado trabajo extra para conseguir la solución.

Capítulo 3

El optimizador *PostgreSQL*

PostgreSQL es un manejador objeto-relacional de base de datos. Fue escogido en la etapa de desarrollo previa de este proyecto para la implementación del operador *Top-kSkyline*[29] en el trabajo de grado de Brando y González [14], cuya extensión denominaron PeaQock. Las razones para su escogencia fueron su condición de *software* libre, por la documentación existente sobre su desarrollo y uso, y por ser una plataforma estable extensamente usada en el mundo.

Este capítulo se enfocará en el optimizador de *PostgreSQL*. Cómo está implementado, qué tipo de optimización realiza, su arquitectura y los algoritmos con los que la realiza. Para mayor información acerca de las otras etapas de procesamiento de consultas en *PostgreSQL* consultar el Apéndice G.

3.1. Arquitectura del optimizador de *PostgreSQL*

El optimizador de PostgreSQL se encuentra dividido en varios componentes que interactúan para poder producir un plan [40]. Entre estos componentes tenemos: *Path*, *Plan*, *Prep*, *Util* y *Geqo*. Cada componente es en realidad una carpeta con archivos que implementan funcionalidades específicas. Cada componente tiene las siguientes funcionalidades:

- **Componente *Path***: Toma el árbol generado por el *parser* de PostgreSQL y genera todos los posibles planes para evaluar la consulta de entrada basado en las restricciones de la cláusula `where`. Para generar los planes utiliza un algoritmo basado en programación dinámica. En este componente también se encuentra implementado el modelo de costo de PostgreSQL.

- **Componente *Plan***: Toma la salida del componente *path*, escoje el plan con menor costo, crea el árbol de evaluación equivalente para esa salida y agrega los operadores adicionales especificados en la consulta [1].

- **Componente *Prep***: Ejecuta procesamiento especial de las consultas, cuando por ejemplo, son anidadas o contienen operaciones de conjunto.

- **Componente *Util***: Contienen rutinas de soporte usadas en el optimizador.

- **Componente *Geqo***: Realiza el mismo trabajo que el componente *path*, pero el mecanismo para la búsqueda de planes es un algoritmo genético. Este componente es usado cuando el número de *joins* en la consulta es grande. Para PostgreSQL este límite es de once *joins*. Su nombre es abreviación de *Genetic Query Optimizer*.

[1] Aquellos operadores que no se originan por especificación de restricciones en la cláusula `where` (order by, group by)

3.2. Optimización de una consulta en *PostgreSQL*

Para explicar en detalla el proceso de optimización de una consulta en PostgreSQL es necesario definir los siguientes conceptos.

Definición 17 *En el proceso de optimización de PostgreSQL, un camino de acceso o Path es una implementación de un operador relacional o lógico, es decir, es un operador físico.*

Definición 18 *Camino de Evaluación, es un árbol formado por operadores físicos, éste representa una forma de evaluación de un plan lógico. La información almacenada en cada nodo del árbol es: la implementación física con la que se va a evaluar, apuntadores a los correspondiente en el plan lógico, información referente a las relaciones a las que tiene que acceder, el costo estimado de evaluación y el costo acumulado estimado.*

Definición 19 *Plan Lógico, es un árbol formado por operadores lógicos o relacionales. Además posee entre otras estructuras, una lista de caminos de evaluación asociados a éste y un conjunto de identificadores de las relaciones base que contiene el plan. Para ilustrar esta explicación, obsérvese la Figura 5*

Definición 20 *Plan de Evaluación, es similar al camino de evaluación, sin embargo, posee toda la información necesaria para que pueda ser procesado por el evaluador. Es el plan con el que se va a evaluar una consulta de entrada.*

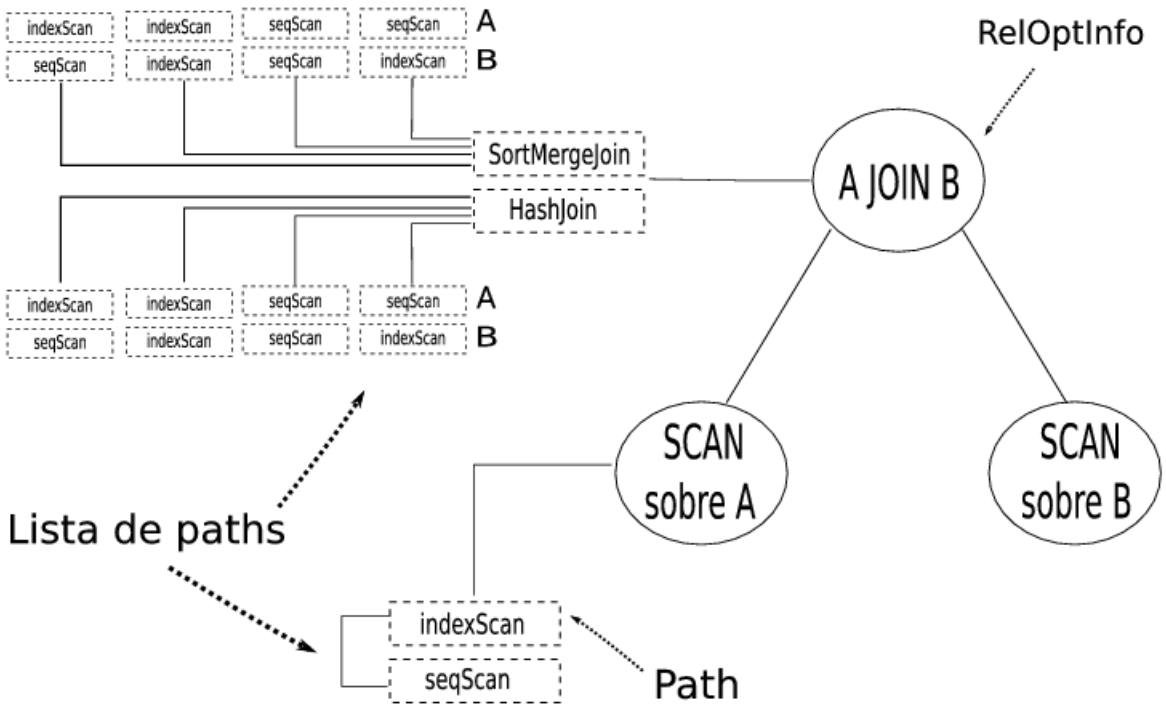


Figura 5: Estructura de un plan en *PostgreSQL*

La optimización en PostgreSQL recibe el árbol generado por el *parser* para transformarlo en un plan de evaluación. El objetivo es hallar un plan de evaluación adecuado para la consulta de entrada.

El optimizador de PostgreSQL está basado en costo, lo que quiere decir que genera varios planes lógicos para una consulta, por consiguiente realiza un esfuerzo adicional para crear los caminos de evaluación posibles de ese plan y utilizando el modelo de costo se calcula el costo de evaluación estimado para cada camino y finalmente escoge el camino de evaluación que menor costo estimado tenga para luego transformarlo en el plan de evaluación de la consulta.

El optimizador de *PostgreSQL* realiza el proceso de optimización en dos etapas. La primera etapa se basa en la construcción de planes lógicos que sólo contengan los operadores **Join** o en su defecto, Producto Cartesiano y *Scan*. En la segunda etapa se transforma el camino escogido en la primera etapa, en el plan de evaluación y se agregan los operadores adicionales que completan la consulta que no se consideraron antes.

3.2.1. Búsqueda del mejor camino de evaluación

En primera instancia se define detalladamente lo que para PostgreSQL es un plan lógico:

La estructura de un plan lógico en *PostgreSQL* está compuesta por nodos llamados `RelOptInfo`. Los `RelOptInfo` representan un operador relacional, generalmente un Join o un Scan. En estos nodos se almacenan distintos campos de información relevantes para el proceso de optimización.

Supóngase que se tiene un `RelOptInfo` r, éste contiene campos con la siguiente información:

- Si r es un *Join*, se tiene apuntadores a los dos sucesores que son los `RelOptInfo` de las relaciones de entrada para el operador.
- También se tiene almacenado el identificador de la o las relaciones involucradas en r, formando un conjunto que se denomina *relids*; si es un *Scan* se almacena solamente el *relid* de la relación involucrada; si es un *join*, se almacena la unión de los *relids* de los `RelOptInfo` sucesores.
- Se mantiene una lista con la información acerca de los diferentes caminos de evaluación considerados para r (`PathList`). Cada árbol del camino de evaluación posee como raíz el camino de acceso para la última relación considerada hasta el momento, la raíz del plan lógico. Los hijos de la raíz corresponden a los caminos de acceso para los hijos de la raíz del plan lógico y así sucesivamente. Para cada camino de acceso se estima el costo de evaluar utilizando ese operador físico.
- Se mantienen estadísticas para la optimización como el número de tuplas que se espera retornar mediante ese operador lógico.

Definición 21 *Se denomina* ***mejor camino de evaluación*** *a áquel que represente físicamente el plan lógico de la consulta y cuyo costo estimado sea el menor.*

El objetivo de esta primera etapa de la optimización es hallar un órden de evaluación de *joins* que se estime no sea tan costoso. Si una consulta posee una sóla relación base, sencillamente se omite esta etapa de procesamiento.

En PostgreSQL, existen dos algoritmos para hallar el mejor camino de evaluación. Estos dos algoritmos serán tratados con detalle más adelante en este capítulo. En todo caso, se puede generalizar en que el mecanismo de búsqueda de este camino se produce de la siguiente manera:

- Para cada tabla involucrada en la consulta se construye un plan lógico *Scan*.
- Partiendo de ese conjunto de relaciones base, se construyen los posibles planes lógico para la consulta de entrada solamente considerando los *scan* y *joins*.
- Finalmente, de ese conjunto de planes lógicos generados, se escoge el camino de evaluación que menor costo estimado posea. Este camino es la respuesta de esta primera etapa y será procesado posteriormente por el componente *plan*.

Al conjunto de operadores representado en este plan lógico se les denomina *down-level*. Los caminos de acceso (operadores físicos en el evaluador) que se encuentran implementados en PostgreSQL son: para el *Scan*: `SeqScan` y `IndexScan`, y para el *Join*: `NestedJoin`, `MergeJoin` y `HashJoin`.

3.2.2. Preparación para el evaluador

En una segunda etapa, se transforma el camino de evaluación en el plan de evaluación agregándole las estructuras adecuadas para ser pasado al evaluador y los operadores presentes en la consulta no considerados en la etapa anterior. Este procesamiento se realiza en el componente *plan* del optimizador.

Primero se realiza una construcción del plan de evaluación en base al mejor camino de evaluación. Esto se realiza obteniendo el `Path` menos costoso de la lista de *paths* y creando para cada nodo un nodo `plan` equivalente, con la información relevante para la evaluación de ese árbol. Seguidamente se agregan en la raíz los nodos `plan` correspondientes a los operadores adicionales no especificados por la cláusula `where` en la consulta. A los nodos agregados en esta etapa se les llama nodos *top-level*.

Para los nodos *top-level*, no se toma la decisión acerca de en qué lugar del árbol se van a colocar, simplemente se van colocando en la raíz del árbol en un orden predeterminado. De la misma manera se encuentra predeterminado el camino de acceso con el que cada operador se va a evaluar.

Las dos funciones del componente *plan* son entonces:

- Procesar el camino de evaluación escogido en la primera etapa, para construir un plan de evaluación equivalente listo para ser evaluado.
- Colocar el resto de los operadores en la raíz del plan de evaluación.

Una vez que el plan de evaluación se encuentra completamente construído y ya se encuentran colocados adecuadamente todos los operadores necesarios para obtener una respuesta correcta a la consulta, está listo para ser evaluado por el evaluador.

3.3. Métodos de búsqueda de plan de PostgreSQL

Como se mencionó anteriormente, PostgreSQL tiene dos métodos para hallar el camino de evaluación. En el componente *path* se encuentra implementado un algoritmo basado en programación dinámica (DYQO) que explora un subconjunto de todos los posibles planes lógicos y sus caminos de evaluación para una consulta. Por otro lado, en el componente *geqo* se encuentra implementado un algoritmo genético (GEQO) orientado a realizar una búsqueda semi-aleatoria de un buen camino de evaluación para evaluar la consulta de entrada.

PostgreSQL contempla solamente un subconjunto de los planes posibles para una consulta. Los planes que se estudian durante el proceso de búsqueda son árboles lineales.

Es así como el espacio de búsqueda se ve reducido en tamaño con el propósito de hacer la búsqueda más manejable. La cantidad de planes lógicos posibles para una consulta con n *Joins* es $\frac{n!}{2^n}\binom{2n}{n}$, mientras que considerando solamente árboles lineales el número de planes lógicos existentes es $\binom{n}{2}(n-2)!$ [22, 36].

El número de planes alternativos para evaluar una consulta aumenta exponencialmente con el número de *Joins* en esta. Por lo tanto, el tamaño del espacio puede convertirse en una cifra muy grande como para explorar todos los planes. Aunque el algoritmo DYQO explora un subconjunto de éste, la cantidad de planes que construye sigue siendo bastante grande. Es ahí cuando PostgreSQL decide usar el algoritmo GEQO en vez del algoritmo DYQO para explorar menos planes y no desperdiciar tanto tiempo en el proceso de optimización. Por defecto, se utiliza el algoritmo GEQO cuando el número de *joins* es doce.

A continuación se presentan en detalle los dos mecanismos de búsqueda utilizados por PostgreSQL para hallar un buen plan de evaluación para una consulta de entrada.

3.3.1. Algoritmo de búsqueda basado en programación dinámica

Este es el algoritmo por defecto de PostgreSQL, el cual se denominará, *Dynamic Query Optimization* (DYQO), realiza una exploración un subconjunto de todos los planes representados por árboles lineales para la consulta de entrada. El algoritmo efectúa la construcción iterativa de los planes, considerando solamente los planes inequivalentes. Para PostgreSQL, dos planes son equivalentes si el conjunto de *relids* hasta el nodo raíz de cada árbol es el mismo. Se explora el espacio de búsqueda construyendo los planes desde las hojas hasta la raíz.

Obsérvese en el algoritmo mostrado en la Figura 6 que se crean inicialmente un plan lógico para cada iésima-relación base mediante CALCULARPLANESBASE(i). Como para cada plan se crea una lista de diversos caminos de evaluación, se eliminan los más costosos en ESCOGERMEJORESCAMINOS(plan) y sólo los de menor costo permanecen en la lista.

Este proceso de creación de planes de tamaño uno [2] considera de una vez las condiciones de selección para cada relación, especificadas en la cláusula `where` de la consulta. Los planes de tamaño uno son almacenados en una lista de planes lógicos que se denominará *listasPlanes*[1].

Luego de la creación de los planes de tamaño uno para cada relación, el algoritmo construye los planes de

[2] El tamaño de un plan viene dado por el, número de relaciones consideradas en ese plan

Algoritmo 3.3.1: CONSTRUIRPLAN(R:relaciones base, l:niveles necesarios)

```
procedure CALCULARPLANESJOIN(plan, planesBase)
  result ← ∅
  for each p ∈ planesBase
    if RELIDSDISJUNTOS(p, plan) and ESFACTIBLEJOIN(p, plan)
     join ← CREARJOIN(plan, p)
     result ← result ∪ join
  return result

main
  listasPlanes ← ∅
  listasPlanes[1] ← ∅
  for each i ∈ |R|
     plan ← CALCULARPLANESBASE(i)
     ESCOGERMEJORESCAMINOS(plan)
     listasPlanes[1] ← listasPlanes[1] ∪ plan
  x ← 2
  while x ≤ l
     listasPlanes[x] ← ∅
     for each i ∈ listasPlanes[x − 1]
        planes ← CALCULARPLANESJOIN(i, listasPlanes[1])
        if planes = ∅
           planes ← CALCULARPRODUCTOCARTESIANO(i, listasPlanes[1])
        ELIMINAREQUIVALENTES(listasPlanes[x], planes)
        listasPlanes[x] ← listasPlanes[x] ∪ planes
     ESCOGERMEJORESCAMINOS(listasPlanes[x])
     x ← x + 1
  return listaPlanes[l][1]      //Al finalizar se tiene un único plan al nivel l.
```

Figura 6: Estructura del Algoritmo de optimización de PostgreSQL basado en programación dinámica: DYQO

tamaño dos partiendo de la unión de dos planes de tamaño uno. Quedando como hijos del nuevo plan lógico los planes de entrada para el operador Join o Producto Cartesiano que son los únicos que permiten enlazar dos planes. En la siguiente iteración se observarán las posibilidades de los planes de tamaño tres, partiendo de la unión de un plan de tamaño dos y uno de tamaño uno y así hasta que se complete el número de *joins* en la consulta.

Detallando un poco más en la construcción de los planes de tamaño mayor que uno, se realiza una iteración donde para cada plan i (creado en la iteración anterior) se efectúa un Join con cada uno de los planes bases $listasPlanes[1]$ al ejecutar CALCULARPLANESJOIN(i,listasPlanes[1]). Cada plan lógico cuya raíz es un join con su respectiva lista de caminos de evaluación es creado con CREARJOIN(plan,p), si este plan fue previamente creado, entonces se devuelve el apuntador de ese plan. Sin embargo, sólo se crea el plan lógico asociado al join si el *relid* de la relación correspondiente al plan base p no pertenece al conjunto de *relids* del plan lógico de entrada, *plan*, (RELIDSDISJUNTOS(p,plan)), ésto indica que hasta ese nivel en el algoritmo, el subárbol *plan* no contiene el plan base p. Otra condición que debe cumplirse es que sea factible realizar el join de *plan* con p (ESFACTIBLEJOIN(p,plan)) para evitar ProductosCartesianos. Una vez realizados todos los Joins para el subárbol *plan*, se devuelve este conjunto *planes*, el cual es almacenado para la siguiente iteración.

Posteriormente si existen planes repetidos o equivalentes en el conjunto *planes* con respecto a los planes generados en ese nivel listasPlanes[x], entonces se descartan del conjunto *planes* dejando el de menor costo en cada caso, utilizando ELIMINAREQUIVALENTES(listasPlanes[x],planes). Nótese que este procedimiento recorta el espacio de búsqueda.

En caso de no existir ningún join en el conjunto *planes* se procede a realizar Productos Cartesianos mediante CALCULARPRODUCTOCARTESIANO(i,listasPlanes[1]).

Este procedimiento se realiza sucesivamente y en el último nivel sobrevive un único plan entre todos los posibles planes generados en ese nivel, ya que son todos equivalentes por tener el mismo conjunto de relaciones y son eliminados mediante ELIMINAREQUIVALENTES(listasPlanes[x],planes). Resultando de esta eliminación un único plan, que tendrá el camino de evaluación de menor costo que será el utilizado para evaluar la consulta.

3.3.2. Algoritmo de búsqueda basado en algoritmos genéticos

El algoritmo genético implementado en PostgreSQL está basado en el algoritmo *Genitor* de Whitley [50], para resolver el problema del agente viajero (TSP) y recibe el nombre de *Genetic Query Optimization* (GEQO).

En este algoritmo se representa al individuo o cromosoma como una cadena de dígitos, donde cada dígito es el *relid* de alguna relación involucrada en la consulta de entrada. La cadena de dgítos es el orden en el que se realizan los *joins* de la consulta. Los individuos en el algoritmo son directamente los mejores caminos de evaluación del plan lógico definido por la secuencia. Puede observarse en la Figura 7 cómo se realiza la transformación de cromosoma a plan.

El algoritmo GEQO se caracteriza por el remplazo de los individuos con la menor función de *fitness* a la

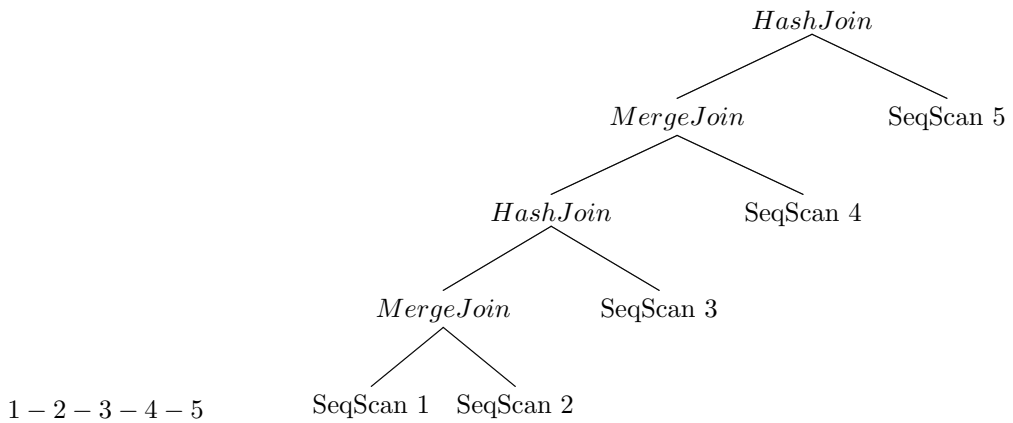


Figura 7: Representación de un plan como individuo en el algoritmo genético de PostgreSQL

población lo que permite una convergencia rápida, importante para la efectividad en el uso de este algoritmo. Además utiliza operadores de recombinación y mutación como mecanismos para variar la población [1, 3].

Algoritmo 3.3.2: GEQO(*poblacion*: ∅)

```
pobTam ← DETERMINARTAM()
gen ← DETERMINARGEN()
poblacion ← INICIALIZARPOBLACION()
while gen < MAXITER
   padre, madre ← SELECCION(poblacion)
   hijo ← RECOMBINACION(padre, madre)
   if TYPE(recombinacion) =" CX"

      MUTACION(hijo)
   CALCULARFITNESS(hijo)
   AGREGARAPOBLACION(hijo)
return MEJOR(poblacion)
```

Figura 8: Estructura del Algoritmo Genético de PostgreSQL (GEQO)

En la figura 8 se muestra el pseudo-código de la implementación realizada en PostgreSQL.

La función de inicialización de la población INICIALIZARPOBLACION(), se crean *pobTam* individuos generados aleatoriamente en base a los *relids* de las relaciones que intervienen en la consulta de entrada. El tamaño de la población y el número de generaciones son determinados en base al número de relaciones que intervienen en la consulta. La condición de parada del algortimo es que se completen el número de generaciones determinados previamente. El tamaño de la población y las tasas de mutación y selección son parámetros configurables por el programador.

GEQO presenta varios operadores de reproducción y uno de mutación que pueden ser configurados por el usuario para su uso. Los operadores de reproducción considerados son: *Edge Recombination Crossover* [49], *Cycle Crossover*[41], *Order Crossover* [46], *Partially Matched Crossover* [23] y *Position Crossover* [38]. El operador de mutación utilizado es *Swap operator*.

La función CALCULARFITNESS(hijo) construye el plan lógico y de alli selecciona el mejor camino de evaluación representativo del individuo y estudiando los caminos de acceso para cada *join*. El *fitness* es el costo estimado de evaluación asociado al plan representativo del individuo. Esta función construye un plan lógico como el que se construye en el componente *Path*, pero con el orden que indica el individuo.

Capítulo 4

Marco Metodológico

En este proyecto de grado, se elaboró una nueva versión correspondiente a la extensión de *PostgreSQL* denominada *PeaQock*, realizada por Brando y González [14]. Esta versión fue creada sobre *PostgreSQL* versión 8.1.4, desarrollada completamente en ANSI C. En la versión inicial de *PeaQock* fue extendido el *parser* para realizar el análisis léxico, sintáctico y semántico de Consultas *Skyline*, además de implementar los algoritmos de evaluación BNL y SFS. En esa versión, el optimizador fue modificado únicamente para colocar el operador *Skyline* como nodo *top-level*, es decir, sólo en la raíz del plan escogido en la optimización y por consiguiente ser evaluado de último como operador agregado.

En trabajos anteriores se muestra que se pueden generar mejores planes si se propaga el *Skyline* en el árbol de evaluación, ya que generalmente los criterios del *Skyline* reducen la cantidad de tuplas y por ende disminuyen los resultados intermedios cuando se produce la evaluación de la consulta [16]. Éste es el fundamento de la extensión propuesta en este trabajo, cuyo objetivo es considerar Consultas *Skyline* en las dos fases del proceso de optimización de *PostgreSQL*, con el fin de encontrar planes de evaluación que reduzcan el tiempo de la obtención de su respuesta.

Debido a que el algoritmo principal de la primera fase de optimización de *PostgreSQL* está basado en programación dinámica, como resultado de este trabajo se busca proporcionar una variante de éste que permita optimizar consultas *Skyline*. A esta variante se le denominó *Dynamic PeaQock Optimization (dPeaQock)*.

Adicionalmente, se realizó, de manera paralela en este proyecto de grado, el prototipo de un algoritmo evolutivo para la optimización de consultas *Skyline*, el cual se encuentra descrito en el Apéndice H. En base a los resultados obtenidos de este prototipo, se estudió la posibilidad y utilidad de desarrollarlo en *PostgreSQL*. Así, como trabajo adicional, se implementó este algoritmo evolutivo denominado *Evolutionary PeaQock Optimization (ePeaQock)*, tomando como fundamento este prototipo y el algoritmo genético *GEQO* de *PostgreSQL*. Se espera que *ePeaQock* sea una buena alternativa en la optimización de consultas *Skyline* para los casos donde el espacio de búsqueda sea inmanejable para *dPeaQock*.

Asímismo, se modificó el optimizador para colocar el *Skyline* como un nodo *top-level*, de esta manera podrá ser comparado el desempeño de los algoritmos tradicionales (*DYQO* y *GEQO*) frente al de los algoritmos propuestos (*dPeaQock* y *ePeaQock*), en la obtención de planes para consultas *Skyline*.

Finalmente, en este capítulo se describe el análisis correspondiente a la fase de inicio del proyecto, luego se explica el diseño e implementación de las estructuras y los algoritmos propuestos para la extensión de *PeaQock*.

4.1. Análisis

La primera fase consistió en el análisis del proyecto, se realizó un estudio inicial de los trabajos previos relacionados sobre la optimización de consultas, el módulo optimizador de consultas de *PostgreSQL*, las

consultas basadas en preferencias y la teoría de optimización de las mismas. Con la ayuda de este estudio inicial, se profundizó en la base teórica que apoya la utilidad de optimizar consultas *Skyline*, se refinaron tanto el alcance como los objetivos del proyecto y se definieron los requerimientos claves de la nueva versión de *PeaQock*.

Posteriormente, se procedió a entender el funcionamiento de *PostgreSQL*, dado que existe una escasa información a nivel técnico de este SMBDR, se realizaron diagramas de paquete, colocados en el Apéndice C, que ejemplificaran las tareas del módulo optimizador con el fin de facilitar el trabajo de implementación. Además, se estudió toda la información relacionada a las estructuras y procedimientos escenciales para extender el módulo optimizador. Este procedimiento se ecuentra descrito en el Apéndice A. Del mismo modo, se analizó la primera versión de *PeaQock* con la finalidad de definir las modificaciones necesarias a ser realizadas previas a la extensión del optimizador.

Una vez entendido el módulo optimizador en su totalidad, se procedió a definir una estrategia inicial para el desarrollo propuesto, estableciendo los requirimientos funcionales y de implemantación presentados a continuación:

- Optimizar consultas *Skyline* de manera eficiente mediante los algoritmos de optimización diseñados para tal fin.
- Permitir la visualización de los valores de las estructuras y costos de los planes generados a partir de una consulta *Skyline*.
- Contar con las estructuras y procedimientos necesarios en las etapas de análisis y re-escritura que facilitaran el trabajo de optimización.
- Procesar un operador *Skyline* como un nodo *down-level*, ya que en la versión de *PeaQock* anterior se procesa como un nodo *top-level*.
- Incorporar un modelo de costo que soporte la optimización de las consultas *Skyline* de manera coherente.
- Incorporar la implementación de los algoritmos *BNL* y *SFS* integradas en el mismo manejador. En la primera versión de *PeaQock*, existe un SMBDR distinto por cada implementación física o algoritmo de evaluación del *Skyline*, y es indispensable tener una versión integrada para que el optimizador, en base a los costos, pueda elegir cuál algoritmo de evaluación se va a utilizar para cada *Skyline* de una consulta.
- Poder recolectar medidas y datos relevantes para la realización del estudio experimental en este trabajo.
- Permitir escoger al usuario con qué algoritmo desea evaluar una consulta; *DYQO*, *GEQO*, *dPeaQock* o *ePeaQock*.

4.2. Diseño e Implementación

El diseño de la solución al problema de optimización de consultas *Skyline* comprende: las modificaciones realizadas a la representación de las cláusulas del *Skyline*, el modelo de costo para los algoritmos *SFS* y *BNL*, los algoritmos de optimización de programación dinámica y evolutivo propuestos, las estructuras necesarias para construir los planes lógicos, caminos de evaluación y planes de evaluación.

En esta sección se detallan los aspectos más importantes del diseño y la implementación de la solución para el problema de optimización de consultas *Skyline* en *PostgreSQL*. Como producto de esta etapa, se encuentran un diagrama de secuencia para el optimizador de *PostgreSQL* y un diagrama de paquetes que indica cómo se incluyó el algoritmo evolutivo propuesto. Además se muestra el manual para la extensión del optimizador de *PostgreSQL*. Los diagramas pueden observarse en el Apéndice C y el manual en el Apéndice A.

4.2.1. Modificaciones en el módulo *parser*

La representación de la cláusula *Skyline* original de la consulta es construída en el *parser* y almacenada en el árbol que es pasado como parámetro al optimizador. En la primera versión de *PeaQock*, esta cláusula, se representó como una lista donde cada elemento es un criterio del *Skyline* a utilizar, pero no almacenaba el identificador o *relid* de la relación a la que pertenece ese atributo.

Debido a que el fundamento de esa versión fue la evaluación de consultas *Skyline*, no tenía relevancia que el *Skyline* fuese definido únicamente para una sola tabla. Sin embargo, para la implementación del proceso de optimización fue necesario definir la estructura de la cláusula *Skyline* como una lista de listas, donde se asocia el *relid* de la relación con la lista de criterios del *Skyline* sobre esa relación. Asimismo, se agregó también el conjunto de *relids* involucrados en esa claúsula original que se denominará $parseSkylineRelids$.

De esta manera, el *Skyline* puede involucrar a más de una tabla y además facilita el trabajo del optimizador, ya que al realizarse la distribución del *Skyline* sobre una tabla base, sólo se necesita copiar el grupo de criterios de la cláusula original que tiene el *relid* de esa tabla.

4.2.2. Implementación del *Skyline* como nodo *top-level*

Como parte del diseño experimental, interesaba poder contar con mecanismos que permitieran evaluar consultas *Skyline* con el proceso de optimización tradicional. Para ello se adaptó el manejador de manera que, cuando el usuario desee, se realice la optimización tradicional y, después de formar el camino de evaluación conformado por *Scans* y *Joins*, el optimizador agregue el *Skyline* como nodo *top-level*. En este caso, el optimizador también realiza un procesamiento extra para eligir cuál implementación estima que es mejor utilizar, *SFS* o *BNL*. El procesamiento como nodo *down-level* será explicado con más detalle en las siguientes subsecciones. Este nodo *top-level* se diferencia al creado en la primera versión de *PeaQock* en que es construido al finalizar la primera fase de optimización, por lo que se le permite decidir al optimizador el operador físico a utilizar. En cambio anteriormente, éste se creaba al finalizar la segunda fase de optimización como un nodo *SFS* o *BNL*, dependiendo del manejador de *PostgreSQL* utilizado.

4.2.3. Implementación del *Skyline* como nodo *down-level*

Para implementar el *Skyline* como nodo *down-level* fue necesario modificar la estructuras `RelOptInfo`, la cual respresenta un plan lógico y `Path` que contiene la información general de un camino de acceso para cualquier operador lógico.

En el `RelOptInfo`, se añadió la cláusula del *Skyline* y el conjunto de *relids*, denominado *skylineRelids*, de las relaciones involucradas en esa claúsula . El objetivo de esto es evitar tener que recorrer el plan lógico para verificar si se pueden aplicar las reglas algebraicas definidas para *Joins* y *Skylines*. De la misma forma, se modificó la estructura *Path*.

Para la implementación de los operadores físicos del *Skyline*, fue necesario crear las estructuras `SFSPath` y `BNLPath` que contienen información específica para un camino de acceso *SFS* y *BNL*, respectivamente. Éstas permiten que *Skyline* pueda ser evaluado como *SFS* o *BNL*, dependiendo de cuál se haya elegido en base al costo.

Con estas estructuras definidas, se desarrollaron las funciones y procedimientos indispensables para tratar al *Skyline* como un nodo *down-level*. Entre las más importantes se encuentran: `make_skyline_rel`, `build_skyline_rel`, `add_paths_to_skyline_rel`, `skyline_cardinality`, `skyline_comp`, `create_sfs_path` y `create_bnl_path`.

El procedimiento `make_skyline_rel` es el más general. Este procedimiento recibe como parámetro de entrada un `RelOptInfo` que representa el plan construído hasta ese momento y retorna el plan lógico resultante luego de agregarle el *Skyline*. En este punto, se ejecuta `build_skyline_rel` que inicializa el nuevo plan lógico y le asigna como hijo izquierdo el `RelOptInfo` de entrada. Inmediatamente, se construye la cláusula *Skyline* asociada al nuevo plan en base a los *SkylineRelids* del plan hijo. Posteriormente, en `make_skyline_rel` se ejecuta `add_paths_to_skyline_rel` que construye los caminos de acceso para el *Skyline* y los caminos de evaluación posibles.

En el procedimiento `add_paths_to_skyline_rel` se calculan la cardinalidad estimada del *Skyline* y el número estimado de comparaciones para el *BNL* y *SFS*. Esto se realiza en base a la cantidad de tuplas que el plan hijo calcula que le va a enviar. La cardinalidad estimada del *Skyline* se calcula por medio de la función `skyline_cardinality` y las comparaciones a través de `skyline_comp`. Estos cálculos se explican más adelante en este capítulo.

Adicionalmente, en `add_paths_to_skyline_rel` se crea la lista de caminos de evaluación. Esta lista se construye creando dos nuevos caminos de evaluación por cada camino del plan lógico hijo. Son creados añadiéndole los operadores *BNL* y *SFS*. Estos caminos de evaluación son creados en `create_bnl_path` y `create_sfs_path`, respectivamente. Al crearse los caminos de acceso, se estima el costo del mismo tomando el costo acumulado hasta el nodo hijo en el camino de evaluación y sumándole el costo estimado de aplicar el operador *BNL* o *SFS*.

La creación del camino de acceso *SFS* implica un mayor procesamiento en el optimizador, debido a que el algoritmo de evaluación para este operador en *PeaQock*, tiene como precondición que la relación producto

de evaluar el plan hasta ese momento, debe estar ordenada por las dimensiones del *Skyline*. Por ello el optimizador, al crear un camino de acceso *SFS*, debe considerar el costo de ordenamiento de la relación entrante en caso de ser necesario.

Cada camino de acceso almacena una lista denominada *pathkeys* que indica la forma en que se encontrará ordenada la relación creada hasta ese momento, despues de aplicar el operador. En el caso del *Skyline*, los *pathkeys* además de indicar el orden de la relación de salida, indican el ordenamiento que debe tener la relación de entrada según las dimensiones de ese nodo *Skyline*.

Los *pathkeys* son verificados para la estimación del costo de un *SFS*. Éste no se suma en el costo del camino si la relación de entrada ya se encuentra ordenada por esos atributos, en cuyo caso se indicará que no hace falta realizar un ordenamiento previo a la ejecución del *SFS*. Las restricciones de ordenamiento o *pathkeys* son colocadas, a lo largo del camino de evaluación, si son necesarias para aplicar un *MergeJoin* o un *SFS*.

Una vez implementado el *Skyline* como nodo *down-level* las características que definen la equivalencia entre los planes lógicos son distintas. Un plan lógico será ahora equivalente a otro, si y sólo si, poseen el mismo conjunto de $relids$ y de $skylineRelids$. Con esto se garantiza que para tener planes equivalentes es necesario que se haya aplicado el mismo conjunto de criterios del *Skyline*, no importa si la cantidad de *Skylines* en el plan es distinta, lo importante es que sea bajo las mismas relaciones involucradas.

En la Figura 9 se observa que los planes *Plan 1* y *Plan 3* son equivalentes, porque tienen los mismos conjuntos $relids = \{A, B, C\}$ y $skylineRelids = \{A, B\}$. Sin embargo, ninguno de ellos es equivalente al *Plan 4* ya que los conjuntos $relids = \{A, B, C\}$ y $skylineRelids = \{A\}$ son distintos.

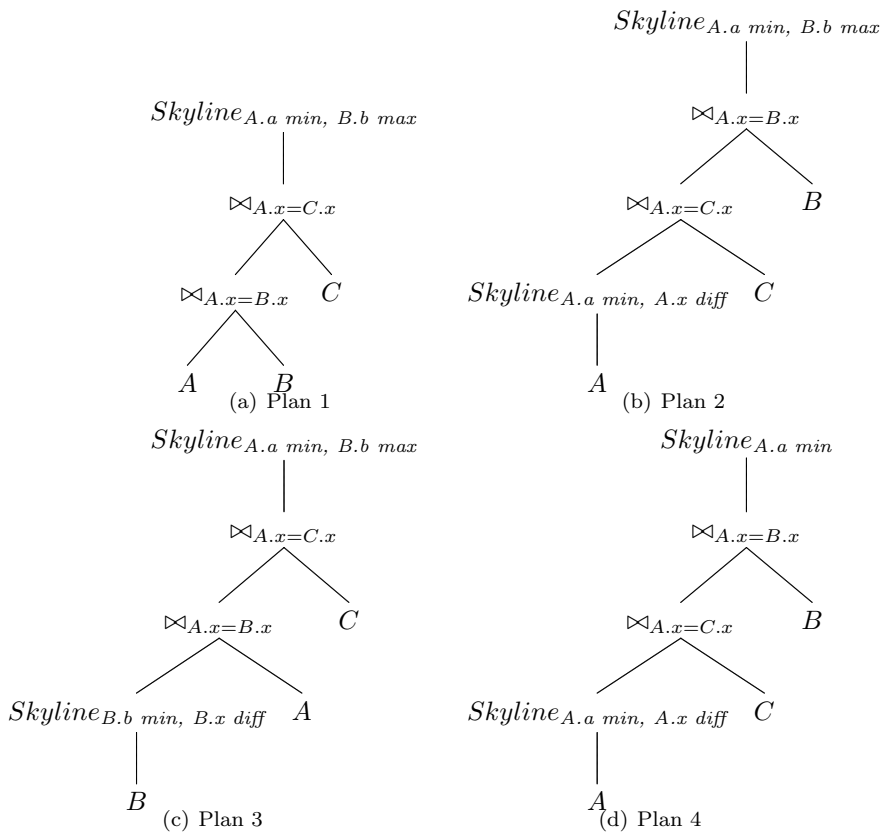
4.2.4. Implementación del modelo de costo para el *Skyline*

Para el *Skyline* se implementó el modelo de costo tomando como referencia las fórmulas de estimación de la cardinalidad del *Skyline* y de estimación del número de comparaciones de *BNL* y *SFS* en [16].

Inicialmente, se implementaron las fórmulas exactamente como estaban definidas. Sin embargo, al ser sumatorias y recursiones en base al número estimado de tuplas de entrada, se realizaba un alto procesamiento durante un período considerable de tiempo lo que resultaba demasiado costoso si se considera una gran cantidad de tuplas.

Debido que estos cálculos se deben realizar muchas veces durante la primera fase de optimización (cada vez que se considera colocar un *Skyline*), no tiene sentido que el porcentaje del tiempo consumido por ellos sea muy alto, ya que perjudica al tiempo total invertido en la optimización. Por ello se intentó mejorar la implementación de estas fórmulas.

Como primera opción, se precalcularon todos los valores de la estimación de la cardinalidad del *Skyline* tomando como entrada hasta 100000 tuplas y 20 dimensiones y se almacenaron en una matriz de tuplas por



(a) Plan 1 (b) Plan 2 (c) Plan 3 (d) Plan 4

Figura 9: Equivalencia e inequivalencia de planes lógicos

dimensiones. De esta manera, para obtener la cardinalidad estimada del *Skyline*, teniendo el número de tuplas de la relación de entrada y el número dimensiones, se ejecutaba un simple acceso constante a memoria.

Por otro lado, para estimar las comparaciones realizadas por *BNL* y *SFS* también se realizaba un cálculo bastante costoso. Por ejemplo, para calcular las comparaciones de *BNL* para 10000 tuplas se necesitaba sumar 9998 términos y, aunque para obtener cada término se hacían dos accesos a la matriz de cardinalidades del *Skyline*, el cálculo resultaba bastante lento. Se pensó entonces en precalcular además el número de comparaciones para *BNL* y *SFS*, pero luego de estudiar la cantidad de memoria que ocuparían estas matrices más la matriz de cardinalidades, se rechazó esta idea.

Aunque precalcular los valores en una matriz era una solución eficiente en cuanto al tiempo de cálculo de la cardinalidad de *Skyline*, no es una solución flexible porque no permite estimar de manera sencilla los valores para aquellos casos donde las dimensiones son más de 100000 tuplas o 20 dimensiones y considerar una matriz más grande podía convertirse inmanejable a nivel del uso de la memoria.

Para solucionar estos problemas presentados se hizo un estudio más amplio con el objetivo de encontrar una buena aproximación a estas fórmulas y reducir los tiempos de cómputo.

4.2.4.1. Aproximación para la estimación de la cardinalidad del *Skyline*

Dado que no existe fórmula cerrada para el cálculo de la cardinalidad del *Skyline* de n tuplas y d dimensiones, $S_{(n,d)}$, y que el precálculo de los valores se hacía inmanejable para valores muy grandes de n y d, se decidió realizar una aproximación de estos valores. En la Figura 10 se muestran las aproximaciones realizadas a esta fórmula.

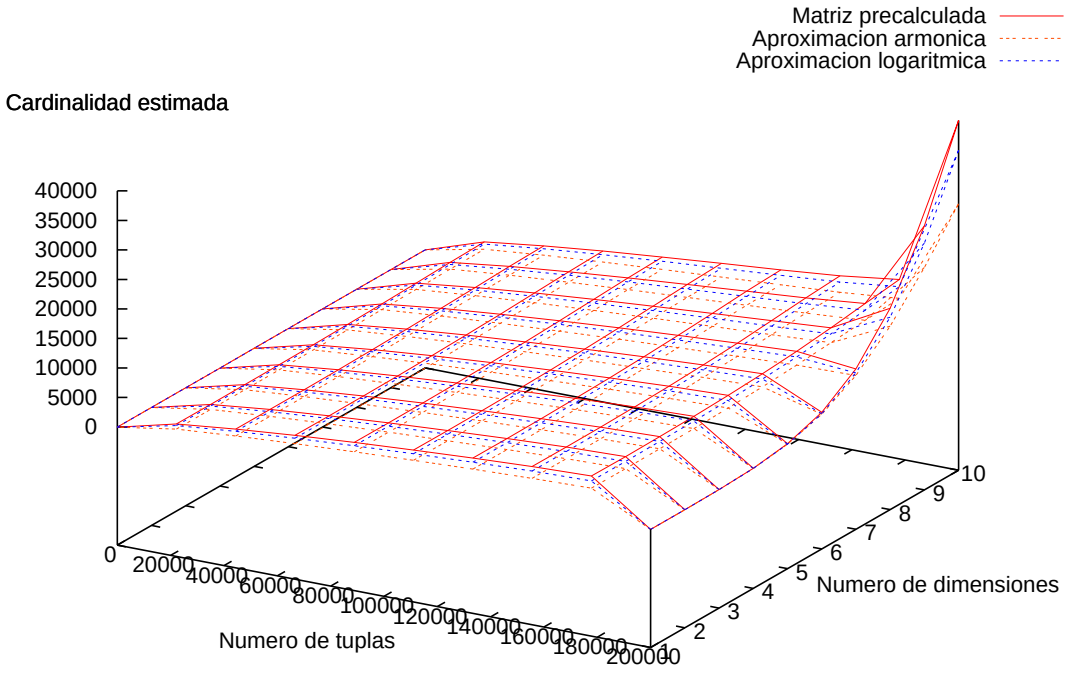


Figura 10: Comparación entre las aproximaciones realizadas

Como se mencionó en el capítulo 2, existen trabajos que aproximan el cálculo de $S_{(n,d)}$. Una de las propuestas es la aproximación basada en los números armónicos bidimensionales [27] mediante la fórmula 8

$$S_{(n,d)} = H_{(n,d-1)} = \frac{(ln(n)+\gamma)^{d-1}}{d-1!} \tag{8}$$

Por otro lado, en [16], se muestra que el comportamiento de la función de cardinalidad del conjunto *Skyline* crece con orden $\Theta(\log n)$ por lo que podría ser aproximado con una expresión de la forma $A \log^B n$. En ese trabajo se utilizó esa afirmación para realizar muestreo sobre las relaciones base a las que se aplicaba el *Skyline* y así poder estimar los valores de las constantes A y B. El muestreo sobre los datos de relaciones base no es una solución que se adapte al problema que se intenta resolver en este trabajo, porque el *Skyline* puede estar en cualquier lugar del plan.

Sin embargo, durante el diseño de la solución se estudiaron estas dos propuestas y se decidió verificar

qué tan buena es la aproximación de los armónicos bidimensionales y, por otro lado, adaptar la propuesta de muestreo para originar funciones que permitieran aproximar la cardinalidad del *Skyline*.

El objetivo de esta adaptación era aproximar una expresión de la forma $A \log^B n$ para cada dimensión. De esta manera, fijada la dimensión, se promediaron los valores de A y B hallados para cien pares de puntos escogidos aleatoriamente. De esta manera para diez dimensiones se obtienen diez coeficientes A_i y B_i donde i es la dimensión.

En la Figura 10 se muestra una gráfica que comprende los valores de $S_{(n,d)}$ hallados para n igual a 250000 y d igual a 10. Como se observa, la mejor aproximación es brindada por la segunda solución: la aproximación logarítmica.

4.2.4.2. Modelo de costo propuesto para el *Skyline*

El modelo de costo explicado en el Capítulo 2 presenta algunas debilidades que podrían ser solventadas para que el optimizador realice una mejor estimación.

En primer lugar, el modelo de costo implementado para el *Skyline* supone que los valores de las dimensiones son independientes, por lo tanto si existen dimensiones correlacionadas o anti-correlacionadas la estimación podría ser bastante inexacta, según Dalvi [16]. El problema es que si se quisiera obtener la correlación de los datos se tendría que modificar el optimizador para que pueda realizar técnicas de muestreo de los datos, explicadas en el mismo trabajo de Dalvi, para estimar la cardinalidad del *Skyline* en base a la correlación.

Este muestreo puede realizarse cuando el *Skyline* se encuentra sobre relaciones base, pero se dificulta una vez que el *Skyline* se encuentre sobre un *Join*. Para los operadores que no son *Scan*, se tendría que efectuar un muestreo de todos los operadores anteriores hasta llegar a las hojas del árbol, para poder así obtener un resultado válido. Esto además, se tendría que realizar para cada plán lógico, lo que perjudicaría en gran medida el tiempo de optimización. Por estas razones, la estimación por muestreo fue descartada para este proyecto.

En segundo lugar, no existe ningun trabajo previo a este proyecto que considere la directiva *diff* en el cálculo de la estimación tanto de la cardinalidad como del número de comparaciones realizadas para el *Skyline*. Las fórmulas actuales sólo toman en cuenta las dimensiones con directivas *min* y *max*. La directiva *diff* en una consulta *Skyline* se comporta de manera distinta a las otras directivas, como se explicó anteriormente en el Capítulo 2.

Por cada grupo de tuplas que posean el mismo valor en el criterio *diff*, el evaluador debe aplicarle a estas tuplas el resto de los criterios (con *min* y *max*) del *Skyline*. Es decir, que el evaluador efectuará el *Skyline* tantas veces como valores distintos existan en los criterios con *diff*. Esto es importante para los planes donde el *Skyline* se encuentra distribuido, ya que la tercera regla del *Skyline* sobre el *Join*, en la Tabla 1, tiene como condición para colocar un nuevo *Skyline*, que el atributo de *Join* con la directiva *diff* debe ser agregado al conjunto de criterios de éste.

Tomando en cuenta esta problemática, se propone en este proyecto considerar las dimensiones indicadas con *diff* como un tipo de dimensiones distintas para estimar tanto la cardinalidad del *Skyline* como el número

de comparaciones. El modelo de costo propuesto en este trabajo se presenta a continuación.

Se asumirá que n es la cantidad de tuplas estimada de entrada para el *Skyline*, g es el número estimado de grupos del *Skyline*, $m = \frac{n}{g}$ es la cantidad de tuplas estimada de entrada para un grupo *Skyline* (suponiendo que se distribuyen uniforme) y d es el número de criterios del *Skyline* sin incluir los criterios con *diff*.

Para la cardinalidad estimada del $SkylineCard(n, d)$ se plantea, en las fórmulas 9 y 10, que el cálculo debe ser efectuado en función del número de los distintos grupos de tuplas definidos por los atributos presentes en los criterios con *diff*, ya que el *diff* se comporta como el operador *Group by* [32] . Asumiendo uniformidad en los datos, si se tienen g grupos entonces la cardinalidad estimada de cada grupo de entrada es m y $\hat{S}_{m,d}$ es la cardinalidad esperada de cada grupo *Skyline*. Se denomina grupo *Skyline* al conjunto de tuplas obtenido después de aplicarle el *Skyline* al grupo original, tomando solamente los criterios correspondientes a las directivas *min* y *max*. Si no se tiene ningún *diff*, g es igual a 1.

$$\hat{S}_{m,d} = \tfrac{1}{m}\hat{S}_{m,d-1} + \hat{S}_{m-1,d} \tag{9}$$

$$Card(n, d) = \hat{S}_{m,d} * g \tag{10}$$

El número de comparaciones de tuplas realizadas en la evaluación, como se dijo anteriormente, depende del operador físico a utilizar *BNL* o *SFS*. En la fórmula 11, se tomará como el número de comparaciones base las realizadas para *BNL*, $Comp(n, d)$. Se sabe que para *SFS* es similar pero restandole uno a d porque se asume que las tuplas ya poseen un orden total para una dimensión. El número estimado de comparaciones para el *Skyline*, considerando el *diff*, será el estimado para un grupo multiplicado por el número de grupos *Skyline*.

$$Comp(n, d) \approx (\textstyle\sum_{j=2}^{m} \frac{\hat{S}_{j-1,d}}{j-1}\hat{S}_{j-1,d+1}) * g \tag{11}$$

El costo estimado de CPU para *BNL*, $C_{BNL}(n, d)$, viene dado en la fórmula 12, por el costo de hacer un *Group by* por los criterios con *diff*, $C_{GroupBy}(n, G)$, más el número de comparaciones $Comp(n, d)$ y el costo de procesar cada tupla de la relación de entrada. En cambio, el costo estimado de CPU para *SFS*, $C_{SFS}(n, d)$, incluye, en la fórmula 13, el costo estimado de ordenar la relación de entrada por todas las dimensiones del *Skyline*, $C_{Sort}(n, C)$, ordenando primero por las correspondientes a la directiva *diff* y luego por las demas y $C_{GroupBy}(n, G)$, $Comp(n, d-1)$ y el costo de procesar cada tupla.

$$C_{BNL}(n, d) \approx C_{GroupBy}(n, G) + Comp(n, d) * k \tag{12}$$

$$C_{SFS}(n, d) \approx C_{Sort}(n, C) + C_{GroupBy}(n, G) + Comp(n, d-1) * k \tag{13}$$

Donde C es el conjunto formado por las dimensiones de todos los criterios *Skyline*, G es el conjunto

formado sólo por las dimensiones con directivas *diff* y k es el costo de procesar una tupla.

Sin embargo, en la implementación del modelo de costo no pudo considerarse esta propuesta debido a que la estimación de costo para el *Group by* en *PostgreSQL* por la forma en que está implementada, sólo puede realizarse cuando el plan es escogido por el optimizador, ya que éste es un operador agregado. Conveniendo para trabajos futuros estudiar la factibilidad de estimar el costo del *Group by* durante la primera fase de optimización y de esta manera, permitir la implementación de éste modelo de costo.

4.2.4.3. Modelo de costo implementado para el *Skyline*

Para la implementación final del modelo de costo del *Skyline* se precalculó una matriz por capas para la función $Comp(n, d)$ variando a n hasta 100 millones y d hasta 15 dimensiones, y donde el intervalo entre una capa y otra son 10000 tuplas. Además, como resultado del estudio de aproximación realizado para la estimación de la cardinalidad del *Skyline*, se decidió implementar la aproximación logarítmica de la fórmula 14, ya que ésta se ajusta mejor al cálculo de $S_{(n,d)}$ definido en [16].

$$\hat{S}_{n,d} \approx a(d) * log(n)^{b(d)} \tag{14}$$

Donde $a(d)$ y $b(d)$ corresponden a las constantes A y B promedio precalculadas para hallar el estimado de la cardinalidad en función de la dimensión d.

De esta manera, el cálculo de la cardinalidad se realiza utilizando una expresión de la forma $A_i(\log n)_i^B$, donde A_i y B_i se encuentran precalculados en una matriz de coeficientes. Aunque esta solución es limitante porque deben existir los coeficientes para la dimensión adecuada, resulta muy sencillo calcular y/o almacenar un número mayor de pares de coeficientes ya que el espacio ocupado por esta matriz no es comparable con la matriz de cardinalidades implementada en primer lugar, lo que brinda mayor flexibilidad a esta solución. Para esto, fue necesario almacenar las constantes A y B para cada dimensión y modificar las funciones de estimación de cardinalidad y costos.

La función que estima el número de comparaciones, fórmula 15, se calcula en base a esta aproximación logarítmica, haciendo uso de la matriz de costos precalculada. Se realiza una sumatoria hasta que se consiga un término que ya se encuentra precalculado en la matriz. Esto hace que a lo sumo, se realice la sumatoria de 9999 términos, para cualquier n menor que un billón.

$$Comp(n, d) \approx \sum_{j=i}^{n} \frac{\hat{S}_{j-1,d}}{j-1} \hat{S}_{j-1,d+1} + Comp'(i, d) \tag{15}$$

Donde i es la cantidad de tuplas más cercana por debajo a n que le corresponde un indice en la matriz de comparaciones y $Comp'(i, d)$ es equivalente a $Comp(i, d)$ con la diferencia que el número de comparaciones es extraído de la matriz precalculada.

Por otro lado, con la finalidad de mantener la coherencia con el modelo propuesto en que el costo estimado de aplicar un *Skyline* sin directivas *diff* es menor que el costo de aplicar uno con directivas *diff*, se tomaron para las fórmulas de costo 16 y 17 de *BNL* y *SFS* los criterios definidos por la directiva *diff* como cualquier

otro criterio.

A continuación se presenta el modelo de costo implementado en esta extensión de *PostgreSQL*.

$$C_{BNL}(n,d) \approx Comp(n,d) * k \tag{16}$$

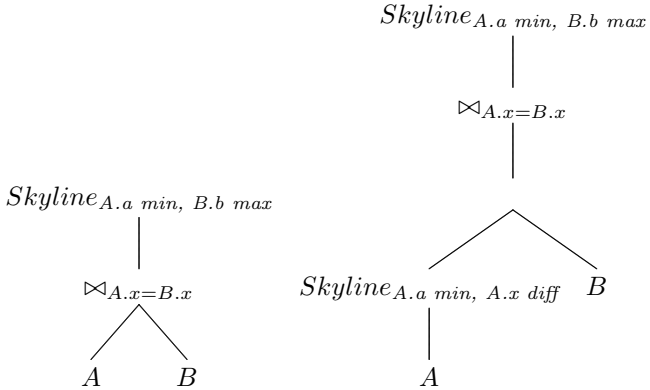
$$C_{SFS}(n,d) \approx C_{Sort}(n,C) + Comp(n,d-1) * k \tag{17}$$

4.2.5. Algoritmo basado en programación dinámica: *dPeaQock*

En relación a la arquitectura de *PostgreSQL* y sus modificaciones, se propuso el algoritmo *Dynamic PeaQock Optimization (dPeaQock)* basado en programación dinámica. En la Figura 12 se presenta el pseudocódigo de ésta variación propuesta para el algoritmo *DYQO*. *dPeaQock* es similar a *DYQO* aunque presenta ciertas modificaciones.

El procedimiento CALCULARPLANESJOIN(i,listasPlanes[1]) es sustituído por el nuevo procedimiento CALCULARPLANESJOINSKYLINE(i,listasPlanes[1]). También es modificado el procedimiento ELIMINAREQUIVALENTES(listasPlanes[x],planes) para que considere la nueva equivalencia entre planes definida anteriormente.

A partir de la ejecución de CALCULARPLANESJOIN(i,listasPlanes[1]) se construyen los posibles planes de la consulta, por lo tanto, es en este procedimiento donde se debe incluir la creación de planes *Skyline*. De esta forma, el *Skyline* puede ser colocado en cualquier lugar del árbol de optimización, permitiendo hacer separación y *push-down* con respecto al árbol inicial de la consulta donde el *Skyline* se encuentra en la raíz como se observa en la Figura 11.



(a) Plan inicial de la consulta (b) Plan transformado

Figura 11: Transformación del *Skyline*

Este cambio se justifica por la necesidad de considerar colocar un nodo *Skyline* donde se pueda ubicar. Debido a su característica de operador que puede filtrar muchas tuplas [32], conviene estudiar su posicionamiento en distintos niveles del plan porque se espera entonces que planes que contengan más distribuído el

Algoritmo 4.2.1: CONSTRUIRPLAN(R:relaciones base, l:niveles necesarios)

```
procedure CALCULARPLANESJOINSKYLINE(plan, planesBase)
  result ← ∅
  for each p ∈ planesBase
    if RELIDSDISJUNTOS(p, plan) and ESFACTIBLEJOIN(p, plan)
      //Caso 1
      join ← CREARJOIN(plan, p)
      result ← result ∪ join

      //Caso 2: Coloca el Skyline sobre p antes de realizar el Join
      if RELIDSSOLAPADOS(p, parseSkylineRelids)
        skyline ← CREARSKYLINE(p, atributoDiff)
        ESCOGERMEJORESCAMINOS(skyline)
        join ← CREARJOIN(plan, skyline)
        result ← result ∪ join

      //Caso 3: Coloca el Skyline sobre plan antes del Join
      if RELIDSSOLAPADOS(plan, parseSkylineRelids)
        skyline ← CREARSKYLINE(plan, atributoDiff)
        ESCOGERMEJORESCAMINOS(skyline)
        join ← CREARJOIN(skyline, p)
        result ← result ∪ join

      //Caso 4: Coloca un Skyline sobre plan y otro sobre p antes del Join
      if RELIDSSOLAPADOS(plan, parseSkylineRelids)
       and RELIDSSOLAPADOS(p, parseSkylineRelids)
        skyline1 ← CREARSKYLINE(plan, atributoDiff1)
        skyline2 ← CREARSKYLINE(p, atributoDiff2)
        ESCOGERMEJORESCAMINOS(skyline1)
        ESCOGERMEJORESCAMINOS(skyline2)
        join ← CREARJOIN(skyline1, skyline2)
        result ← result ∪ join
  return result
```

Figura 12: Pseudocódigo del algoritmo de optimización propuesto basado en programación dinámica

Skyline, se evalúen en menor tiempo.

En consecuencia, los planes construidos en CALCULARPLANESJOINSKYLINE(i,listasPlanes[1]) corresponden a un espacio de búsqueda mayor, ya que los planes lógicos comprenden ahora *Scans*, *Joins* y *Skylines*. Esto implica un mayor tiempo empleado en el proceso de optimización. Sin embargo, se espera que sea compensado con una disminución considerable en el tiempo de evaluación.

El *Skyline* sobre un plan se produce ejecutando CREARSKYLINE(p, atributoDiff) (en *PostgreSQL*, esta función es `make_skyline_rel`), no obstante, en vista de que éste debe ser manipulado bajo ciertas reglas para no alterar su semántica, en la optimización se utiliza la regla más general [32] para construir los planes. Ésta regla permite distribuir el *Skyline* sobre un *Join* y si se aplica recursivamente permite repartir el *Skyline* en diversos lugares del plan lógico. Sin embargo, para crear un *Skyline* sobre un plan p empleando esta regla, se debe verificar mediante RELIDSSOLAPADOS(p,skylineRelids) que el conjunto de los *relids* del plan p se encuentra solapado con el conjunto *skylineRelids* que abarca la claúsula de *Skyline* original de la consulta, esto es para verificar que se puede aplicar un *Skyline* a ese nivel.

Seguidamente, se debe agregar a la cláusula del *Skyline* a construir, el atributo por el cual realiza el *Join* en forma de *diff*, *atributoDiff*. Es importante destacar, que el nuevo *Skyline* incluye todos los criterios que corresponden a las relaciones que ya han ocurrido en el plan más el criterio *diff* agregado por la regla.

El *Skyline* creado, al igual que el *Join*, tiene asociado la lista de los caminos de evaluación para el plan. Se escogen los mejores caminos de evaluación para el plan formado por el *Skyline* por medio de ESCOGERMEJORESCAMINOS(skyline) con el motivo de podar el espacio de búsqueda.

Luego, se construye el *Join*. Los posibles hijos dependiendo del caso para el *Join*, pueden ser:

- Caso 1: un *Join* (o un plan base) y un plan base.
- Caso 2: un *Join* (o un plan base) y un *Skyline* (sobre un plan base).
- Caso 3: un *Skyline* (sobre un *Join* o un plan base) y un plan base
- Caso 4: un *Skyline* (sobre un *Join* o un plan base) y un *Skyline* (sobre un plan base).

Un corrida simplificada de este algoritmo se muestra en la Figura 13, en la cual pueden visualizarse los planes lógicos de cada nivel. La corrida corresponde a la consulta de entrada:

```
SELECT * FROM A, B
WHERE A.x=B.x
SKYLINE OF A.a min, B.b min
```

La notación $S_{A.a\ min,\ B.b\ min}$ representa el operador lógico para el *Skyline* por los criterios *A.a min, B.b min*. Se observa que para el segundo nivel, se crean los planes con *Joins* entre A y B y todos las posibles formas de colocar el *Skyline* en ese nivel. Si hubiesen más tablas con las que hacer *Join* se realizaría un tercer nivel. Finalmente, se escoge el plan que contenga el camino de evaluación menos costoso y se agrega el nodo

Skyline de la raíz. Recuerde que el algoritmo es más complejo, ya que en cada nivel se crean todos los posibles caminos de evaluación para cada plan lógico.

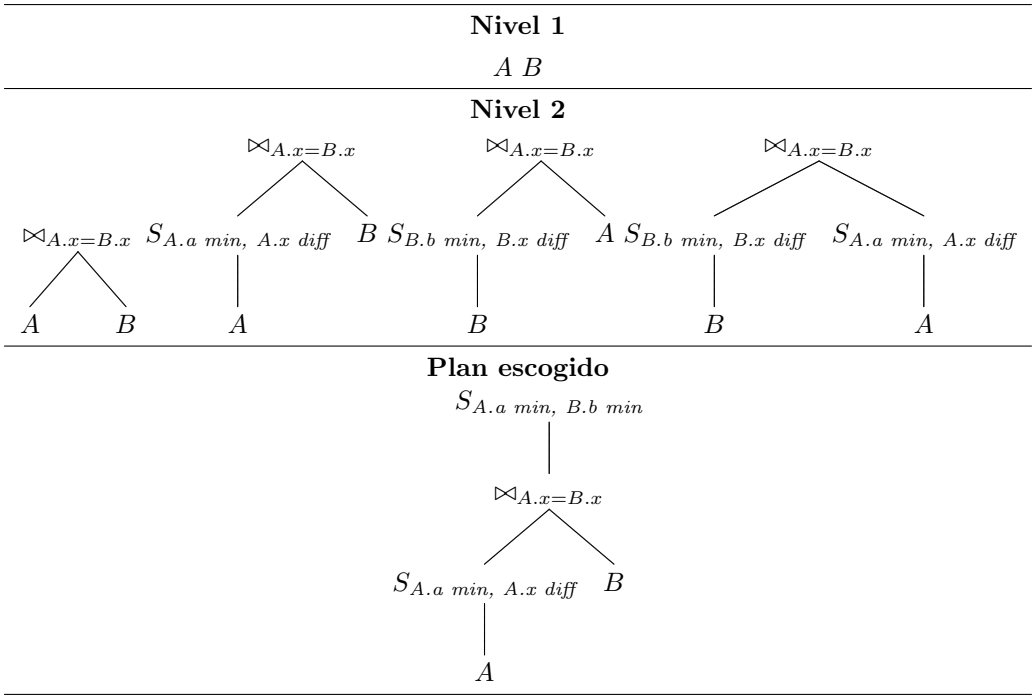


Figura 13: Corrida simplificada del algoritmo *dPeaQock*

4.2.6. Algoritmo evolutivo: *ePeaqock*

A continuación, se abarcan los problemas presentados y las soluciones brindadas en el diseño e implementación del algoritmo *Evolutionary PeaQock Optimization (ePeaQock)*, así como la motivación de la inclusión de este algoritmo dentro de *PostgreSQL*. *ePeaqock* es un algoritmo evolutivo propuesto para brindar una alternativa en la búsqueda de planes para evaluar una consulta *Skyline*. Este algoritmo responde a la necesidad de mecanismos que sean capaces de manejar un espacio de búsqueda de gran tamaño.

Al introducir al *Skyline* dentro del proceso de optimización, el espacio de búsqueda se amplía [1], haciendo que la utilidad del algoritmo de programación dinámica se vea disminuida por invertir demasiado tiempo en encontrar un camino de evaluación. El diseño preliminar de este algoritmo se encuentra descrito en el trabajo que se encuentra anexo en el Apéndice H. Sin embargo, para la adaptación del algoritmo a la arquitectura de *PostgreSQL*, se tuvo que realizar un estudio adicional para determinar cuáles cambios se tenían que realizar respecto al prototipo original.

Las características principales de *ePeaqock* son:

- **Generación de planes válidos**: se decidió que la generación y mutación generaran siempre individuos válidos.

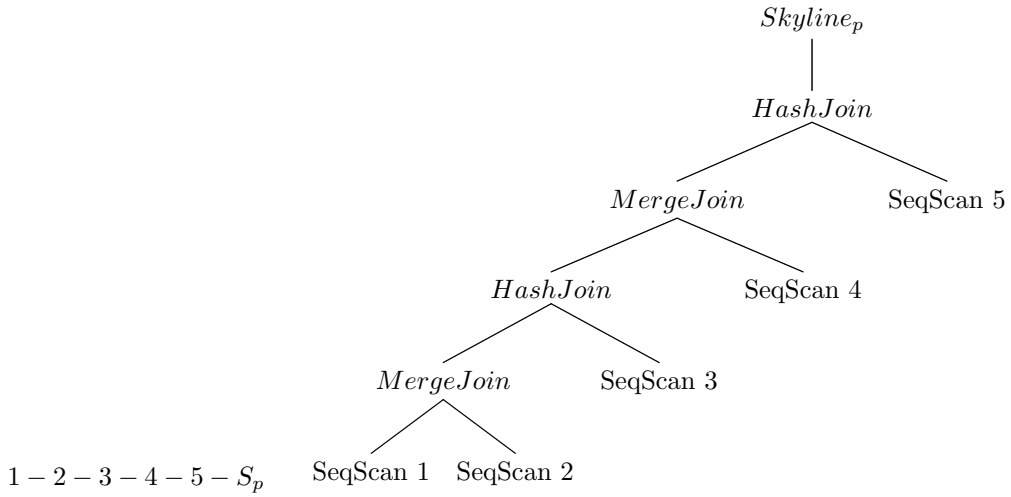
[1] ver Apéndice B

- **Ausencia de *crossover* o reproducción**: como instrumento para variar la población, se implementaron solamente operadores de mutación. Como se dijo en secciones previas, las consultas consideradas por este trabajo involucran los operadores *Join* y *Skyline*. Considerando que las operaciones que involucran estos operadores no son conmutativas o asociativas, los individuos son de longitud variable y que se desea que los individuos generados sean siempre válidos, el implementar un operador de *crossover* se hace no sólo complicado sino costoso en ejecución debido a las muchas verificaciones (y reordenamientos) que se tienen que realizar para garantizar que al unir dos individuos el plan sea válido.
- **Tasa de mutación alta**. Como el algoritmo solamente posee mutaciones como el mecanismo para variar la población y por la naturaleza de los operadores no se garantiza que un individuo se puede mutar, se fijó una tasa de mutación alta.
- **Múltiples operadores de mutación**: el algoritmo utiliza dos operadores de mutación para variar la población, uno para modificar el orden del *Join* y otro para distribuir el *Skyline* en el árbol. La aplicación de un operador u otro es aleatoria con igual probabilidad de escogencia.
- **Condición de parada compuesta**: La condición de parada del algoritmo es que se cumpla un número máximo de iteraciones o que el mejor individuo se estabilice en una misma solución en un número indicado de iteraciones.
- **Selección elitista**: en este algoritmo el mejor individuo nunca es mutado porque se desea conservarlo.

En la Figura 15, se muestra la estructura de *ePeaqock* de manera muy general. La inicialización de la población realizada por INICIALIZARPOBLACION() genera individuos que representan planes donde el *Skyline* se encuentra como operador raíz del plan. Cada individuo está caracterizado por un costo y un arreglo de genes. Los genes representan los operadores relacionales *Join* o *Skyline* y cada gen contiene un identificador de la tabla base que se involucra en el plan con su aplicación. Existe una correspondencia directa entre un individuo y un camino de evaluación. En la Figura 14 se muestra la representación de un individuo inicial y el respectivo camino de evaluación que representa.

Los operadores de mutación utilizados son: MUTARSKYLINE(ind) y MUTARJOIN(ind). El primero, revisa si a un operador *Skyline* se le puede realizar un *push-down* de acuerdo a las reglas algebraicas explicadas en 2.3.2. El segundo, revisa si puede intercambiar el orden de dos *Joins* escogidos de manera aleatoria fundamentandose en las reglas de asociatividad y conmutatividad del *Join* descritas en el Apéndice E.

La función CALCULARFITNESS(ind), es la que permite medir qué tan buena solución es la que representa un individuo. La función de *fitness* es el costo de evaluación estimado del plan que representa. Mientras menor costo posea mejor es su *fitness*. Esta función se encarga de transformar la estructura del individuo a un plan lógico con sus respectivos caminos de evaluación, mediante la llamada correspondiente a `make_join_rel` y `make_skyline_rel`. El valor del *fitness* de un individuo es el costo estimado asociado al mejor camino de evaluación para el árbol lógico representado por él. Esta función también es de utilidad para construir el mejor camino de evaluación que será la entrada a la etapa de construcción del plan de evaluación.



(a) Individuo (b) Plan representativo del individuo

Figura 14: Representación de un plan como individuo en el algoritmo ePeaqock

Algoritmo 4.2.2: GEQO(*poblacion*: ∅)

```
procedure MUTACION(poblacion)
  mutados ← 0
  while mutados < tasaMutacion * poblacion.tam
     ind ← ESCOGERINDIVIDUO(poblacion)
     metodo ← ESCOGERMETODO()
     if metodo = Skyline
       MUTARSKYLINE(ind)
      else MUTARJOIN(ind)
     ind.costo ← CALCULARFITNESS(ind)
     gen ← gen + 1

main
  pobTam ← DETERMINARTAM()
  gen ← DETERMINARGEN()
  poblacion ← INICIALIZARPOBLACION()
  temporal ← MEJOR(poblacion)
  while gen < MAXITER and estable < MAXSTABLE
     MUTACION(poblacion)
     gen ← gen + 1
  return MEJOR(poblacion)
```

Figura 15: Estructura del Algoritmo Evolutivo propuesto ePeaqock

4.2.6.1. Respecto a la inclusión de un nuevo algoritmo

Dado que *PostgreSQL* cuenta con un mecanismo de optimización basado en algoritmos genéticos (*GEQO*), se analizó la posibilidad de realizar una extensión del algoritmo genético existente, de la misma manera que se extendió el algoritmo de programación dinámica. Sin embargo, esta idea fue descartada casi inmediatamente por la diferencia en la representación de los problemas de optimización tradicional y optimización de consultas *Skyline*. Además de esto, resultaba interesante realizar estudios comparativos entre *GEQO* y el algoritmo evolutivo propuesto en este trabajo que se hubiesen visto afectados con la alteración de la implementación original.

Se propuso entonces implementar el algoritmo evolutivo de manera separada a *GEQO*. Para ello se decidió incluir *ePeaqock*, un tercer componente de búsqueda del planificador además de *GEQO* y *DYQO*. Aunque se tomó esta decisión de diseño, la implementación de *GEQO* representó una base para *ePeaqock* y se trató en lo posible de respetar las estructuras de representación ya implementadas.

4.2.7. Creación del plan de evaluación

Al finalizar la primera fase de optimización, se transforma el camino de evaluación en el plan de evaluación que contiene la información relevante para el evaluador. Cada operador físico es transformado en en un nodo correspondiente para ser evaluado.

La función `create_skyline_plan` que creaba el nodo *Skyline* como *top-level* en la versión anterior de *PeaQock*, fue modificada para que permita crear el nodo *Skyline* de evaluación a partir de un nodo `SFSPath` o `BNLPath`.

Se implementaron las funciones `create_bnl_plan` y `create_sfs_plan` para construir el plan de evaluación correspondiente utilizando `create_skyline_plan`, sea *SFS* o *BNL*. En caso de construirse un plan *SFS*, si se índica en los *pathkeys* del `SFSPath`, se crea un nodo *Sort* que ordene la relación de entrada al *Skyline* de acuerdo a sus dimensiones. Luego de este nodo, es colocado el nodo del *SFS*.

Otras funciones más especificas fueron implementadas para procesar el *Skyline* en esta segunda fase de optimización, para mayor detalle ver el Apéndice A.

4.2.8. *Explain* de la consulta

En *PostgreSQL* una consulta puede ser ejecutada colocando al principio la cláusula *Explain* que permite visualizar el plan de evaluación escogido por el optimizador. Para consultas Skyline se implementó esta funcionalidad, de manera que se puede observar cuál fue el plan escogido por el optimizador para una consulta de este tipo. Además, si el usuario lo desea puede visualizar todos los planes lógicos y su lista de caminos de evaluación considerados por el algoritmo de optimización.

Capítulo 5

Diseño Experimental y Análisis de Resultados

El desempeño de los algoritmos propuestos, *dPeaQock* y *ePeaQock*, para la optimización de consultas *Skyline* fue analizado mediante la realización de experimentos que permitieran comparar los costos de los planes generados, el tiempo de optimización y el tiempo de evaluación, bajo diferentes condiciones.

El conjunto de datos con los que se decidió experimentar está compuesto por datos uniformes y datos reales. Los datos uniformes se crearon mediante el uso de un programa generador de datos existente en [4], ligeramente modificado para este trabajo. Se generaron aproximadamente 60 tablas de prueba, diez tablas por cada tamaño considerado. La cantidad de tuplas varió desde 1000 hasta 20000 tuplas. Antes de realizar los experimentos se realizó el *analyze* [1] de todas las tablas.

Para controlar las selectividades, se generaron en cada tabla tres columnas. En cada columna se tomaba un porcentaje de las filas para colocar valores que hicieran *Join* con otras tablas, dependiendo de la selectividad que se quisiera estimar. Estos valores se escogían aleatoriamente de un conjunto de valores comunes para todas las tablas y se estimó de tal manera que cada tupla hiciera *Join* aproximadamente con tres tuplas de otra tabla. El resto de las filas que no eran escogidas para hacer *Join*, se les colocaba un valor único, asegurando de ésta manera que no hiciera *Join* con otra tabla. Los valores de la selectividad escogidos fueron 0.001, 0.0006 y 0.0002.

El valor único escogido para las filas que no hacen *Join* se produjo de dos maneras. Para las tablas del experimento II, se tenía un número único por tabla que le era colocado a los registros que no se quería que hiciesen *Join*. Para el resto de las tablas de los demás experimentos se colocó un número único por tabla y por fila. Esto, aunque no afectó de manera impactante los resultados generales, originó observaciones interesantes que serán detalladas en la discusión de resultados. La razón para variar el cálculo de la selectividad en el experimento II es su diversidad y amplitud en las variables consideradas.

La selectividad fue generada de esta manera porque se quería experimentar con *Joins* que simularan de forma natural lo que en realidad sucede, que es que nunca se conoce la selectividad exacta sino estimada de éste.

Para los datos reales, se obtuvo una base de datos con las estadísticas del *Baseball* estadounidense de todos los tiempos, encontrada en [44] y está compuesta de veinte tablas cuyas cardinalidades varían desde 1000 a 100000 tuplas. Para las pruebas se escogió un conjunto de seis tablas cuyos tamaños varían en ese mismo orden. Para utilizar estos datos se tuvo que realizar un procesamiento para su inserción en la base de datos porque el formato en el que se encuentran los datos es *comma separated values* [2].

La razón por la que no se probó con datos correlacionados es que experimentos previos muestran que el modelo de costo no se comporta adecuadamente ante la presencia de correlación y anticorrelación [16]. Se

[1] El *analyze* es una herramienta de *PostgreSQL* que permite recolectar estadísticas sobre las tablas para el catálogo. Entre las estadísticas importantes para los experimentos se encuentran el número de valores distintos y la cardinalidad real de las tablas

[2] Archivos donde se tiene una fila por cada registro y cada atributo está separado por comas

prefirió entonces, probar con datos reales que pueden o no tener correlación.

Las consultas con las que se realizaron los experimentos sobre datos uniformes fueron generadas aleatoriamente verificando que en el *Join* producido no existieran productos cartesianos y que además, todas las tablas pudieran hacer *Join* con todas las restantes.

El marco experimental se dispuso de la siguiente manera: cinco experimentos, cuatro con datos uniformes y uno con datos reales. Las mediciones recolectadas para los experimentos son: costo estimado por para cada plan, tiempo de optimización, tiempo de evaluación, el número de comparaciones estimadas y reales para cada nodo *Skyline* y, el número y tipo de nodos *Skyline* en el plan. Adicionalmente se almacenaron otros datos para verificar las observaciones: número de tuplas que entraban al *Skyline* y el número de tuplas que el modelo de costo estima que resultaba de aplicar el *Skyline*.

Cuando en este capítulo, se hace mención a la cantidad de tablas en la que se reparte el *Skyline*, se está indicando el número de tablas que se encuentran involucradas en él. Si se dice que "se tiene una consulta de 6 tablas con un skyline de 6 dimensiones repartidas en 3 tablas", se indica que es una consulta con cinco *Joins* cuya cláusula *SKYLINE OF* tiene seis dimensiones que involucran atributos de solamente tres relaciones base de la consulta.

La configuración de los experimentos se muestra en la Tabla 2. La notación xt-yd-zr se refiere a que se realizaron consultas con x tablas, y dimensiones y el *Skyline* está repartido en z dimensiones. Las variables consideradas durante todos los experimentos son: el número de *Joins*, la cardinalidad de las tablas, la selectividad de los *Joins*, el número de dimensiones y la cantidad de tablas en la que están repartidas las dimensiones. La notación que aparece al lado de algunos términos entre paréntesis será usada más adelante para referirse al experimento o valor de variable que acompaña. La arquitectura de las computadoras donde fueron realizados los experimentos es presentada en la Tabla 3.

Para determinar entre dos algoritmos cuál se comporta mejor, se utilizará la razón de sus tiempos totales de ejecución. El tiempo total de ejecución es la suma del tiempo de optimización, que es el tiempo en que el algoritmo tarda en hallar un plan, y tiempo de evaluación o tiempo que tarda en evaluarse el plan hallado por un algoritmo. Esta razón será calculada por medio de la fórmula 18.

$$R_{te} = \frac{T_{alg1}}{T_{alg2}} \tag{18}$$

Donde T_{alg1} y T_{alg2} son los tiempos totales de ejecución promedio de los algoritmos alg1 y alg2, respectivamente. Si R_{te} es mayor que uno, el algoritmo *alg2*, es considerado mejor o que realiza un mejor desempeño.

La hipótesis general del marco experimental es que es mejor realizar el proceso de optimización que incluye el *Skyline* en contraste con las técnicas tradicionales de optimización. Por lo tanto, en todos los experimentos se hará mención a resultados que muestren o, por el contrario, contradigan esta hipótesis.

En este capítulo se detallan los experimentos realizados. El marco experimental se compone de cinco experimentos. En cada uno se explicará el procedimiento, las hipótesis y variables consideradas y se mostrará la discusión de los resultados obtenidos en él. La mayoría de las figuras aquí presentadas se encuentran en escala

Experimento	**Algoritmos utilizados**	**Variables:valores**	**Configuración de las consultas**
Experimento I (exp1)	*DYQO dPeaQock*	Selectividad: 0.001 (S3), 0.0006 (S2), 0.0002 (S1). Cardinalidad: 1000 (C1), 2000 (C2), 5000 (C3).	4t-4d-4r. 5 consultas para cada grupo. Datos uniformes.
Experimento II (exp2)	*DYQO dPeaQock*	*Joins*: 3,5,7 Tablas involucradas en el *Skyline*: 2,4 y 6 Dimensiones: 4,6 y 8	4t-4d-4r. 4t-4d-2r. 6t-6d-4r. 6t-6d-6r. 6t-6d-2r. 8t-8d-4r. 8t-8d-6r. 8t-8d-2r. 10 consultas de cada combinación con tablas de 1000 tuplas y selectividad de 0.0006. Datos uniformes.
Experimento III (exp3)	*DYQO dPeaQock GEQO ePeaQock*	*Joins*: 3,5,7. Tablas involucradas en el *Skyline*: 6, 7. Dimensiones: 4,6 y 7.	4t-4d-4r. 6t-6d-4r. 6t-6d-6r. 8t-6d-6r (Sólo para la segunda parte). 8t-7d-7r (Sólo para la segunda parte). 5 consultas de cada combinación con tablas de 1000 tuplas y selectividad de 0.0006. Datos uniformes.
Experimento IV (exp4)	*DYQO dPeaQock*	Cardinalidad: 10000 (C4), 15000(C5), 20000 (C6)	4t-3d-3r. 5 consultas con selectividad de 0.0002. Datos uniformes.
Experimento V (exp5)	*DYQO dPeaQock*		2 consultas donde los *Joins* fueran por atributos clave. 2 consultas donde los *Joins* fueran por atributos no clave. Datos reales.

Tabla 2: Experimentos y arquitectura de la computadora donde se realizaron.

Experimento	**Arquitectura de la plataforma de prueba**
Experimento I	Procesador Intel(R) Pentium(R) 4 CPU 3.0GHz, 1024Kb de caché y 1Gb de memoria RAM.
Experimento II	Procesador Intel(R) Pentium(R) 4 CPU 3.0GHz, 1024Kb de caché y 1Gb de memoria RAM.
Experimento III	Intel(R) Pentium(R) D CPU 3.20GHz, 1024Kb de caché y 1Gb de memoria RAM.
Experimento IV	Intel(R) Pentium(R) D CPU 3.20GHz, 1024Kb de caché y 1Gb de memoria RAM.
Experimento V	Intel(R) Pentium(R) 4 CPU 2.80GHz, 1024Kb de caché y 1Gb de memoria RAM

Tabla 3: Experimentos y arquitectura de la computadora donde se realizaron.

logarítmica y en caso de presentarse un gráfico en escala real se mencionará apropiadamente.

5.1. Estudio experimental de los algoritmos

En esta sección se plantea estudiar el comportamiento de los algoritmos propuestos, usando como suposición general que *dPeaQock* y *ePeaQock* generan planes para consultas *Skyline* que son mejores a los generados por *DYQO* y *GEQO*. Se presentan cinco experimentos que intentan estudiar cómo influyen la variación de algunos parámetros y la distribución de los datos en la estimación realizada por el proceso de optimización implementado en este trabajo. Además se realizarán algunas observaciones acerca del comportamiento del modelo de costo respecto al tiempo de evaluación, sin entrar en mucho detalle porque este tema será tratado más adelante en este capítulo.

5.1.1. Experimento I

El objetivo de este experimento es observar cómo es el comportamiento conjunto de los algoritmos *DYQO* y *dPeaQock* respecto a la variación de la selectividad y la cardinalidad de las tablas.

En este experimento se manejaron las siguientes hipótesis:

H1 A medida que aumenta la cardinalidad, mejora el desempeño de *dPeaQock* respecto a *DYQO*.

H2 A medida que aumente el valor de la selectividad del *Join*, mejora el desempeño de *dPeaQock* respecto a *DYQO*.

Obsérvese en la Tabla 4 que el algoritmo *dPeaQock*, en todos los casos allí expuestos, es mejor que *DYQO* en cuanto al tiempo total de ejecución. Destacándose una mejora de 442 veces para las consultas con tablas de cardinalidad C3. Además se puede observar en la Figura 27 que los planes escogidos por *dpeaQock* tienen dos nodos *Skyline*, lo que quiere decir que el optimizador escogió planes donde se distribuye el *Skyline*. Esto nos permite aseverar que para este conjunto de resultados realizar la optimización del *Skyline* es mejor que realizar la optimización tradicional, porque se halla un plan que disminuye el tiempo total en que la consulta se ejecuta y que no se pudo obtener con *DYQO* .

Cardinalidad	**Razón**
C1	2.36445979261
C2	24.2810279955
C3	442.129029317

Tabla 4: Tabla de $T_{te} = T_{DYQO}/T_{dPeaQock}$ respecto a la cardinalidad

Debe mencionarse también que al aumentar la cardinalidad de las tablas aumenta la razón del tiempo de ejecución. El crecimiento es aproximadamente 12 veces entre C1 (1000 tuplas) y C2 (2000 tuplas), y 18 veces entre C2 y C3 (5000 tuplas). Esto es consecuencia de la diferencia en el crecimiento del tiempo total de ejecución para *DYQO* y *dPeaQock*.

En la Figura 16, se observa que al aumentar la cardinalidad de las tablas, el tiempo de optimización y de evaluación para *DYQO* crece más rápido que para *dPeaQock*. La diferencia en la tasa de crecimiento en el tiempo de optimización se debe a que cuando se aumenta la cardinalidad de las tablas, es probable que el número de tuplas que se estima que resultan de hacer solamente el *Join* es mayor ya que no se está filtrando ningún dato, por lo que el cálculo del costo del *Skyline* al final tarda más tiempo al tener que sumar más términos. Lo dicho anteriormente indica que *dPeaQock* funciona mejor respecto a *DYQO* a medida que las tablas consideradas tienen más tuplas, lo que nos permite confirmar **H1** para este conjunto de experimentos.

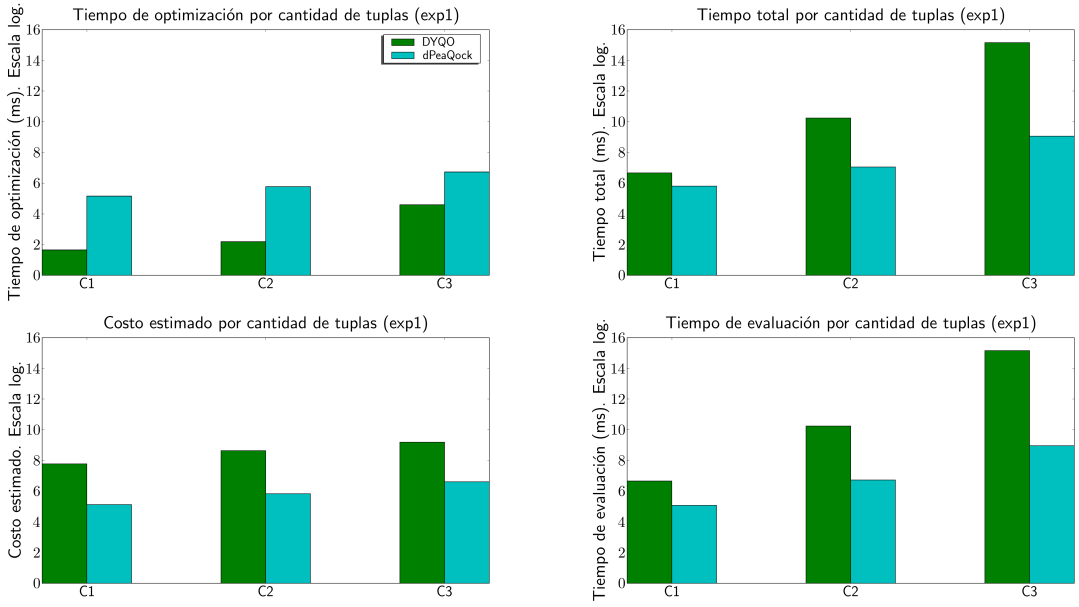


Figura 16: Comportamiento para 1000, 2000 y 5000 tuplas

El mismo análisis puede ser realizado cuando se varía la selectividad. Nótese en la Tabla 5 que la razón del tiempo de ejecución aumenta de S1 a S2 y de S2 a S3. Se debe recordar que de S1 a S2 y de S2 a S3 el valor de la selectividad aumenta. A diferencia de los resultados variando la cardinalidad, observamos que la tasa de crecimiento de la razón no es monótona, es decir, de S1 a S2 *dPeaQock* mejora el tiempo total un poco más del 100 % respecto a *DYQO*, mientras que de S2 a S3 mejora aproximadamente en un 10 %.

El aumento de estas razones se debe a la cantidad de tuplas que son filtradas con la distribución del *Skyline* en el árbol, ya que a un mayor valor en la selectividad, mayor es la cantidad de tuplas que son retornadas y por lo tanto los planes con el *Skyline* distribuido resultaran mejores.

Se observa que al aumentar la selectividad el algoritmo *dPeaQock* aumenta su ventaja en tiempo total de evaluación respecto a *DYQO*.

En la Figura 17, se observa que la selectividad no afecta significativamente los tiempos de optimización o evaluación, lo que quiere decir que el modelo de costo es más suceptible a variaciones en la cardinalidad que

Selectividad	Razón
S1	165.74886394
S2	432.974576009
S3	475.413161761

Tabla 5: Tabla de T_{te} respecto a la selectividad y la cardinalidad

en la selectividad. Con los resultados observados sobre la selectividad se puede decir que para este conjunto de resultados se cumple **H2**.

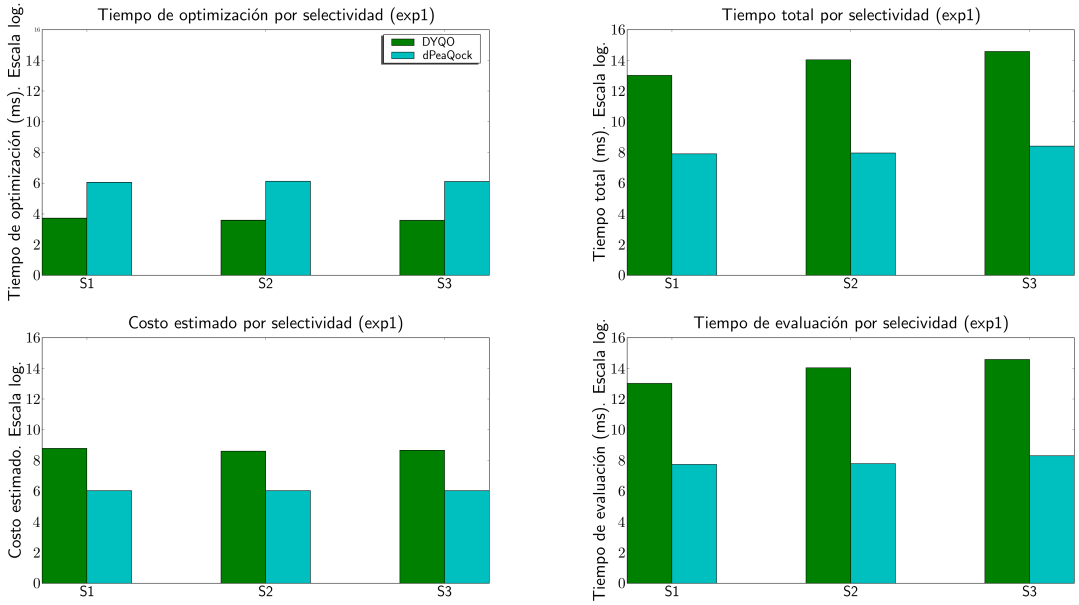


Figura 17: Comportamiento para 0.0002, 0.0006 y 0.001 selectividades

Como comentario final de este experimento, vale la pena destacar que el comportamiento de la estimación de costo y los tiempos de evaluación presentan el mismo patron tanto cuando se varían las selectividades como cuando se varían las cardinalidades. Esto es indicador de que el modelo de costo está estimando de manera acertada el tiempo de evaluación de los planes.

5.1.2. Experimento II

El objetivo de este experimento es observar cómo se ve afectado el tiempo de ejecución y la escogencia del plan al ampliar el espacio de búsqueda en dos sentidos: aumentando los *Joins* conjuntamente con las dimensiones o la repartición de las dimensiones en las tablas.

En este experimento se manejaron las siguientes hipótesis:

H1 A medida que aumenta la cantidad de *Joins*, mejora el desempeño de *dPeaQock* respecto a *DYQO*.

H2 A medida que aumenta la cantidad de tablas involucradas en el *Skyline*, mejora el desempeño de *dPeaQock* respecto a *DYQO*.

En la Tabla 6 se puede observar que siempre *dPeaQock* resulta mejor que *DYQO* para estos casos de prueba, dado que el coeficiente o razón T_{te} es mayor que uno en todos los casos. Se observa también que este coeficiente no crece monótonamente respecto al crecimiento del número de *Joins* y dimensiones. Aún así se observa al final un incremento en la razón para el caso de 7 *Joins* y 8 dimensiones. No se puede decir que se acepte *H1* porque en el segundo caso mostrado en el Tabla 6, se observa una disminución de la efectividad de *dPeaQock* respecto a *DYQO*.

Joins	**Razón**
3j-4d	1.78371380769
5j-6d	1.37998726834
7j-8d	3.18633978257

Tabla 6: Razón de $T_{te} = T_{dPeaQock}/T_{DYQO}$ para la variación del número de joins y dimensiones

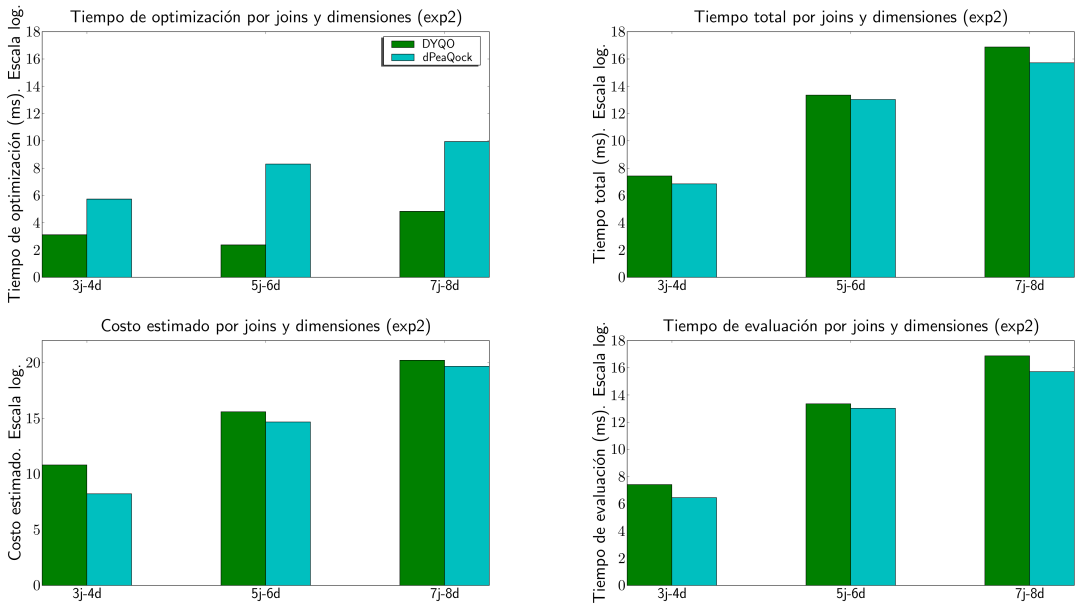


Figura 18: Comportamiento para 3, 5 y 7 *Joins* y 4, 6 y 8 dimensiones

Por otro lado, en la Tabla 7 donde se observa la razón T_{te} variando el número de tablas del *Skyline*, se puede observar que *dPeaQock* siempre es mejor que *DYQO* ya que todos los coeficientes son mayores a uno. Otra observación importante es que a medida que se reparte el *Skyline* en más tablas la mejoría de *dPeaQock* respecto a *DYQO* aumenta notablemente de 140 % a 665 % aproximadamente. Esto permite aseverar que para este conjunto de experimentos se cumple *H2*.

Tablas	Razón
2t	2.47409991774
4t	3.50352003126
6t	23.3106839371

Tabla 7: Razón de $T_{te} = T_{dPeaQock}/T_{DYQO}$ para la variación del número de tablas involucradas en el *Skyline*

En las mediciones realizadas para este experimento se refleja un hecho interesante. Se esperaba que para *DYQO*, el tiempo de evaluación aumentara a medida que se incrementaban la cantidad de dimensiones y las tablas involucradas en ellas. Se observa que el tiempo de evaluación es creciente respecto al número de dimensiones y *Joins*, pero decreciente respecto al número de tablas involucradas en el *Skyline*. Estos resultados se atribuyen al modo en que se evalúa el *Skyline*.

Para *dPeaQock* se observa el mismo comportamiento pero más acentuado. Esto puede ser atribuído a las mismas razones mencionadas anteriormente y a que cuando se divide en más tablas el *Skyline*, se tienen menos dimensiones por tabla y por ende es probable que al aplicar el *Skyline*, se filtre mayor cantidad de tuplas, lo que se traduce en un mejor desempeño del plan escogido por *dPeaQock*.

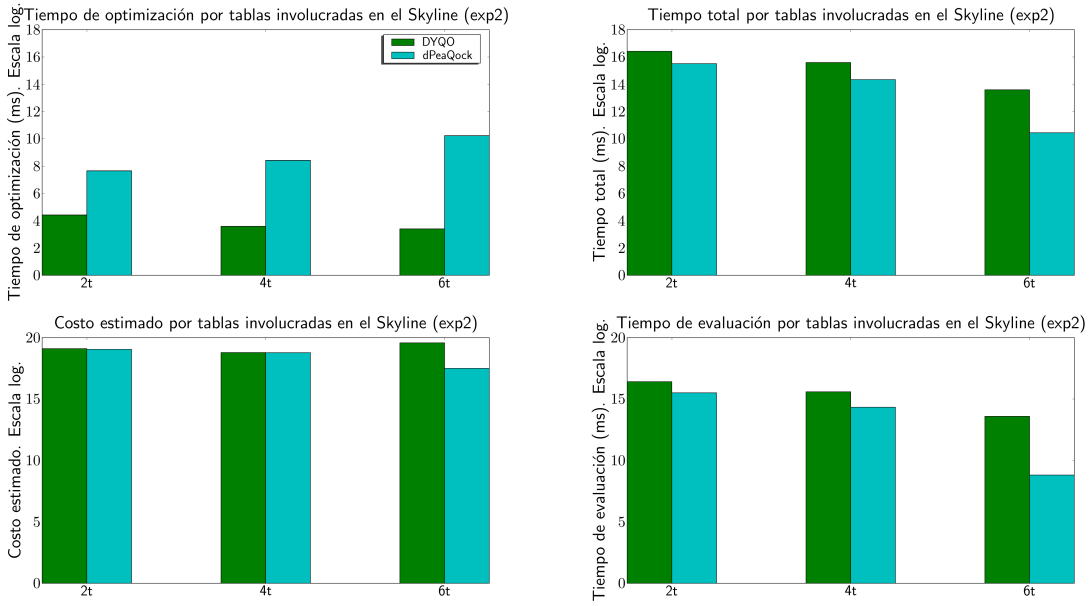


Figura 19: Comportamiento para 2, 4 y 6 tablas involucradas en el *Skyline*

Detallando un poco en la Figuras 18 y 19 se observa un comportamiento coherente. Los tiempos de optimización de *dPeaQock* aumentan con mayor velocidad que los de *DYQO*. Esto era de esperarse porque el número de planes depende directamente del número de *Joins* y el número de tablas involucradas en el *Skyline*.

Por último, nótese en ambas figuras, que el comportamiento del costo estimado y el tiempo de ejecución es similar. Esto indica, como se ha dicho anteriormente, que el optimizador es capaz de estimar tiempos coherentes para los planes generados por *dPeaQock*.

5.1.3. Experimento III

En este experimento se intenta comparar el desempeño de los cuatro algoritmos estudiados. El objetivo es observar el comportamiento de los costos y los tiempos para poder determinar si, en las pruebas realizadas, siempre resulta mejor utilizar los algoritmos propuestos en este trabajo antes que los algoritmos tradicionales de optimización.

La hipótesis manejada en este experimento es la planteada en la introducción de este capítulo:

H1 Resulta mejor realizar una optimización especializada cuando se procesan consultas *Skyline*.

Obsérvese en los Tablas 8 y 9 que las razones del tiempo total de ejecución siempre son mayor a uno. Esto indica que los algoritmos *ePeaQock* y *dPeaQock* son siempre mejores que *DYQO* y *GEQO*, respectivamente. Nótese que la mejora mínima alcanzada es de un 25 % y puede llegar a ser uns 193 % para este experimento.

Tablas	**Razón 1**	**Razón 2**
4t	2.9314468	2.060584608
6t	1.764349362	1.309546621

Tabla 8: Razones de $T_{te} = T_{DYQO}/T_{dPeaQock}$ (Razón 1) y $T_{te} = T_{GEQO}/T_{ePeaQock}$ (Razón 2) variando la cantidad de tablas involucradas en el *Skyline*

Por otro lado, se observa que los valores de las razones están comprendidos en el intervalo $[1,253; 2{,}931]$ y se comportan de manera similar. Esto puede ser indicativo de que el número de *Joins* y dimensiones, y el número de tablas involucradas contribuyen de igual manera a los resultados obtenidos.

Joins-Dim	**Razón 1**	**Razón 2**
3j-4d	2.5588604	1.3815881734
5j-6d	2.82595315	1.253567638

Tabla 9: Razones de $T_{te} = T_{DYQO}/T_{dPeaQock}$ (Razón 1) y $T_{te} = T_{GEQO}/T_{ePeaQock}$ (Razón 2) variando el número de joins y dimensiones

Como se observa en las Figuras 20 y 22, de manera coherente con el resto de los resultados mostrados anteriormente, el tiempo de optimización de *dPeaQock* y *ePeaQock* incrementa respecto a los tiempos de optimización de *DYQO* y *GEQO* debido a que el espacio de búsqueda es mayor. Sin embargo, el tiempo consumido en la optimización es compensado con una disminución en el tiempo total de ejecución

El patrón que se observa en estas dos figuras es que el promedio del tiempo de ejecución total para *dPeaQock* es menor que el de *DYQO*. De manera similar es el comportamiento del algoritmo *ePeaQock* en comparación con el algoritmo *GEQO*.

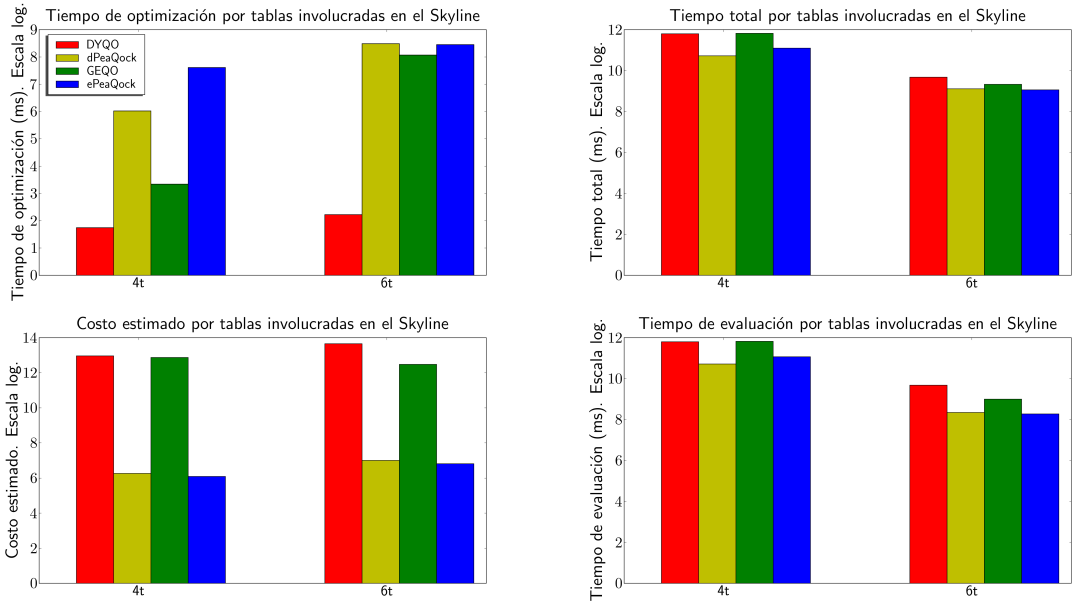


Figura 20: Comportamiento para 4 y 6 tablas involucradas en el *Skyline*

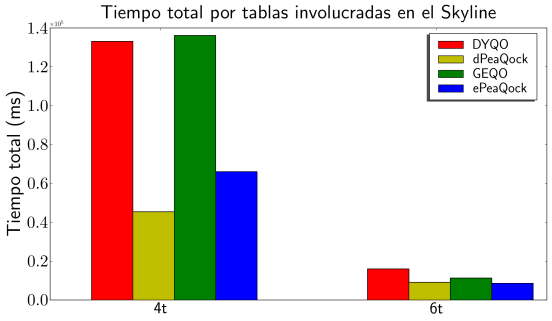


Figura 21: Comportamiento para 4 y 6 tablas involucradas en el *Skyline*. Escala real

Para reiterar las observaciones de este experimento se puede observar la Figura 21, que muestra la **escala real** de los tiempos totales de ejecución para el mismo caso que la Figura 20, el mismo comportamiento. Haciendo un estudio por separado de la forma en que afectan los *Joins*, las dimensiones y las tablas involucradas en el *Skyline* se obtienen resultados similares. Sin embargo, la diferencia en el comportamiento parece ser más sensible a la variación de la cantidad de *Joins*, lo que es esperado porque éste valor es el que más contribuye al aumento del espacio de búsqueda. Esto indica que los algoritmos propuestos en este trabajo, para este conjunto de pruebas, obtienen siempre mejores resultados. Esto permite afirmar que para este conjunto de experimentos *H1* se corrobora.

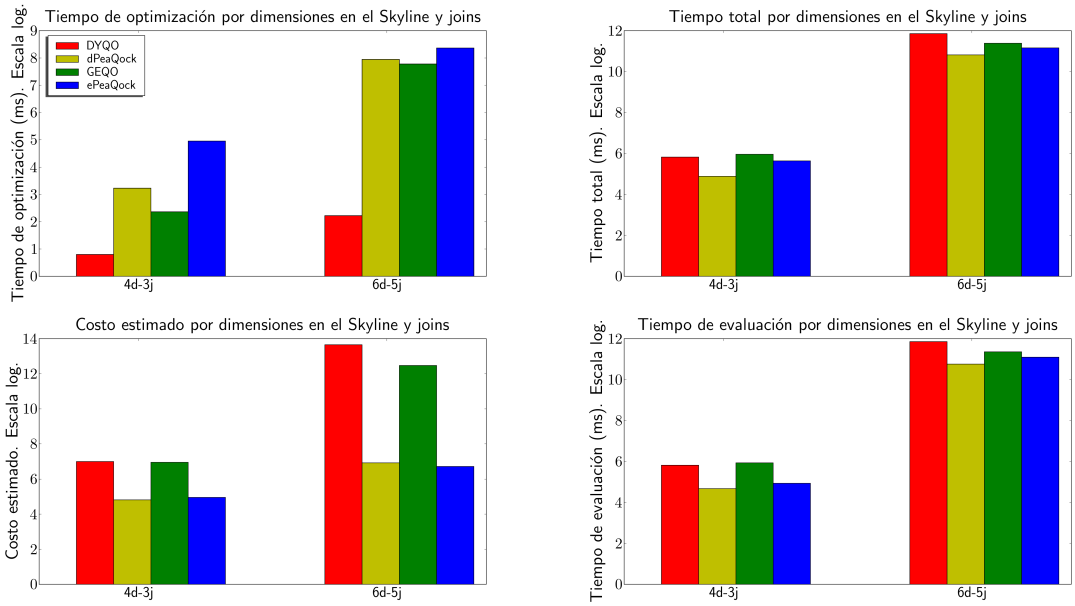


Figura 22: Comportamiento para 4 y 6 dimensiones y 3 y 5 *Joins*

Respecto a la correspondencia entre el costo estimado y el tiempo de evaluación real, se puede observar nuevamente en las Figuras 20 y 22, que las estimaciones realizadas no son acertadas respecto al tiempo de evaluación de la consulta. La estimación realizada en los casos de *dPeaQok* y *ePeaQok* son muy bajas respecto al tiempo real, sin embargo, si el costo estimado crece, el tiempo de evaluación también aumentará. Con esto se quiere decir que aún cuando subestima notablemente, el modelo de costo es capaz de escoger planes que disminuye el tiempo de evaluación de las consultas.

dPeaQock y ePeaQock

Con el objeto de seguir comparando los algoritmos, se realizó una prueba con las corridas generadas en este experimento adicionando además los resultados para la corrida de cinco consultas con seis *Joins* y dimensiones repartidas en siete tablas. El objetivo de esta prueba es comparar el comportamiento de los dos algoritmos propuestos y verificar si en algún punto de crecimiento del espacio de búsqueda en realidad es útil aplicar el algoritmo evolutivo.

En base a lo que se fundamentó en el capítulo 2, se espera que al crecer el espacio de búsqueda, el algoritmo *dPeaQock* decremente su utilidad. En la gráfica 23 se observa precisamente el comportamiento esperado. Ya cuando se tienen siete *Joins* y dimensiones repartidas en 7 tablas, se observa que el desempeño del evolutivo es mejor en cuanto a tiempo de optimización.

Por otro lado es importante resaltar que el tiempo de optimización promedio del algoritmo *ePeaQock* crece de manera más lenta que el algoritmo *dPeaqock*. Esto es un resultado coherente porque se supone que *dPeaQock* por ser una algoritmo basado en programación dinámica es más sensible al crecimiento del espacio

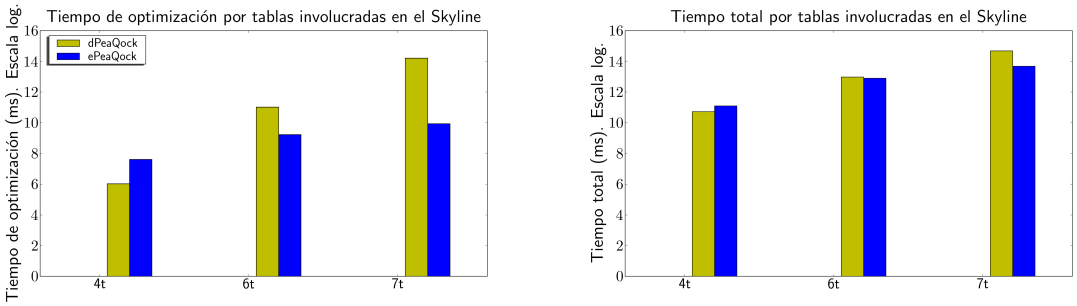


Figura 23: Comportamiento para 4, 6 y 7 tablas involucradas en el *Skyline*

de búsqueda. Para observar un mayor estudio sobre el algoritmo evolutivo, obsérvese el Apéndice H.

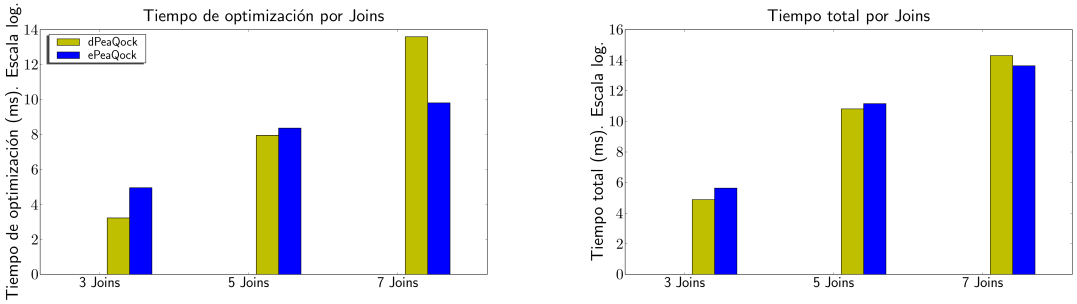


Figura 24: Comportamiento para 3, 5 y 7 *Joins*

5.1.4. Experimento IV

Este estudio se realizó con el objetivo de validar los resultados mostrados en los experimentos anteriores. La hipótesis manejada durante esta prueba es:

H1 Al cambiar la configuración a pocos *Joins* y uso de tablas grandes, el comportamiento de *dPeaQock* respecto a *DYQO* es similar al de los experimentos anteriores.

En un principio se intentó ejecutar las consultas *Skyline* usando *DYQO*. Sin embargo, se colocó un tiempo límite de dos semanas para este experimento, tiempo en el cual no se obtuvieron resultados porque ninguna consulta terminó de ejecutar. Por lo tanto, para realizar el estudio con *DYQO*, se trabajó con las mismas consultas sin la cláusula *SKYLINE OF* y así se obtuvo una cota inferior del tiempo que tardaría en ejecutarse la consulta original. Éstas son mostradas en la Figura 25 como *DYQO-joins*.

Obsérvese en la Tabla 10, que la razón para todos los casos es bastante mayor que uno. En el peor de los casos *dPeaQock* es mejor que *DYQO* en un 621 %. Esto muestra coherencia con los experimentos anteriores

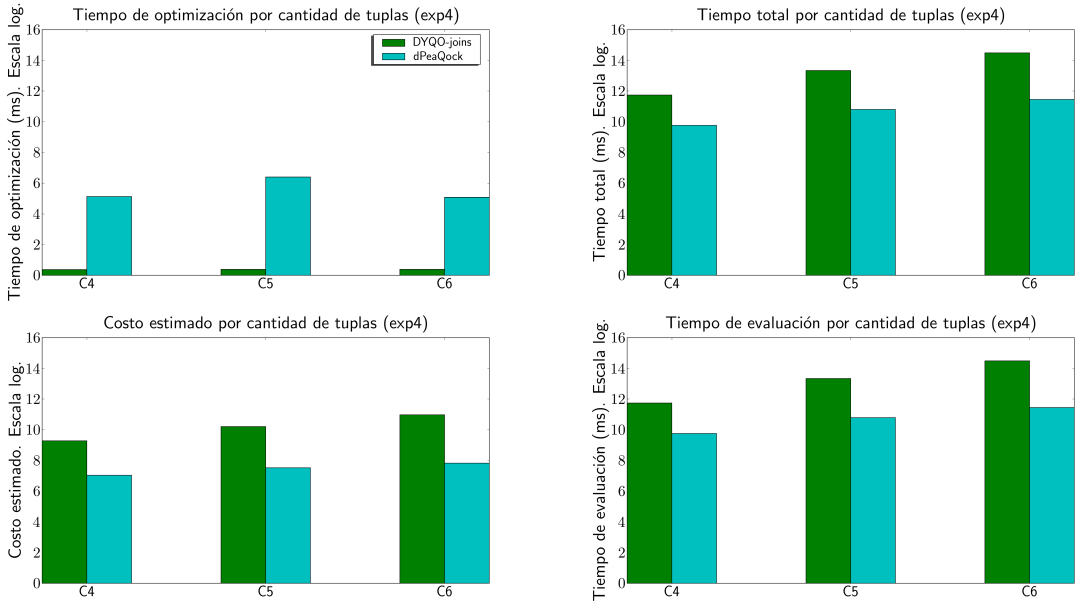


Figura 25: Comportamiento para 10000, 15000 y 20000 tuplas

donde los algoritmos propuestos en este trabajo resultan mejores que los tradicionales. Por otro lado se muestran resultados similares a los del experimento I, donde el comportamiento al aumentar las cardinalidades, el algoritmo *dPeaQock* tiene un mejor desempeñno. Hay que recordar que las consultas evaluadas con *DYQO* estaban compuestas solamente por los *Joins* de la consultas originales, por lo que se deduce que los valores mostrados pudieran ser aún mayores.

Cardinalidad	**Razón**
C4 (10000 tuplas)	7.21110919566
C5 (15000 tuplas)	12.6362493978
C6 (20000 tuplas)	20.8648385542

Tabla 10: Razones de $T_{te} = T_{DYQO}/T_{dPeaQock}$ variando la cardinalidad en 10000, 15000 y 20000 tuplas

Por otro lado el comportamiento observado en la Figura 25 es similar con respecto a las observaciones realizadas anteriormente: cuando se utiliza *dPeaQock* existe un incremento en el tiempo de optimización y se produce una disminución del tiempo de evaluación y del tiempo total de ejecución. Esto permite corroborar *H1* para este conjunto de experimentos.

Es importante mencionar que en la misma figura, se puede observar que el comportemiento del tiempo de optimización aumente de C4 a C5 pero disminuye de C5 a C6. Se espera que esto suceda por la implementación del modelo de costo realizada. Al calcularse el costo con una matriz escalada, precisamente lo que se quiere es acotar a un máximo de 9999 términos la sumatoria que se realiza para tal estimación. Debido a esto, si un *Skyline* es situado sobre una relación base de 15000 tuplas, entonces se tendrían que realizar cinco mil iteraciones para estimar el costo del operador. En cambio para el cálculo para 10000 y 20000 tuplas no se

necesitaría ninguna iteración para ello.

5.1.5. Experimento V

Este experimento se realizó con el objetivo de comparar los resultados obtenidos con datos uniformes, con pruebas realizadas sobre un conjunto de datos reales. La hipótesis manejada en este experimento son:

H1 El tiempo total de ejecución usando el algortimo *dPeaQock* es menor que el tiempo total de ejecución utilizando *DYQO*.

Se tomaron dos consultas donde los *Joins* eran una clave foránea y dos consultas donde el atributo por el cual se hace *Join* no es clave. Debido a que con *DYQO* ninguna ejecución de las consultas 1 y 2 había terminado en un tiempo prudencial, se realizaron las pruebas para *DYQO* con las mismas consultas 1 y 2 sin la cláusula *SKYLINE OF*. Éstas son mostradas en la Figura 26 como *DYQO-joins*.

Las consultas utilizadas se listan a continuación:

- **Consulta 1:**Obtener el equipo que cuente con los bateadores con buen rendimiento y mala remuneración en los últimos años.
- **Consulta 2:** Obtener el equipo que cuente con los *pitchers* con buen rendimiento y mala remuneración en los últimos años.
- **Consulta 3:** Obtener los *pitchers* que se encuentren en el *Hall of Fame* con un buen rendimiento como lanzador y deficiente como bateador.
- **Consulta 4:** Obtener los bateadores pertenecientes al *Hall of Fame* con buen rendimiento como bateador y mala remuneración en los últimos años.

Nótese que las consultas 1 y 2, están orientadas a realizar la búsqueda por equipos por lo que se espera que la cantidad de valores distintos del atributo del *Join* sea baja. Se espera lo contrario para las consultas 3 y 4. En todas las consultas se intentó utilizar valores que pudieran estar anticorrelacionados para probar como se adapta el modelo de costo a esta situación.

Para las consultas 1 y 2, se observa un comportamiento coherente respecto a los resultados de experimentos con datos uniformes. Las razones observadas en la Tabla 11 permiten deducir que *dPeaQock* es mucho mejor que *DYQO* para estos casos. Más aún, obsérvese que las consultas 1 y 2 son muy similares pero en la consulta 2 se cambian los bateadores por pitchers. La tabla de bateadores cuenta con aproximadamente 80000 tuplas, mientras que la de *pitchers* comprende un tercio aproximadamente de ese tamaño. Con esto se observa que al aumentar la cardinalidad, aumenta la mejora de *dPeaQock* respecto a *DYQO*, lo que sigue siendo coherente con los experimentos I y IV. Recuérdese también que las consultas fueron realizadas tomando en cuenta solamente los *Joins* de las consultas originales, por lo que se espera que en la realidad, las razones mostradas para las consultas 1 y 2 sean mayores.

Consulta	Radio
Consulta1	2455.42478843
Consulta2	1707.88488633
Consulta3	0.0157903735993
Consulta4	0.00744743743172

Tabla 11: Tabla de la razón $T_{te} = T_{DYQO}/T_{dPeaQock}$ para las consultas soobre datos reales

En la Figura 26 y en la Tabla 11 se observa que para las consultas 3 y 4 el menor tiempo total de ejecución se obtuvo con el algoritmo *DYQO*, siendo la razón considerablemente menor que uno. Más aún el algoritmo *DYQO* para estas consultas es aproximadamente 62 y 134 veces mejor, respectivamente.

Se pronosticaba que la estimación de costo y la escogencia de los planes para las consultas 3 y 4 no sería acertada, pero no se esperaba que los resultados mostraran que los planes determinados con *DYQO* son mejores para evaluar estas consultas. Estos resultados motivaron una discusión detallada del modelo de costo observada más adelante en este capítulo. En base a estas observaciones, no se puede corroborar *H1* para este conjunto de pruebas.

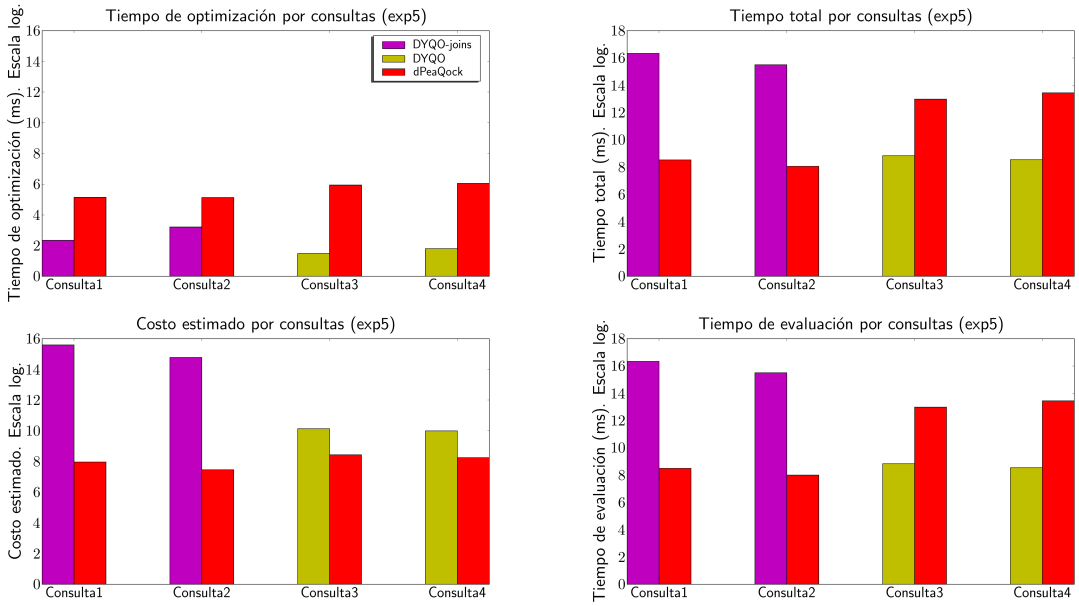


Figura 26: Comportamiento para las consultas 1, 2, 3 y 4

En las Figura 26 se puede observar con más detalle lo dicho anteriormente. Para las dos primeras consultas los gráficos tienen el mismo comportamiento que en experimentos anteriores. Para las consultas 3 y 4, se observa cómo el modelo de costo se comporta de manera deficiente al estimar que planes que son buenos, en realidad no lo son, lo que se traduce en lo mencionado anteriormente: aumento del tiempo total de ejecución para estas consultas.

5.2. Estudio del modelo de costo y planes de evaluación

Se realizó un estudio a partir de los resultados obtenidos con el objetivo de analizar el modelo de costo implementado y la calidad y forma de los planes de evaluación escogidos en las consultas *Skyline*.

Con respecto a la calidad y forma de los planes de evaluación, se tomó en cuenta el número y los tipos de *Skylines*. En el análisis de estos planes conseguidos por los algoritmos *dPeaQock* y *ePeaQock* se puede observar, en la Figura 27, que la cantidad de operadores *Skyline* varía según el experimento realizado.

Para el experimento I se colocan en general dos *Skylines* en cada plan. Cuando se tienen las consultas sobre las tablas de cardinalidades C1 y C2 se coloca en todas las veces un operador *BNL* y otro *SFS*. En cambio, para las consultas con cardinalidades C3 son colocados únicamente operadores *BNL*. La causa es el costo de ordenamiento que se debe pagar con *SFS*, ya que a medida que aumentan las cardinalidades es más costoso realizar un ordenamiento y aunque el número de comparaciones para *SFS* es menor que para *BNL*, el modelo de costo supone que es preferible utilizar *BNL*.

Para el experimento III, al igual que para el experimento I, se tienen en promedio dos *Skylines*, con la diferencia en que generalmente predomina el operador *SFS*. Esto sucede porque a medida que incrementa el número de tablas, resulta más útil para los *Joins* este ordenamiento.

Sin embargo, para el experimento II se observa una cantidad de *Skylines* considerablemente mayor, se colocan entre 3 y 7 *Skylines* en el plan. El aumento en la cantidad es consecuencia de la diferencia existente entre las tablas de este experimento y las del experimento I y III. Las tablas presentan una disimilitud en la forma en que se generan los valores de los atributos de *Join*, ya que para las tuplas que no deben hacer *Join* el número de valores distintos es uno. Sin embargo, esto no afecta la cantidad de tuplas destinadas a estar en el *Join*, únicamente influye en las estadísticas generadas por el manejador, ya que siempre asume que los datos tienen una distribución uniforme. La finalidad de realizar estos cambios en las tablas fue estudiar la forma en que esto influye en la distribución del *Skyline* en el plan.

Para la comprensión de estos resultados, se entenderá que una notación $D - xyz$ y $E - xyz$ corresponde a los resultados obtenidos para las corridas con el algoritmo *dPeaQock* (D) y *ePeaQock* (E) de x tablas, y dimensiones en el *Skyline* y repartidas en z tablas.

Observando la Figura 27, se percibe que la cantidad de *Skylines* para $D-444$ es mayor que para $C1-S2$ del experimento I (estas consultas poseen las mismas características). De las misma manera, las cantidades de *Skylines* en un plan para los casos $D-444$, $D-664$ $D-666$ son mayores que las del experimento III. Esto trajo como consecuencia, observando el Tabla 12, un mayor tiempo de optimización porque se colocaron más *Skylines* en los planes, aunque es compensado con una mejora notable en el tiempo de evaluación.

Para los experimentos IV y V, los cuales utilizan las tablas de mayor cardinalidad, se observa una mayor cantidad de *Skylines* comparada con las de los experimentos I y III. Cabe destacar que debido a que estas consultas poseen entre dos y tres dimensiones del *Skyline*, en la mayoría de las veces se distribuye lo más posible el *Skyline* en el plan, ya que son colocados dos ó tres sobre las tablas base más el que es dejado en la raíz. Esto apoya la teoría de que el tiempo de evaluación disminuye al colocar más operadores *Skyline* en el

Experimento	Lote	Tiempo optimización	Tiempo evaluación
Experimento II	D-444	363.893	39.876
Experimento III	D-444	25.198	107.102
Experimento II	D-664	1937.211	6245.560
Experimento III	D-664	798.063	89851.435
Experimento II	D-666	9285.834	119.744
Experimento III	D-666	4844.863	4209.587

Tabla 12: Tabla de tiempos promedios para los experimentos II y III

plan aún cuando el cálculo del *Skyline* resulte costoso.

En promedio, se tienen en el plan mas operadores *BNL* que *SFS*, además los *BNL* siguen mantenienendose en las tablas bases, dejando el *SFS* para ser evaluado de último en el plan.

De este análisis se concluye que:

- Al menos se colocan dos operadores *Skyline* en el plan escogido.
- Si las tablas involucradas en el *Skyline* contienen más de 5000 tuplas entonces se coloca mínimo un *BNL* sobre alguna de esas tablas.
- Las estadísticas sobre los datos, aportadas por el manejador, afectan de forma directa la cantidad de *Skylines* en el plan.

Por otro lado, para realizar un mayor análisis del modelo de costo, se estudiaron los experimentos II y V y se observaron las comparaciones estimadas y reales para cada *Skyline* en los planes obtenidos. Las figuras analizadas indican el número estimado y real de comparaciones de los *Skylines* para cada consulta. Tal que *Skyline* 1 y *Skyline* n corresponden al primero y al n-ésimo *Skyline* evaluado en la consulta. Las lineas discontinuas representan las comparaciones estimadas y las continuas a las reales.

En cuanto al experimento II, se analizó el caso que obtuvo una mayor cantidad de *Skylines* para determinar la calidad de la estimación. En la Figura se observa que aunque las curvas estimadas y reales son distintas, éstas no se encuentran demasiado alejadas y presentan un patrón similar, lo que influye en una buena elección del algoritmo en el plan escogido.

Para el experimento V se compararon los resultados de las Figura 29 correspondientes al set de corridas para la consulta 1 y 4, respectivamente. Como los planes escogidos para la consulta 1 fueron mejores que los de la consulta 4, se realizó este estudio para determinar la causa de esta escogencia inadecuada de los planes para la consulta 4.

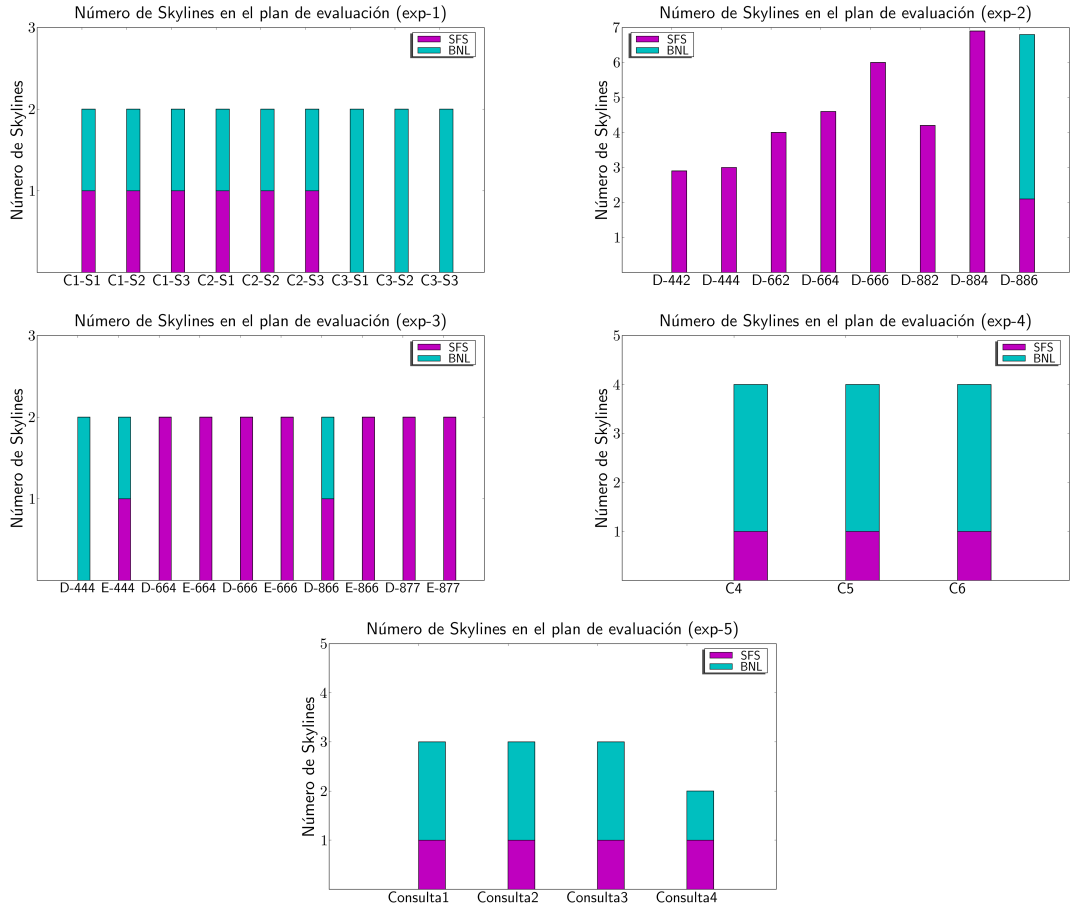


Figura 27: Número promedio de operadores *Skyline* en el plan de evaluación por cada experimento realizado

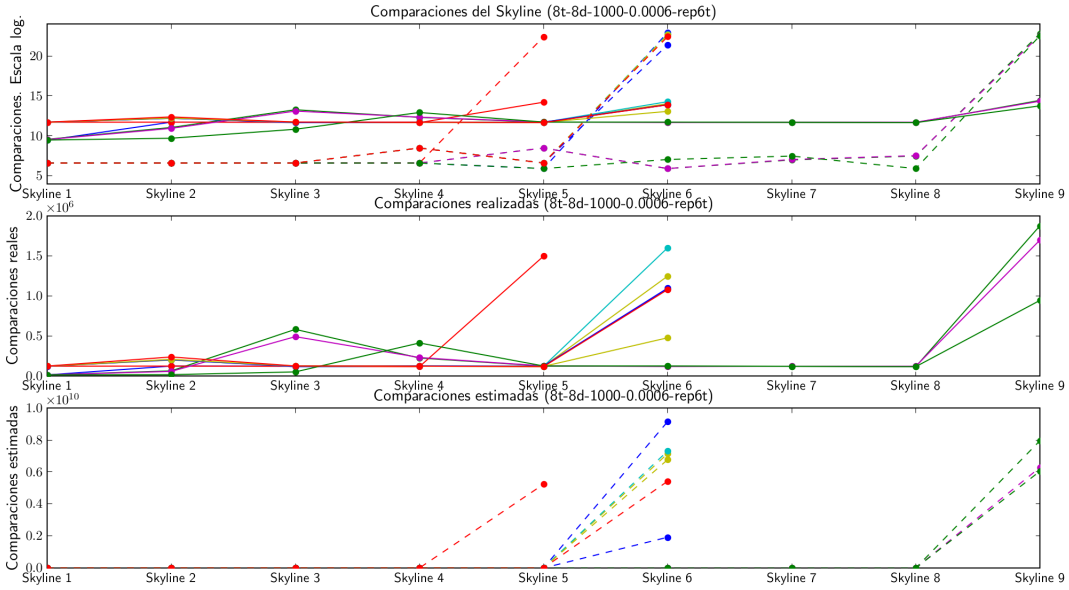


Figura 28: Comparaciones estimadas y realizadas para las consultas del lote D-886 del experimento II

En las comparaciones reales y estimadas para la consulta 1, se observa en el primer *Skyline* una mayor aproximación a la curva de comparaciones reales que para la consulta 4. Está es la causa fundamental de una escogencia errada, ya que el primer *Skyline* es colocado, en ambos casos, sobre una tabla y éste posee además el *diff* por el atributo de *Join*. Como la consulta 4 comprende *Joins* por valores únicos, al realizar el *Skyline* sobre una tabla con un *diff* este devolverá la misma cantidad de tuplas. Este resultado refuerza la necesidad de implementar un modelo de costo que permita realizar una mejor elección en estos casos. Se espera que una implementación en PostgreSQL del modelo de costo propuesto en el capítulo anterior pueda resolver este problema.

5.3. Resumen de los resultados

En general, para las pruebas realizadas se obtuvieron buenos resultados al aplicar los algoritmos *dPeaQock* y *ePeaQock* en relación con *DYQO* y *GEQO*. El único caso en el que se observó que *DYQO* mejoraba respecto a *dPeaQock* fueron las consultas 3 y 4 del experimento V. Las fallas del diseño a las que se atribuye estos resultados también fueron explicadas.

En todo el estudio realizado, se obtuvo un mejora mínima en el tiempo total de ejecución de 1.76 y una máxima de 2455.42 veces de *dPeaQock* respecto *DYQO*. De la misma manera, para *ePeaQock* se tuvo una mejora mínima de 1.25 y máxima de 2.06 con respecto al *GEQO*.

Además se observó, que para cuando se tienen siete *Joins* y siete tablas involucradas en el *Skyline* el espacio de búqueda para *dPeaQock* es demasiado grande por lo que resulta mejor utilizar *ePeaQock*.

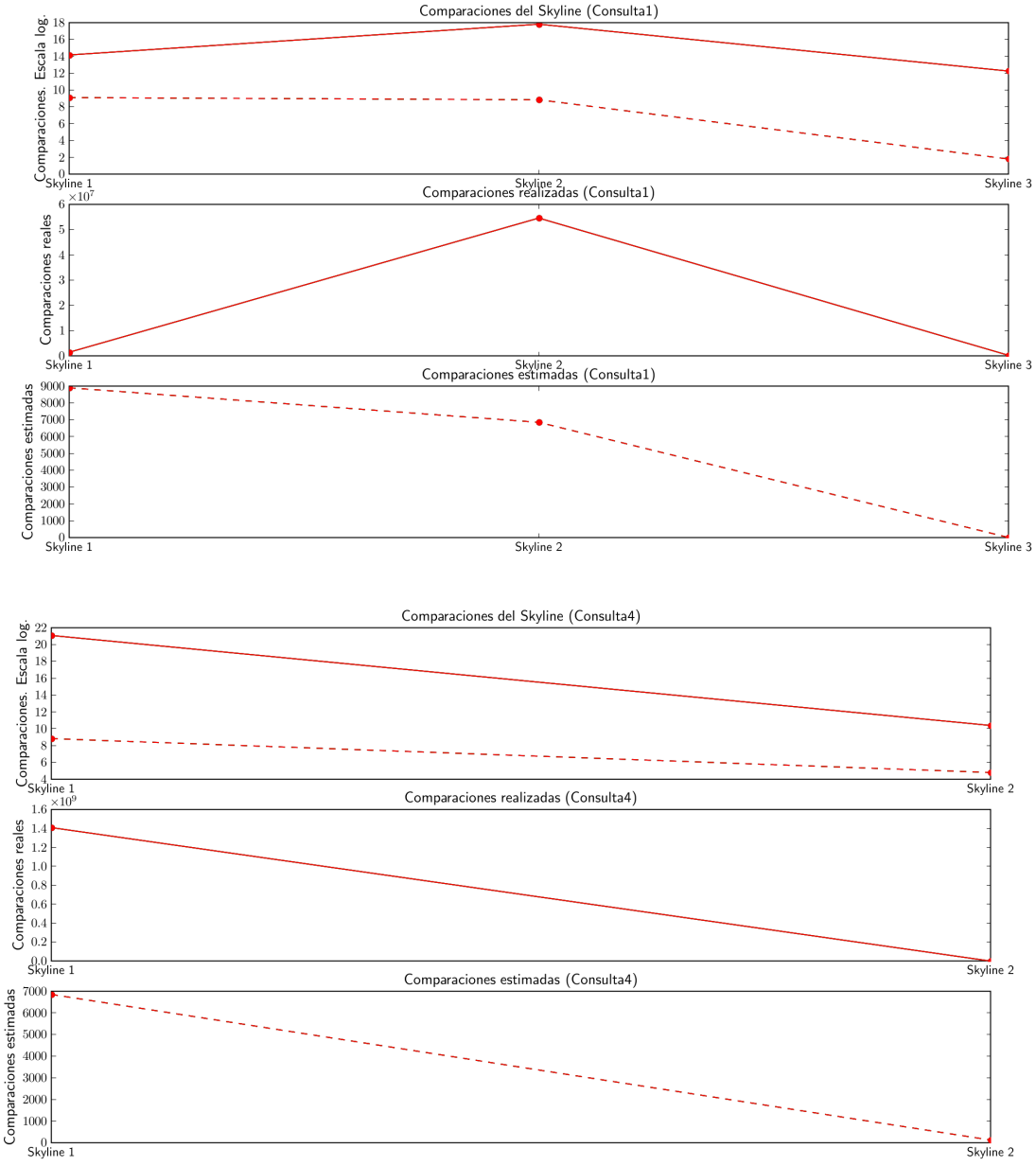


Figura 29: Comparaciones estimadas y realizadas para la consulta 1 y 4 del experimento V

De acuerdo al número de *Skylines* en el plan, en promedio se tiene un mínimo de dos y un máximo de siete *Skylines*. Para el experimento de datos reales se observan entre dos y tres *Skylines* y, según los planes escogidos, todos son colocados en las relaciones base. La escogencia de planes que siempre distribuyen el *Skyline* otorga validez a la teoría de que la optimización especializada del *Skyline*.

Además se observó en el experimento III, que a mayor cantidad de *Joins* resulta mejor optimizar con el algoritmo *ePeaQock* porque se hace demasiado amplio el espacio de búsqueda para *dPeaQock*. Aunque en la mayoría de las veces *dPeaQock* encuentra un mejor plan, cuando el espacio de búsqueda se hace muy grande, el desempeño de *dPeaQock* empeora.

Otra observación general es que a medida que aumenta la cardinalidad de las tablas involucradas en las consultas, aumenta la utilidad del uso de los algoritmos propuestos. A mayor cardinalidad se incrementa el tiempo de evaluación por lo que resulta mejor filtrar la cantidad de tuplas mediante la distribución del *Skyline* en el plan. Además se estima que para estos casos es mejor utilizar *BNL* que *SFS* para evitar un mayor tiempo de evaluación ordenando las relaciones. Una alta cantidad de valores distintos en los atributos correspondientes a las directivas *diff* del operador *Skyline* ocasiona una estimación de costo inadecuada y por lo tanto el optimizador escoge un plan errado, causando un tiempo evaluación extremadamente mayor.

Capítulo 6

Conclusiones y Recomendaciones

Como resultado de este trabajo de grado, se logró añadir a *PostgreSQL* la funcionalidad que permite optimizar consultas *Skyline*. El componente principal de este proyecto es la extensión del planificador y el modelo de costo de *PostgreSQL*, para que pueda ser considerada la inclusión de operadores *Skyline* durante la primera fase de optimización. De esta manera es posible generar planes donde el *Skyline* puede estar distribuido en el árbol que representa el plan y en consecuencia, la obtención de la respuesta a este tipo de consultas puede ser más rápida.

El modelo de costo fue extendido en base a una propuesta realizada en este trabajo, cuyo fundamento se soporta en la bibliografía relacionada y cuyo objetivo era considerar los criterios con directiva *diff* dentro de la estimación del costo. Debido a limitaciones de *PostgreSQL*, tuvo que ser implementada una propuesta distinta que considerara de manera diferente el criterio con la directiva *diff*. La implementación de este modelo representó un reto, debido a que está compuesto por expresiones costosas de evaluar y difíciles de aproximar eficientemente.

El algoritmo de optimización tradicional de *PostgreSQL*, referido en este trabajo como *DYQO*, fue extendido con la creación de las estructuras, funciones y procedimientos necesarios para poder manipular el operador *Skyline* como un nodo *down-level*. La implementación realizada se apoyó en las reglas algebraicas definidas para *Skyline* y *Joins*, de manera de asegurar que los planes generados por el algoritmo propuesto fueran semánticamente correctos. A esta extensión se le denominó *Dynamic PeaQock Optimization* o *dPeaQock*.

Adicionalmente, se planteó la implementación de un algoritmo evolutivo como alternativa para la optimización de consultas *Skyline* ya que, hasta el momento, no se conocía de soluciones similares que además permitieran manejar espacios de búsqueda de gran tamaño. Esta propuesta surge de la implementación, prueba y estudio previo de un prototipo del algoritmo. En base a los resultados obtenidos, se decidió incorporarlo al planificador de *PostgreSQL*. De esta manera, surgió un segundo procedimiento para optimizar las consultas *Skyline* denominado *Evolutionary PeaQock Optimization* o *ePeaQock*.

Asimismo, debido a la escasa información de *PostgreSQL*, se documentó la arquitectura y funcionamiento de su módulo optimizador, como contribución para aquellos desarrolladores que deseen extenderlo. La documentación realizada comprende los diagramas de paquetes y secuencia del optimizador de *PostgreSQL* extendido y un manual que explica los pasos para lograr esta extensión.

En cuanto al estudio experimental realizado, en general se obtuvieron resultados muy satisfactorios para *dPeaQock* y *ePeaQock*. Se encontraron relaciones interesantes entre la variación de las características consideradas en el estudio experimental y el desempeño de los algoritmos. Ejemplo de ésto es que los resultados mostraron que al aumentar la cardinalidad de las tablas de una consulta *Skyline*, los algoritmos propuestos mejoraron su desempeño, especialmente *dPeaQock*. De igual manera es interesante resaltar que la implementación del algoritmo evolutivo resultó de gran utilidad porque al mostrarse cómo *dPeaQock* es un algoritmo

bastante sensible al incremento del espacio de búsqueda, *ePeaQock* se convierte en una buena opción para realizar el trabajo de optimización en ese caso.

Resulta interesante destacar cómo el optimizador siempre escoge planes que distribuyen el *Skyline* y en la mayoría de los casos, reducen considerablemente el tiempo de evaluación de las consultas. Las suposiciones generales de este trabajo se ven apoyadas por los resultados obtenidos en el estudio experimental.

Por otro lado, en un detallado estudio del modelo de costo se observó que, aunque realizaba estimaciones acertadas en muchos casos, es muy dependiente de la distribución de los valores de los datos con los que se trabaje. Existen evidencias acerca de la incompletitud del modelo de costo actual por considerar el criterio *diff* únicamente como una dimensión más en la cláusula.

Por último, es interesante resaltar que la implementación escogida para el modelo de costo permitió acotar los tiempos de optimización, lo que resulta bastante beneficioso debido a la complejidad de las fórmulas tratadas.

Respecto a las recomendaciones para trabajos futuros, sería interesante la implementación y prueba del modelo de costo aquí presentado. El comportamiento de la directiva *diff* es distinto a las directivas *max* y *min*, por lo tanto debe ser considerado de manera distinta para poder obtener una estimación más acertada del costo de aplicarla.

Una alternativa es realizar una aproximación de las funciones de cardinalidad y costos propuestas. Adicionalmente, es interesante intentar proponer e implementar modelos de costo para el resto de los algoritmos existentes para evaluar al operador *Skyline*.

Finalmente, es de gran interés realizar un estudio completo del algoritmo *ePeaQock* dentro de *PostgreSQL*, debido a que los resultados que se manejan en cuanto a su convergencia y comportamiento como algoritmo evolutivo se encuentran disponibles solamente para el prototipo. El estudio que se realice deberá estar orientado a su entonación, así como a comparar cómo es el desempeño del mismo cuando se varían los operadores y tasas de selección y mutación.

Bibliografía

[1] Sandia national laboratories. Genetic algorithms, evolutionary programming and genetic programming, 1993.

[2] PostgreSQL Global Development Group. PostgreSQL programmer's guide, 1998.

[3] Guía del programador de PostgreSQL. Genetic query optimization, 2000.

[4] Computer science department. Columbia University. RANK: Top-k query processing, 2005.

[5] PostgreSQL global development group. PostgreSQL 8.1.10 documentation, 2007.

[6] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J.Ñ. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[7] W. Balke, U. Güntzer, and J. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

[8] R. Bannon, A. Chin, F. Kassam, and A. Roszko. MySQL conceptual architecture. 2002.

[9] M. Bayir, I. Toroslu, and A. Cosar. Genetic algorithm for the multiple-query optimization problem. *IEEE Transactions on Systems, Man and Cybernetics, Part C*, 37:147–153, 2007.

[10] K. Bennett, M. Ferris, and Y. Ioannidis. A genetic algorithm for database query optimization. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 400–407, 1991.

[11] J. Bentley. On the average number of maxima in a set of vectors and applications. 1978.

[12] J. Bentley. Fast linear expected-time algorithms for computing maxima and convex hulls. 1990.

[13] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *IEEE Conf. on Data Engineering*, pages 421–430, 2001.

[14] C. Brando and V. González. Extension de PostgreSQL con mecanismos de evaluación de consultas basadas en preferencias, 2007.

[15] D. C. *A guide to the SQL standard (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[16] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. *ICDE*, page 64, 2006.

[17] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB'99*, pages 397–410, 1999.

[18] J. Chomicki. Querying with intrinsic preferences. In *EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology*, pages 34–51, London, UK, 2002. Springer-Verlag.

[19] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[20] J. Chomicki. Semantic optimization techniques for preference queries. *Inf. Syst.*, 32(5):670–684, 2007.

[21] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting, 2002.

[22] C. cong Xing and B. P. Buckles. On the size of the search space of join optimization. *J. Comput. Small Coll.*, 20(6):136–142, 2005.

[23] A. L. Corcoran and R. L. Wainwright. LibGA: a user-friendly workbench for order-based genetic algorithm research. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 111–117, New York, NY, USA, 1993. ACM Press.

[24] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[25] R. Elmasri and S.Ñavathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings, 1994.

[26] E. Geschwinde and H. Schoenig. PostgreSQL, developer's handbook. *SAMS*, 2002.

[27] P. Godfrey. Cardinality estimation of skyline queries. Technical Report CS-2002-03, York University, 2002.

[28] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 229–240, 2005.

[29] M. Goncalves and M. E. Vidal. Top-k skyline: An unified approach. 2002.

[30] J. Gottlieb and T. Kruse. Selection in evolutionary algorithms for the traveling salesman problem. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 415–421, New York, NY, USA, 2000. ACM Press.

[31] J. J. Grefenstette, R. Gopal, B. J. Rosmaita, and D. V. Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 160–168, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[32] B. Hafenrichter and W. Kießling. Optimization of relational preference queries. In *ADC '05: Proceedings of the 16th Australasian database conference*, pages 175–184, 2005.

[33] C. Haubelt, J. Gamenik, and J. Teich. Initial population construction for convergence improvement of moeas, 2005.

[34] J. Hong, C. Kao, and B. Liu. A genetic algorithm for database query optimization. In *Proceedings of the First IEEE World Congress on Computational Intelligence.*, pages 350 – 355, 1994.

[35] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 312–321, New York, NY, USA, 1990. ACM Press.

[36] Y. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 168–177, New York, NY, USA, 1991. ACM Press.

[37] W. Kießling and G. Kostler. Preference sql - design, implementation, experiences. Technical report, University of Augsburg, GERMANY, OPUS [http://www.opus-bayern.de/uni-augsburg/oai/oai2.php] (Germany), 2001.

[38] Larra, Kuijpers, Murga, Inza, and Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artif. Intell. Rev.*, 13(2):129–170, 1999.

[39] Z. Michalewicz. Genetic algorithms + data structures = evolution programs. *SIGART Bull.*, 4(2):5, 1996. Reviewer-Michael de la Maza.

[40] B. Momjian. PostgreSQL backend directories, 2005.

[41] A. Moraglio, Y.-H. Kim, Y. Yoon, B.-R. Moon, and R. Poli. Generalized cycle crossover for graph partitioning. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1421–1422, New York, NY, USA, 2006. ACM Press.

[42] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478, New York, NY, USA, 2003. ACM Press.

[43] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[44] PinderSoft. Baseball stats 3.0, 2007.

[45] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[46] S. S. Ray, S. Bandyopadhyay, and S. K. Pal. Genetic operators for combinatorial optimization in TSP and microarray gene ordering. *Applied Intelligence*, 26(3):183–195, 2007.

[47] M. Stonebraker and L. A. Rowe. The design of postgres. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 340–355, New York, NY, USA, 1986. ACM Press.

[48] K. Tan, P. Eng, and B. Ooi. Efficient progressive skyline computation. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 301–310, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[49] G. Tao and Z. Michalewicz. Inver-over operator for the tsp. In *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 803–812, London, UK, 1998. Springer-Verlag.

[50] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989. Morgan Kaufman.

[51] D. Xinyi, L. Yuan, and Z. Lijie. Conceptual architecture of PostgreSQL, 2005.

Apéndice A

Manual para la extensión del optimizador de *PostgreSQL*

El proceso de implementación del operador *Skyline* para que pueda ser considerado en la primera fase de optimización de *PostgreSQL* contempló la modificación de las estructuras, funciones y la creación de nuevos nodos utilizados en el optimizador.

A.1. Parser

Como se sabe, ya el análisis léxico, semantico y sintáctico del *Skyline* se desarrolló en la primera versión de *PeaQock*. Sin embargo se tuvo que modificar la forma en que se almacenaban los criterios del *Skyline* y eliminar el nodo *Sort* que era colocado antes del *Skyline*.

Cambios y creación de funciones:

- En el archivo *../src/backend/parser/gram.y* se modificó la función que agrega las preferencias de *Skyline* al árbol `insertPreferenceSelectOption` para evitar colocar una cláusula *Sort*.
- En el archivo *../src/backend/parser/parse_clause.c* se modificó la función *addTargetToSkylineList* para que utilice la estructura del `SkylineClause` modificada en este proyecto. Se crearon las funciones `makeSkylineClause` y `makeSkylineAtt` las cuales van llenando la estructura `SkylineClause` del árbolo del parser (`SelectStmt`) agregando primero a la lista `skylineClause` un grupo de criterios de una misma tabla con `makeSkylineClause` que a su vez llama a `makeSkylineAtt` por cada dimensión que se vaya a colocar en la lista de `SkylineClause`.

Cambios y creación de nuevas estructuras:

- Se modificó el nodo Query (*../src/include/parser/parsenodes.h*) para que agregue también el conjunto de relids que involucra la claúsula *Skyline* `skylineRelids`, de esta manera alamacenará los datos recibidos de la estructura SelectStmt que pertenecen al *Skyline*.
- Se modificó la estructura `SkylineClause` para que los criterios se encuentren agrupados por tablas involucradas:

```
typedef struct SkylineClause
{
  NodeTag   type;
  List      *atts;  /*lista de criterios con igual relid*/
  Index     relid; /* relid de la tabla*/
```

```
} SkylineClause;
```

- Se creo la estructura `SkylineAtt` la cual corresponde a los datos de un criterio en el *Skyline*. donde el atributo

```
typedef struct SkylineAtt
{
  NodeTag   type;
  Oid       skylineOp; /* tipo del operador Skyline */
  int  skyline_kind; /* MIN, MAX or DIFF */
  Expr   *tleExpr; /* referencia a la exp del TargetEntry correspondiente
                      al atributo de la dimension del Skyline, con el fin de
                      identificarlo univocamente*/
} SkylineAtt;
```

A.2. Planner/Optimizer

En este módulo se realizó la mayor parte de la implementación, ya que es el encargado de construir los planes y escoger el que estima como mejor plan para luego ser enviado al evaluador. En esta estapa se toma todos los datos provenientes del `Query` para crear y encontrar un plan de evaluación.

Se modificaron y crearon estructuras para considerar el *Skyline* como un nodo *down-level*:

- Todos los nodos en *PostgreSQL* tienen una etiqueta o identificador. Por lo tanto para cualquier estructura que se quiera agregar se deben definir las etiquetas de los nodos en *../src/include/nodes/nodes.h*.
- En la primera etapa de optimización se crean caminos de acceso (`Paths`), por esto se deben crear los caminos de acceso para el BNL y SFS en *../src/include/nodes/relation.h* :

```
typedef struct SkylinePath
{
  Path     path;          /* Nombre del nodo, en este caso BNLPath
                             o SFSPath*/
  Path    *sonpath;    /* camino de acceso del hijo izquierdo y
                             'unico hijo del Skyline*/
  List    *skylineClause; /* SkylineClause, lista de criterios del Skyline*/
  List    *sonsortkeys;   /* Ordenamiento que se lleva, lista de PathKeys*/
  int      skylineDim;     /* Numero de dimensiones para ese Skyline*/
  double ntuples;          /* Numero de comparaciones estimadas para SFS o BNL*/
```

```
} SkylinePath;

typedef SkylinePath SFSPath; /* Camino de acceso para el SFS */
typedef SkylinePath BNLPath; /* Camino de acceso para el BNL */
```

- Los planes lógicos en la optimización son representados por estructuras `RelOptInfo`. Para evitar demasiadas comparaciones en la optimización se agregan las mismas cláusulas del `SkylinePath` en este nodo, ademas del conjunto de *relids* para el *Skyline*, el número de comparaciones y un entero que indica si es el último nodo del árbol o no. Este cambio fue realizado en *../src/include/nodes/relation.h* :

```
typedef struct RelOptInfo
{ .
  .
  List    *skylineClause;  /* Clausula Skyline tomada para los
                              caminos de acceso de este RelOptInfo*/
  Relids  skylineRelids;   /* Conjunto de relids pertenecientes al Skyline*/
  int     lastRel;         /* Valores: 0 o 1. Si es 1 entonces
                              se le agrega el Skyline final */
  double  ntuplesSFS;      /* Comparaciones estimadas para SFS */
  double  ntuplesBNL;      /* Comparaciones estimadas para BNL */
}
```

- El árbol que se le pasa como parametró al evaluador se compone de nodos del tipo `Plan`, por lo tanto, incialmente se tenía el nodo plan para el *Skyline*, pero como no se elegía que operador físico utilizar no se hacía ninguna distinción del mismo. Entonces se crearon los nodos para evaluar el *Skyline* como SFS o BNL en *../src/include/nodes/plannodes.h*:

```
typedef Skyline SFSSkyline;
typedef Skyline BNLSkyline;
```

Funciones modificadas e implementadas para la primera fase de optimización:

- La función `make_rels_by_joins` es la llamada principal para la optimización dinámica. Se encuentra en (*../src/backend/optimizer/path/joinrels.c*). Fué modificada para agregar el nodo reloptinfo del *Skyline* cuando el optimizador se encuentre en el último nivel, ya que en el se ejecuta la construcción de los diferentes planes para el nivel dado. Aqui también se generan los planes bushy.
- La función `make_rels_by_clause_joins` (*../src/backend/optimizer/path/joinrels.c*), es ejecutada por la función anterior, esta se encarga de construir todos los planes partiendo de la relación anteriormente

construida y efectuando un *Join* con una relación base. Para considerar el *Skyline* como un nodo *down-level* en ésta deben agregarse los nodos RelOptInfo del *Skyline* si es factible agregarlo según la regla algebraica para éste. Aqui se generan varios planes, los tradicionales formados por *Joins*, y si es factible crear el *Skyline* este puede ser colocado en cualquier lado o ambos lados del *Join*. En la lista de planes lógicos, se agregan todas estas posibilidades, para que en los pasos posteriores se pueda el que se estima que es el mejor plan. Esta función llama a `make_join_rel` y `make_skyline_rel`.

- La función `make_join_rel` (*../src/backend/optimizer/path/joinrels.c*) fue modificada para que vaya acumulando los `skylineRelids`.
- Se modificó `find_join_rel` (*../src/backend/optimizer/util/relnode.c*) porque en esta función se incluye la verificación de que devuelva el mismo árbol lógico si el plan a crear posee el mismo conjunto de *relids* y *skylineRelids*, ya que la equivalencia en los planes cambia al agregar el *Skyline* como *doqn-level*, si el `RelOptInfo` existe se devuelve ese, sino devuelve null.
- Se creó la función `make_skyline_rel` (*../src/backend/optimizer/path/joinrels.c*) la cual construye un `RelOptInfo` con la información importante del *Skyline*.
- El `RelOptInfo` es creado por la función implementada `build_skyline_rel` (*../src/backend/optimizer/util/relnode.c*). También las funciones `add_skyline_clause` y `add_skyline_diff` se implementaron para agregarle al `RelOptInfo` la información de la cláusula *Skyline* a crear.
- Además se desarrolló la función `add_paths_to_skyline_rel` (*../src/backend/optimizer/path/joinpaths.c*) la cual agrega todos los caminos de acceso del *Skyline* a la lista de caminos de evaluación.
- Se implementaron para la creación de los caminos de acceso las funciones `create_bnl_path` y `create_sfs_path` (*../src/backend/optimizer/path/joinpaths.c*) que crean los caminos para *SFS* y *BNL*. Aquí se llamana a las otras destinadas para la estimación del número de comparaciones y el costo del operador.

Funciones modificadas e implementadas para la segunda fase de optimización:

- Se modificó la función `create_plan` (*../src/backend/optimizer/plan/createplan.c*), la cuál crea un plan de evaluación desde un camino de evaluación. Por lo tanto este procedimiento debe considerar la creación de un nodo plan `Skyline` a partir de un `SFSPath` y `BNLPath`.
- Se modificó la función `create_skyline_plan` porque el *Skyline* puede tener ahora diferentes implementaciones (*SFS* o *BNL*). Para esto, se implementaron las funciones `create_sfs_plan` y `create_bnl_plan`, que ejecutan a `make_sfsskyline` y `make_bnlskyline` respectivamente. Cabe destacar que allí se almacenan los datos necesarios en las estructuras adecuadas para que el plan pueda ser enviado al evaluador.

Apéndice B

Número de planes para una consulta *Skyline*

Estimar la cantidad de planes lógicos asociados a una consulta *Skyline* no es un problema sencillo. Hasta estos momentos, no se conocen trabajos que estimen este número. En este trabajo se propone una forma de contar los planes, haciendo una simplificación del problema.

Una vez que se decide colocar un *Skyline* sobre una relación, todos las dimensiones que tengan que ver con esa relación serán consideradas en ese *Skyline*. Esto lo que quiere decir es que intuitivamente, no importa cuántas dimensiones se tengan definidas en un *SKYLINE OF*, lo que influye es el número de tablas que se encuentran involucradas en el *Skyline*. Para comprender lo que aquí se menciona obsérvese el siguiente ejemplo.

Ejemplo 2 *Se tienen dos relaciones A y B. Se tienen además dos Consultas Skyline sobre la relación A* $\bowtie$ *B. La primera consulta define 10 dimensiones sobre la tabla A, mientras que la segunda consulta define dos dimensiones, una sobre cada tabla.*

Haciendo una relajación del problema, supóngase que se puede decidir colocar los *Skyline* sobre la relación base o no. Es decir, los subconjuntos de relaciones involucradas considerados para repartir el *Skyline* son de tamaño uno.

Para la primera consulta existen dos posibles planes, el que se decide mantener el *Skyline* en la raíz y el que decide colocarlo sobre la relación base. Obsérvese la figura 31.

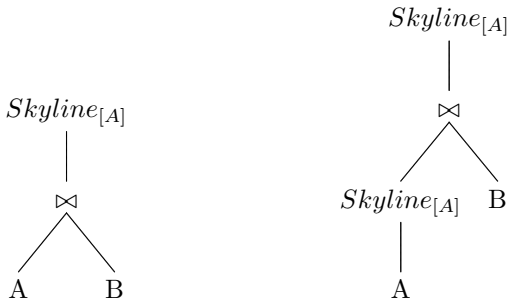


Figura 30: Planes posibles para la Consulta 1

Para la segunda consulta la cantidad de planes aumenta a cuatro, justamente porque se tienen más tablas involucradad en el *Skyline* y por lo tanto se tienen más subconjuntos de tamaño uno, que pueden decidirse colocar o no sobre su tabla blase correspondiente.

Esto muestra que el número de dimensiones no afecta directamente el número de planes lógicos que se pueden generar a partir de una consulta Skyline. Cabe destacar que cuando no se tienen *Joins* en la consulta, el número de planes lógicos siempre va a ser uno y por supuesto el número de tablas en las que el Skyline va a estar distribuído es acotado por el número de dimensiones. Entonces, el número de planes lógicos que se

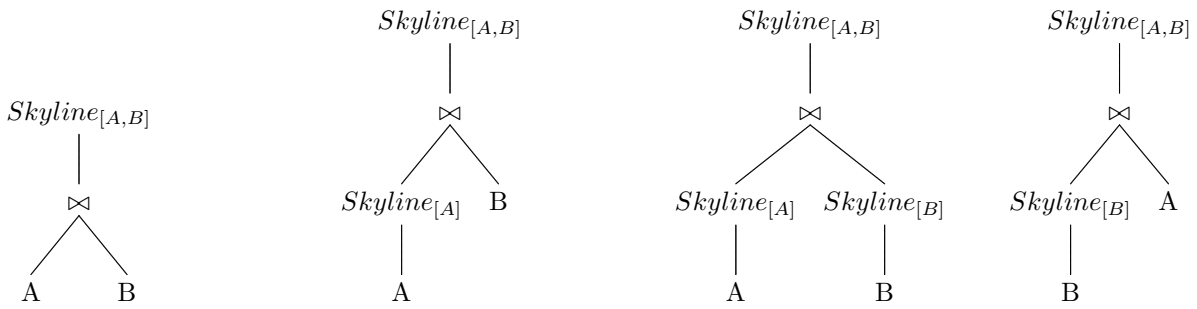


Figura 31: Planes posibles para la Consulta 1

pueden hallar bajo la relajación del problema explicada anteriormente es:

$$C_p = 2^r \binom{2}{n} (n-2)!$$

Donde r es el número de tablas involucradas en el *Skyline* y n es el número de relaciones en la consulta. La razón de realizar una simplificación y no buscar el número exacto de planes lógicos es que se quería era obtener una idea de cuánto podía crecer el espacio de búsqueda de las consultas *Skyline* respecto al espacio de búsqueda manejado tradicionalmente.

Apéndice C

Diagramas

A continuación se presenta el diagrama general simplificado del optimizador de *PostgreSQL*y el diagrama de paquetes propuesto para la inclusión de *ePeaQock*.

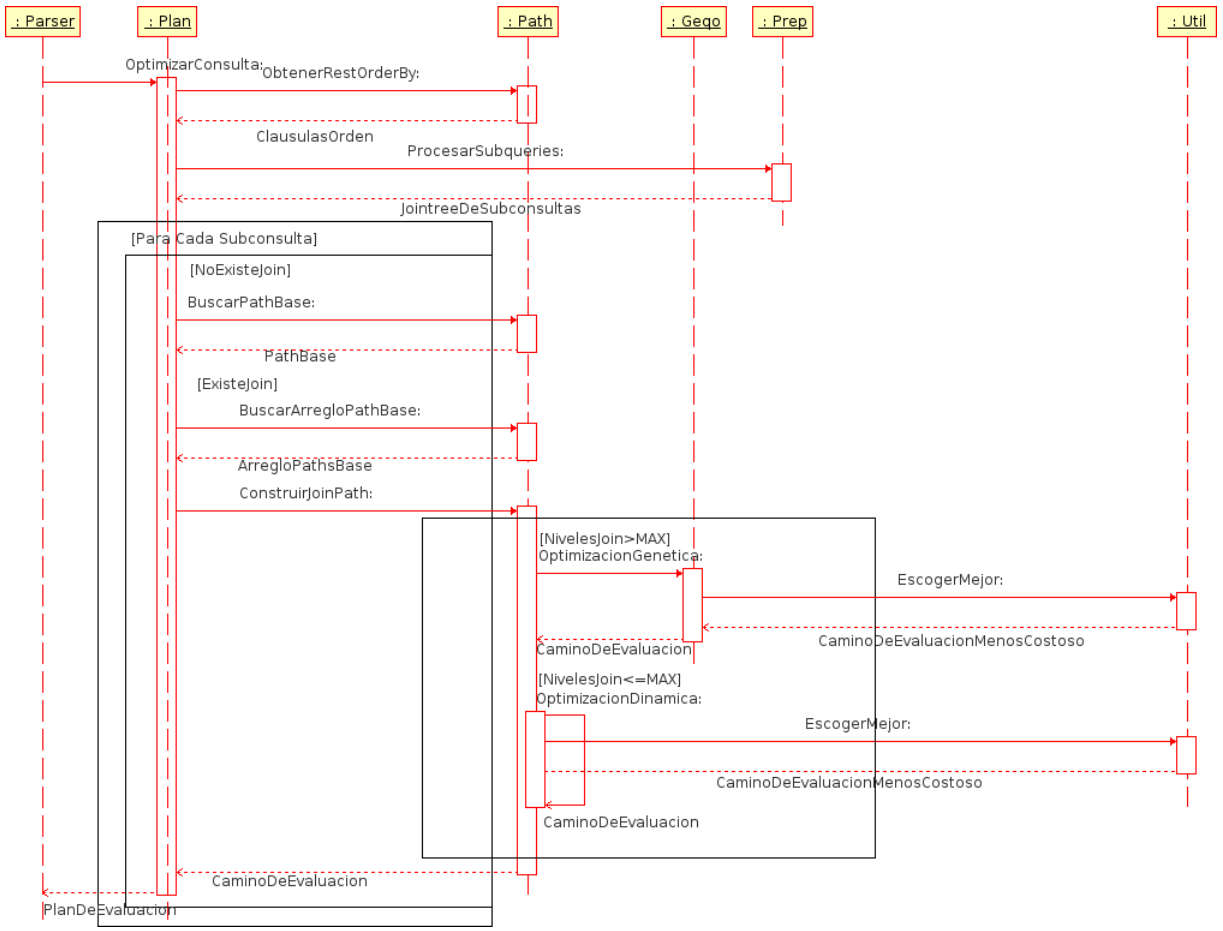


Figura 32: Diagrama de secuencia para el optimizador de *PostgreSQL*.

En el diagrama de secuencias mostrado en la Figura 32, se observa de manera bastante simplificada el hilo de ejecución del optimizador. A manera de resumen, el optimizador lo que hace es crear un arreglo con todas las subconsultas que están involucradas en la consulta principal y para cada una de ellas realiza el proceso de búsqueda de un camino de evaluación. El algoritmo escogido para realizar tal búsqueda dependerá del número de *Joins* que tenga la expresión a optimizar.

Por otro lado, en la figura 33 se muestra que la inclusión del paquete *ePeaQock*, que contiene la implementación del algoritmo evolutivo propuesto en este trabajo. Se observa que interactúa con los demás paquetes de la misma manera que lo hace *GEQO*. Esto es porque las estructuras implementadas son totalmente independientes del resto del optimizador del mismo modo que *GEQO*. Adicionalmente, en la Figura 34, se muestra

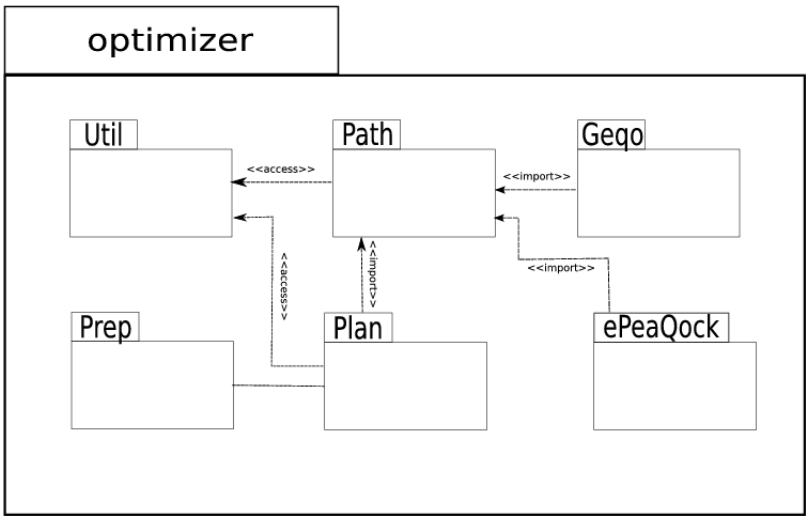


Figura 33: Diagrama de paquetes para el optimizador de *PostgreSQL*

el diagrama de clases para el paquete *ePeaQock*.

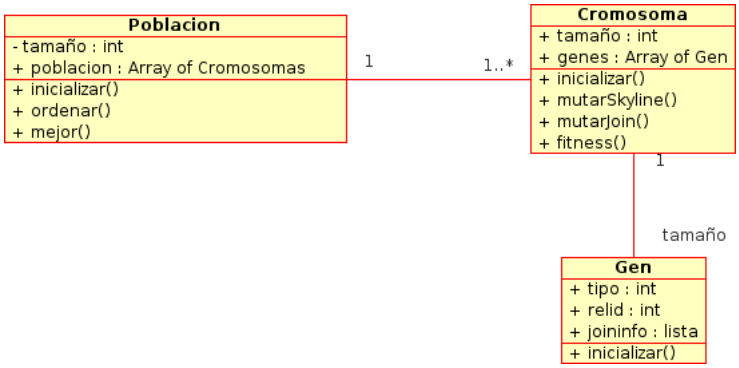


Figura 34: Diagrama de Clases para *gPeaQock*

A continuación se muestra el diagrama de secuencia extendido para el optimizador de *PostgreSQL*en la Figura 35.

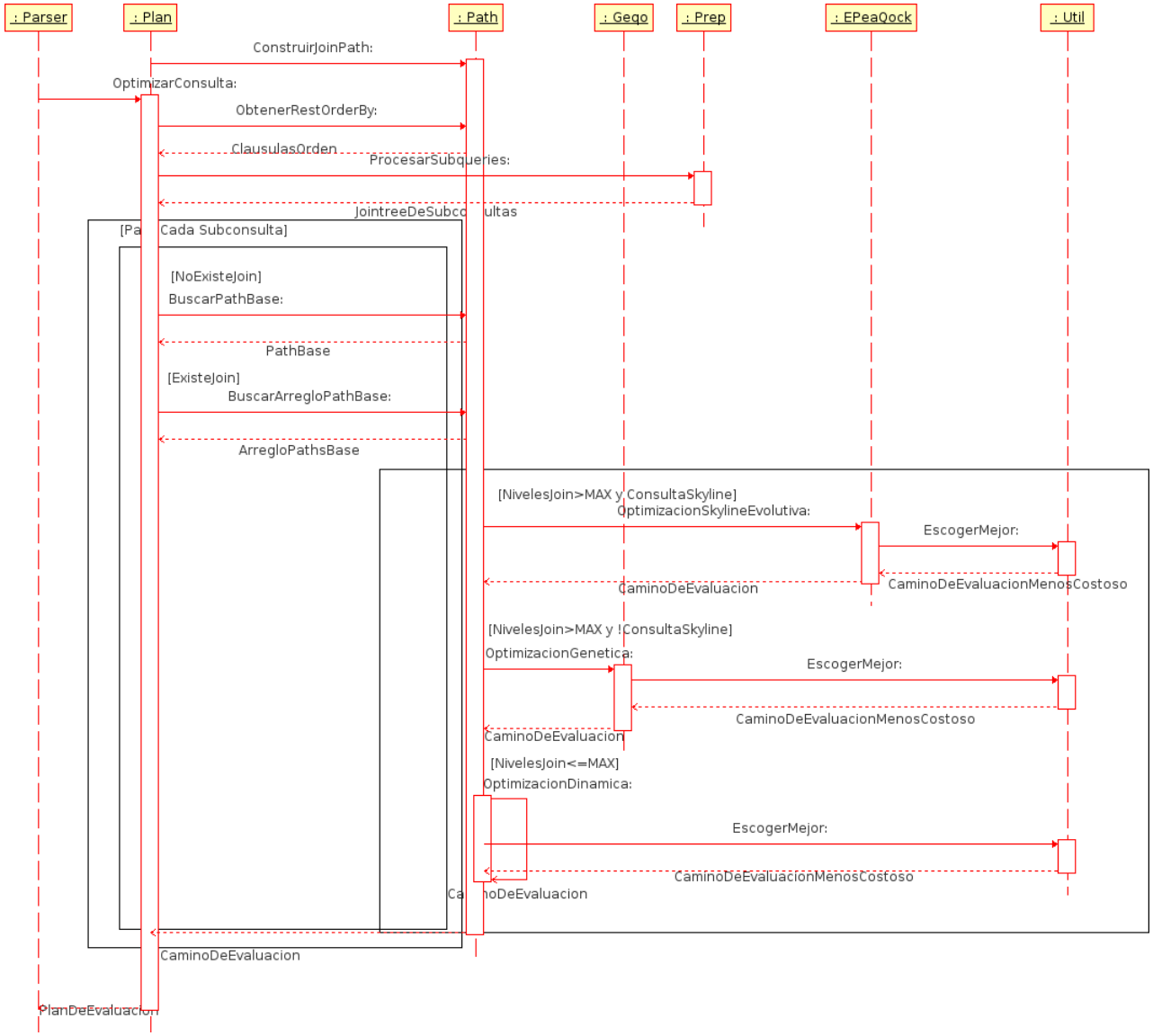


Figura 35: Diagrama de secuencia extendido para el optimizador de *PostgreSQL*.

Apéndice D

Modelo de costo y Reglas algebraicas

En este apéndice se muestra información relevante al proceso de optimización en un manejador relacional común. Se presentan las reglas algebraicas más comunmente usadas y el modelo de costo para el operador *Join*.

D.1. Modelo de costo para el *join*

Según [45], para obtener funciones de costo razonablemente exactas para el *Join* entre dos relaciones S y R, se necesita calcular la selectividad de *Join*. La selectividad de *Join* (js) se calcula como la razón del número de tuplas que pertenecen a la respuesta de un *Join* entre el número de tuplas originadas por el producto cartesiano.

$$js = \frac{|(R \bowtie_c S)|}{|(R \times S)|}$$

La selectividad de *Join* estará entre 0 y 1, $0 \leq js \leq 1$. Si js es igual cero, significa que el *Join* no produjo ningún resultado. Si js es igual a uno, significa que el *Join* que se realizó es equivalente a realizar el producto cartesiano de las relaciones.

Por lo tanto, si se conoce la selectividad del *Join* entonces se puede utilizar la fórmula para estimar la cardinalidad de la relación resultante de la operación *Join* dadas las dos relaciones de entrada. Un *join* se expresa como $R \bowtie_{A=B} S$, donde R es la relación externa a ejecutar que ocupa b_r bloques y S es la relación interna a ejecutar que ocupa b_s bloques. El factor de bloqueo [1] de la relación resultante se denota como fbl_{RS}, y D es el costo de leer/escribir un bloque en disco. Existen varias implementaciones del operador *Join* que difieren en la manera como son preocesados los datos y en el costo de su ejecución. A continuación se presentan las fórmulas de costo para *Nested Loop Join, Index Nested Loop Join* y *Sort Merge Join* en base a los parámetros fijados anteriormente.

Nested Loop Join El *Nested Loop Join* es una implementación del *Join*, la cuál tiene un costo de ejecución orden $O(n^2)$. Para cada tupla de R se revisa cuáles tuplas de S cumplen con la condición del *Join*. Suponiendo que se conoce la selectividad de *Join*, se tiene la siguiente función de costo de lectura y escritura a disco con este método:

$$C_{I/O} = [b_r + (b_r * b_s) + \frac{js * |R| * |S|}{fbl_{RS}}] * D$$

Existe una mejora de este algoritmo llamada *block nested loop join* y lo que hace es que en vez de leer cada tupla, se accede a un bloque de disco y las tuplas contenidas en ese bloque son comparadas de una vez

[1] Factor de bloqueo se calcula como el número de tuplas de una relación entre el tamaño de un bloque de disco y expresa cuántas tuplas pueden ser almacenadas en un bloque

contra las de la otra relación. La mejora de esto es que en vez de leer toda la relación S por cada tupla de la relación R, se lee tantas veces como el valor de b_s.

Index Nested Loop Join Este algoritmo trabaja del mismo modo que el anterior solo que busca los valores de S que hacen *match* por indexación y no realizando un *scan* sobre todas las tuplas.

Para el *Index Nested Loop Join*, si existe un indice para el atributo de *Join* B de S de x_B niveles, se pueden obtener todas las tuplas de R y luego usar el índice para obtener todas las tuplas coincidentes de S que satisfagan la condición.

$$C_{I/O} = [b_R + (|R| * (X_B + 1)) + \frac{js * |R| * |S|}{fbl_{RS}}] * D$$

Sort Merge Join El *Sort Merge Join* toma como precondición que ambas relaciones están ordenadas por el atributo de interés y lo que hace es recorrer en orden lineal las tuplas para ir uniendo los atributos de igual valor. El costo de entrada y salida del operador si las relaciones ya están ordenadas según el atributo de *Join* es:

$$C_{I/O} = [b_R + b_S + \frac{js * |R| * |S|}{fbl_{RS}}] * D$$

Si ambas relaciones se encuentran desordenadas, se debe realizar un ordenamiento previo, por lo que se agrega el costo de ordenar cada relación, quedando la función de costo de la siguiente forma:

$$C_{I/O} = [k * ((b_R * log_2 b_R) + (b_S * log_2 b_S))b_R + b_S + \frac{js * |R| * |S|}{fbl_{RS}}] * D$$

D.2. Reglas algebraicas más comunes

A continuación se presentan las reglas algebraicas más comunmente usadas para la optimización de consultas en manejadores relacionales.

- Cascada de la selección: $\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))))$.
- Conmutatividad de la selección: $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$.
- Conmutatividad del *Join*: $R \bowtie S \equiv S \bowtie R$.
- Asociatividad del *Join*: $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$.
- Distributividad de la selección sobre el *Join*. Suponga c una condición de selección sobre R, entonces: $\sigma_c(R \bowtie S)\sigma_c(R) \bowtie S$

Apéndice E

Reglas algebraicas más comunes

A continuación se presentan las reglas algebraicas más comunmente usadas para la optimización de consultas en manejadores relacionales.

- Cascada de la selección: $\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))))$.
- Conmutatividad de la selección: $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$.
- Conmutatividad del *Join*: $R \bowtie S \equiv S \bowtie R$.
- Asociatividad del *Join*: $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$.
- Distributividad de la selección sobre el *Join*. Suponga c una condición de selección sobre R, entonces: $\sigma_c(R \bowtie S)\sigma_c(R) \bowtie S$

Apéndice F

Implementaciones del *Skyline*

Como los demás operadores definidos en un manejador cualquiera, existen varias maneras de implementar el operador relacional *Skyline*. En esta sección se describirán algunas variantes propuestas en trabajos anteriores.

En primer lugar, las consultas que utilizan el operador *Skyline* pueden ser traducidas a una consulta anidada en SQL y ser ejecutadas de esa manera. Sin embargo, la ejecución de una consulta por este método es ineficiente. Por ello se propone implementar el operador *Skyline* como un operador relacional más y no transformarlo en operadores existentes. Para esto se han propuesto varios algoritmos existentes. El problema de calcular el conjunto Skyline es el problema del vector máximo [28], y ha sido estudiado anteriormente. A continuación se presenta la descripción de algunos algoritmos que intentan resolver este problema.

F.1. Algoritmos que mantienen una ventana

Se presentan dos algoritmos cuya estrategia es la comparación entre bloques de tuplas por cada iteración del algoritmo. Estos algoritmos mantienen una ventana con las tuplas que ya han sido procesadas y que serán comparadas con otras para comprobar si pertenecen al conjunto Skyline o no.

En primer lugar observamos el algoritmo *Block Nested Loop (BNL)*. La idea de este algortimo es mantener una ventana de tuplas incomparables en memoria. En cada iteración se leen tuplas para ser procesadas. Una tupla p y es comparada con cada una de las tuplas que se encuentran en la ventana. Como resultado de esta comparación puede suceder que:

- p es eliminada si se consigue que es dominada por alguna de las tuplas de la ventana
- p es agregada a la ventana si domina algunas tuplas en la ventana y se eliminan dichas tuplas.
- p es colocada en un archivo temporal si es incomparable con todas las tuplas de la ventana y no cabe en la ventana.

En cada iteración, se recrea la ventana de tuplas con las que se van leyendo tanto del disco como del archivo temporal. Al final de cada iteración la ventana resultante es el conjunto Skyline de la consulta. En el peor de los casos, el algortimo tiene que hacer más de una iteración para poder hacer el escaneo contra las tuplas que fueron almacenadas en el archivo temporal. Nótese que este algoritmo tiene una estrategia de comparación de todos contra todos lo que lo puede hacer muy costoso en ejecución.

Para optimizar la ejecución de este algortimo, se puede optar por estrategias como mantener la ventana de tuplas organizada. Como la mayoría del tiempo se consume comparando las tuplas entre todas, al tener un orden establecido. Cuando se consigue que una tupla domina alguna tupla de la ventana, se coloca al inicio para que sea el primer punto de comparación de las demás.

Por otro lado existe una mejora del BNL propuesta en el articulo "Skyline with Presorting"[21], llamada *Sort Filter Skyline (SFS)* que se diferencia con el BNL en que la tabla de entrada debe estar ordenada por todos los atributos a considerar, según el criterio de skyline fijado. Es decir, si se desea realizar el skyline por las tuplas que maximizen los atributos a1, ..., ak se procede a ordenar la tabla por todos estos atributos de forma descendiente. De esta forma una tupla comparada con las tuplas pertenecientes a la ventana puede resultar ser incomparableo dominada, pero nunca se realiza un reemplazo en la ventana de tuplas procesadas. Por lo tanto realiza siempre el menor número de pasos, ya que el conjunto de tuplas almacenado en la ventana pertenecen al conjunto respuesta o Skyline.

En el artículo "Maximal Vector Computation on large data sets" [28], se propone un nuevo algoritmo basado en SFS, BNL, y FLET [12]. LESS (*linear elimination sort for skyline*), filtra las tuplas por medio de una ventana de filtro de skyline, como se realiza en SFS. El conjunto de tuplas tiene que estar ordenado. Los cambios que realiza LESS sobre SFS se resumen en que usa una ventana de filtrado de tuplas en el paso cero del algoritmo (cuando se está realizando el ordenamiento) para poder eliminar tuplas de manera más rápida lo que conlleva a combinar el último paso del ordenamiento con el primer paso de filtrado con la ventana de filtrado skyline, partiendo la ejecución con un conjunto de tuplas de menor tamaño.El proceso de ejecución es similar al SFS.

F.2. Algoritmos divide&conquer

Este algoritmo trata de aprovechar las bondades de la busqueda binaria. El procedimiento consiste en los siguientes pasos:

H1 Calcular el valor medio para una de las dimensiones d_1, entonces se particiona el conjunto de tuplas inicial en dos P_1 y P_2. Nótese que hay una partición "mejor" respecto a d, supongamos que es P_1.

H2 Se calculan los *Skyline* S_1 y S_2 de cada partición P_1 y P_2 recursivamente. Es decir, se vuelven a particionar en dos nuevos subconjuntos y así hasta que no se pueda particionar más. Cuando se ha llegado al caso base de la recursión (una o pocas tuplas), entonces se calcula el *Skyline* para ese conjunto, lo que debería consumir poco tiempo y recurso.

H3 Luego se hace la unión de cada *Skyline*calculado. La mezcla entre dos particiones se realiza eliminando de S_2 los elementos que son dominados por alguna tupla de S_1. Nunca va a suceder que alguna tupla de S_1 sea dominada por alguna de S_2 porque S_1 contiene la "mejor" partición.

La complicación de este algoritmo se encuentra en 3, porque representa el trabajo más costoso. Para hacer la mezcla de S_1 y S_2 y así obtener el *Skyline* resultante, se vuelve a particionar cada uno de los dos conjuntos S_1 y S_2 usando la misma estrategia del inicio. Es decir, se toma una segunda dimensión d_2 y se realiza el mismo trabajo de calcular la media para esa dimensión y separar cada S_i en dos particiones en base a ese valor.

El objetivo es hacer una mezcla de $S_{1,1}$ con $S_{2,1}$ y de $S_{1,2}$ con $S_{2,2}$. La mezcla se realiza aplicando nuevamente el algoritmo. La recursión termina cuando se han considerado todas las dimensiones d_i o se llega a una partición vacía o con una tupla.

Este algoritmo requiere de gran utilización de recursos de la computadora, en especial, de la memoria. En consultas donde se procese una gran cantidad de tuplas, se requeriría de accesos a disco para almacenar las tuplas que no cupieran en memoria. Para ello, se puede, en vez de particionar el conjunto total de tuplas en dos, hacer m particiones. La idea es crear un bloque de tuplas que si pueda ser procesado en memoria. También, para solucionar el problema de memoria, puede pre-filtrarse la entrada. Se toma un bloque grande de tuplas, y se aplica el divide&conquer usual. Luego al *Skyline*resultante, se le unen las tuplas restantes y se aplica el algoritmo m-particionando los conjuntos resultantes de cada división.

Apéndice G

El manejador PostgreSQL

Los inicios de *PostgreSQL* se remontan a 1970 cuando su antecesor, *Ingres*, comenzó a ser desarrollado como un manejador relacional de base de datos en la Universidad de Berkley, California. Este manejador experimental pasó a ser un producto comercial cuando la compañía *Relational Technologies*, luego *Ingres Corporation*, lo adquirió para tales efectos [2]

En 1986, nuevamente un grupo de la Universidad de Berkley trabajó en una extensión de *Ingres*, llamada *Postgres*, que añadía funcionalidades orientadas a objeto. De nuevo, el producto fue adquirido por una empresa para su comercialización. Esta vez es *Illustra* quien se encarga de llevar a la venta el producto.

El desarrollo de *Postgres* por parte de *Illustra* terminó oficialmente en 1994 con la versión 4.2, debido a el gran esfuerzo que requería el mantenimiento de la aplicación [26]. Sin embargo, ese mismo año, dos graduados de Berkley, Andrew Yu y Jolly Chen, incorporaron a *Postgres* soporte para SQL (el lenguaje de consultas manejado anteriormente era uno propio llamado Postquel [47]), convirtiendo el proyecto *Postgres* en *PostgreSQL*. La última comercialización de este producto fue realizada por *RedHat* con el nombre de *RadHat Database*.

PostgreSQL sigue existiendo como manejador *open source*, se encuentra disponible en la red y sigue siendo desarrollado y mantenido por un grupo llamado *PostgreSQL Global Development Group* que está conformado por personas de todo el mundo.

Actualmente, *PostgreSQL* es uno de los manejadores de base de datos más aceptados a nivel mundial. Compite como una de las mejores aplicaciones *open source* de este tipo. *PostgreSQL* está implementado en ANSI C, con una estructura cliente–servidor. Una sesión de *PostgreSQL* se encuentra soportada por la interacción de tres procesos: un demonio llamado *postmaster*, que se encarga de estudiar y aceptar o rechazar los pedidos de conexión a una base de datos; un proceso cliente *psql*, que se encarga de hacer el pedido de conexión al postmaster; y un proceso *backend* llamado *postgres*, que es el servidor como tal. Básicamente el *postmaster* maneja las conexiones a las distintas bases de datos existentes en un *host*, esperando que algún proceso *psql* haga una petición para la conexión al servidor de bases de datos.

Cuando una petición de conexión es aceptada se inicia un nuevo proceso *postgres* el cual es conectado con el proceso *psql* que realizó la petición y así poder comunicarse directamente sin intervención del *postmaster* [5]. Por cada proceso cliente (*psql*), existe en el servidor un hilo que mantiene un proceso *postgres*, pero solo existe un proceso *postmaster* por servidor.

A continuación se presenta la arquitectura de *PostgreSQL* y se brindará un breve panorama acerca del procesamiento de una consulta en el manejador.

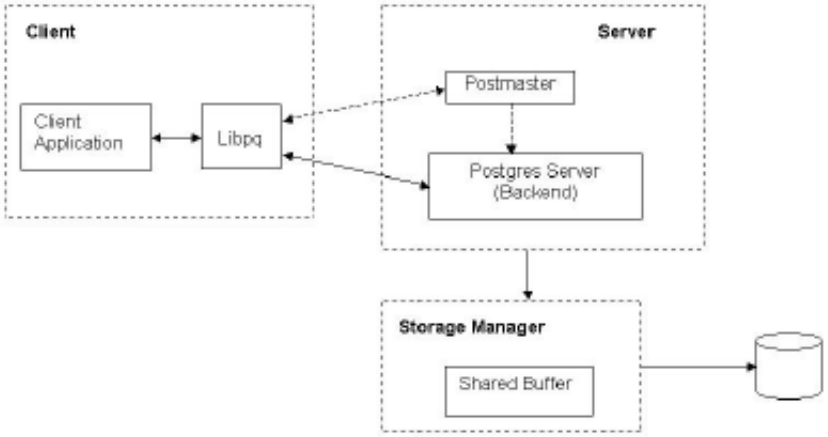


Figura 36: Arquitectura cliente-servidor de *PostgreSQL* Fuente [14]

G.1. Arquitectura de *PostgreSQL*

Según [51], la arquitectura de la implementación de *PostgreSQL* está estructurada como se muestra en la Figura 37.

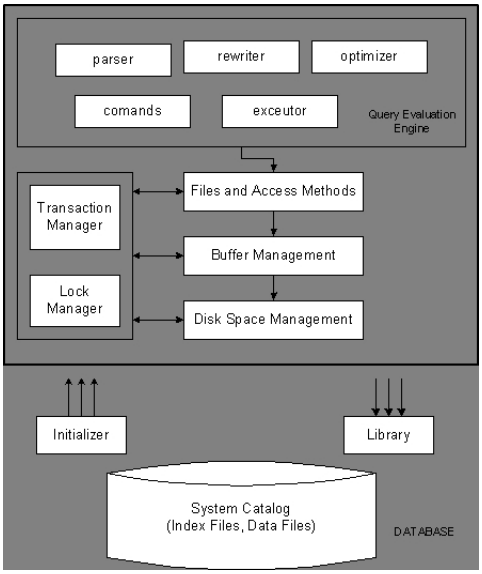


Figura 37: Arquitectura de implementación de PostgreSQL

Como puede observarse en la Figura 37, los componentes mostrados se encuentran estructurados de manera separada según su funcionalidad y dependen o interactúan con otros componentes del manejador. A grandes rasgos, estos módulos son:

- **Evaluación de consultas:** acepta las consultas escritas en SQL desde la aplicación del usuario y realiza el procesamiento de la consulta para retornar el resultado al usuario.

- **Manejador de archivos y métodos de acceso:** soporta el concepto de archivo como un conjunto de páginas o registros, en sus distintas implementaciones: *heap* e índices.
- **Manejador del *buffer*:** controla el uso de la memoria, mantiene y accede las páginas que han sido leídas desde el disco.
- **Manejador del espacio de disco:** maneja el espacio del disco donde se almacenan los datos. Es el módulo donde se encuentran implementadas las operaciones de lectura y escritura de páginas.
- **El manejador de Transacciones:** gestiona la ejecución de las transacciones del manejador.
- **El manejador de *Locks*:** Maneja las peticiones y liberaciones de *locks*. En conjunto con el manejador de transacciones provee el control sobre las operaciones concurrentes, *rollback* y recuperación de fallas.
- **El subsistema de inicialización y el subsistema de librerías** realizan acciones de inicialización y comunicación con el *kernel*, respectivamente.

Es de especial interés el módulo de evaluación de consultas, porque es el que se estudió y modificó durante la realización de éste proyecto.

G.2. Proceso de evaluación de consultas en PostgreSQL

En el desarrollo de este proyecto es de gran interés el módulo que maneja el procesamiento de las consultas, especialmente la etapa de optimización. En *PostgreSQL* una consulta es procesada a través de cuatro etapas: análisis sintáctico–léxico y semántico, reescritura, optimización y evaluación. En cada una de las etapas es producido un árbol en representación de la consulta entrante con información acerca de los operadores, relaciones y atributos relevantes.

Esta estructura de árbol está compuesto por nodos que estan implementados como estructuras de C[1] que varían también según la etapa de procesamiento en la que se encuentre la consulta.

Este módulo es lo bastante complejo como para descomponerse en varias partes. Obsérvese en la figura38 el flujo de procesamiento de las consultas, donde cada recuadro representa un componente de software dedicado a ejecutar alguna de las etapas del procesamiento. En primer lugar tiene que estar establecida una conexión del proceso del cliente *psql* con un proceso servidor *postgres*, de esta manera pueden realizarse las consultas desde la aplicación cliente y el servidor se encargará de realizar el procesamiento de la consulta a través del *parser*, luego el reescritor, el optimizador y finalmente el evaluador de la consulta. A continuación se detallará un poco más en estos cuatro módulos haciendo especial énfasis en el optimizador.

G.2.1. Parser

El *parser* es el primer módulo que procesa la consulta. Éste recibe como entrada una cadena de caracteres enviada desde el proceso cliente y le realiza un análisis léxico, sintáctico y semántico. En primer lugar se

[1] ANSI C. lenguaje en el que se encuentra implementado PostgreSQL

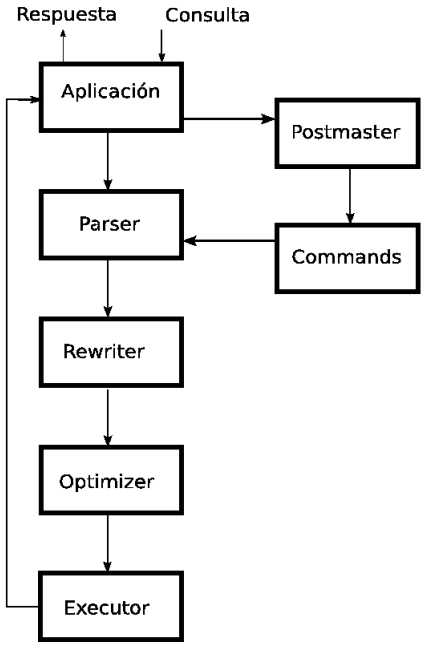


Figura 38: Procesamiento de una consulta en *PostgreSQL*

separa la cadena de caracteres en *tokens*, uno por cada palabra. En este proceso se estudian identificadores y palabras claves para verificar que se encuentran escritas correctamente. Luego, mediante el uso de una gramática, se realiza el análisis sintáctico para determinar la correctitud en el orden de la escritura de la consulta. En este punto, se ha creado una estructura de árbol con la información de la consulta cuyo nodo raíz es un `SelectStatement`[2]. Por último, se realiza una verificación de las tablas y atributos a los que se hace referencia en la consulta para constatar su existencia. En este punto se crea una estructura llamada `Query` que contiene la información de la consulta con las referencias necesarias a tablas y atributos.

Es así como se termina de crear el *Querytree* que contiene la información acerca de la estructura de la consulta, las operaciones relacionales que se deben realizar y la información sobre las relaciones que se encuentran involucradas en la consulta.

G.2.2. Reescritor

La entrada de este módulo es el *Querytree* producido en el *parser* y su objetivo es transformar el árbol de acuerdo a las definiciones de vistas, *triggers* y reglas que estén involucradas con la consulta de entrada. El procesamiento consiste en construir un árbol con las referencias e información necesarias para ejecutar subqueries correspondientes a vistas o reglas y señalar la apropiada ejecución de *triggers*. En caso de que no existan asociadas reglas, vistas o *triggers* a la consulta, se crea un árbol correspondiente uno a uno con el *Querytree*. Esta nueva estructura es llamada *Rewritten-tree*.

[2]Estructura de datos que permite construir un árbol inicial de la consulta, sin ninguna otra información que las cláusulas presentes en la misma

G.2.3. Optimizador

En el optimizador se transforma el *Rewritten-tree* en un nuevo árbol llamado *Plantree*, que será el plan de evaluación para la consulta de entrada. Este módulo se encuentra detallado en el capítulo 3.

G.2.4. Evaluador

El evaluador toma el *Plantree* y lo procesa para realizar la evaluación del plan y retornar el resultado de la consulta al usuario. En el evaluador se realizan tres procesos fundamentales para la evaluación del plan: inicialización, ejecución y finalización. En la inicialización se construye un árbol equivalente al *Plantree* pero con la información relevante para su ejecución. Es en esta etapa cuando se inicializan las estructuras con la información correspondiente para la ejecución de cada operador físico.

En la etapa de ejecución, como es natural pensar, se produce la evaluación de cada operador. La ejecución se realiza de manera recursiva y la transmisión de la información entre dos operadores es de tipo *pipeline*, es decir, se transmite tupla por tupla. Cuando una tupla sobrevive hasta la raíz del árbol de ejecución, entonces es retornada al usuario.

Luego que termina la ejecución, la etapa de finalización se encarga de liberar todas las estructuras que se usaron durante la ejecución.

Apéndice H

Propuesta inicial de *ePeaQock*

An Evolutionary Algorithm for Skyline Query Optimization

Fabiola Di Bartolo, Marlene Goncalves, Ivette Martínez and Francelice Sardá

Grupo de Bases de Datos/Grupo de Inteligencia Artificial
Universidad Simón Bolívar
Caracas 1080-A, Venezuela
fabiola@gia.usb.ve, {mgoncalves,martinez}@ldc.usb.ve, francelice@gia.usb.ve

Abstract. SQL Query optimization is the process to select a "good" evaluation plan. A good plan is one that has a low estimated execution time. A plan is composed of Relational Algebra operators. One of them is the Join which may affect query execution time substantially. Recently a new operator, the Skyline, has been introduced to evaluate user-preference queries. Skyline-Join query optimization could improve execution time. Existing solutions for the optimization are based on dynamic programming, losing effectiveness as the size of search space increases. On the other hand, semi-aleatory search algorithms have shown to be successful to solve the optimization problem for standard queries. In this work we propose an evolutionary algorithm for Skyline-Join query optimization. Our algorithm comprises three specialized mutation operators. The experiments evidence that our algorithm obtains good plans in few iterations, when mutation over the Skyline is used.

1 Introduction

A query is a request for information from a database. For relational DBMSs (Database Management Systems), a query may be specified by means of the standard Structured Query Language (SQL) [3]. SQL is a declarative query language, that means, it is user-oriented and different evaluation plans with different execution times may exist for a single query. Therefore, it is necessary for its execution to select an evaluation plan. An evaluation plan is a sequence of operators that are evaluated on data in certain order to retrieve a set of tuples that satisfies a query. These operators correspond to Relational Algebra operators [14].

Both the evaluation order and suitable operator selection influence query execution time. This influence is consequence of the differences in the number of tuples that are returned by each operator. A bad selection of an evaluation plan produces long execution time of the query and high memory use. Optimal evaluation plan computation is a combinatorial problem [1]. In practice, to determine an optimal evaluation plan might take a longer time than to evaluate a suboptimal one. Therefore, DBMSs use heuristics that allow to get adequate plans, or at least to avoid the worst.

On the other hand, preference queries emerge as a new paradigm for specifying user criteria [5]. They are not intended for strict constraints on data, but they define certain desires or criteria that data must satisfy. So, a preference query retrieves a set of interesting tuples for the user. For example, an user could want to get information about which are the hotels cheaper and closer to the beach. This query may be specified in SQL by means of the ORDER BY clause [7], ordering price and distance ascendly; however, the user must be aware of filtering the tuples that satisfy their preferences. Hence, SQL was extended with a new clause: Skyline of [2]. Skyline allows to express user-preferences in terms of a multicriteria function defined by the user. Multicriteria functions induces partially ordered set.

Time complexity for answering Skyline queries is high and depends on the size of the input and the number of multicriteria function probes performed [1]. Many solutions have been proposed for improving Skyline evaluation time [2, 13, 7]. A Skyline query can be translated into a SQL query; however this strategy is expensive. A better strategy has been to implement Skyline as an operator [2] and to combine it with the rest of SQL operators. Even though Skyline evaluation is expensive, it may produce less tuples than other operators and reduce query execution time [6].

By its semantic, Skyline is evaluated after the other operators and a single Skyline operator is put at the end of the evaluation plan. But, there exists several algebraic rules that allow to transform this plan into another, where Skyline is intercalated with the other operators [12, 14]. This type of plan could have less cost than a plan composed of a single Skyline operator. Thus, the DBMS may select a better evaluation plan because it considers a wider subset of search space.

Current DBMSs optimize SQL queries using an algorithm based on dinamic programming [15]. Nevertheless, this algorithm is not effective as the size of the search space increases [1]. Thus, it is necessary to define other mechanism in order to select an adequate plan for queries that include Skyline and Join operators (Skyline-Join queries).

In this work, we propose to use an evolutionary algorithm for optimization of Skyline-Join queries, using the algebraic rules defined for them [12, 14]. In order to find a good evaluation plan for a query that contains Skyline and Join operators, the DBMS requires to determine: the Join evaluation order, the evaluation algorithm for each operator and the correct position of Skyline in the plan. To the best of our knowledge, there is no optimization algorithm based on evolutionary algorithms for Skyline queries. We have adopted evolutionary algorithms in this work due to their successful use in the optimization of standard queries [1].

[1] A study of complexity Skyline problem is presented in [11].

2 Skyline Operator

A Skyline query identifies interesting or non-dominated tuples based on a multicriteria function that maximizes or minimizes some attributes. Some tuple a is said to dominate another tuple b, if a is as good or better than b for all attributes, and strictly better than b in, at least, one attribute [11].

The Skyline operator was proposed as a SQL extension to answer queries that satisfy desirable conditions, not necessarily strict on data [2]. In SQL, a Skyline query looks like:

```
Select *
From tabla_1, tabla_2, ... tabla_n
[Where ... ]
Skyline Of att1 [max|min|diff] [, att2 [max|min|diff], ...] ...
```

A SKYLINE OF clause body represents a set of preferences that contains attributes or dimensions used to rank the dataset. Each dimension can be an integer, float, or a date and may be annotated with the directives: min, max and diff. Min and max indicate minimum or maximum values and the diff directive selects the best choice for each different value of the diff attribute.

3 Query Optimization

Query optimization is the process to select a "good" evaluation plan. A "good" plan is one that has a low estimated execution time. This process consists of three steps. First, several possible plans that describe different execution strategies are studied. These plans are generated considering algebraic rules for each operator. Second, execution costs for the generated plans are estimated. Finally, the least expensive plan is chosen. In this process, just a subset of plans are generated (left-deep strategy), due to the high cost of searching in the complete solutions space.

A plan is a sequence of Relational Algebra operators that will be evaluated in order to retrieve a set of tuples that satisfies an SQL query. One of these operators is the Join. A Join is an operator to select tuples of the Cartesian Product among two tables that satisfy a condition defined by the user. A Join operator is represented by $R \bowtie_{A=B} S$, where R and S are tables, A is an attribute of R, B is an attribute of S and A=B is the Join condition defined by the user.

On the other hand, the DBMS uses estimated costs because data is updated constantly and it is too expensive to calculate actual cost after updating. Thus, for each plan, its cost is calculated using metadata and cost formulas. Cost formulas estimate cost of the Join operator depending of the evaluation algorithm. Block Nested Loop Join (BNLJ), Index Block Nested Loop Join (IBNLJ) and Sort Merge Join (SMJ) are three evaluation algorithms for the Join operator [9, 14].

To estimate cost formulas for Join, the Join Selectivity and the Blocking factor must be calculated. The Join Selectivity is the proportion of Cartesian's

Product tuples that are expected to be retrieved after applying join operator and it is computed as:

$$js = \frac{|(R \bowtie_c S)|}{|(R \times S)|} \tag{1}$$

The blocking factor is the proportion of number of tuples per block. Let b_R be the number of blocks of R; b_S, the number of blocks of S; fbl_{RS}, the blocking factor of $R \bowtie S$; x_B, the index levels and D, the input/ouput cost of a disk block. The formulas in Table 1 allow to estimate the generated plan's cost during optimization process. We only show input/output cost, for more details on processing cost refer to [14].

Join Algorithm	Cost Formula
Nested Loop	$C_{I/O} = [b_r + (b_r * b_s) + \frac{js * \|R\| * \|S\|}{fbl_{RS}}] * D$
Index Nested Loop	$C_{I/O} = [b_R + (\|R\| * (X_B + 1)) + \frac{js * \|R\| * \|S\|}{fbl_{RS}}] * D$
Sort Merge	$C_{I/O} = [b_R + b_S + \frac{js * \|R\| * \|S\|}{fbl_{RS}}] * D$

Table 1. Cost Function for Join

Also, Skyline operator may be integrated with Relational Algebra operators by means of algebraic rules. Hence, it may be included in an evaluation plan. Block Nested Loops (BNL) [2] was one of the first algorithms defined for Skyline evaluation in relational databases. To show the benefits of intercalating Skyline in a plan we are just considering BNL as the algorithm for Skyline implementation.

BNL scans all the table and in consequence, I/O cost function is equal to table cardinality. Let fbl_R be the blocking factor of the input table. BNL's I/O cost is:

$$C_{I/O} = (fbl_R * n) * D \tag{2}$$

Nevertheless, BNL has a high CPU cost in terms of number of comparisons that the algorithm has to probe. This cost was defined in [4] supposing pair-wise independence between values:

$$c_{BNL}(n, P) \approx (\sum_{i=2}^{n} \frac{\hat{S}_{j-1,P}}{j-1} \hat{S}_{j-1,P+1}) * k \tag{3}$$

where P is the cardinality of the set of Skyline dimensions and k is the proccesing cost for a tuple.

Godfrey [10] propose a Skyline cardinality estimation based on the size n of input tables, dimensions d of the Skyline operator and the membership probability of a tuple to be in the answer set. Godfrey supposed pair-wise independence between values and no duplicate values. Thus, Skyline cardinality estimation is:

$$\hat{S}_{d,n} = \frac{1}{n} \hat{S}_{d-1,n} + \hat{S}_{d,n-1} \tag{4}$$

Many authors have studied Skyline query evaluation. However, Skyline query processing time could be reduced if the DBMS generates plans taking into account the posibility of intercalated Skylines. Consider four tables: A, B, C and

D, a join between the four previous tables and a Skyline defined on attributes of A and C by A.a max, C.c max. By Skyline semantic, we can build the following plan: Skyline Of A.a max, C.c max$(((A \bowtie B) \bowtie C) \bowtie D)$ with the Skyline as the last operator to evaluate, we refer this as canonical plan. This plan represented as a left linear tree [2] is shown in 1. If table A has a great number of tuples and Skyline size [3] is small, this plan will be expensive because many tuples will be processed for each Join. However, the DBMS may push down Skyline, reducing the number of tuples evaluated at the beginning and therefore, the plan's cost. Finally, a better plan may be generated as: Skyline Of A.a max, C.c max$((((\text{Skyline Of A.a max}) \bowtie B) \bowtie C) \bowtie D)$.

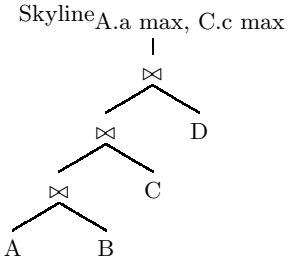


Fig. 1. Left linear tree for the Skyline-Join query

3.1 Algebraic Rules

Different algebraic rules have been defined for the Join and Skyline operators. Algebraic rules allow to keep equivalences between Relational Algebra and SQL queries. Therefore, it is possible to get a Relational Algebra query equivalent to an SQL query. In case of Skyline, this operator is integrated with the Relational Algebra operators.

Given R and S two relations. We denote Skyline$_{[p]}(R)$ as a Skyline defined on a relation R, characterized by a set of preferences p. Let P_r be the subset of preferences defined on R and P_s the subset of preferences defined on S. Some of the algebraic rules [12] necessary to this work are shown in Table 1. Condition C_0 is true if each tuple of R joins with at least one tuple of S and c is the Join condition $R.x = S.x$.

[2] Many DBMS generate left linear trees. A left linear tree is an expression tree, where the right (inner) child of every Join node represents a single table in the query.
[3] Operator size refers to the number of retrieved tuples

$\text{Skyline}_{[P_r]}\,(\text{R} \bowtie_c \text{S}) = \text{Skyline}_{[P_r]}\,(\text{R}) \bowtie_c \text{S} \qquad \textbf{if } C_0$
$\text{Skyline}_{[P_r]}(\text{R} \bowtie_c \text{S}) = \text{Skyline}_{[P_r]}(\text{Skyline}_{[P_r \cup \{\text{R.x diff}\}]}(\text{R}) \bowtie_c \text{S})$
$\text{Skyline}_{[P_r \cup P_s]}(\text{R} \bowtie_c \text{S}) = \text{Skyline}_{[P_r \cup P_s]}(\text{Skyline}_{[P_r \cup \{\text{R.x diff}\}]}(\text{R}) \bowtie_c \text{S})$

Table 2. Algebraic Rules for Skyline

4 Evolutionary Algorithm Design

Evolutionary algorithms (EA) are population based search techniques inspired by the natural selection process [8]. They have been widely used for optimization in diverse domains.

To characterize an EA is necessary to describe its basic elements: the representation of individuals (or chromosome encoding), the fitness function, operators for selection and replacement, and variation operators (mutation and crossover). Here we present our design of these elements for the Skyline-Join query optimization problem.

4.1 Chromosome Encoding

Each individual is represented as a gene array of variable length, this representation is an extension of the used in [1]. Each gene represents an operation. An operation involves an operator, Join or Skyline, the tables and the attributes used in the operation. Each individual denotes a left-deep strategy, represented as a left linear tree. These trees are translated in the array through using a pre-order traversal.

$J^{BNLJ}_{A,B}$	$J^{SMJ}_{B,C}$	$J^{BNLJ}_{B,D}$	$S_{A.a\ max,\ C.c\ max}$

Fig. 2. Encoded individual

Figure 2 shows the resulting code of applying this process to the plan presented in figure 1. $J^{BNLJ}_{A,B}$ is a Join operator implemented as $BNLJ$ between tables A and B. To simplify the notation we assume that all Join conditions are over the attribute x. $S_{A.a\ max,\ C.c\ max}$ is a Skyline that maximizes attributes a and c from tables A and C respectively.

4.2 Selection and Replacement

On each iteration we select the 20% of the population to be mutated. This selection is performed uniformly random over the previous population. Resulting individuals from mutation replace the original ones. The best plan in the population is preserved, i.e., our algorithm is elitist.

4.3 Fitness Function

The individual's fitness is the estimated cost of the plan represented by the chromosome. This cost is calculated using the following equation:

$$Fitness = Cost_{plan} = \sum_{i=1}^{n} Cost(gene_i) \quad (5)$$

Where n is the chromosome length and $Cost(gene_i)$ is the cost of the operation represented by $gene_i$. Gene's estimated cost is calculated using the cost models presented in Section 3. It is important to notice that an operator's cost depends on the results of the application of its previous operators.

4.4 Mutation

Since standard queries genetic optimization works only with Join operators [4], it does not require major validations on the individuals after mutations and crossovers [1]. However, the Skyline operator is not commutative nor associative. Thus, to avoid a great amount of validations to generate valid individuals [5], we are just considering mutation as variation operator.

Individuals can be affected by two kinds of mutations: complete and partial. Complete mutations affect all Join and Skyline operators while partial mutations just affect Joins.

There were some proofs over mutation rates, the best value was 20%, because it gave the best tradeoff between population's diversity and execution time. As mutation was the only mechanism used to generate new individuals, we expected a relatively high mutation rate.

For each individual a mutation operator is selected. We implement three mutation operators: *JoinOrderMutation*, *JoinImplementationMutation* and *Skyline Mutation*. The probabilities of selection of these mutation operators are 50%, 10%, and 40% respectively. Each one of these operators modifies individuals in a different way: changing the query operators order, modifying or adding new genes to the chromosome.

JoinOrderMutation For a given individual (plan), P_1, we select randomly two Join genes. If there not exist an Skyline between them and the result of the left child of the first Join (from left to right) contains the first relation of the second Join, then both Joins positions are swapped to produce a new plan P_2. Figure 3 shows a JoinOrderMutation between Joins at positions 1 and 2. Restrictions over the swapping are check in order to generate valid individuals.

JoinImplementationMutation This operator randomly selects a Join gene and changes its implementation for a different one. The Join implementation could be: Block Nested Loop Join (BNLJ), Index Nested Loop Join (INLJ) or

[4] Join operators are associative and commutative

[5] those semantically equivalent to the original query

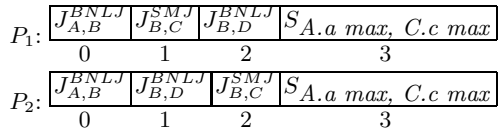


Fig. 3. Example of *JoinOrderMutation* operation

Sort Merge Join (SMJ). This mutation has the smaller probability of selection because we are more interested in the population's changes introduced by the skyline's mutation.

Figure 4 shows an example of the *JoinImplementationMutation* operator. From the plan P_1 the Join at position 0 is selected to be mutated, then its implementation as BNLJ is replaced by SMJ to produce P_2.

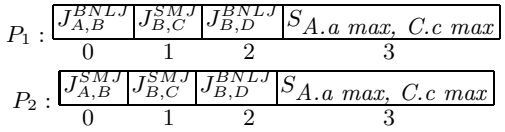


Fig. 4. Example of *JoinImplementationrMutation* operation

SkylineMutation This operator selects randomly an Skyline gene that could be mutated. The selected gene is modified using one of the algebraic rules for Skylines and Joins presented in Subsection 3.1. This operator can transform an Skyline operator by creating a new Skyline gene with a condition that is a sub-condition of the original Skyline. Then, the new gene is pushed into the chromosome. Pushing is done by adding the new Skyline gene at the position where all the involved tables have appeared. Figure 5 shows an example of the *SkylineMutation* operator. We select the Skyline at position 3 from P_1, then from the random selection of algebraic rules we got the third rule of the table 2. Then to form P_2 we will aggregate a new Skyline at the first chromosome position.

4.5 Population Initialization and termination condition

The initial population is filled with canonical individuals. A canonical individual takes the n Joins for the initial query and put them as the first n genes of the chromosome. At gene $n+1$ is located the Skyline defined for the initial query.

Population sizes are proportional to the number of relations of the query (10 times). The search space depends on the possible valid combinations of Joins and Skylines.

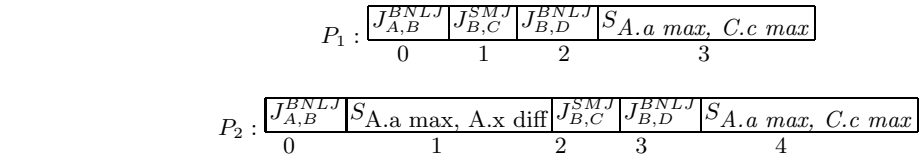


Fig. 5. Example of *SkylineMutation* operation applying table's 2 third rule with the substitution R:=A,S:=C,P_r:=A.a max, P_s:=C.c max

We use two termination conditions: arriving to 1000 iterations or detecting that the best individual hasn't change in 60 iterations.

5 Experiments and Results

In order to cover a wide range of query conditions, to check of our results, we generated queries with different charateristics. Each query is characterized by four variables: number of dimensions, skyline's spreading, cardinality (tuples per table) and Join's selectivity.

For number of dimensions we consider three categories: small (4), medium (8), and large (12). Skyline's spreading values varies according to number of dimensions. For small dimensions the spreading is over 2 and 4 tables, for medium is over 2 and 8 tables, and for large is over 2 and 10. The cardinality can be small (100) or large (1000000). We considered three cardinality combinations: small for tables involved in the Skyline and large for other tables, large for tables involved in the Skyline and small for others, and large for all tables. For Join's selectivity we used the following ranges: $[1.0, 1.0]$, $[0.8, 0.9]$, $[0.4, 0.6]$ and $[0.1, 0.2]$.

For our experiments we generated ramdomly 72 chained queries[6] over 10 tables[7]. Each query is obtained by combination of the possible values of these variables.

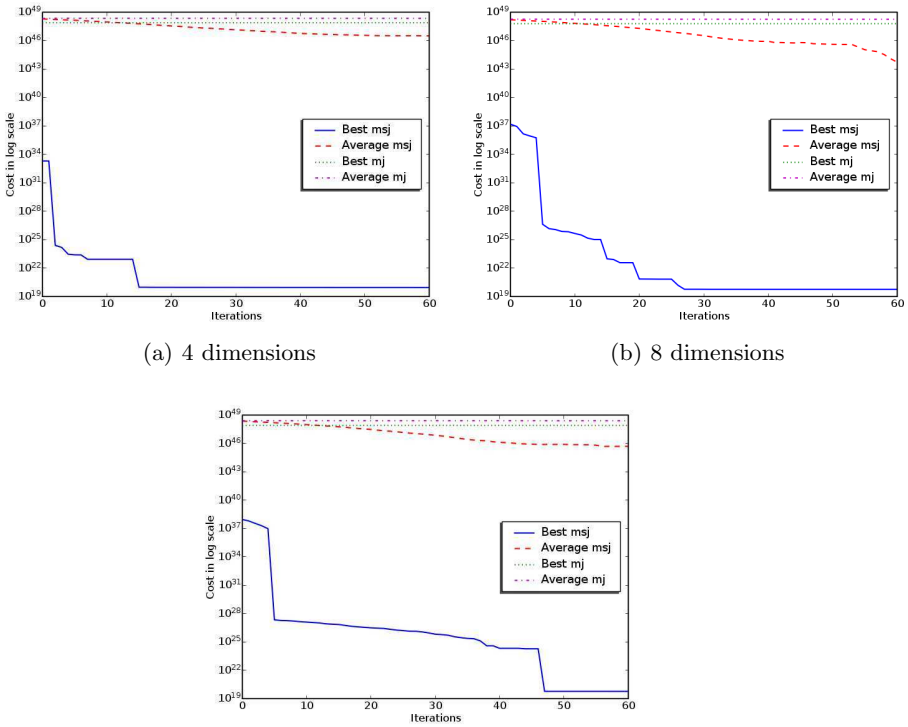
For each query we made 40 runs: 20 with partial mutations and 20 with complete mutations. Partial mutations only affect Joins' arrangement while the Skyline operator position is fixed; the Skyline is the last operator in being evaluated. Complete mutations affects Skylines and Joins arrangement. Showed results are the values obtained of averaging over the 72 queries (20 runs each).

To study how the EA works over different number dimensions we split our queries in three categories, with 24 queries on each. Figure 6 shows the dynamics of plans's cost evolution for each one of our dimensions categories. This figure presents the best plan's cost and the average of the population cost for each kind of mutation. Complete mutations are denoted as msj while partial mutations (that do not perform the *SkylineMutation*) are denoted as mj.

We observe how our algorithm, when the SkylineMutation was actived, found a "good" plan in few iterations; being this plan sufficiently good for the DBMS's

[6] Queries where Joins are linearly chained
[7] 10 tables than involve 10 Joins, wich is considered a large number of Joins

execution time requirements. Even when the number of iterations needed to get a good plan was larger as dimensions increased; for the longest dimensions studied they where less than 50. For all dimensions, the best found plans are in average almost 15 orders of magnitude cheaper than the best plan in the initial population and 30 orders of magnitude cheaper than the initial population average.



(a) 4 dimensions

(b) 8 dimensions

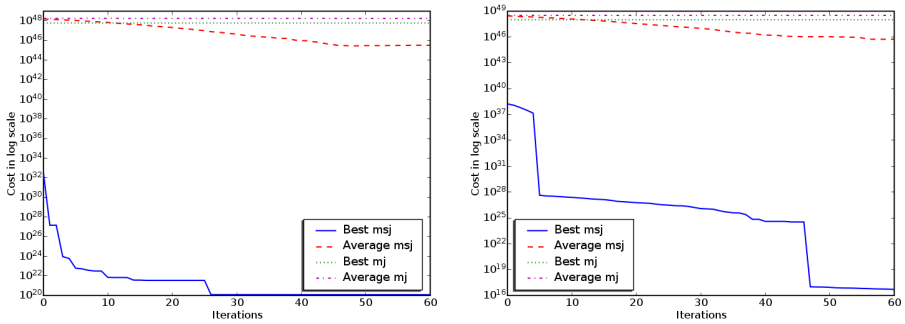
(c) 12 dimensions

Fig. 6. Dynamic view of the effect of number of dimensions over cost's evolution

Figure 7 shows the effect of the skyline spreading over queries with 12 dimensions. Figure 7(a) shows the cost evolution for a Skyline-Join query with the 12 dimensions and spreading of 2. That means, the Skyline is defined over attributes of two tables. The same way for the figure 7(b).

It can be observed that for a wider spread the algorithm took more time to found a good solution, but this solution was better (4 orders of magnitude).

To observe the convergence of the algorithm in the long term, we took the worst-case query and ran it for 1000 iterations. Figure 8 shows that with a long



(a) over 2 tables (b) over 10 tables

Fig. 7. Effects of the skyline spreading over cost's evolution

run the algorithm with complete mutations not only found a good solution, but the whole population converged. Nevertheless, when the algorithm did not use the *SkylineMutation*, there were little improvement on the best individual and on the population's average.

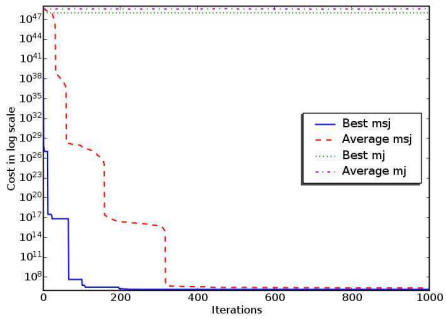


Fig. 8. Cost evolution in the long term (1000 iterations

6 Conclusions and Future Work

We have proposed the use of an EA for the Skyline-Join query optimization problem. Our results show that our EA produces "good" plans in few iterations. As quality of plans depends on the adequacy of the cost model, it is necessary to test our EA on actual data to check that the plan selected by the algorithm really has a good evaluation cost. Experiments also show that if we only use Join mutations, cost's improvements are poor. But, if we use Skyline mutations along

Join mutations, our EA found "good" plans in few iterations. This observation implies that plans with intercalated Skylines are better than canonicals plans (those whose Skyline is at the root of the evaluation tree). Also, we noticed that our algorithm converges faster in queries where the Skyline involves few tables.

As we got good results with this approach, we are implementing a version of this algorithm into a real DBMS (PostgreSQL). An extension of our algorithm is to include several evaluation techniques for the Skyline operator. Also, we plan to compare our algorithm against other optimization algorithms for Skyline-Join queries.

Acknowledgments

This research was supported in part by FONACIT under Grant G-2005000278, and by DID of the Universidad Simn Bolvar.

References

1. K. Bennett, M. Ferris, and Y. Ioannidis. A genetic algorithm for database query optimization. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 400–407, 1991.
2. S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *IEEE Conf. on Data Engineering*, pages 421–430, 2001.
3. Date C. *A guide to the SQL standard (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
4. S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. *ICDE*, page 64, 2006.
5. Jan Chomicki. Preference queries. *CoRR*, cs.DB/0207093, 2002.
6. Jan Chomicki. Semantic optimization techniques for preference queries. *Inf. Syst.*, 32(5):670–684, 2007.
7. Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.
8. A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.
9. R. Elmasri and S. Navathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings, 1994.
10. P Godfrey. Cardinality estimation of skyline queries. Technical Report CS-2002-03, York University, 2002.
11. P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 229–240, 2005.
12. B. Hafenrichter and W. Kießling. Optimization of relational preference queries. In *ADC '05: Proceedings of the 16th Australasian database conference*, pages 175–184, 2005.
13. Tan K., Eng P., and Ooi B. Efficient progressive skyline computation. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 301–310, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
14. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
15. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. pages 141–152, 1998.