UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Informe de Proyecto de Grado

Uso de aceleradores de hardware en sistemas de Bases de Datos relacionales

Autores:
Alejandro Barreiro
Anthony Cabrera

Tutores:
Lorena Etcheverry
Pablo Ezzatti

Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Diciembre de 2019

Resumen

En los últimos años el uso de co-procesadores gráficos, o GPUs por su sigla en inglés, para acelerar la resolución de problemas de propósito general ha ido en aumento. Estos dispositivos han fomentado la evolución del uso de arquitecturas de cómputo homogéneas y con un solo procesador a plataformas de hardware heterogéneas y masivamente paralelas. Los motores de bases de datos no son ajenos a esta tendencia. Notar que los sistemas manejadores de bases de datos (DBMS) requieren de grandes volúmenes de cómputo. Los manejadores que aprovechan las GPUs, conocidos como GDBMS, en general sacan partido de las bondades de estos dispositivos en contextos de consultas que implican grandes volúmenes de datos.

En este trabajo se relevan GDBMS existentes, analizando oportunidades de mejora, priorizando aquellos GDBMS que sean capaces de incorporar técnicas de aprendizaje automático para determinar qué dispositivo utilizar en cada contexto. En particular, en este proyecto se propone trabajar sobre un manejador existente (CoGaDB), y más específicamente en una versión que fue extendida en un proyecto de grado anterior, con el fin de mejorar el aprovechamiento de los recursos ociosos (y, por tanto, reducir los tiempos de ejecución de consultas con altos requerimientos de cómputo). El principal foco del trabajo es el desarrollo de una heurística para automatizar la distribución del trabajo (y datos) entre los diferentes dispositivos de cómputo disponibles en la plataforma de ejecución para consultas del tipo *Join*. La herramienta desarrollada se basa en el uso de técnicas de aprendizaje automático. La evaluación experimental muestra que, para consultas que involucran columnas con gran cantidad de valores, la utilización del *Join* híbrido concurrente y el uso de la heurística formulada es fundamental para la obtención de mejores tiempos de ejecución, así como para alcanzar un mejor aprovechamiento de los recursos disponibles.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Las bases de datos tienen una gran relevancia en los sistemas informáticos, y el desempeño de estos sistemas está fuertemente marcado por su capacidad de responder en tiempo y forma a las consultas que sobre él se realicen. En particular, y en el contexto del procesamiento de grandes volúmenes de información o *big data*, la necesidad de mejorar el desempeño de las consultas lleva a explorar nuevas arquitecturas que exploten, entre otros aspectos, la naturaleza paralelizable de algunas de las subtareas vinculadas al procesamiento de dichas consultas. En los últimos años, la utilización de Unidades de Procesamiento Gráfico (GPUs) ha evolucionado y aparecen como una alternativa ideal para el procesamiento de operaciones paralelizables. En este contexto, parece razonable adaptar el procesamiento de consultas del sistema de bases de datos a estos procesadores *many-core* para lograr el máximo rendimiento.

En este proyecto, se continúa con el trabajo realizado en el proyecto de grado realizado por Acuña y Parula [3]. En el mismo se abordó el estudio de la utilización de las GPUs como aceleradores de hardware de manejadores de base de datos. Dicho proyecto realizó un análisis de los manejadores disponibles que utilizan GPUs para acelerar su procesamiento, del que se desprende que existen tres enfoques diferentes de utilización de las GPUs. Un primer enfoque donde todo el procesamiento es realizado en GPU, utilizando la CPU para despachar las consultas y obtener los resultados. El segundo, es un enfoque híbrido en el que se realiza parte del procesamiento en GPU y parte del procesamiento en CPU. Y un tercer enfoque, llamado enfoque híbrido concurrente que consiste en realizar el procesamiento en paralelo en la GPU y en la CPU. El objetivo del proyecto consistió en minimizar el tiempo de recursos ociosos,

reduciendo a su vez los tiempos de ejecución de las consultas en general, por lo que se extendió una herramienta existente para incorporar el enfoque híbrido concurrente. En dicho proyecto se logró un mejor uso de los recursos, obteniendo como resultado mejores tiempos de ejecución y se plantearon varias oportunidades de mejora a futuro, que son de interés para abordar en el marco de este proyecto.

1.1. Objetivos del proyecto

El objetivo principal de este proyecto es actualizar el estado del arte en cuanto al uso de GPUs para implementar bases de datos relaciones. En primera instancia, se extiende el relevamiento a los nuevos manejadores de bases de datos que utilizan GPUs posteriores al año 2014, fecha en la cual fue realizada la comparación en el proyecto de grado presentado por Acuña y Parula [3]. En particular, se comparan dichos manejadores con la herramienta CoGaDB, que fue la base del trabajo anterior. Del análisis de las herramientas actuales, se decidió continuar extendiendo la herramienta CoGaDB con el objetivo de identificar estrategias que permitan mejorar el desempeño computacional de la herramienta, tales como compresión de datos, planificación eficiente de qué operadores y datos ejecutar en cada co-procesador utilizando métodos de aprendizaje estadístico, e incluso la utilización simultánea de múltiples GPUs.

1.2. Estructura del documento

Se detalla a continuación la estructura del documento. En el Capítulo 2, se incluyen los conceptos preliminares necesarios para el posterior desarrollo del documento, abordando conceptos fundamentales de sistemas de bases de datos relacionales, conceptos básicos de las GPUs y se realiza una breve introducción de los métodos de aprendizaje estadísticos. En el Capítulo 2 también se describe el uso de GPUs por parte de los GDBMS, y se realiza un relevamiento de los sistemas existentes a la fecha, en el que se analizan y comparan ciertos aspectos importantes para el proyecto.

En el Capítulo 3, se describe CoGaDB, el GDBMS utilizado en el marco de este proyecto. Se detalla su arquitectura y cómo éste genera el plan lógico asociado. También se describen los conceptos fundamentales del optimizador

físico, el uso de modelos de aprendizaje y de distintos criterios de optimización para generar el plan físico con el menor tiempo de ejecución. Finalmente se listan los problemas existentes en el GDBMS elegido, junto con la elección de uno de ellos para su abordaje posterior. En el Capítulo 4, se describe el problema abordado: la aplicación de técnicas de aprendizaje estadístico para la distribución del cómputo en CoGaDB. Se detalla el diseño y la implementación de la solución propuesta, los casos de prueba utilizados junto a los resultados obtenidos en la evaluación experimental. Finalmente, en el Capítulo 5 se detallan las conclusiones y se identifican posibles desafíos a futuro.

Capítulo 2

Conceptos preliminares

En este capítulo se resumen conceptos que son la base para el desarrollo del proyecto. En primer lugar, se introducen conceptos relevantes referentes a las bases de datos relacionales, definiendo el modelo relacional y sus principales operaciones. También se introducen conceptos de las bases de datos columnares, cuyo modelo de almacenamiento es el utilizado por CoGaDB. Luego, se definen los principales conceptos de las GPUs y se avanza en cómo su arquitectura permite obtener mayor capacidad de procesamiento, ejecutando programas utilizando el framework CUDA. Finalmente, una breve introducción a los métodos de aprendizaje estadísticos, los cuales son uno de los principales pilares de esta investigación.

2.1. Bases de datos relacionales

Las bases de datos relacionales son un tipo de base de datos que cumple con el modelo relacional. El modelo relacional representa una base de datos como una colección de relaciones. Una relación puede ser pensada como una **tabla**, cuyas columnas corresponden a **atributos** de tipo atómico, y cada fila representa una colección de valores relacionados. Un **dominio** D es un conjunto de valores atómicos. Un **esquema de relación** R denotado por $R(A_1, A_2, ..., A_n)$, está constituido por un nombre de relación R y una lista de atributos $A_1, A_2, ..., A_n$, siendo D el dominio de A_i. Una **relación** r del esquema $R(A_1, A_2, ..., A_n)$ es un conjunto de tuplas, donde cada **tupla** t es una lista ordenada de n valores que representan un elemento del dominio D o un valor especial NULL.

El **álgebra relacional** es el conjunto de operaciones básicas que permiten al usuario especificar peticiones al modelo relacional. Algunas de sus operaciones principales son las siguientes:

- **Selección** (*Select*): Se emplea para seleccionar un subconjunto de las tuplas de la relación que satisfacen una condición de selección.

$$\sigma < condicion > (R)$$

- **Proyección** (*Project*): Se emplea para seleccionar ciertas columnas de la tabla y descarta otras. Si solo se está interesado en ciertos atributos de la relación, se usa la proyección para obtener una relación sobre esos atributos. La forma general de la operación proyección es:

$$\pi < atributos > (R).$$

- **Unión** (*Union*), **Intersección** (*Intersection*), **Menos** (*Minus*): Grupo de operaciones de álgebra relacional correspondientes a la operativa matemática sobre conjuntos. $R \cup S$ es una relación que incluye todos las tuplas que están tanto en R como en S. $R \cap S$ es una relación que incluye todas las tuplas que están en R y en S. $R - S$ es una relación que incluye todas las tuplas que están en R pero no en S.
- ***Join***: Operación empleada para combinar tuplas relacionadas de dos relaciones en una sola. La forma general de un *Join* de dos relaciones $R(A_1, A_2, ..., A_n)$ y $S(B_1, B_2, ..., B_n)$ es:

$$R \bowtie_{<condiciondeconexion>} S.$$

  El resultado de un *Join* es una relación Q que tiene una tupla por cada combinación de éstas (una para R y otra para S) siempre que dicha combinación satisfaga la condición de conexión [5].

2.1.1. Tipos de *Join*

El *Join* se emplea para combinar datos precedentes de múltiples relaciones, de forma que la información pueda presentarse en una única relación o tabla. Estas operaciones también se conocen como *INNER JOIN* y *OUTER JOIN*.

El lenguaje de consulta estructurado (SQL por sus siglas en inglés) implementa el modelo relacional y es considerado estándar en los sistemas manejadores de bases de datos relacionales. En SQL se utilizan los términos de tabla, fila y columna para los términos relación, tupla y atributo definidos previamente en el modelo relacional. El estándar ANSI de SQL define cinco tipos de *Join*: *INNER JOIN*, *LEFT OUTER JOIN*, *RIGHT OUTER JOIN*, *FULL OUTER JOIN* y *CROSS JOIN*.

INNER JOIN

Cómo se definió en la operación *Join* del álgebra relacional, se combinan las tuplas de una tabla R, con las correspondientes de la tabla S, que satisfagan las condiciones que se especifiquen en el predicado del *Join*. La condición de conexión es especificada sobre los atributos de las dos tablas (relaciones) R y S, y es evaluada para cada combinación de tuplas, incluyéndose en el resultado en forma de tupla combinada en caso que la condición evalúe verdadero. A los operadores de comparación de la condición de conexión se les llama θ (*theta-join*). Las tuplas que evalúen falso en la condición de conexión, o sus valores sean *NULL*, no aparecen en el resultado.

Existen distintas variaciones de *Join*, que pueden ser clasificados como de igualdad, naturales y cruzadas.

***EQUI JOIN*:** Cuando el *theta-join* define condiciones de conexión únicamente con comparaciones de igualdad en el predicado del *Join*, recibe el nombre de *EQUI JOIN*.

***NATURAL JOIN*:** Es una extensión del *EQUI JOIN*, utilizada para no incluir en el resultado los atributos de conexión de la segunda relación, ya que son idénticos a los de la primera relación. Esta operación, necesita que ambos atributos de conexión tengan el mismo valor para las dos relaciones, en caso contrario, se debe aplicar en primer lugar una operación de renombrado.

***SEMI JOIN*:** En la ejecución convencional del *Join*, la relación resultante incluye los atributos de conexión de ambas relaciones. En caso de querer obtener en la relación resultante los atributos de una de las relaciones, se puede utilizar el *SEMI JOIN*. Es equivalente a realizar el *Join* de las dos relaciones, seguido por una proyección, que descarta los atributos que no se quiere incluir en el resultado.

OUTER JOIN

A diferencia del *INNER JOIN* en el cual se eliminan las tuplas que no satisfagan la condición de conexión, el *OUTER JOIN* mantiene en el resultado todas las tuplas de R, o de S, o de ambas, es decir, no hay perdida de información ya que no se eliminan tuplas del resultado.

Existen diferentes variantes de *OUTER JOIN*:

- *LEFT OUTER JOIN*: Mantiene cada tupla de la primera relación, o de la relación izquierda R en $R \bowtie S$. Si no se encuentran tuplas en S relacionadas con R, la concatenación resultado se rellena con valores *NULL*.
- *RIGHT OUTER JOIN*: De forma análoga, se mantiene cada tupla de la segunda relación.
- *FULL OUTER JOIN*: Mantiene todas las tuplas de ambas relaciones, en caso de no existir coincidencias, se rellena la concatenación resultado con valores *NULL* [5].

2.1.2. Implementaciones del *Join*

Se expone un breve estudio sobre algunas posibles implementaciones de la operación, dado que es una de las operaciones que más tiempo consume durante el procesamiento de una consulta. Para explicar el funcionamiento de cada algoritmo se usa como ejemplo la siguiente operación:

$$R \bowtie_{A=B} S,$$

donde A y B son atributos compatibles en el dominio de R y S respectivamente.

NESTED LOOP JOIN

Este algoritmo simple de fuerza bruta consiste en un par de bucles anidados, en el cual se obtienen todos los registros r de R (bucle externo), luego se toman todos los registros s de S (bucle interno) y finalmente se agregan al resultado si se satisface que $r[A] = s[B]$.

Un ejemplo de procedimiento que implementa la operación:

$$R \bowtie_{A=B} S,$$

Algoritmo 2.1.1: Pseudocódigo *NESTED LOOP JOIN*

```
1 para cada tupla s ∈ S hacer
2     para cada tupla r ∈ R hacer
3         si s(b) = r(a) entonces
4             Join r y s;
5             agregar a la relación resultado
```

se puede ver en el Algoritmo 2.1.1.

SORT-MERGE JOIN

En este caso, los registros de R y S se encuentran físicamente ordenados por el valor de los atributos de concatenación A y B respectivamente. Cada registro de R se empareja contra el registro de S una única vez. Un ejemplo de método se puede ver en el Algoritmo 2.1.2, el cual reduce la cantidad de comparaciones de tuplas respecto al *NESTED LOOP JOIN*.

Algoritmo 2.1.2: Pseudocódigo *SORT-MERGE JOIN*

```
1 ordenar R según A;
2 ordenar S según B;
3 para cada tupla r ∈ R hacer
4     mientras s(b) < r(a) hacer
5         leer siguiente tupla de s ∈ S;
6         si s(b) = r(a) entonces
7             Join r y s;
8             agregar a la relación resultado;
```

HASH JOIN

En este algorito, los registros de R y S se encuentran clasificados en el mismo hash, utilizando la misma función de dispersión sobre los atributos de A de R y B de S como claves de dispersión. Esta implementación reduce la cantidad de comparaciones que se realizan entre tuplas, ya que se compara tuplas de S con un conjunto limitado de R con las que se les podría hacer *Join*. Un ejemplo de implementación del *Hash Join* se puede ver en el Algoritmo 2.1.3. [5].

Algoritmo 2.1.3: Pseudocódigo *HASH JOIN*

```
1  para cada tupla s ∈ S hacer
2      hash sobre s(b);
3      guardar tuplas en la tabla hash;
4  para cada tupla r ∈ R hacer
5      hash sobre r(a);
6      si la función de hash apunta a tuplas de s entonces
7          para cada tupla s de las obtenidas hacer
8              si s(b) = r(a) entonces
9                  Join r y s;
10                 agregar a la relación resultado;
```

Como se verá en el Capítulo 4, el *Hash Join* será el algoritmo utilizado en el desarrollo de este proyecto.

2.1.3. Bases de datos columnares

Una de las ventajas de las bases de datos columnares sobre las tradicionales es la compresión de los datos [6]. Para una misma columna los datos son del mismo tipo y los valores posibles difieren en menor cantidad a comparación con las bases de datos tradicionales donde cada tupla tiene la información de todas sus columnas. Esto permite a los algoritmos de compresión ser más efectivos sobre datos que se almacenan en columnas.

Cada columna puede tener su propio método de compresión eligiéndose el más eficiente dependiendo de ciertas características de los mismos como lo son el ordenamiento de la columna, el tipo de datos, o incluso tener en cuenta la cantidad de valores distintos de la misma.

Los métodos de compresión más utilizados en la actualidad por las DBMS de tipo columnar son las siguientes:

- *Run-length Encoding (RLE)*: consiste en comprimir el mismo valor en una columna en una única representación. Este tipo de compresión es útil en columnas que han sido ordenadas por valor y tienen una cantidad razonable de valores repetidos. La idea consiste en sustituirlas utilizando una tripleta que indica el valor que se esta representando, la posición de comienzo y la cantidad de ocurrencias.

- *Bit-Vector Encoding*: consiste en almacenar cada valor una única vez, representado como una colección de bits de tamaño fijo la cantidad de elementos de la columna, donde se tiene un 1 en la posición i-esima si ese valor existe en la posición i-esima de la columna original o un 0 en caso contrario. El mismo es útil cuando las columnas tienen una limitada cantidad de valores posibles.
- *Dictionary*: consiste en reemplazar cada valor de la columna por un único entero que lo identifique y generar una estructura de datos de tipo diccionario almacenada en una tabla aparte la cual mapea este identificador con los correspondientes valores. La técnica provee compresión dado que los códigos tienen tamaño fijo y, en general, son de menor largo que el valor real. A su vez este método permite comprimir con otra técnica el listado de identificadores resultantes.
- *Frame Of Reference (FOR)*: consiste en representar los valores de una columna como una constante más un valor. Por ejemplo la secuencia de valores 1003, 1001, 1007 puede ser representado, como la constante 1000 y luego la suma de los valores 3, 1, 7. Este método es útil para tipos de datos cuyos valores posibles no es amplio.

Operación con datos comprimidos

Las técnicas de compresión vistas anteriormente permiten en muchos casos operar directamente sobre los datos comprimidos, lo cual mejora en gran medida la performance de las consultas producto de no pagar el costo de descompresión de datos. Un ejemplo de esta mejora se ve en el método RLE, en el cual el operador de agregación SUM de SQL puede ser ejecutado multiplicando el valor de la tripleta por la cantidad de ocurrencias sin necesidad de descomprimir cada valor para realizar el conteo. Otro ejemplo ocurre para el caso de compresión de Diccionario cuando este preserva el orden de la codificación, es decir, si un código *c1* es mayor a otro, significa que el valor asociado a *c1* también es mayor. Nuevamente esto permite que operaciones de filtrado, sean realizadas directamente sobre los datos codificados, es decir, sin la necesidad de descomprimirlos (a lo sumo codificando los valores utilizados como filtro). Lo mismo ocurre en operaciones como *Joins*, producto que la mayoría de los *Joins* son del tipo igualdad en claves y claves foráneas, y estas se rigen sobre el mismo dominio de valores, los *Joins* sobre claves con valores comprimidos

dará los mismos resultados que un *Join* normal sobre valores de claves descomprimidas [7]. Resultados sobre experimentos en la literatura demuestran que la compresión es importante tanto para ahorrar espacio como para la mejora de performance de las consultas. Sin embargo, sin realizar operaciones sobre datos comprimidos las ventajas en desempeño decrecen en órdenes de magnitud, principalmente en CPU, donde el costo de cómputo de la descompresión, es también costoso. Además la mayoría de las mejoras se puede ver especialmente en columnas ordenadas [6].

2.2. Tarjetas Gráficas (GPUs)

En esta sección, se proporciona una breve descripción de la arquitectura de las tarjetas gráficas y el modelo de programación que propone NVIDIA para explotar el poder de cómputo de dichos dispositivos.

2.2.1. Arquitectura de una GPU

La Figura 2.1 muestra la arquitectura de un sistema con una CPU multiprocesador y una GPU. Las GPUs cuentan con varios multi-procesadores (*Stream-Multiprocessors*) que consisten en un número fijo de procesadores escalares (*Streaming Processors*) el cual depende de la arquitectura y un chip de memoria compartida. El multiprocesador está diseñado para ejecutar cientos de subprocesos simultáneamente, administrando una gran cantidad de hilos siguiendo una arquitectura SIMT (*Single Instruction, Multiple-Thread*). La GPU también dispone de una memoria global que típicamente tiene una capacidad de entre 2 a 16GB. Se comunica con la CPU por medio de un bus, como por ejemplo QPI y *PCIExpress*, los cuales varían en el ancho de banda que ofrecen para la transferencia de datos, el QPI admite una velocidad de transferencia de 25,6 GB/s y el *PCIExpress* de 16 GB/s aproximadamente [1].

2.2.2. Programación en GPU

La computación de propósito general en unidades de procesamiento gráfico es un modelo de programación que aprovecha el motor de ejecución con alto nivel de paralelismo, y permite resolver diversos problemas complejos de computación de una manera más eficiente que en una CPU. El mayor reto de

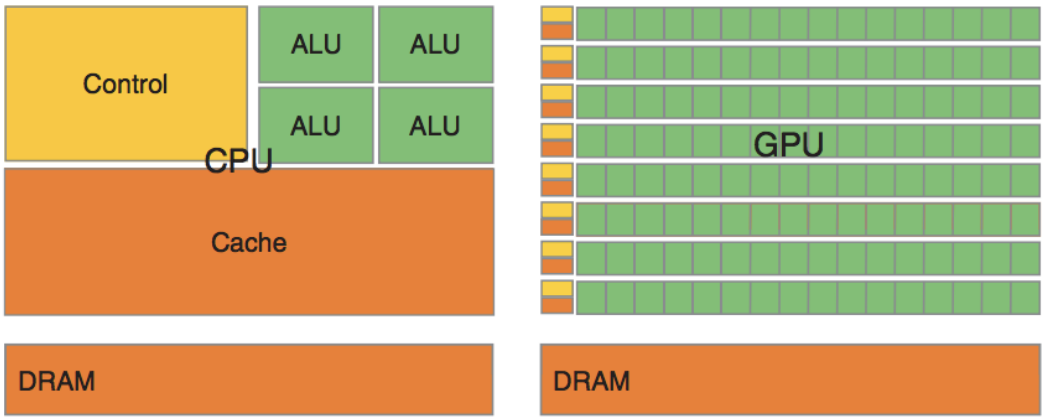

Figura 2.1: Diferencias en la Arquitectura de una GPU y CPU. Extraído de [1].

desarrollar una aplicación para la GPU es aprovechar el alto grado de paralelismo, y el ancho de banda de la memoria en las GPU. Para ejecutar programas en GPU se dispone principalmente de dos frameworks: CUDA y OpenCL. Ambos frameworks implementan una API para interactuar con los *kernels* desde el *host*. CUDA fue introducido por NVIDIA. El diseño de CUDA contiene tres puntos claves: la jerarquía de grupos de hilos, la memoria compartida y las barreras de sincronización [8].

2.3. Métodos de aprendizaje estadísticos

Resulta de particular interés realizar una breve introducción sobre los métodos de aprendizaje estadístico, ya que como se verá en la Sección 3.3, es un concepto clave que se aplica con el objetivo de realizar estimaciones para optimizar el plan físico de una consulta.

El problema de aprendizaje consiste de un conjunto de variables independientes $(X_1, \cdots, X_n)$ también llamadas características, por medios de las cuales se quiere predecir una variable Y dependiente. La dependencia entre la respuesta y las características se formaliza por medio de una función f que describe la relación de dependencia existente entre ellas [9].

$$Y = f(X_1, \cdots, X_n) + \epsilon_i$$

La función f es desconocida y necesita ser estimada, con el fin de lograr predecir nuevas respuestas Y para características conocidas, cometiendo un error ϵ_i. Para poder predecir Y a partir de X, es necesario estimar la función

f desconocida, su estimación se denota $\hat{f}$ y las respuestas estimadas como $\hat{Y}$. Por lo que, la función real resulta:

$$f(X) = \hat{f}(X) + \epsilon_r.$$

Los métodos de aprendizaje estadísticos buscan por medio de mecanismos reducir el error de forma que la función estimada sea lo más parecida a la función objetivo.

2.3.1. Métodos de Regresión

El objetivo de regresión consiste en aprender una función $\hat{f}(X)$ que predice, de manera precisa, una respuesta Y para cada observación (X, Y) llamada conjunto de entrenamiento, lo que se busca entonces es $Y \approx \hat{f}(X)$. Estos se pueden organizar en métodos paramétricos y no paramétricos.

Los métodos paramétricos realizan suposiciones sobre la forma de la función f y se configuran en dos pasos bien definidos. En el primer paso se define la forma de la función que se quiere aprender, luego en una segunda etapa se requiere ajustar la función a los datos de entrenamiento estimando los coeficientes de la misma. El problema de estimar la función f pasa a ser un problema de estimar los coeficientes de una función cuya forma es conocida.

Por otro lado, los métodos no paramétricos, no realizan suposiciones sobre la forma de la función a estimar. Por lo tanto evitan el error proveniente de seleccionar una forma de f que no sea la correcta para la realidad dada, dando más flexibilidad y por tanto teniendo en general una mayor precisión en las predicciones realizadas. Este tipo de métodos suele obtener mejores resultados a medida que el tamaño del conjunto es mas grande.

Dentro de los métodos paramétricos se destacan la regresión lineal y la regresión de K-vecinos más cercanos. La regresión lineal es un método de tipo paramétrico cuyo modelo es lineal al conjunto de observaciones, es decir que el mismo asume que las n variables independientes $X_1, \cdots, X_n$ tienen una relación lineal con la variable dependiente Y. Para cada variable independiente X_i se define un coeficiente β_i, que es el que cuantifica esta relación, la forma funcional para n variables independientes se formaliza como:

$$Y = \beta_n X_n + \cdots + \beta_1 X_1 + \beta_0 + \epsilon.$$

Siendo ϵ el error cometido. Para ajustar el modelo al conjunto de entrenamiento se debe estimar cada coeficiente β_i. La función f estimada $\hat{f}$ se define como:

$$\hat{f}(X) = \beta_n X_n + \cdots + \beta_1 X_1 + \beta_0.$$

Para estimar estos coeficientes se pueden usar distintos métodos como son: método de mínimos cuadrados, descenso por gradiente, regularización, etc.

El método K-vecinos más cercanos asume que todas las instancias observadas corresponden a puntos en $\Re^n$, y que cuanto más cercanas sean dos instancias entre si, más similares serán entre ellas [10].

Para hallar la distancia entre dos instancias x_i y x_j se utiliza en general la distancia euclidiana aplicada a las características de las mismas. Sea $a_r(x_i)$ la característica r-esima de la instancia x_i, la distancia euclidiana aplicada se define como:

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}.$$

Teniendo definida la función de distancia, el algoritmo consiste en almacenar como conjunto de entrenamiento a la totalidad de las instancias observadas y a partir de ellas predecir una respuesta desconocida $\hat{Y}$, por medio de la búsqueda de las K instancias más cercanas. $\hat{Y}$ puede ser estimada de la siguiente manera:

$$\hat{f}(X) = \frac{1}{K} \sum_{(x_i, y_i) \in N} y_i.$$

2.4. Uso de GPUs en manejadores de bases de datos

Las GPUs permiten acelerar consultas sobre manejadores de basos de datos. El concepto de GPGPU (*General-Purpose Computing on Graphics Processing Units*) aplica también en el campo de los manejadores de bases de datos acelerados por GPUs (GDBMS). Notar que los componentes internos, tales como estructuras de datos, procesamiento de consultas y optimización, están optimizados tradicionalmente para las estrategías de cómputo de las CPUs y no para las GPUs. Las GPUs tienen una arquitectura de hardware diferente, con un

gran número de co-procesadores homogéneos, y para lograr sacar partido de dicho hardware es necesario re-implementar los operadores utilizando el *toolkit* ofrecido por las GPUs, tales como CUDA u OpenCL. Dadas las características de las GPUs, un algoritmo no es necesariamente más rápido en GPU que en CPU, entre otras cosas por el costo de transferencia de datos entre la memoria de la CPU y de la GPU. Hay que utilizar estrategias inteligentes para mitigar dicho sobrecosto presente en el intercambio de datos entre la memoria principal y la memoria de la GPU.

2.4.1. Comparación de GDBMS existentes

A continuación se presenta una actualización del relevamiento de los manejadores de las bases de datos relacionales que utilizan hardware gráfico para acelerar el tiempo de ejecución de las consultas.

Para esto se toma como referencia el estudio realizado por Acuña y Parula [3], en el cual se realiza un análisis de los GDBMS existentes hasta finales del año 2014, tomando como manejador principal del estudio a CoGaDB [11].

Se realiza un breve análisis y comparación sobre los GDBMS implementados posteriores a la fecha, es decir, a partir del año 2015, clasificando sus propiedades y el modo de optimización de consultas. Finalmente se propone una comparación con CoGaDB fundamentando la decisión de continuar la extensión del prototipo utilizando dicha herramienta.

Blazingdb

Desarrollada en el 2015, Blazingdb [12] es una base de datos SQL que utiliza la GPU para la aceleración de consultas. Soporta los operadores *Select*, *Join*, *Group*, *Order*, *Union*, y *Aggregations*.

Alenka

Alenka [13] es un motor de base de datos desarrollado en 2015 basado en bases de datos columnares, que utiliza la GPU para aplicar de forma concurrente una operación a todo el set de datos. Usa comprensión de datos en la GPU, y opera con datos comprimidos. Soporta los operadores *Select*, *Filter*, *Join*, *Group* y *Order*. Fue utilizada por Furst [14] para una prueba de concepto de optimización de consultas sobre el *benchmark* TCP-H.

PG-Strom

PG-Strom [15] es una extensión diseñada para PostgreSQL v9.5 o posteriores, para optimizar las consultas sobre grandes conjuntos de datos haciendo uso de la GPU. Soporta los operadores *Select*, *Filter*, *Join*, *Group* y *Aggregations*. No soporta transacciones.

Comparación con CoGaDB

En esta sección se comparan los DBMS previamente descritos, viendo las principales características de los mismos. En la Tabla 2.1 se presentan por orden cronológico los GDBMS que serán tenidos en cuenta.

Manejador	Año	Código abierto	Columnar	Compresión de datos	Lenguaje en GPU
CoGaDB	2013	Si	Si	Si	CUDA
Blazingdb	2015	No	-	-	CUDA
Alenka	2015	Si	Si	Si	CUDA
PG-Strom	2016	Si	No	No	CUDA

Tabla 2.1: Comparación de CoGaDB y los DBMS posteriores a 2015.

Tanto Alenka como CoGaDB almacenan sus datos de manera columnar, y operan con datos comprimidos. El autor de Alenka no especifica que mecanismo de compresión utiliza, y carece de una documentación extensa o artículos que refieran a dicha herramienta. Respecto a PG-Storm, aparece como una extensión específica para el motor PostgreSQL, el cual tiene un almacenamiento de datos basado en filas. Ghodsnia [16] en su comparación respecto a la conveniencia utilizar almacenamiento por columna o fila para la GPU, concluyó que el almacenamiento por columnas es más adecuado que el almacenamiento por filas. Una de las características más importantes al trabajar con GPU, consiste en reducir el volumen de datos que debe transferirse y, al mismo tiempo, minimizar el tamaño de datos almacenado debido a las limitaciones de memoria existentes en las GPUs. Con bases de datos columnares se puede lograr una tasa de compresión más alta, favoreciendo ambos aspectos.

Los principales argumentos para continuar trabajando sobre la herramienta CoGaDB son la cantidad de artículos académicos que referencian esta herramienta, y las oportunidades de mejora encontradas en el componente de

estimación y planificación de consultas de CoGaDB, continuando así con los avances realizados por Acuña y Parula [3].

Capítulo 3

CoGaDB

En este capítulo se describe CoGaDB, un DBMS orientado a columnas que utiliza la GPU para acelerar consultas. Además, se describe la biblioteca HyPE, utilizada por CoGaDB para obtener el plan físico a partir del plan lógico de la consulta a ejecutar, y estimar tiempos de ejecución de los operadores de la base de datos sin un conocimiento detallado del *hardware* subyacente.

3.1. Arquitectura de CoGaDB

La arquitectura de CoGaDB se compone de cuatro módulos principales, tal cual describe la Figura 3.1.

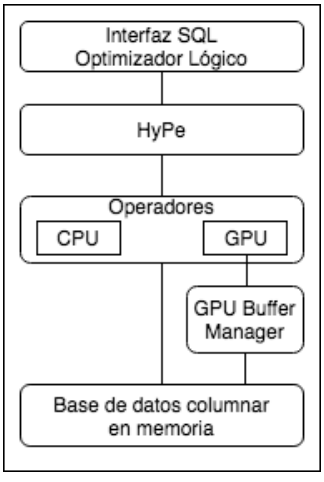

Figura 3.1: Arquitectura de CoGaDB. Adaptada de [2].

Como la mayoría de los DBMSs, CoGaDB cuenta con una interfaz SQL que es utilizada para ejecutar consultas y a su vez construye el árbol lógico,

el cual es convertido a un plan lógico. Luego, el optimizador lógico aplica una serie de reglas a dicho plan con el fin de hacerlo más eficiente.

CoGaDB utiliza la biblioteca HyPE *(Hybrid Query Processing Engine)* [17] como optimizador físico, el cual consta de tres componentes: un optimizador híbrido para CPU/GPU, un selector de algoritmo, y un componente de estimación para una operación en determinado dispositivo (GPU o CPU). HyPE crea el plan físico a partir del plan lógico utilizando el selector de algoritmo y estimador de costos, y luego ejecuta la consulta usando el motor de ejecución de HyPE. Internamente CoGaDB registra todos sus operadores en HyPE, implementandolos a partir de su interfaz. En la Sección 3.3 se brindan más detalles sobre la arquitectura e implementación de HyPE.

El módulo *Buffer Manager* es el encargado de copiar la información a la GPU; además se encarga de optimizar consultas si la información de la consulta a ejecutar ya está disponible en GPU.

Por último, se tiene el módulo de almacenamiento de datos el cual persiste en memoria principal y está orientado a columnas, ya que tiene un uso más eficiente de la jerarquía de memoria y mayor compresión de datos. En caso que el espacio en memoria principal no sea suficiente para la base de datos, el administrador de memoria virtual del sistema operativo gestiona el *buffer* de la base de datos para intercambiarlos a memoria principal. Esto es beneficioso ya que para obtener el mejor rendimiento de las operaciones en GPU los datos deben estar en memoria principal.

3.2. Plan de consultas

CoGaDB implementa un optimizador lógico que utiliza dos heurísticas básicas: mover las selecciones lo más abajo posible en el árbol lógico y cambiar secuencias de selecciones y productos por *Joins* naturales. Para lograr esto, CoGaDB aplica cuatro reglas de optimización:

1. Cambia las selecciones conjuntivas en una secuencia de selecciones que consisten en a lo sumo una disyunción.
2. Mueve hacia lo más abajo posible las selecciones.
3. Se remueve la condición de *Join* expresada por la selección y el producto cruzado, y lo remplaza por un operador de *Join* semánticamente equivalente.

4. El optimizador combina todas las selecciones sucesivas a selecciones complejas en forma conjuntiva.

El plan lógico generado por CoGaDB, es procesado por la biblioteca HyPE para generar el plan físico, el cual asigna a cada operador del plan de consultas un dispositivo de procesamiento, y el algoritmo más adecuado para el dispositivo elegido.

3.3. El optimizador HyPE

En el contexto de ejecutar consultas en procesadores heterogéneos, es crucial que el plan de consultas haga un uso eficiente de los recursos y la memoria disponible de todos los procesadores. Para lograr este objetivo, CoGaDB utiliza la biblioteca HyPE como optimizador físico. HyPE es una biblioteca que es agnóstica a la arquitectura y modelo de la base de datos y también a la arquitectura de los dispositivos de procesamiento. Para lograr estimar y optimizar un plan de consultas, HyPE aprende en cada ejecución, utilizando regresión para refinar la estimación del modelo. De esta forma la herramienta permite predecir el costo de ejecutar una operación, utilizando un determinado algoritmo, en un dispositivo de procesamiento [2, 18].

3.3.1. Arquitectura de HyPE

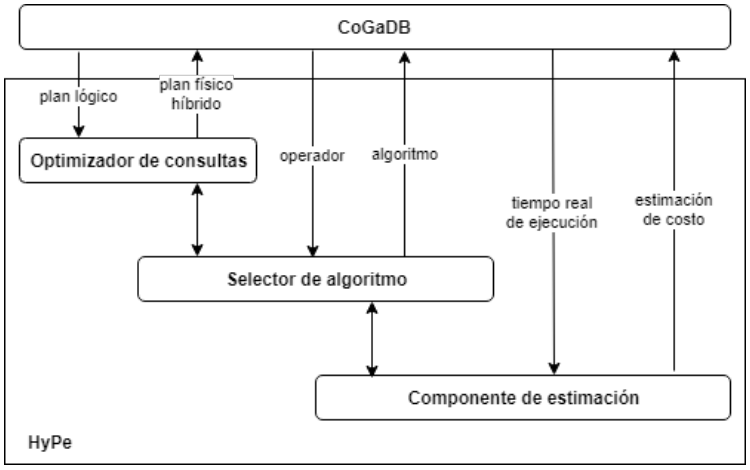

Figura 3.2: Arquitectura de HyPE. Adaptada de [2].

HyPE consta de tres componentes principales: el componente de estimación (STEMOD - *selftuning execution time estimator*), el selector de algoritmo

(ALRIC - *algorithm selector*), y el optimizador de consultas híbrido (HOPE - *hybrid query optimizer*), un resumen gráfico se puede observar en la Figura 3.2.

- **Componente de estimación:** HyPE no requiere tener contexto del *hardware* o detalles de implementación de los algoritmos de la base de datos. Para esto, el componente de estimación usa modelos estadísticos simples (por ejemplo, mínimos cuadrados o regresión lineal sobre datos de aprendizaje) para aproximarse al comportamiento de los algoritmos de la base de datos en los diferentes dispositivos. El modelo recibe como entrada un conjunto de algoritmos disponibles para ejecutar determinada operación, y un conjunto de características o parámetros. Estos parámetros son propiedades que influyen en el tiempo de ejecución, tales como propiedades de los datos de entrada (por ejemplo, el tamaño de los mismos), propiedades de la operación (por ejemplo, selectividad del *Join*) o si los datos se encuentran en *caché*. HyPE requiere de una etapa inicial de entrenamiento, luego de la ejecución de los algoritmos refina constantemente el modelo a partir del tiempo real de ejecución de estos, logrando así progresivamente mayor exactitud en sus estimaciones.
- **Selector de algoritmo:** Basado en el componente de estimación, el optimizador necesita elegir para cada operador del plan de consultas, qué algoritmo debería ser utilizado. Para cada operación, hay un conjunto de algoritmos disponibles que ejecutan en determinada unidad de procesamiento. Con esta información, el componente de estimación calcula para cada algoritmo un tiempo de ejecución estimado y, finalmente, se toma la decisión del algoritmo óptimo de acuerdo al criterio de optimización definido por el usuario. El criterio de optimización es configurable, un ejemplo sería utilizar el algoritmo que brinde el menor tiempo de respuesta. Por defecto HyPE utiliza el criterio WTAR (*Waiting Time Aware Response Time*) que considera además del menor tiempo de respuesta, el tiempo de espera estimado en cada procesador, lo que a su vez genera que se tienda a balancear la carga automáticamente entre los distintos dispositivos.
- **Optimizador de consultas híbrido:** Es el módulo encargado de construir un plan físico a partir de un plan lógico. El optimizador de consultas utiliza el módulo selector de algoritmo para cada operador del plan de consultas, obteniendo un dispositivo de procesamiento y un algoritmo

adecuado según los criterios vistos anteriormente. HyPE a su vez genera un conjunto de planes candidatos y utiliza la heurística del camino crítico (*Critical Path*) para determinar el plan óptimo.

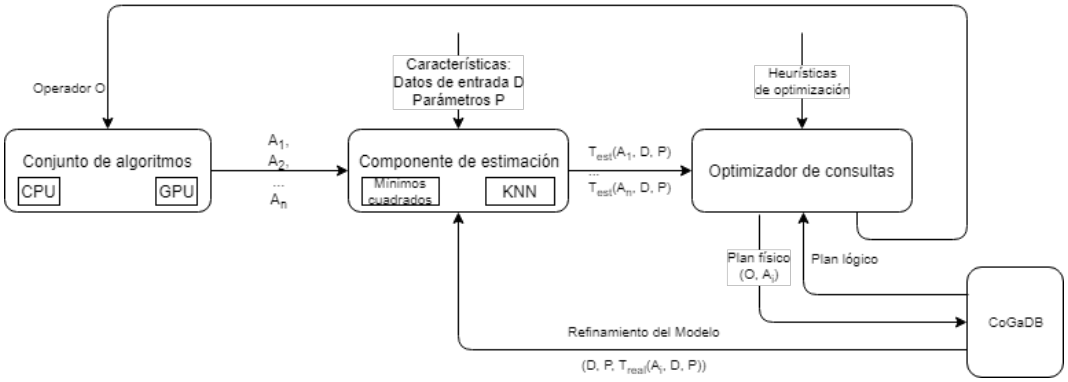

Figura 3.3: Interacción de los módulos de HyPE. Adaptada de [2].

En la Figura 3.3 se resume, en alto nivel, cómo interactúan los distintos componentes de HyPE, qué información reciben de entrada, y la información que brindan de salida, para poder elegir un plan físico óptimo a partir de un plan lógico. En las siguientes secciones se explicará en mayor detalle conceptos de HyPE tales como operaciones, algoritmos, modelos de aprendizaje, criterios de optimización y modelos de refinamiento.

3.3.2. Operaciones y algoritmos

El concepto general de HyPE es tomar la decisión de qué algoritmo ejecutar para determinada operación. Para esto, en primer lugar se deben especificar las operaciones de la aplicación (por ejemplo, *Join*) y en segundo lugar especificar los algoritmos para dicha operación (por ejemplo, *Hash Join*). Al especificar un algoritmo en HyPE, se debe definir también a qué dispositivo de procesamiento (GPU y/o CPU) está asociado dicho algoritmo. También es posible configurar un modelo de aprendizaje, un criterio de optimización y de refinamiento para el algoritmo.

3.3.3. Modelos de aprendizaje utilizados por el componente de estimación

En primer lugar es importante identificar los parámetros o características que impactan en el tiempo de ejecución del algoritmo, así pueden ser tenidos

en cuenta por el modelo. Por convención de la biblioteca, las características consisten en una primer entrada conteniendo el tamaño de la primer columna, y como segunda entrada, el tamaño de la segunda columna. También, en caso que aplique, se puede tener una entrada que contenga la selectividad del operador, así como también otra entrada que señale si ambas columnas se encuentran almacenadas en *caché*. El modelo de aprendizaje aprende en función del tiempo de ejecución de los algoritmos utilizados, y los parámetros definidos para esa ejecución. HyPE utiliza bibliotecas para dar soporte a los modelos mínimos cuadrados, regresión multi-lineal y KNN (*k-nearest neighbors algorithm*). Una de las ventajas de utilizar este último método es que soporta n características de aprendizaje, mientras que mínimos cuadrados acepta una sola, y regresión multi-lineal acepta hasta dos.

El modelo tiene una fase de entrenamiento inicial que se alimenta a partir de los tiempos reales de ejecución de los algoritmos seleccionados, los cuales son seleccionados de a turnos durante esta etapa de entrenamiento. Cuando un algoritmo supera el umbral mínimo de ejecuciones, el modelo computa su tiempo estimado de ejecución. Además, el modelo se puede re-computar periódicamente para adaptarse a escenarios donde los tiempos de ejecución de los algoritmos varíen significativamente, por ejemplo debido a un cambio drástico en el tamaño de la base de datos. Para re-computar la función de aproximación HyPE ofrece dos heurísticas:

- *Periodic Recomputation*: La re-computación periódica volverá a calcular la función de aproximación de un algoritmo después de un número X de ejecuciones (configurable). Es recomendable en escenarios donde se necesita que HyPE adapte la estimación debido a datos cambiantes, o variaciones grandes en el tamaño de datos.
- *Oneshot Recomputation*: Computa la función de aproximación por una única vez luego de finalizada la etapa de entrenamiento del algoritmo.

3.3.4. Criterios de optimización

El criterio de optimización define la estrategia para encontrar el algoritmo "óptimo". HyPE define varias opciones:

- *Response Time*: Reduce la elección a aquel algoritmo que minimice el tiempo estimado de ejecución para determinada operación.

- *Waiting Time Aware Response Time*: Es una extensión a la elección según el menor tiempo de respuesta, pero además tiene en cuenta el tiempo de espera para todos los dispositivos de procesamiento, balanceando la carga entre ellos. Es el criterio de optimización recomendado por HyPE.
- *Round robin*: Esta estrategia distribuye en turnos las operaciones en todos los dispositivos de procesamiento. Funciona bien en casos que el tiempo de ejecución sea aproximadamente el mismo en todos los dispositivos.
- *Threshold-based Outsourcing*: Es una extensión al criterio de elección según el menor tiempo de respuesta, pero la idea es utilizar el dispositivo de procesamiento más lento, para liberar carga de dispositivos más rápidos. Además, se tiene que cumplir que por ejecutar la operación en el dispositivo más lento, no se disminuya el tiempo estimado un cierto umbral.

3.4. Problemas, extensiones y posibles mejoras

El uso de tarjetas gráficas para acelerar el procesamiento de datos tiene dos desafíos principales: El primero, la transferencia de datos entre la CPU y la GPU. Hay que utilizar estrategias inteligentes para mitigar el cuello de botella existente en la transferencia de datos desde la memoria principal de la CPU mediante el bus PCI-Express a la GPU (o viceversa). En segundo lugar, los componentes internos del manejador de bases de datos, tales como estructuras de datos, procesamiento de consultas y optimización, están optimizados tradicionalmente para las CPU y no para las GPU. La GPU tiene una arquitectura diferente, con un gran número de co-procesadores homogéneos. El mismo código que se ejecuta en la CPU al ejecutarlo en la GPU no implica que sea eficiente, y entre otras cosas, para lograr performance es necesario reimplementar los operadores utilizando el *toolkit* de la GPU, tales como CUDA u OpenCL, para poder utilizar y sacar ventaja de la arquitectura de la GPU.

Con respecto a la transferencia de datos entre el *host* y el *device* notar que el acceso a memoria principal es de varios ordenes de magnitud más rápido comparado con el envío de datos utilizando el bus PCI-Express.

3.4.1. Compresión de datos

La compresión en sistemas DBMS otorga ciertas ventajas como son la reducción del espacio total de almacenamiento utilizado y en algunos casos se obtiene también una mejora en la performance al ejecutar las consultas. Si los datos son comprimidos, menor es el tiempo consumido por I/O ya que menor es el volumen de datos que deben ser leídos de disco hacia la memoria y luego a la CPU. Para el caso de los GDBMS la transferencia de datos entre la memoria principal y la memoria de GPU ocurre sobre un *bus* de datos PIC-E de bajo ancho de banda, el *overhead* de transferir datos sin comprimir es un factor sumamente importante, es especial considerando que esta transferencia puede contribuir con el tiempo total del procesamiento de una consulta en un 15 a 90 % [19]. Por esta razón diversas investigaciones se han centrado en cómo puede reducir la compresión los tiempos de transferencia y, por lo tanto, mejorar el tiemplo total de co-procesamiento de consultas en sistemas de bases columnares.

Más importante aún es el hecho de hacer un uso óptimo tanto de la CPU como de la GPU, ya que los tiempos de procesamiento de ambos dispositivos son de menor magnitud que los tiempos de transferencia de datos desde la memoria, por lo que el costo de acceder a los datos cuesta más que la ejecución de los ciclos de una CPU/GPU.

Compresión de datos en CoGaDB

Dado que CoGaDB utiliza un almacenamiento de tipo columnar, concepto que fue visto en la Sección 2.1.3, el mismo tiene la capacidad de comprimir los datos [20] utilizando los siguientes algoritmos:

- *Run Length Encoding*
- *Bit Vector Encoding*
- *Dictionary Compression*
- *Delta Coding*

En particular, la compresión de tipo diccionario se utiliza en columnas de tipo *String*. Dado que el diccionario no es ordenado, solo se pueden evaluar predicados cuyas condiciones sean de igualdad o diferencia [21].

3.4.2. Estimaciones y planificación

Una propiedad inherente de los co-procesadores masivamente paralelos es que su desempeño puede diferir significativamente dependiendo de la operación a ser ejecutada. El DBMS necesita decidir qué operadores de una consulta deben ejecutarse en qué co-procesador utilizando los recursos de forma eficiente para obtener el mejor rendimiento. Este problema es conocido como el problema de ubicación del operador, y se enfrenta a dos desafíos principales:

1. Similar a otros problemas de optimización, el problema de ubicar el operador tiene un espacio de búsqueda de orden exponencial: para n co-procesadores y m operadores en el plan de consulta, el optimizador necesita explorar n^m planes si fuera a recorrer de forma exhaustiva. Esta situación justifica el uso de heurísticas (eficientes).
2. Los DBMS generalmente usan funciones de costo para estimar las cardinalidades del resultado de cada operador y luego, eligen un plan de consulta que estime el tamaño mínimo en los resultados intermedios. Sin embargo, las cardinalidades estimadas no son suficientes para ubicar el operador, ya que el tiempo varía según el tipo de procesador. Por lo tanto, es necesario estimar el tiempo de ejecución real de los operadores de consulta en cada procesador. Para resolver este problema una opción es analizar la arquitectura de cada unidad de procesamiento, como también cada algoritmo posible, y basado en esto, crear un modelo de costo analítico para cada tipo de co-procesador.

Los modelos de costos analíticos tienen la ventaja de que sus estimaciones son predecibles y rápidas, pero su principal desventaja radica en el gran esfuerzo que implica crearlos y mantenerlos. Algunos de los parámetros que tienen que ser tenidos en cuenta son: las características de la tabla de entrada, la transferencia de datos entre procesadores, copias a memoria, tamaño de memoria de caché y latencia de memoria, detalles de implementación del algoritmo (por ejemplo, tablas de *hash* particionadas o sin particiones). La mayor complejidad proviene del *hardware* y de los parámetros dependientes del algoritmo.

3.4.3. *Join* híbrido concurrente

Como se mencionó en secciones anteriores, la última versión de CoGaDB en conjunto con HyPE, ejecutan el algoritmo seleccionado en un solo dispositivo. HyPE decide qué algoritmo utilizar para cada operador, como también el mejor dispositivo en el cual ejecutar en ese momento, es decir, aquella unidad de procesamiento que retorne el menor tiempo de ejecución para una consulta dada. Dado que la cantidad de memoria en los co-procesadores es escasa, siempre existe la posibilidad de que un operador no pueda asignar la memoria necesaria. En este caso, CoGaDB descarta el trabajo del operador y comienza a computar desde cero en la CPU, dado que en general se cuenta con mayor memoria disponible en este dispositivo. Dependiendo la carga de trabajo de procesamiento que haya, se puede generar una degradación significante en el sistema, ya que el esfuerzo computacional realizado es descartado en su totalidad, y además pueden haber dependencias entre operadores que obliguen también la ejecución en la CPU de aquellos operadores posteriores en el plan físico.

Acuña y Parula propusieron una solución a este problema, llamada *Join* híbrido concurrente [3] (CoGaDB extendido). La misma consiste en la realización de un algoritmo híbrido - el *Join* en primer instancia - que pueda ser ejecutado en ambos dispositivos, CPU y GPU en simultaneo y computando distintas secciones en cada dispositivo. La ejecución del *Join* híbrido concurrente está conformado por tres etapas bien definidas. La primera de ellas es la etapa de partición, en la misma se toma la columna de mayor tamaño y se dividen los cómputos asociados a ella, una porción será ejecutada en la CPU mientras que el restante será ejecutada en la GPU. De tal manera que permita el mayor aprovechamiento de las capacidades de cómputo, esto es, que las unidades de procesamiento existentes tengan la menor cantidad de tiempo ocioso. Esto permite además que si alguno de los dispositivos tiene algún error en la ejecución del operador, no se comprometa todo el cómputo realizado debiendo comenzar nuevamente, como sucede con el *Join* nativo de CoGaDB.

La forma por la cual se decide el tamaño de dicha partición es un tema de interés en el marco de este proyecto, ya que actualmente se configura de manera manual, y es un punto a explorar con el fin de obtener la partición óptima, reduciendo así los tiempos totales de ejecución de una consulta, esta discusión se realizará en secciones posteriores. En una segunda etapa se tiene

la ejecución del operador en cada dispositivo. Previamente se realiza una copia a la memoria de cada unidad de procesamiento, con la proporción de datos de la primera columna asignados a cada uno, como también la totalidad de la segunda columna involucrada en el *Join*. En este paso se reutiliza el *Hash Join* implementado por CoGaDB de cada dispositivo concurrentemente, dando como resultado un *LookupColumn* por cada dispositivo. En una tercer y última etapa, se unen los datos computados para tener finalmente el resultado del *Join*. Todos los *LookupColumn* generados en la etapa anterior son transferidos al dispositivo principal (CPU en este caso) y el mismo es el encargado de la unión de los *LookupColumn* así como del armado del resultado final con los datos de cada tupla.

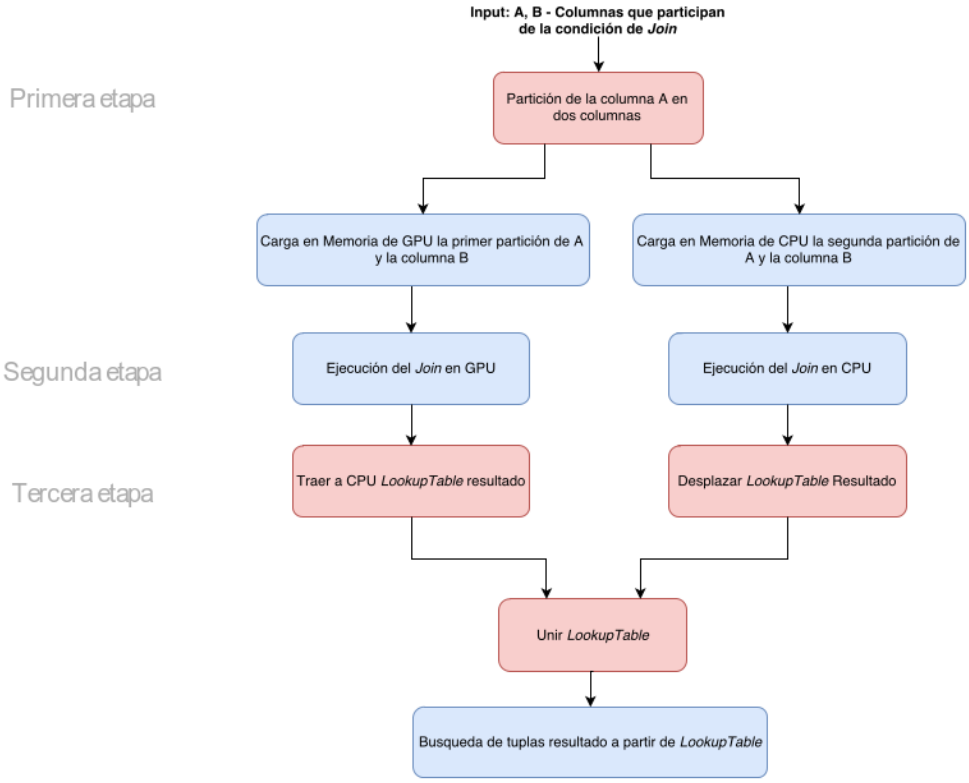

Figura 3.4: Etapas del *Join* híbrido concurrente. Extraído de [3].

En la Figura 3.4 se muestra en detalle las operaciones realizadas en cada etapa, diferenciando de color azul aquellas que se realizan nativamente, mientras que las rojas son las extensiones realizadas por el proyecto de Acuña y Parula [3], de manera de ejecutar consultas de manera concurrente. Acuña y Parula tomaron una decisión simple en cuanto a la elección del

tamaño de cada partición, la cual consiste en que el usuario debe definir el porcentaje, específicamente definieron como *cpu_per* al porcentaje de datos de la columna a particionar que son asignados y ejecutados por la CPU. El resto del porcentaje es ejecutado por los demás dispositivos, en este caso una única GPU. El *cpu_per* define dos cantidades de elementos sobre la columna a dividir, las cuales se denominan *cpu_elements* y *gpu_elements*, que es la cantidad de elementos asignada a la CPU y GPU respectivamente, es de notar que para el caso en el que *cpu_elements* no sea entero el mismo se redondea. Para determinar el punto de corte de la columna de mayor tamaño, se definió una variable global del sistema que indica el porcentaje de elementos a ejecutar por la CPU, la cual se configura en tiempo de ejecución mediante el comando *set cpu_per=valor*, los valores que puede tomar son cualquier número entre 0 y 1, ya que su representación es porcentual en términos decimales. Esta variable es configurada manualmente, y en términos generales, la partición elegida no será óptima, es de interés seleccionar aquel *cpu_per* que reduzca los tiempos de ejecución y lo que es más, que esta elección sea automática teniendo en cuenta los dispositivos disponibles, como también su contexto actual. Esto es lo que se denomina *cpu_per* óptimo, y el tiempo asociado se le llama tiempo de ejecución óptimo.

En el marco del proyecto de Acuña y Parula se menciona como una mejora en la implementación la elección automática del *cpu_per* mediante una heurística genérica, que permita hallar el óptimo sin depender de la plataforma y teniendo en cuenta el contexto de ejecución de los distintos dispositivos disponibles. HyPE utiliza algoritmos de aprendizaje para tomar decisiones, esto es, elegir el algoritmo y el dispositivo cuya ejecución del operador es la de menor tiempo de ejecución. HyPE permite además tanto la modificación de los algoritmos de aprendizaje existentes, como también la extensión de nuevos algoritmos, lo cual permite un alto grado de personalización adaptando la elección del dispositivo en base a distintas características. Una de estas características a tener en cuenta a la hora de aprender es la de *cpu_per*, algunas de las posibles ideas que surgen son:

- Estimar por medio de aprendizaje automático el *cpu_per* óptimo a partir de distintas ejecuciones.
- Ejecutar HyPE para un conjunto de posibles *cpu_per*, y tomar la opción que alcance el menor tiempo de ejecución. Actualmente HyPE toma to-

dos los posibles algoritmos y dispositivos realizando una estimación, por lo que se podría tomar la misma estrategia con el *cpu_per*, es decir probar un rango o espacio de posibles *cpu_per*.

3.4.4. Elección del problema: distribución automatizada del cómputo

En este apartado se presenta el problema en el cual se hará foco en el marco de este proyecto, y para el cual en posteriores secciones se diseñará una solución.

Dado que el *Join* Híbrido Concurrente, está en sus fases iniciales, se observa un amplio camino a futuro en el cual trabajar siendo de suma importancia continuar aventando en la misma línea. Motivados principalmente por las mejoras de performance observadas en la prueba de concepto llevada a cabo en el proyecto de Acuña y Parula. En particular, se quiere hacer foco en la búsqueda de una solución para obtener el *cpu_per* óptimo, sin depender de la plataforma, y basandose en las ejecuciónes pasadas, logrando como consecuencia de esto reducir los tiempos totales de ejecución de una consulta y mejorar aún más la performance, evitando la perdida de procesamiento computado y minimizando los tiempos ociosos existentes en cada dispositivo.

Por otra parte, los otros problemas mencionados parecen estar ya abordados, CoGaDB posee actualmente soluciones de compresión de datos para minimizar la transferencia de datos entre los dispositivos y su almacenamiento, se considera que el trabajo faltante a realizar en esta área no es muy amplio y del cual no se obtendría el máximo de provecho.

Por otra parte, el componente de estimación y planificación de HyPE está lo suficiente maduro en cuanto a su implementación y la investigación realizada por Bress et al.[2] Además su performance está vinculada en gran parte en el aprendizaje realizado en ejecuciones pasadas.

Por las razones mencionadas, se decide entonces aplicar técnicas de aprendizaje automático para la distribución óptima del cómputo.

Capítulo 4

Aplicación de técnicas de Aprendizaje Automático para la distribución del cómputo en CoGaDB

En el presente capítulo se describe el problema abordado, detallando el diseño y la implementación de su solución propuesta. También se describen los experimentos realizados y un análisis de los resultados experimentales obtenidos.

4.1. Distribución automatizada del cómputo en CoGaDB

Como se mencionó en la Sección 3.4.3, CoGaDB extendido permite la ejecución de *Joins* de manera concurrente, configurando de manera manual la distribución del cómputo en cada dispositivo. Para eso se utiliza la bandera *cpu_per* que representa el porcentaje de la columna que se debe ejecutar en la CPU, mientras que la porción restante debe ser procesada por la GPU.

El problema radica en la selección manual de esta configuración sin tener en cuenta características importantes a la hora de la ejecución de está operación como lo son la carga de trabajo actual de cada dispositivo, como también características inherentes a los datos a ser procesados como el tamaño y su selectividad.

Se consideró entonces la extensión a la herramienta CoGaDB y a la biblioteca HyPE, de manera que la elección del *cpu_per* sea automática y de manera "óptima". Para ello se utilizan técnicas de aprendizaje automático para la distribución del cómputo.

En términos generales, la solución consiste por un lado en la extensión de HyPE, definiendo nuevos algoritmos que contengan un *cpu_per* asociado, también se define un nuevo criterio de optimización híbrido, que seleccione el mejor par de algoritmos a ejecutar en cada dispositivo. Por otra parte, se extendió CoGaDB, soportando nuevos algoritmos de tipo híbrido, tomando de ellos su *cpu_per* asociado, distribuyendo el cómputo de manera óptima entre los distintos dispositivos y teniendo en cuenta la carga de cada dispositivo como también las características de los datos de entrada del *Join* híbrido concurrente.

4.2. Diseño

En primera instancia se analizó la arquitectura de HyPE, la cual ofrece una interfaz con la que CoGaDB interactúa, y además es de tipo *plug-in* lo que permite extender la implementación subyacente con optimizadores personalizados [2]. Los tipos de *plug-in* ofrecidos por la biblioteca son: métodos estadísticos, heurísticas de recomputación y criterios de optimización. Este último es de interés ya que un criterio de optimización es aquel que configura los objetivos de optimización utilizados por el *Scheduler*, utilizando alguna estrategia de *scheduling*.

La idea detrás del diseño realizado, es que HyPE retorne a CoGaDB planes físicos donde el algoritmo especifique el *cpu_per* con el que debe ejecutar, y además el plan físico puede contener más de un algoritmo para un mismo operador, ya que por ejemplo para el *Join* puede ejecutar cierto porcentaje en CPU usando un algoritmo implementado para CPU, y otro cierto porcentaje en GPU usando otro algoritmo implementado para GPU. Por lo tanto hay tres objetivos principales:

1. Que HyPE retorne un plan físico híbrido.
2. Que CoGaDB soporte algoritmos de tipo híbrido, cuyo *cpu_per* permita una ejecución concurrente en los dispositivos asociados.
3. Adaptar las técnicas de aprendizaje automático de HyPE para aprender y refinar la estimación del *cpu_per* óptimo para cada ejecución.

Cómo se detalló en la Sección 3.3 el optimizador de HyPE tiene dos responsabilidades principales: la primera, consiste en proveer el servicio para decidir un algoritmo óptimo para cierta operación, y la segunda, implementar una interfaz para agregar nuevas observaciones, es decir, obtener estimaciones de tiempo para la ejecución de algoritmos. En cuanto a la primera responsabilidad, como se vio en la Sección 3.3.2 los algoritmos y las operaciones son conceptos relevantes y la extensión diseñada en este apartado, permite a HyPE - para cada operación - aprender indirectamente cual es el *cpu_per* "óptimo", por medio del *cpu_per* asociado del algoritmo seleccionado. Este nuevo tipo de algoritmos híbridos, tiene asociado además un criterio de optimización híbrido.

Los criterios de optimización vistos en el Apartado 3.3.4 fueron extendidos para que *Response Time* y *Waiting Time Aware Response Time* puedan optimizar teniendo en cuenta que ahora los algoritmos pueden ser híbridos, y la decisión óptima es tomada en base a pares compatibles, esto es, que sus *cpu_per* sean iguales. Cuando se tiene un par, para estimar su tiempo de ejecución se escoge aquel tiempo que sea máximo en el par. Por lo tanto, dado un operador, se obtienen todos los algoritmos que están asociados a dicho operador, luego se agrupan en pares compatibles, y se aplica la criterio de optimización definido para el algoritmo.

Finalmente, el componente de estimación ahora debe ser capaz de recibir observaciones de tiempo de ejecución híbridas, es decir para un mismo operador refinar el tiempo de dos algoritmos, uno ejecutado en CPU y otro ejecutado en GPU.

4.2.1. Detalles de la solución

En esta sección se describe el diseño de la solución en detalle. Se analizará cada módulo de HyPE y CoGaDB mencionando las extenciones realizadas a los mismos para integrar el soporte híbrido.

En la Figura 4.1 se presenta en alto nivel el flujo de interacción entre los módulos que son de interés en el marco de la solución diseñada.

Los cambios realizados a HyPE fueron los siguientes:

- **Extensión de la especificación de algoritmos**: los mismos pasan a tener un *cpu_per* asociado, y esto será lo que los diferencie de los algoritmos ya existentes. Entonces, un algoritmo es híbrido si tiene un *cpu_per* asociado.

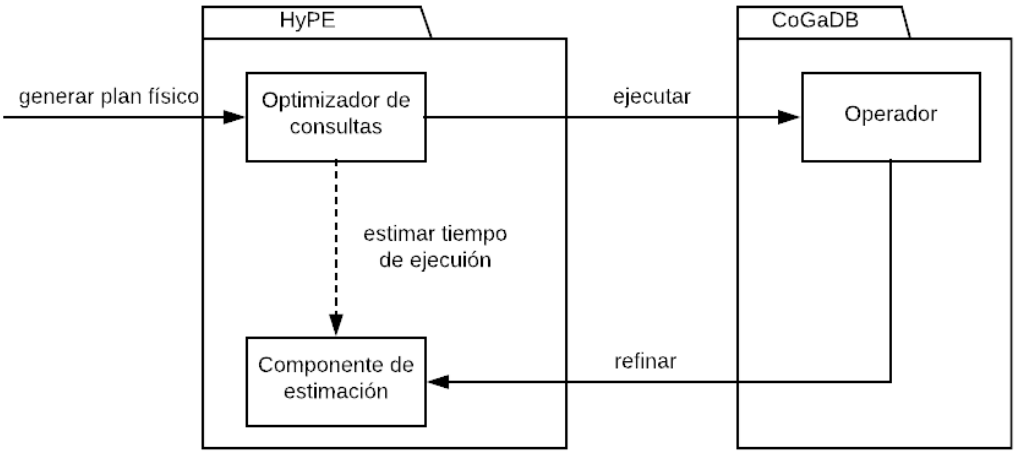

Figura 4.1: Los módulos de HyPE y CoGaDB que fueron modificados junto con las interacciones existentes entre ellos.

- **Extensión del optimizador de consultas híbridos**: como se mencionó en la Sección 3.3.1, este módulo es el encargado de construir un plan físico a partir de uno lógico. La extensión realizada considera en el plan construido un nuevo tipo de operaciones híbridas, las operaciones híbridas tendrán asociadas más de un algoritmo y correrán (potencialmente) en más de un dispositivo en simultaneo.
- **Extensión del componente de estimación**: este componente originalmente agregaba por cada operación ejecutada una observación al modelo de aprendizaje. En el soporte híbrido realizado, este componente considera operaciones híbridas, agregando más de una observación, ya que cada una de ellas tiene asociado más de un algoritmo híbrido, uno para la CPU y otro para la GPU.

Por otra parte, CoGaDB también fue extendido para soportar operaciones híbridas. Se modificó el operador de *Join*, de manera de diferenciar *Joins* híbridos, de los no híbridos. Para el caso de los primeros, se realizó un conteo del tiempo de ejecución en cada dispositivo, el cual es utilizado luego por el componente de estimación al refinar los modelos de aprendizaje.

En las siguientes secciones se describen, con mayor nivel de detalle, los cambios mencionados anteriormente a nivel de cada módulo.

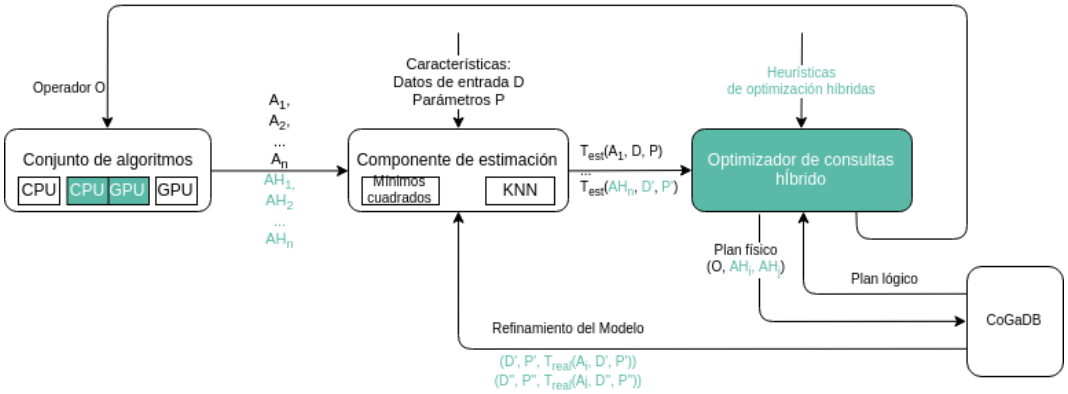

Figura 4.2: Extensiones realizadas a los módulos de HyPE definidos en la Figura 3.3.

Generación del plan físico híbrido

Para generar un plan físico híbrido, fue necesario modificar el optimizador de consultas existente en HyPE. El objetivo en esta etapa consiste en construir un plan físico a partir de un plan lógico que considere algoritmos híbridos. En la Figura 4.2 se puede observar la interacción de los sub-módulos de HyPE involucrados en la generación del plan físico híbrido. El primer desafío consistió en que HyPE pudiera definir algoritmos híbridos los cuales estarán asociados a una operación existente del sistema, es decir, que los mismos contengan un *cpu_per*.

HyPE especifica los algoritmos mediante la clase *AlgorithmSpecification*, la cual define toda la información relevante sobre un algoritmo como el nombre del mismo y el de la operación a la que pertenece. En la Figura 4.3 se muestran los cambios realizados a la clase, se agrega un nuevo atributo a la clase que almacena el valor del *cpu_per*, el cual solo está definido en algoritmos propiamente híbridos. Esto permite especificar algoritmos híbridos, que contengan información sobre su *cpu_per*, un ejemplo de definición de algoritmo utilizando la biblioteca HyPE es la siguiente:

```
gpu_alg("GPU_alg", "JOIN", KNN, Periodic, ResponseTime, 0.5)      [1]
```

Donde se indica el nombre del algoritmo, el nombre de la operación, el método y el modo de aprendizaje utilizado, el criterio de optimización y el *cpu_per* asociado.

HyPE mantiene un conjunto de algoritmos, que son creados a partir de

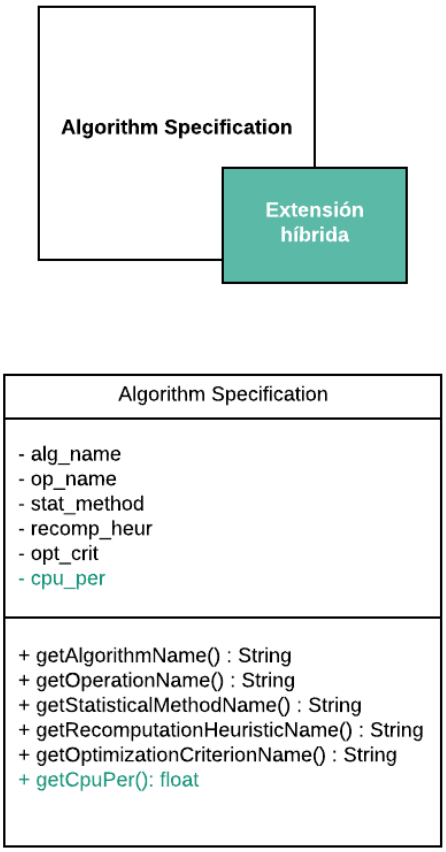

Figura 4.3: Extensión híbrida realizada a la clase *AlgorithmSpecification*.

la especificación de algoritmos. Básicamente se mantiene un mapeo, que dado una especificación de algoritmo devuelve el algoritmo concreto. La clase que lo representa se llama *Algorithm* y su función es abstraer y encapsular características importantes e información del algoritmo, como lo son, el nombre del algoritmo, la operación a la que pertenece, el dispositivo en el que ejecuta, entre otros. La extensión híbrida implementada se puede observar en la Figura 4.4, nuevamente se agrega un atributo que contenga la información del *cpu_per* y algunos métodos que permitan su manejo. Con estos cambios realizados, el módulo de *Conjunto de algoritmos* está listo para especificar y contener *Algoritmos Híbridos*.

Siguiendo en el proceso de generar un plan físico, los optimizadores son los encargados de dada una operación y ciertas características de la misma, retornar la decisión óptima sobre qué algoritmo ejecutar y en cual dispositivo hacerlo.

El siguiente desafío consta de dos etapas:

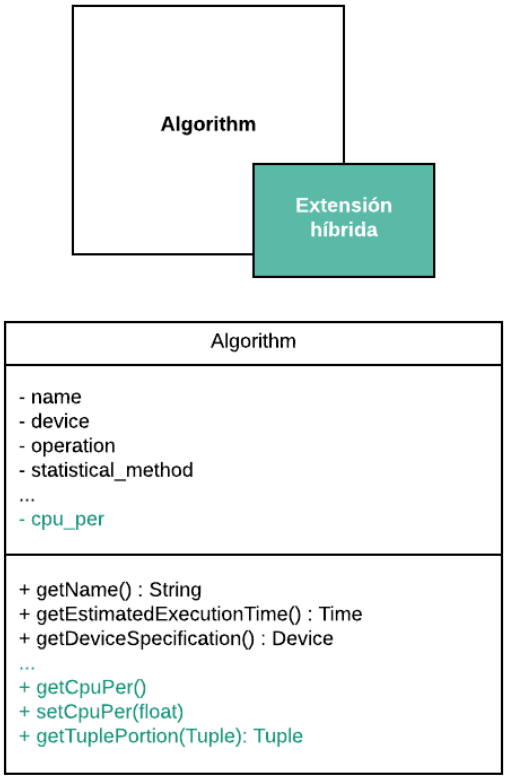

Figura 4.4: Clase *Algorithm* de HyPE con los cambios realizados en verde.

1. Modificar el criterio de optimización base *OptimizationCriterion*
2. Crear un nuevo criterio de optimización o heurística: que sea capaz de recibir algoritmos híbridos y de generar decisiones óptimas que permitan ejecutar en más de un dispositivo a la vez.

En primer lugar se modificó la clase *OptimizationCriterion* de HyPE, la cual es encargada de obtener el algoritmo óptimo para una operación, teniendo en cuenta los *features* de los datos a procesar.

Un diagrama de flujo se puede ver en la Figura 4.5. Como se observa, primero se entrenan los algoritmos, luego, una vez que todos los algoritmos están entrenados, se procede a la siguiente fase que consiste en delegar el trabajo al criterio de optimización concreto que se haya configurado. En el caso de este proyecto se utiliza un nuevo criterio de optimización llamado *WaitingTimeAwareResponseTimeHybrid* (*WTARTH*).

En el diagrama de flujo se muestra la extensión realizada que consiste en una nueva decisión, que se pregunta sí el algoritmo a entrenar es híbrido. En el caso de ser híbrido, es decir, tiene un *cpu_per* asociado, se toma el algorítmo

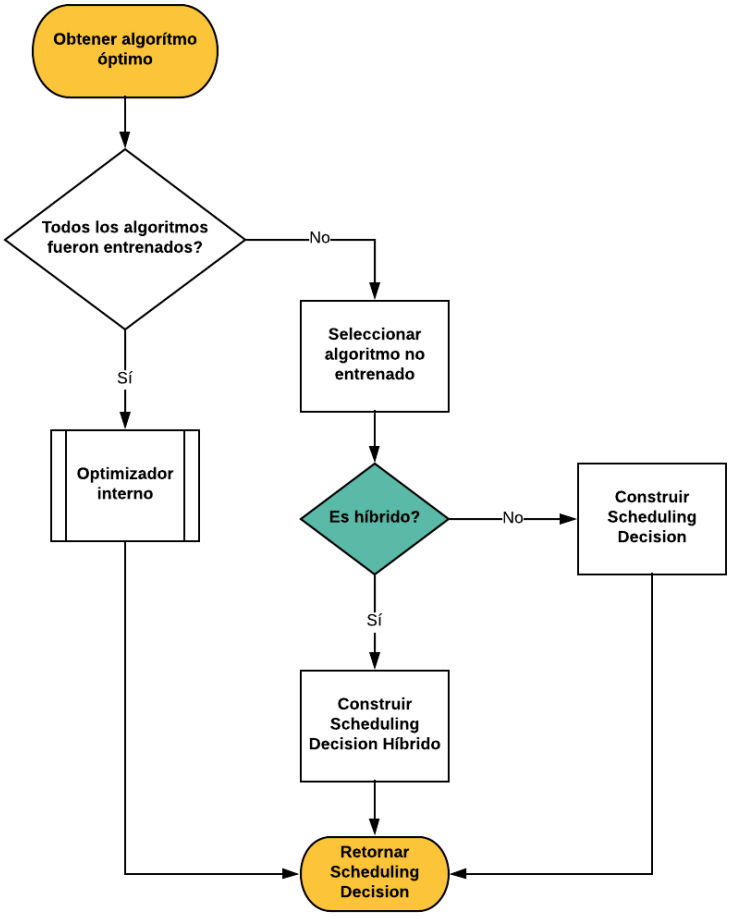

Figura 4.5: Diagrama de flujo de los pasos seguidos por el criterio de optimización, en verde las extensiones realizadas.

que tenga el mismo *cpu_per* pero que ejecute en el otro dispositivo. Por ejemplo, si el algoritmo seleccionado ejecuta en CPU con *cpu_per* 0.3, entonces también se selecciona el algoritmo que ejecuta en GPU con *cpu_per* 0.3. Entonces la modificación que se realizó consiste básicamente en forzar a que los algoritmos híbridos cuyo *cpu_per* coincida, entrenen en conjunto y no por separado.

HyPE permite crear nuevos criterios de optimización de manera sencilla heredando de su clase *hype::core::OptimizationCriterion* e implementando ciertos métodos de la clase padre. Luego como HyPE utiliza una arquitectura de *plug-in* basada en *factories*, se debe registrar el nuevo optimizador en el *hype::core::PluginLoader*.

El pseudocódigo del nuevo criterio de optimización puede verse en el Algoritmo 4.2.1.

Algoritmo 4.2.1: Pseudocódigo del método *response_time_advanced_hybrid*

```
   Entrada: operacion, features, dev_constr
1  algoritmos = obtener_todos_los_algoritmos(operacion);
2  agrupados = agrupar_algoritmos_cpu_per(algoritmos);
3  alg_optimo = optimo_para_dispositivo(agrupados, features, dev_constr);
4  scheduling_decision = construir_scheduling_decision(alg_optimo);
5  devolver scheduling_decision
```

Primero se obtienen todos los algoritmos, tanto híbridos como no híbridos. Luego se los agrupa por *cpu_per* como se observa en el Algoritmo 4.2.2. Finalmente se construye la decisión óptima o *scheduling_decision* a partir del par de algoritmos óptimos. Para seleccionar el par de algoritmos óptimos se tiene en cuenta el tiempo estimado de ejecución, el tiempo estimado de espera del dispositivo y el *dev_constr* que representa la restricción sobre cuales son los únicos dispositivos que tienen permitido ejecutar la operación.

Es de interés describir el funcionamiento del método *optimo_para_dispositivo* (Algoritmo 4.2.3), ya que es el encargado de seleccionar el algoritmo óptimo. Hasta la línea 4, se discute teniendo en cuenta la restricción del dispositivo a ser utilizado, en caso de ejecutar sólo en la CPU, se retorna aquel algoritmo cuyo *cpu_per* valga 1, mientras que en caso de ejecutar sólo en la GPU, se retorna el algoritmo cuyo *cpu_per* valga 0. A partir de la línea 6, comienza el *core* de la decisión óptima. La idea consiste en recorrer cada *cpu_per*, tomar sus algoritmos asociados y obtener el tiempo del par. En la línea 15, puede observarse la medida que se utiliza para calcular el tiempo del par - utilizando el componente de estimación -, el cual está dado por el dispositivo cuyo tiempo total sea mayor, el tiempo total del dispositivo está dado por la suma

Algoritmo 4.2.2: Pseudocódigo del método *agrupar_algoritmos_cpu_per*

```
   Entrada: algoritmos
1  mapa_algoritmos_agrupados = crear_mapa()
2  para algoritmo in algoritmos hacer
3  |  clave = obtener_cpu_per(algoritmo);
4  |  valor = obtener_alg_ref(algoritmo);
5  |  agregar_a_mapa(clave, valor, mapa_algoritmos_agrupados)
6  fin
7  devolver mapa_algoritmos_agrupados
```

Algoritmo 4.2.3: Pseudocódigo del método *optimo_para_dispositivo*

```
   Entrada: algoritmos_agrupados, features, dev_constr
 1 si solo_cpu(dev_constr) entonces
 2 |  alg_optimos = obtener_algoritmo_cpu(algoritmos_agrupados);
 3 si no, si solo_gpu(dev_constr) entonces
 4 |  alg_optimos = obtener_algoritmo_gpu(algoritmos_agrupados);
 5 en otro caso
 6 |  menor_tiempo = max_time();
 7 |  para par_algoritmos in algoritmos_agrupados hacer
 8 |  |  cpu_alg, gpu_alg = desagrupar(par_algoritmos);
 9 |  |  features_cpu = particionar(features, cpu_alg);
10 |  |  features_gpu = particionar(features, gpu_alg);
11 |  |  t_cpu = estimar_tiempo_ejec(cpu_alg, features_cpu);
12 |  |  t_esp_cpu = estimar_tiempo_espera(cpu);
13 |  |  t_gpu = estimar_tiempo_ejec(gpu_alg, features_gpu);
14 |  |  t_esp_gpu = estimar_tiempo_espera(gpu);
15 |  |  tiempo_par = max(t_cpu + t_esp_cpu, t_gpu + t_esp_gpu);
16 |  |  si tiempo_par < menor_tiempo entonces
17 |  |  |  menor_tiempo = tiempo_par;
18 |  |  |  alg_optimos = agrupar(cpu_alg, gpu_alg);
19 |  |  fin
20 |  fin
21 fin
22 devolver alg_optimos
```

del tiempo estimado de ejecución del algoritmo más la espera del dispositivo para quedar libre de carga. Finalmente, se toma el par cuyo *tiempo_par* sea el menor. Notar que se particionan las características (tamaño de datos, etc.), con el fin de poder estimar el tiempo correctamente para cada dispositivo.

Queda un último paso para poder finalizar la extensión del *optimizador de consultas*, y es que la decisión óptima retornada por el criterio de optimización, tiene que ser capaz de contener más de un algoritmo. La decisión óptima está representada por la clase *hype::core::SchedulingDecision*, la cual contiene información importante sobre la operación a ser ejecutada, el algoritmo óptimo y el dispositivo en el que ejecuta. Esta clase juega un rol preponderante en la solución diseñada ya que está presente en cada interacción entre HyPE y CoGaDB, es por tanto donde se hace énfasis para almacenar información que

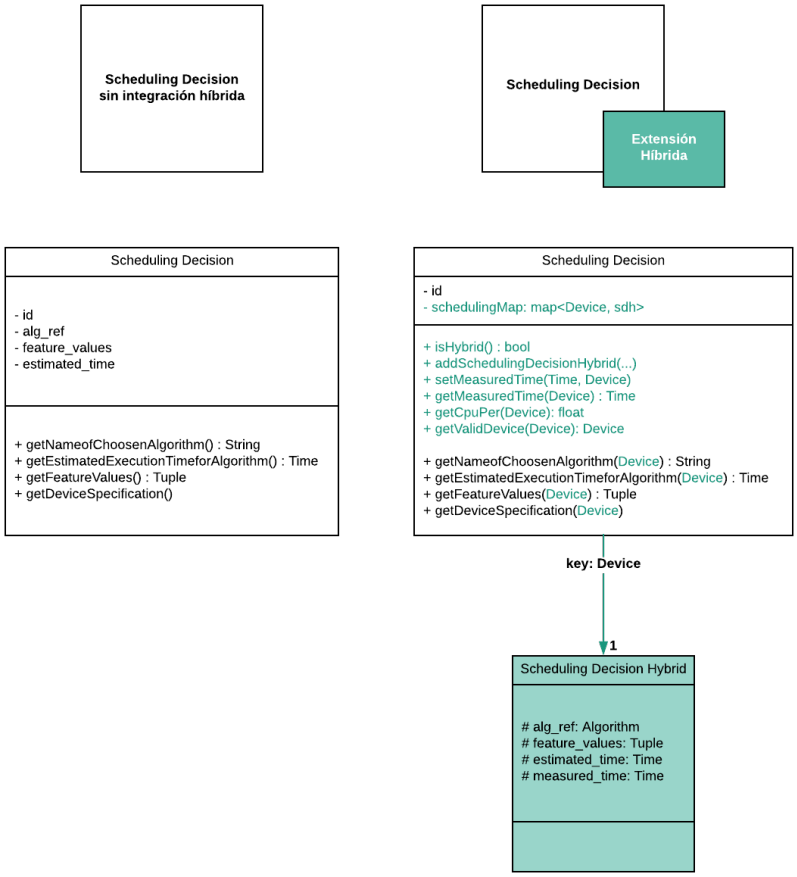

Figura 4.6: Comparación del *Scheduling Decision* original (figura izquierda) con la extensión híbrida implementada (figura derecha). En verde se mencionan los métodos agregados a la implementación como también la nueva clase *Scheduling Decision Hybrid* que permite almacenar datos relevantes del par de algoritmos híbridos.

da soporte a la ejecución híbrida.

En la Figura 4.6 se muestra el UML de la clase en cuestión y los cambios realizados sobre la misma. En la clase original, se tenía una relación 1 a 1 con un algoritmo y su dispositivo, por lo que el *scheduling decision* no era capaz de ejecutar varios algoritmos en varios dispositivos a la vez.

Los cambios realizados se resumen en el siguiente listado:

- Creación de la clase *hype::core::SchedulingDecisionHybrid*: se movieron los atributos originales existentes en la clase *SchedulingDecision*, agregando el tiempo medido y además se la relacionó con ésta última, a través de un mapa, cuya clave es el device. Esto permite tener más de un algoritmo asociado y por tanto tener más de un dispositivo ejecutando

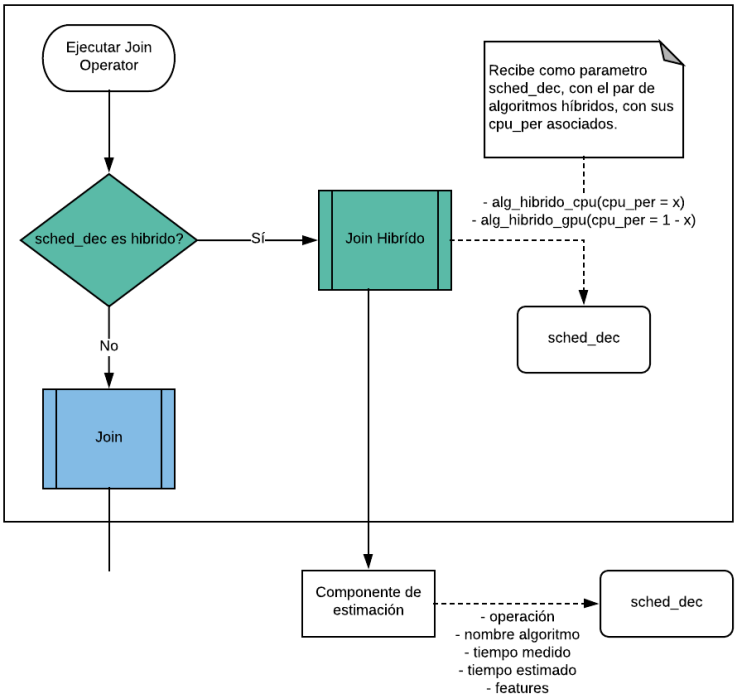

Figura 4.7: Interacción de los componentes internos al módulo de CoGaDB que ejecuta un operador para el caso de *Join Operator*.

en simultaneo.

- Se crearon nuevos métodos que permiten el manejo híbrido. Algunos de ellos consisten en: saber si la instancia actual de *SchedulingDecision* es híbrida o no, agregar un *SchedulingDecisionHybrid* a la relación, obtener el *cpu_per* asociado a un dispositivo y también el manejo de los tiempos medidos, los cuales se almacenan en la nueva clase, ya que cada algoritmo va a tener un tiempo estimado y un tiempo medido.
- Se mantuvieron los métodos originales agregando un parámetro que identifica el dispositivo y por tanto la información contenida en el *SchedulingDecisionHybrid* relacionado, esto se hizo con el fin de ser retrocompatibles, ya que el parámetro tiene un valor por defecto para las llamadas no híbridas. En el caso de algoritmos no híbridos, la relación es 1 a 1 y dado que se tiene el parámetro *device::DEFAULT* por defecto, se logró replicar el funcionamiento que existía previo al cambio.

  Con estas modificaciones se tiene el contexto necesario para poder generar un plan físico híbrido. Resumiendo a grandes rasgos entonces se

Algoritmo 4.2.4: Pseudocódigo del método *hybrid_join* con los conteos de tiempo para cada dispositivo

```
   Entrada: col1, col2, sched_dec
 1 cpu_timer = start_timer();
 2 gpu_timer = start_timer();
 3 Ejecutar de manera concurrente;
 4    col11 = copiar_primer_mitad(col1, gpu);
 5    col12 = copiar_segunda_mitad(col2, cpu);
 6 Ejecutar de manera concurrente;
 7    id1 = join_en_GPU(col11, col2);
 8    id2 = join_en_CPU(col12, col2);
 9 gpu_timer = end_timer(gpu_timer);
10 id = juntar_ids(id1, id2);
11 Se juntan los esquemas de las tablas;
12 Se busca en las tablas involucradas los resultados;
13 cpu_timer = end_timer(cpu_timer);
14 set_tiempo_medido(sched_dec, cpu_timer, cpu);
15 set_tiempo_medido(sched_dec, gpu_timer, gpu);
16 devolver tabla resultado
```

definieron algoritmos híbridos, de los cuales un nuevo criterio de optimización selecciona el par de algoritmos óptimo, finalmente retornando una decisión híbrida que los contiene.

Ejecución de operadores híbridos

Como se mencionó en la sección anterior, se generó un nuevo tipo de plan físico, el cual puede contener operaciones de tipo híbrido. CoGaDB ahora debe ser capaz de manejar este nuevo tipo de operadores, en particular el nuevo tipo de operador a contemplar es el *Join* híbrido.

Las extensiones fundamentales constan de una primer etapa la cual consiste en discernir si el *Join* es de tipo híbrido o no. En el caso de ser híbrido, el *Join* debe recibir el *cpu_per* asociado a cada dispositivo y hacer la partición de datos correspondiente. Otra etapa consiste en medir los tiempos medidos de ejecución de cada dispositivo y así poder refinar los componentes de estimación.

Como se detalló en la sección anterior, el *sched_dec* contiene información sobre si es híbrido o no, esto permite ejecutar el *Join* híbrido implementado por Acuña y Parula en [3]. Ahora el método recibe como parámetro el *cpu_per* de cada algoritmo, el cual es nuevamente obtenido del *sched_dec*, recordar que

Algoritmo 4.2.5: Pseudocódigo de la sección agregada al método *HyPE::Scheduler::add_observation*

```
   Entrada: sched_dec
 1 ...;
 2 si es_hibrido(sched_dec) entonces
 3 |   para dispositivo in obtener_dispositivos(sched_dec) hacer
 4 |   |   t_medido = tiempo_medido(sched_dec, dispositivo);
 5 |   |   t_estimado = tiempo_estimado(sched_dec, dispositivo);
 6 |   |   f = obtener_features(sched_dec, dispositivo);
 7 |   |   alg_dispositivo = algoritmo_dispositivo(sched_dec, dispositivo);
 8 |   |   mapa_operaciones = obtener_mapa_operaciones(scheduler);
 9 |   |   op = obtener_operacion(alg_dispositivo, mapa_operaciones);
10 |   |   refinador_est = construir_refinador(t_medido, t_estimado, f);
11 |   |   nom_alg = obtener_nombre(algoritmo);
12 |   |   add_observation(op, nom_alg, refinador_est);
13 |   fin
14 |   devolver true
15 fin
16 ...;
```

en versiones anterior del *Join* Híbrido el *cpu_per* se configuraba por medio de una bandera global. Finalmente, el *Join* Híbrido también se extiende con el fin de contar los tiempos asociados a ejecutar el algoritmo en cada dispositivo de manera simultanea, y almacenar dicha información en la clase en cuestión, esto permite luego refinar el componente de estimación con los tiempos medidos. El diagrama de las interacciones entre los módulos puede verse en la Figura 4.7. El algoritmo extendido del *Join* híbrido y como se midieron los tiempos de cada dispositivo, se puede observar en el Algoritmo 4.2.4. Los tiempos de CPU serán mayores ya que es en la CPU donde se juntan los esquemas de las tablas y se buscan los resultados en las tablas involucradas. Estos tiempos luego son almacenados en el *sched_dec*, para ser utilizados por el componente de estimación, tanto para estimar los tiempos de ejecución como para estimar los tiempos de espera de cada dispositivo. Estos son los cambios necesarios para que CoGaDB soporte la ejecución de operadores híbridos, en particular esta solución aplica para la operación de *Join*.

Refinamiento del componente de estimación para ejecuciones híbridas

El último componente de interés para la solución es el componente de estimación de HyPE. Hasta el momento el componente de estimación tomaba el resultado de cada operación y actualizaba el modelo del algoritmo asociado. Ahora se tienen operaciones híbridas, las cuales contienen más de un tiempo medido, para cada algoritmo contenido en el *sched_dec*.

La sección de código agregada al método *hype::Scheduler::addObservation* se encuentra en el pseudocódigo del Algoritmo 4.2.5. En el mismo se discute si el *sched_dec* es híbrido, en caso afirmativo, se recorren todos los dispositivos del mismo, y para cada uno de ellos se agrega una observación para la operacion asociada, luego esto desencadenará un refinamiento de los modelos de aprendizaje, que estiman los tiempos de ejecución de un algoritmo de una operación ejecutado en cierto dispositivo.

4.3. Implementación

En la presente sección se describen detalles de la implementación realizada. La Sección 4.3.1 presenta una prueba de concepto realizada con un prototipo, donde se analizaron los resultados en una aplicación que interactúa con HyPE simulando ejecuciones de algoritmos de manera híbrida. En la Sección 4.3.2 se detallan particularidades de los algoritmos, como también decisiones tomadas a la hora de la implementación.

4.3.1. Prueba de concepto

Como se mencionó en la Sección 3.3, HyPE es una biblioteca independiente de CoGaDB, por lo que puede ser integrada en sistemas que tengan la necesidad de optimizar la elección de un algoritmo y el co-procesador en el que ejecuta, obteniendo mejoras en los tiempos de ejecución. Esto permitió implementar un prototipo sencillo que consiste en una pequeña aplicación que integra la biblioteca HyPE con el fin de verificar la correctitud de las extensiones propuestas en la etapa de diseño.

Como se describió en la Sección 3.3.1, para generar el modelo se deben definir los posibles algoritmos que implementan las distintas operaciones, y un conjunto de parámetros que influyen en el tiempo de ejecución.

Algoritmo 4.3.1: Pseudocódigo prototipo

```
1  cpu_pers = [0, 0.3, 0.5, 1];
2  datos = [tamaño];
3  operacion = 'JOIN';
4  algoritmos_cpu = obtener_algoritmos_cpu(cpu_pers, operacion);
5  algoritmos_gpu = obtener_algoritmos_gpu(cpu_pers, operacion);
6  sched_dec = obtener_decision(operacion, datos);
7  si es_hibrido(sched_dec) entonces
8  |  para dispositivo in obtener_dispositivos(sched_dec) hacer
9  |  |  cpu_per = sched_dec.obtenerCpuPer(dispositivo);
10 |  |  si es_CPU(dispositivo) entonces
11 |  |  |  tiempo_cpu = tamaño * cpu_per;
   |  |  |   sched_dec.medir(tiempo_cpu, 'CPU');
12 |  |  fin
13 |  |  si es_GPU(dispositivo) entonces
14 |  |  |  tiempo_gpu = tamaño * (1 - cpu_per) / 2 + 25;
   |  |  |   sched_dec.medir(tiempo_gpu, 'GPU');
15 |  |  fin
16 |  fin
17 fin
18 en otro caso
19 |  si es_CPU(dispositivo) entonces
20 |  |  tiempo_cpu = tamaño * cpu_per;
21 |  |  sched_dec.medir(tiempo_gpu, 'CPU');
22 |  fin
23 |  si es_GPU(dispositivo) entonces
24 |  |  tiempo_gpu = tamaño * (1 - cpu_per) / 2 + 25;
   |  |   sched_dec.medir(tiempo_gpu, 'GPU');
25 |  fin
26 fin
```

Como se puede observar en el Algoritmo 4.3.1, para la prueba de concepto se definió una única operación ficticia *"Join"*, y ocho algoritmos que la implementan, cuatro en GPU, y la misma cantidad en CPU, utilizando los siguientes *cpu_per*: 0, 0.3, 0.5 y 1. La característica definida para el modelo es el tamaño de los datos, y los algoritmos implementados simulan el tiempo de ejecución de la operación para cada dispositivo, en función del tamaño de los datos y el *cpu_per*.

Como el tiempo de ejecución de cada algoritmo es simulado, el modelo se entrena utilizando tiempos calculados lo que hace posible verificar si el algoritmo seleccionado por el modelo es el de menor tiempo. El tiempo simulado

de ejecución para el algoritmo de CPU es:

$$cpu\_per * tamaño\_datos \tag{4.1}$$

mientras que para la GPU es:

$$\frac{((1 - cpu\_per) * tamaño\_datos) + 25}{2} \tag{4.2}$$

De esta manera, luego de hacer varias ejecuciones de entrenamiento variando el tamaño de los datos, se verificó que en todos los casos HyPE seleccionaba correctamente el algoritmo que simulaba el menor tiempo de ejecución. Luego de realizadas estas extensiones a la librería HyPE, junto con el nuevo criterio de optimización híbrido, las mismas fueron adaptadas e integradas a la librería de CoGaDB, como se describió en la Sección 4.2.

4.3.2. Decisiones de implementación

En este apartado se detallan particularidades de los algoritmos como también decisiones tomadas durante la implementación.

Espacio de *cpu_per* definidos

El espacio de *cpu_per* queda definido por medio de las especificaciones de algoritmos que se registren a nivel de HyPE. En el marco de este proyecto se definieron únicamente algoritmos cuya operación asociada sea la de *Join* y los *cpu_per* asociados tengan valor $0, 0.1, 0.2, \cdots, 1$. Dado que internamente se utiliza el *Join* híbrido implementado en [3] y este realiza la partición de datos específicamente para el tipo *Hash Join*, se extendieron nuevos algorítmos únicamente para la operación de *Hash Join*. Para los demás algorítmos existentes, como lo son *Sort Merge Join*, *Nested Loop Join* y *Radix Join*, entre otros, no se extendieron a su versión híbrida.

Notar en la Tabla 4.1 que para los algoritmos no híbridos puros, como los son el caso de *cpu_per* 0 y 1, no es necesario definir su algoritmo complementario en el otro dispositivo. A su vez para cada uno de los algoritmos listados, se los definió para cada uno de los métodos de aprendizaje existentes para la operación (hype::Multilinear_Fitting_2D y *hype::KNN_Regression*).

Etiqueta	Porción de datos	Dispositivo	Tipo de *Join*
CPU_J_Alg_Hybrid_cper_01	10 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_02	20 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_03	30 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_04	40 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_05	50 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_06	60 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_07	70 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_08	80 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_09	90 %	CPU	*Hash Join*
CPU_J_Alg_Hybrid_cper_1	100 %	CPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_0	100 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_01	90 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_02	80 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_03	70 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_04	60 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_05	50 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_06	40 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_07	30 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_08	20 %	GPU	*Hash Join*
GPU_J_Alg_Hybrid_cper_09	10 %	GPU	*Hash Join*

Tabla 4.1: Algoritmos híbridos registrados en HyPE.

Partición lineal de las características de aprendizaje

Las características de aprendizaje se modelan como una tupla cuya primer y segunda entrada son el tamaño de cada una de las columnas involucradas en el *Join*, y en una tercer entrada se almacena un 1 si ambas columnas se encuentra en *caché*, o un 0 en caso contrario. Todas las características son proporcionadas por CoGaDB sin tener en cuenta que el algorítmo puede ser híbrido y por tanto cierta porción de datos de la primer columna ejecuta en la CPU y el restante en la GPU. Al tener distintas porciones de datos para cada dispositivo, es claro que el tamaño de la primer columna involucrada en el *Join* va a ser menor a la original y por lo tanto se debio ajustar estas características de aprendizaje en el Algoritmo 4.2.3, para que el refinamiento de tiempos sea concordante con el tamaño de datos efectivamente procesado por cada dispositivo. Las demás características no sufren cambios y por lo tanto se utilizan las provistas originalmente por CoGaDB.

Si el algoritmo se ejecuta en CPU, la característica asociada al tamaño de

la primer columna se reajusta como:

```
round(característica[0] * cpu_per).
```
[2]

Mientras que para algoritmos que ejecutan en GPU, el reajuste se calcula utilizando el *cpu_per* de la siguiente manera:

```
round(característica[0] * (1 - cpu_per)).
```
[3]

Tiempo de ejecución del *Join* Híbrido para cada dispositivo

Una de las extensiones realizadas al algorítmo de *Join* Híbrido consiste en agregar *timers* que permitan contar el tiempo de ejecución del mismo para cada dispositivo, CPU y GPU. El cálculo de estos tiempos es de suma importancia ya que luego son utilizados para refinar los modelos de aprendizaje, como también para estimar el tiempo de espera de cada dispositivo.

Como se observa en el Algoritmo 4.2.4 el conteo para la CPU comienza en el inicio de la ejecución del mismo, mientras que para la GPU el conteo comienza desde la primer transferencia al realizar la copia de las columnas a la memoria del *device*. Luego de realizado el *Join* y la transferencia desde la GPU, el conteo de tiempo de ejecución de la misma es finalizado, mientras que en la CPU el conteo continua ya que además de realizar el *Join*, tambien se deben juntar los esquemas y buscar los resultados en las tablas involucradas.

El tiempo del algorítmo híbrido óptimo utilizado por los optimizadores híbridos es aquel par de algoritmos que minimiza la siguiente medida:

```
max(t_espera_cpu + t_ejec_cpu, t_espera_gpu + t_ejec_gpu)
```
[4]

Para un par de algoritmos híbridos complementarios uno de los dispositivos va a demorar más que el otro y es este el que en definitiva determina el tiempo del par, luego se toma como par óptimo el que minimice este tiempo.

Variable de ambiente para la ejecución híbrida

CoGaDB permite configurar el dispositivo a ser utilizado para ejecutar las distintas operaciones. Para ello se utiliza el comando *setdevice* indicando *cpu*, *gpu*, *any* o *hybrid*.

Para el caso de ejecución híbrido, se creó una nueva variable de ambiente global llamada *hype::ANY_DEVICE_HYBRID*, la cual representa que el operador a ejecutar puede tener en cuenta algorítmos híbridos - si los tuviera - asociados.

Esta variable actua de similar manera que la variable *hype::ANY_DEVICE* ya que no restringe el dispositivo a ser utilizado, por lo que se modificaron distintas clases de HyPE agregando la nueva variable de manera pertinente para lograr compatibilidad con la funcionalidad ya existente.

Luego el optimizador híbrido es capaz de discernir a partir del *device constraint*, que dispositivos y cuales algorítmos son los candidatos para la ejecución del operador en cuestión. Para el caso de los nuevos optimizadores, estos solicitan todos los algoritmos, incluidos los híbridos cuando el *device constraint* tenga valor *ANY_DEVICE_HYBRID*. Mientras que los optimizadores existentes, no tendran en cuenta esta nueva variable y por lo tanto manejan únicamente algoritmos que no son híbridos.

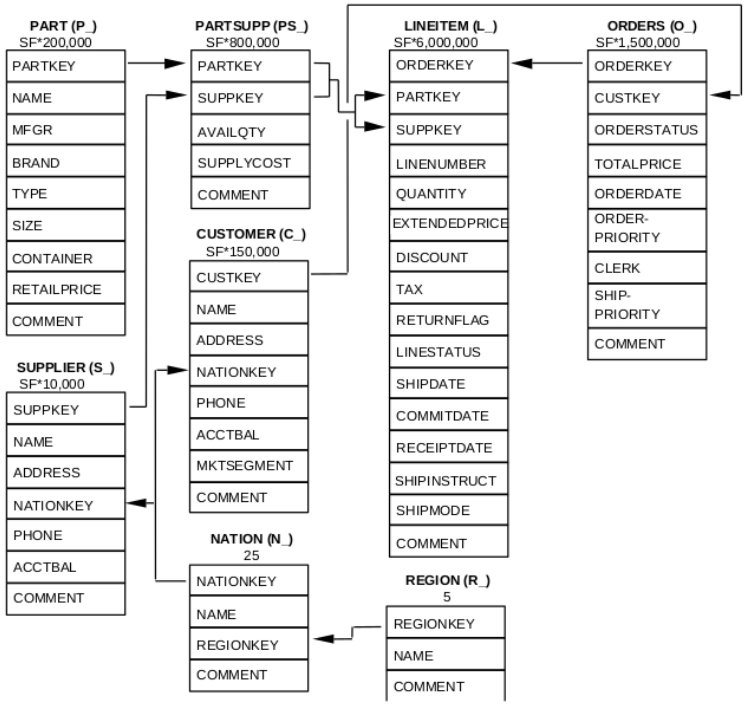

Figura 4.8: Esquema Relacional de TPC-H extraído de [4].

4.4. Evaluación experimental

En esta sección se presentan los resultados obtenidos de la ejecución de distintas consultas, evaluando la elección del *cpu_per* y realizando comparaciones de los tiempos de ejecución obtenidos por los algoritmos de CoGaDB y los híbridos implementados a partir de este proyecto. Se describe a continuación la plataforma de hardware sobre la que se llevaron a cabo las pruebas definidas, las consultas y datos utilizados, y un análisis de los resultados obtenidos.

4.4.1. Plataforma de hardware

La evaluación experimental se ejecutó sobre una PC con sistema operativo Linux Ubuntu 14.04 LTS, con procesador Intel Core i7-8550U (1.80GHz), 16GB de memoria RAM DDR4 y una tarjeta gráfica NVIDIA GeForce MX 150 con 4GB de memoria y 384 CUDA *cores*. Se utilizó la version 0.4.2 de CoGaDB con las extensiones realizada en el proyecto de grado de Acuña y Parula, compilado con CUDA 7.5 y GCC 4.8.8

Caso	Consulta
Q_{s1}	select * from nation join region on n_regionkey = r_regionkey
Q_{s2}	select * from supplier join nation on s_nationkey = n_nationkey
Q_{m1}	select * from partsupp join part on ps_partkey = p_partkey
Q_{m2}	select * from orders join customer on o_custkey = c_custkey
Q_{l1}	select * from lineitem join part on l_partkey = p_partkey
Q_{l2}	select * from lineitem join partsupp on l_partkey = ps_partkey

Tabla 4.2: Conjunto de consultas experimentales.

4.4.2. Contexto de pruebas

Las pruebas fueron realizadas sobre el *benchmark* TPC-H, una base de datos orientada al negocio y creada con el fin de comparar el desempeño de los distintos sistemas manejadores de bases de datos. TPC-H se caracteriza por generar grandes volúmenes de datos y poseer consultas con alto grado de complejidad que utilizan una amplia variedad de operaciones y restricciones de selectividad [4]. El modelo relacional utilizado por el *benchmark* se puede observar en la Figura 4.8.

TPC-H incluye el programa DBGEN el cual genera los datos para cargar las tablas del *benchmark*. DBGEN genera 8 archivos con extensión .tbl, donde cada uno contiene los datos a ser cargados para cada una de las tablas definidas en el esquema de la base de datos TPC-H.

Dado que se quiere evaluar la correcta elección del *cpu_per*, se utilizaron la totalidad de las tablas de manera de poder generar distintos contextos con tamaño variado, en la Tabla 4.2 se definen las consultas utilizadas para cada uno de los casos de prueba mencionados en la Sección 4.4.3.

En la ejecución de las pruebas se deshabilitó la impresión de resultados para la correcta medición de los tiempos. Para deshabilitar dicha opción, se configuró la variable `print_query_result`.

A no ser que se especifique lo contrario para algún caso particular, en todos los casos de pruebas se utilizaron las siguientes configuraciones relacionadas al aprendizaje:

- Criterio de optimización para el *Join* híbrido: *WTARTH*
- Método de aprendizaje: *k-nearest neighbors* (KNN)
- Valor del parámetro k: 3
- Largo de la etapa de entrenamiento: 10
- Refinamiento del modelo: activado
- Periodo para refinamiento del modelo: 100
- Reentrenamiento del modelo: desactivado
- Largo de la etapa de reentrenamiento: 1

Los valores configurados son los que se tienen por defecto en la versión de CoGaDB utilizada en este proyecto, a excepción del valor del parámetro k, el cual se cambió de 10 a 3. Esto es producto de que el largo de la etapa de entrenamiento definido en los casos de prueba es 10, y si se utilizara el valor por defecto para k se evaluarían todas las ejecuciones de entrenamiento como vecinos cercanos para estimar el tiempo de ejecución, esto no es realista si se entrena el modelo con un conjunto de consultas dispares en tamaño. El nuevo valor asignado parece ser razonable si se tiene en cuenta que a lo sumo se entrenan los modelos con 3 consultas diferentes, esto hace que los 3 vecinos más cercanos sean posiblemente más similares a la consulta evaluada.

En cuanto a los algoritmos de *Join* considerados en los casos de prueba solo se evalúan aquellos que internamente son implementados como *Hash Join*. Para los casos de prueba serán objeto de estudio, el *Hash Join* nativo ejecutando en CPU o GPU y los distintos *Hash Join* híbridos.

4.4.3. Casos de prueba

Se definen en esta sección los distintos casos de prueba realizados para evaluar la elección del *cpu_per* y los tiempos de ejecución obtenidos.

Variación del tamaño de las columnas

Se probó el comportamiento del sistema para distintos tamaño de columnas, ejecutando consultas que involucren columnas de pequeño tamaño (aproximadamente 25 valores) y otras de gran tamaño (aproximadamente 6 millones de valores). Se definió un conjunto acotado de consultas descriptas en la Sección 4.4.2, las cuales se nombran como Q_s, Q_m, Q_l, estos son conjuntos que

Modelo entrenado	Conjunto de consultas de entrenamiento	Tamaño de la columnas involucradas
M_s	Q_s	Pequeña.
M_m	Q_m	Mediana.
M_l	Q_l	Grande.
M_r	Q_s, Q_m, Q_l	Cantidad variada.

Tabla 4.3: Modelos generados y sus respectivos entrenamientos.

contienen a las consultas de *Join* que involucran columnas pequeñas, medianas y grandes respectivamente. También se definieron tres conjuntos extras Q'_s, Q'_m, Q'_l los cuales son parecidos - en cuanto a la cantidad de valores involucrados - a los conjuntos Q_s, Q_m, Q_l respectivamente.

En un primer paso se calcula el *cpu_per* óptimo para cada una de estas consultas. Se realizó una exhaustiva ejecución de todas las consultas sobre todos los algoritmos existentes. De esta manera tomando el algoritmo cuyo promedio de tiempos fue menor para todas las ejecuciones, se obtiene el *cpu_per* óptimo para cada una de las consultas evaluada.

En un segundo paso se realiza el entrenamiento, el cual se separa en tres casos posibles, en los cuales se utilizan consultas variadas de distinto tamaño. Cada uno de estos casos devuelve un modelo distinto.

El modelo M_s consiste en un modelo entrenado con consultas que involucran columnas de pequeño tamaño es decir con aquellas consultas contenidas en el conjunto Q_s, análogamente se tiene el modelo M_m y M_l restringiendo su entrenamiento para las consultas pertenecientes a Q_m y Q_l respectivamente. Finalmente, se obtiene un modelo más realista M_r, el cual es entrenado con todas las consultas definidas en los conjuntos, y por tanto su entrenamiento es realizado sobre consultas con columnas de tamaño variado. En la Tabla 4.3 se tiene un resumen de los modelos generados para cada caso.

En un tercer paso se realiza la ejecución distintas consultas sobre los modelos definidos, se busca probar sobre consultas ya conocidas o similares (en tamaño de columna a particionar) como también consultas nunca antes ejecutadas sobre los mismos.

En la Tabla 4.4 se resumen los casos de ejecución sobre cada uno de los modelos.

Con los distintos casos de ejecuciones se buscó: ejecutar el modelo sobre

Caso de ejecución	Modelo	Entrenamiento del modelo	Evaluación del modelo
Caso MS1	M_s	Q_s	Consulta conocida.
Caso MS2	M_s	Q'_s	Consulta similar.
Caso MS3	M_s	Q_m, Q_l	Consulta diferente.
Caso MM1	M_m	Q_m	Consulta conocida.
Caso MM2	M_m	Q'_m	Consulta similar.
Caso MM3	M_m	Q_s, Q_l	Consulta diferente.
Caso ML1	M_l	Q_l	Consulta conocida.
Caso ML2	M_l	Q'_l	Consulta similar.
Caso ML3	M_l	Q_s, Q_m	Consulta diferente.
Caso MR1	M_r	Q_s, Q_m, Q_l	Consulta conocida.
Caso MR2	M_r	Q'_s, Q'_m, Q'_l	Consulta similar.

Tabla 4.4: Listado de casos de ejecuciones para cada uno de los modelos.

las consultas de entrenamiento, ejecutar el modelo sobre consultas similares a las de entrenamiento, y ejecutar consultas muy diferentes a las utilizadas para el entrenamiento.

En cuanto a los casos:

- Como se observa MS1, MM1, ML1 y MR1 son similares en el hecho de que se prueba el modelo sobre las mismas consultas sobre las que se entrenaron. Por esta razón es de esperar que para estos casos el desempeño del modelo sea muy bueno.
- Por otra parte MS2, MM2, ML2 y MR2 son similares ya que se ejecutan consultas no conocidas por el modelo pero que involucran una cantidad de valores de las columnas similar que las consultas de entrenamiento.
- Finalmente, MS3, MM3 y ML3 prueban el modelo sobre consultas no conocidas, y cuya cantidad de valores involucradas en las columnas de *Join* es muy distinta a las cantidades de datos manejados en la etapa de entrenamiento.

En una cuarta y última etapa, se evaluó el desempeño de cada uno de los modelos entrenados para los distintos casos.

Variación de los algoritmos híbridos definidos

Se realizaron distintas pruebas para las cuales se definieron distintos conjuntos de *cpu_per*, es decir, se definieron casos teniendo una distinta cantidad

de algoritmos híbridos, es de notar que a mayor algoritmos híbridos definidos mayor es la cantidad de posibles *cpu_per* ocupados.

Variación de las variables de aprendizaje

En otro juego de pruebas se optimizaron los distintos parámetros de aprendizaje provistos por HyPE:

- **Período de refinamiento:** número de ejecuciones para refinar el modelo con lo aprendido hasta el momento.
- **Período de reentrenamiento:** número de ejecuciones de la operación sin que ejecute un algoritmo para volver a reentrenarlo.
- **Largo del reentrenamiento:** número de ejecuciones de cada algoritmo para completar el reentrenamiento.

4.4.4. Resultados obtenidos

En esta sección se reportan los resultados obtenidos a partir de los casos de prueba definidos en la Sección 4.4.3.

Variación del tamaño de las columnas

Los tiempos reportados corresponden a 10 ejecuciones de entrenamiento para cada uno de los algoritmos de *Join* existentes, dando como resultado un modelo para el cual se realiza la ejecución de una única consulta de evaluación, luego el modelo elige el *cpu_per* óptimo teniendo en cuenta las características de la misma, esto es, el tamaño de las columnas que intervienen en el *Join* y si se encuentran en la *caché*, obteniendo de esta manera el *cpu_per* que arrojó el menor tiempo.

En la Tabla 4.5 se detalla para cada consulta el *cpu_per* óptimo seleccionado. A su vez en las Figuras 4.9, 4.10 y 4.11 se muestra una comparativa con los tiempos promedios de cada *cpu_per* para las mismas consultas. Los resultados obtenidos son coherentes para los distintos casos, teniendo en cuenta la plataforma de hardware utilizada, y el tamaño de cada consulta. Para las consultas pequeñas, Q_{s1} y Q_{s2}, se observa claramente que la CPU es la que tiene mejor rendimiento, en comparativa con la GPU y los restantes algoritmos híbridos que empeoran los tiempos, principalmente por el costo adicional que implica la transferencia de datos al co-procesador. Respecto a la consulta Q_{m1},

Consulta	*cpu_per* óptimo	Tamaño columnas	Tamaño resultado
Q_{s1}	1	$25 * 5$	25
Q_{s2}	1	$10.000 * 25$	10.000
Q_{m1}	0	$800.000 * 200.000$	800.000
Q_{m2}	0.1	$1.500.000 * 150.000$	1.500.000
Q_{l1}	0.2	$6.001.215 * 200.000$	6.001.215
Q_{l2}	0.2	$6.001.215 * 800.000$	24.004.860

Tabla 4.5: Consultas experimentales con su *cpu_per* óptimo.

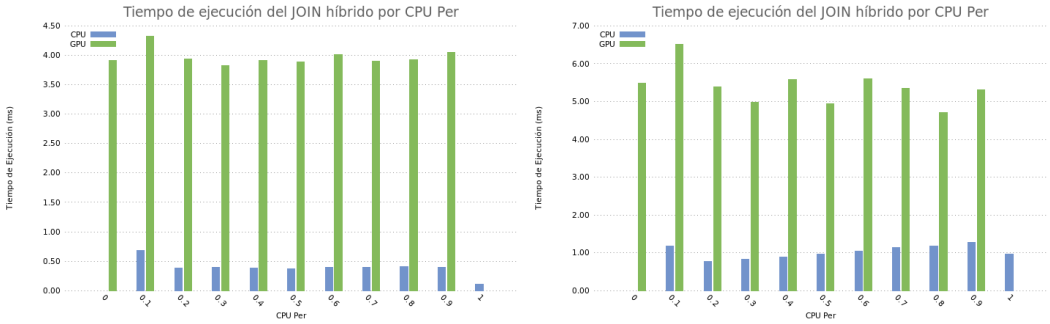

Figura 4.9: Tiempo de ejecución para cada dispositivo por *cpu_per* para las consultas Q_{s1} y Q_{s2}.

se observa que el mejor tiempo se da ejecutando la consulta enteramente en GPU, a diferencia de Q_{m2}, Q_{l1} y Q_{l2} que obtienen mejores tiempos con una ejecución híbrida. Esto tiene sentido, ya que la capacidad de cómputo de la GPU es suficiente para ejecutar la consulta Q_{m1} sin la cooperación de la CPU, mientras que para consultas de mayor tamaño, se obtienen mejores tiempos otorgando una porción de los datos a la CPU.

Modelo evaluado con consulta conocida

Esta etapa experimental consiste de la ejecución de los casos MS1, MM1, ML1 y MR1. Para estos casos se prueba el modelo sobre la misma consulta con la que se entrenó.

En la Tabla 4.6 se detalla el caso de prueba ejecutado junto con la consulta evaluada del conjunto de consultas definidas en la Tabla 4.2 y para cada uno de estos se indica el *cpu_per* elegido por el modelo.

Como se observa en los resultados el *cpu_per* seleccionado coincide en todos los casos con el *cpu_per* óptimo calculado previamente en la Tabla 4.5. Tal y como se esperaba para este caso, el desempeño del modelo en cuanto a la

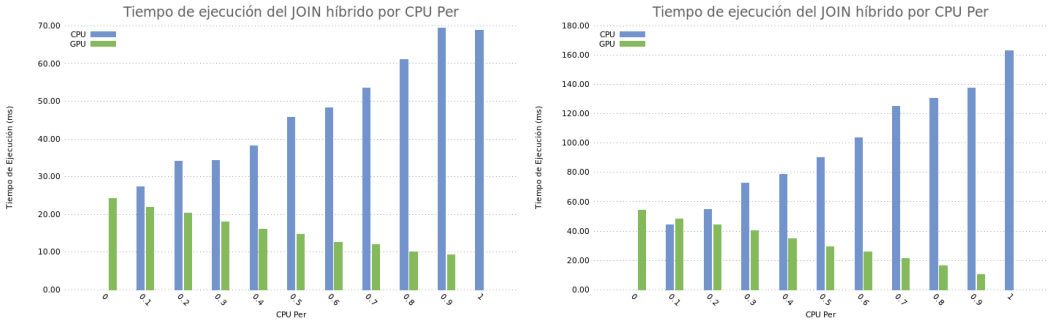

Figura 4.10: Tiempo de ejecución para cada dispositivo por *cpu_per* para las consultas Q_{m1} y Q_{m2}.

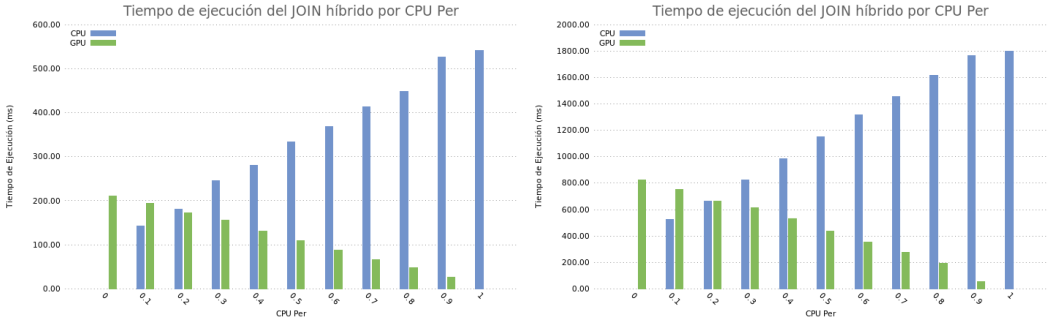

Figura 4.11: Tiempo de ejecución para cada dispositivo por *cpu_per* para las consultas Q_{l1} y Q_{l2}.

elección del *cpu_per* óptimo, es bueno dado que es una consulta conocida.

Modelo evaluado con consulta desconocida pero de similar tamaño

Esta etapa experimental consiste de la ejecución de los casos MS2, MM2, ML2 y MR2. Para los cuales se ejecutan consultas no conocidas por los modelos, pero que involucran columnas con similar cantidad de valores que las columnas utilizadas en las consultas de entrenamiento. En la Tabla 4.7 se detallan los resultados obtenidos para los distintos casos. Como se puede observar, el *cpu_per* seleccionado coincide con el *cpu_per* óptimo, esto es debido a que el tiempo estimado para cada algoritmo en el modelo tiene características similares en tamaño a la de las columnas en la consulta evaluada, obteniendo una estimación óptima.

Modelo evaluado con consulta totalmente desconocida

Esta etapa experimental consiste de la ejecución de los casos MS3, MM3, ML3. Para estos casos, presentados en la Tabla 4.8, se prueba el modelo sobre con-

Caso de Prueba	**Consulta entrenada**	**Consulta evaluada**	***cpu_per* esperado**	***cpu_per* seleccionado**
Caso MS1	Q_{s1}	Q_{s1}	1	1
Caso MS1	Q_{s2}	Q_{s2}	1	1
Caso MM1	Q_{m1}	Q_{m1}	0	0
Caso MM1	Q_{m2}	Q_{m2}	0.1	0.1
Caso ML1	Q_{l1}	Q_{l1}	0.2	0.2
Caso ML1	Q_{l2}	Q_{l2}	0.2	0.2
Caso MR1	Q_{s1}, Q_{m1}, Q_{l1}	Q_{s1}	1	1
Caso MR1	Q_{s1}, Q_{m1}, Q_{l1}	Q_{m1}	0	0
Caso MR1	Q_{s1}, Q_{m1}, Q_{l1}	Q_{l1}	0.2	0.2

Tabla 4.6: Casos de pruebas con el *cpu_per* seleccionado para la consulta conocida evaluada.

Caso de Prueba	**Consulta entrenada**	**Consulta evaluada**	***cpu_per* esperado**	***cpu_per* seleccionado**
Caso MS2	Q_{s1}	Q_{s2}	1	1
Caso MS2	Q_{s2}	Q_{s1}	1	1
Caso MM2	Q_{m1}	Q_{m2}	0	0
Caso MM2	Q_{m2}	Q_{m1}	0.1	0.1
Caso ML2	Q_{l1}	Q_{l2}	0.2	0.2
Caso ML2	Q_{l2}	Q_{l1}	0.2	0.2
Caso MR2	Q_{s1}, Q_{m1}, Q_{l1}	Q_{s2}	1	1
Caso MR2	Q_{s1}, Q_{m1}, Q_{l1}	Q_{m2}	0.1	0.1
Caso MR2	Q_{s1}, Q_{m1}, Q_{l1}	Q_{l2}	0.2	0.2

Tabla 4.7: Casos de pruebas con el *cpu_per* seleccionado para la consulta desconocida con tamaño similar evaluada.

sultas no conocidas y cuya cantidad de valores de las columnas involucradas en el *Join* es muy distinta a las cantidades de datos manejados en la etapa de entrenamiento. Por este mismo motivo, las estimaciones obtenidas no coinciden con lo esperado, ya que el modelo no conoce lo que se esta evaluando. Más adelante se evalúan estos mismos casos modificando las variables de aprendizaje de HyPE, analizando en cuantas ejecuciones se puede obtener una estimación estable al *cpu_per* óptimo.

Caso de Prueba	Consulta entrenada	Consulta evaluada	*cpu_per* esperado	*cpu_per* seleccionado
Caso MS3	Q_{s1}, Q_{s2}	Q_{m1}	0	1
Caso MS3	Q_{s1}, Q_{s2}	Q_{l1}	0.2	1
Caso MM3	Q_{m1}, Q_{m2}	Q_{s1}	1	0
Caso MM3	Q_{m1}, Q_{m2}	Q_{l1}	0.2	0.1
Caso ML3	Q_{l1}, Q_{l2}	Q_{s1}	1	0.2
Caso ML3	Q_{l1}, Q_{l2}	Q_{m1}	0	0.2

Tabla 4.8: Casos de pruebas con el *cpu_per* seleccionado para la consulta desconocida evaluada.

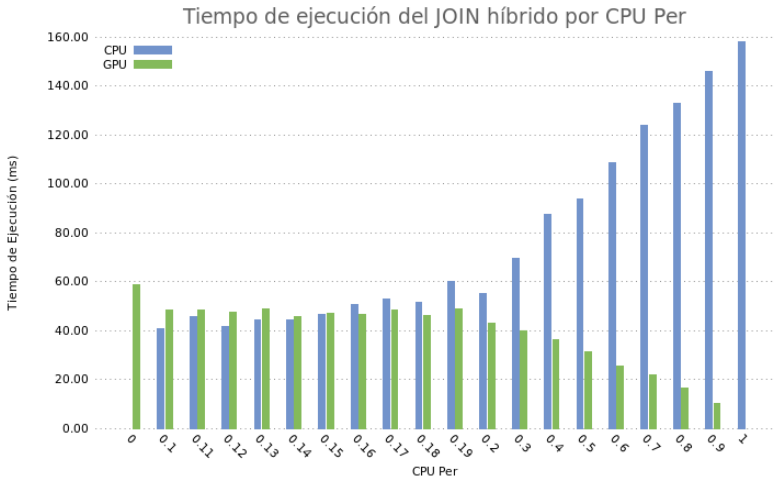

Figura 4.12: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM1 entrenado con la consulta Q_{m2} para ser evaluado con la consulta Q_{m2} con el nuevo espacio de *cpu_per*.

Variación de los algoritmos híbridos definidos

El objetivo de esta sección es aumentar la cantidad de algoritmos híbridos de tal manera que se mejore el tiempo de ejecución para uno de los casos de prueba analizados en secciones anteriores.

El enfoque es sobre el caso MM1 entrenado y evaluado con la consulta Q_{m2} cuyo *cpu_per* óptimo es 0.1 en el espacio de *cpu_pers* con valores $0.1, 0.2, \ldots, 1$. Para extender el espacio de *cpu_pers* se agregan los valores decimales entre 0.1 y 0.2 $(0.11, 0.12, \ldots, 0.19)$, ya que estos se encuentran próximos al *cpu_per* óptimo del caso en estudio.

En la Tabla 4.9 y en la Figura 4.12 se tienen las ejecuciones y sus tiempos para cada dispositivo con cada *cpu_per*. Se observa que cuatro de los nuevos

cpu_per	Tiempo CPU	Tiempo GPU
0	–	58.65
0.1	41.02	48.62
0.11	45.98	48.59
0.12	41.55	47.55
0.13	44.28	48.73
0.14	**44.49**	**45.73**
0.15	46.49	47.12
0.16	50.50	46.81
0.17	52.93	48.26
0.18	51.63	46.10
0.19	60.32	48.73
0.2	55.30	43.01
0.3	69.45	40.15
0.4	87.76	36.21
0.5	93.68	31.31
0.6	108.74	25.63
0.7	123.92	21.90
0.8	133.01	16.33
0.9	146.18	10.30
1	158.05	–

Tabla 4.9: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM1 entrenado y evaluado con la consulta Q_{m2} con el nuevo espacio de *cpu_pers*.

algoritmos híbridos definidos mejoran los tiempos obtenidos respecto al espacio de *cpu_per* anterior. En el nuevo espacio el *cpu_per* óptimo es 0.14 con una mejora del tiempo de ejecución de aproximadamente 2.75 milisegundos.

Esto hace que el espacio de *cpu_per* que se defina juega un rol importante a la hora de decidir el *Join* híbrido que dará los mejores resultados y este puede variar a partir de los datos o de las capacidades computacionales que se tengan. Queda fuera del alcance de este proyecto, pero parece interesante que la definición del espacio de *cpu_per* se genere de forma dinámica e inteligente teniendo en cuenta el contexto sobre el cual se está ejecutando.

Variación de las variables de aprendizaje

Período de refinamiento

El período de refinamiento consiste en el número de ejecuciones que deben de ocurrir en el algoritmo para refinar el modelo de aprendizaje asociado.

Período de refinamiento	Primera ejecución estable	Número de cambio de elección	Número de decisiones correctas
1	34	22	165
2	45	21	154
3	43	14	158
4	45	11	154
5	51	10	144
6	61	10	134
7	71	10	124
8	89	11	114
9	100	11	104
10	111	11	94

Tabla 4.10: Para el caso MS3 entrenado con las consultas Q_{s1} y Q_{s2} y evaluado con la consulta Q_{m1}, se varía el período de refinamiento, registrando la primera ejecución estable, el número de cambios de elección de los distintos algoritmo hasta llegar al óptimo, y el número de decisiones correctas.

HyPE almacena todo el historial de ejecuciones junto con los datos necesarios para poder actualizar el modelo, pero este refinamiento del modelo no ocurre en cada ejecución ya que puede ser costoso, la variable de ambiente `HYPE_RECOMPUTATION_PERIOD` es la que permite configurar este comportamiento.

En esta sección el enfoque está centrado sobre el caso MS3 entrenado con las consultas Q_{s1} y Q_{s2} y evaluado con la consulta Q_{m1}. De la Tabla 4.8 se sabe que el *cpu_per* óptimo es 0, es decir, que toda la ejecución ocurra en la GPU, sin embargo HyPE decide ejecutar la consulta completamente en la CPU, lo cual no es del todo performante ya que la consulta es lo suficiente grande para sacar provecho en la paralelización que se puede llevar a cabo en la GPU. Se verá como el período de refinamiento permite reajustar el modelo paso a paso, hacía la elección del algoritmo óptimo.

A diferencia de la prueba realizada en la Tabla 4.8, la evaluación es realizada 181 veces sobre la consulta Q_{m1}, justamente con el fin de poder reajustar la elección del algoritmo.

En la Tabla 4.10 se observa como a medida que se disminuye el valor del período de refinamiento, se accede en menos ejecuciones al *cpu_per* correcto, como contra parte, se prueba una mayor cantidad de algoritmos hasta estabi-

lizarse en el correcto.

En la Figura 4.13 se muestra como varía el *cpu_per* en relación al número de ejecución para el caso en el cual el período de refinamiento vale 1.

Ajustando esta variable de aprendizaje se logró corregir con éxito el caso MS3 que lejos estaba de seleccionar el *cpu_per* óptimo, sino que por el contrario elegía el menos eficiente.

Período y largo del reentrenamiento

El período de reentrenamiento se define como el número de ejecuciones en el cual un algoritmo no fue ejecutado y por lo cual debe de reentrenar, mientras que el largo del reentrenamiento consiste en el numero de ejecuciones del algoritmo para dar por finalizado su reentrenamiento. HyPE permite configurar ambos comportamientos con las variables `HYPE_ALGORITHM_MAXIMAL_IDLE_TIME` y `HYPE_RETRAINING_LENGTH`, respectivamente. Para poder reentrenar obligatoriamente `HYPE_RECOMPUTATION_PERIOD` debe de valer 1, para que los modelos de los algoritmos reentrenados se ajusten con la información de la nueva ejecución.

El objetivo de esta sección consiste en configurar ambas variables de una manera óptima para poder mejorar un caso de prueba anterior para el cual la elección del *cpu_per* no haya sido el óptimo. Se analiza como caso de prueba el MM3 entrenado con las consultas Q_{m1} y Q_{m2} y evaluado con la consulta Q_{s1}. De la Tabla 4.8 se sabe que el *cpu_per* óptimo es 1, es decir, que toda la

Figura 4.13: Variación del *cpu_per* elegido por ejecución para período de refinamiento con valor 1.

Período de reentrenamiento	Primera ejecución estable	Reentrenamientos requeridos
2	9	4
3	13	4
4	17	4
5	21	4

Tabla 4.11: Para el caso de prueba MM3 entrenado con las consultas Q_{m1} y Q_{m2} y evaluado con la consulta Q_{s1}, se varía el período de reentrenamiento, registrando la primer ejecución estable y la cantidad de reentrenamientos requeridos para que HyPE tome una correcta decisión. El largo del reentrenamiento para estas ejecuciones es 1.

ejecución ocurra en la CPU, sin embargo HyPE hace todo lo contrario, ejecuta la consulta completamente en la GPU, lo cual no es performante ya que la consulta es pequeña como para transferir parte de ella a la GPU, sabiendo que la transferencia de datos de un dispositivo a otros es un cuello de botella cuando los tiempos de ejecución se encuentran en magnitudes bajas. Se verá como el período de reentrenamiento permite reentrenar el modelo con una consulta desconocida y como este comienza a tomar mejores decisiones hasta llegar al *cpu_per* óptimo.

A diferencia de la prueba realizada en la Tabla 4.8, la evaluación es realizada 181 veces sobre la consulta Q_{s1}, con el fin de poder reentrenar los distintos algoritmos que no fueron seleccionados.

Para las ejecuciones con las que se construyó la Tabla 4.11, se fijó el valor del largo del reentrenamiento en 1 y se fue variando el período de reentrenamiento con el fin de analizar cuantos reentrenamientos son necesarios para llegar al óptimo, y en que número de ejecución sucede. A medida que el período de reentrenamiento aumenta, lo mismo ocurre con la primera ejecución estable, esto es porque el período de reentrenamiento afecta directamente en la cantidad de ejecuciones del algoritmo que está siendo seleccionado actualmente. Sin embargo variar el período de reentrenamiento no afecta en la cantidad de reentrenamientos requeridos para que HyPE se decante por el *cpu_per* óptimo.

Se ejecutó el mismo caso de prueba, pero ahora fijando el periodo de reentrenamiento en 2, ya que este fue el que dio mejores resultados, y se fue variando el largo del reentrenamiento con el fin de analizar la cantidad de reentrenamientos requeridos para este caso. Se observa en la Tabla 4.12 que a

Largo del reentrenamiento	Primera ejecución estable	Reentrenamientos requeridos
2	5	2
3	2	1
4	2	1

Tabla 4.12: Para el caso de prueba MM3 entrenado con las consultas Q_{m1} y Q_{m2} y evaluado con la consulta Q_{s1}, se varía el largo del reentrenamiento, registrando la primer ejecución estable y la cantidad de reentrenamientos requeridos para que HyPE tome una correcta decisión. El período de reentrenamiento para estas ejecuciones es 2.

medida que se aumenta el largo del reentrenamiento la cantidad de reentrenamientos requeridos es menor y a su vez se llega con mayor rapidez a la primera ejecución estable. En general se tenía que la cantidad de reentrenamientos requeridos era 4, lo mismo parece ocurrir para este caso, si tenemos en cuenta que la cantidad de unidades de reentrenamientos realizados esta dado por el largo del reentrenamiento multiplicado por la cantidad de etapas de reentrenamiento. El mejor caso se da cuando el largo del reentrenamiento es 3, ya que se necesitan tan solo 3 unidades de reentrenamiento a comparación de las 4 necesarias en todas las demás ejecuciones.

Se logró el objetivo que consistía en mejorar uno de los casos de pruebas anteriores para los cuales no se habían tenidos buenos resultados. De cualquier manera es importante detallar que en un ambiente real de producción no parece conveniente reentrenar en pequeños períodos de ejecución, por un lado porque el reentrenamiento es costoso y por tanto el sistema en general decaería en performance en cuanto a la cantidad de pedidos simultaneos que puede atender. Por otro lado mientras se está reentrenando, HyPE no toma el mejor algorítmo si no que va seleccionandolos con el método de *Round-robin*, haciendo que en esta etapa, la ejecución de las consultas no den necesariamente tiempos de ejecución óptimos. Por lo tanto debe de existir un correcto estudio a la hora de configurar estas variables.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se describen las conclusiones que se obtuvieron por medio de la realización del proyecto. Se definen además posibles mejoras a futuro que se detectaron.

5.1. Conclusiones

El objetivo de este proyecto consistía en el uso de aceleradores de hardware en sistemas de Bases de Datos relacionales. En particular, se buscaron opciones de mejora a este tipo de sistemas con el fin de reducir los tiempos de ejecución para una consulta, haciendo un uso óptimo de los recursos computacionales.

En primera instancia se realizó una investigación de los manejadores acelerados mediante el uso de GPUs, existentes a la fecha. Se realizó una comparación, viendo características de los mismos y proveniente de la investigación se optó por continuar utilizando la herramienta CoGaDB con las extensiones realizadas en el proyecto antes mencionado.

Luego de un análisis en profundidad del manejador y de los posibles problemas a los que se enfrentaba, se decidió mejorar el *Join* híbrido concurrente. En particular se implementó un método que permita seleccionar el *cpu_per* óptimo dependiente del tamaño de las columnas involucradas en la operación de *Join*.

Una vez implementado, se realizo una evaluación experimental, con distintos casos de prueba aplicados sobre varios conjuntos de datos. Sobre los resultados obtenidos, se observa que:

- Utilizando el esquema TPC-H, para consultas que involucran el opera-

dor de *Join* sobre columnas pequeñas (10.000 valores aproximadamente) se selecciona *cpu_per*=1, ejecutando únicamente en la CPU. Cuando el tamaño de las columnas involucradas en el operador de *Join* es mediano (800.000 valores aproximadamente), el dispositivo seleccionado es la GPU con *cpu_per*=0. Hasta este punto no se ejecuta el *Join* híbrido concurrente, luego para consultas grandes (de 1 millon a 24 millones de tuplas) se toman *cpu_per* híbridos, partiendo con valores 0.1 y tendiendo a 0.2 a medida que aumenta la cantidad de datos. Para estos casos el *Join* híbrido concurrente obtiene mejores tiempos que la ejecución en un único único dispositivo.

- En el proyecto anterior el *cpu_per* óptimo tendía a 0.3, mientras que la misma consulta en este proyecto tiende a 0.2, esto es debido a las mejoras computacionales en los dispositivos utilizados. A medida que los coprocesadores sean más dispares, es decir, que la GPU sea sumamente superior en capacidades de computo que la CPU, menor sera la utilización de la CPU por ser un cuello de botella y por tanto el *cpu_per* tenderá hacia el valor 0.
- A medida que los dispositivos son más dispares en cuanto a su capacidad de computo, no parece provechoso la utilización de algorítmos híbridos dado que las mejoras de tiempos no serían demasiado beneficiosas, sumado al mantenimiento a realizar referente a los modelos (entrenamiento, reentrenaiento y refinamiento). Sin embargo existe una limitante que es la capacidad de memoria existente en la GPU. Para consultas lo suficientemente grandes que no puedan ser ejecutadas en su totalidad en la GPU, el *Join* híbrido concurrente con su partición de datos hace que sea capaz de transferir una cantidad adecuada a la GPU, de manera de paralelizar la ejecución de manera óptima, sin desaprovechar datos ya computados. El análisis en profundidad de esté tema escapa del alcance de este proyecto, quedando como línea de trabajo a futuro.
- Teniendo varios modelos definidos utilizando técnicas de aprendizaje estadístico, sus desempeños dependen directamente de la similitud de la consulta evaluada con las utilizadas en la etapa de entrenamiento. Cuando el modelo es evaluado con una consulta para la cual fue previamente entrenado, el desempeño del mismo en cuanto a la elección del *cpu_per* es óptimo. En segundo lugar, cuando la consulta no es conocida por el modelo, pero involucra *Joins* cuyo tamaño de las columnas es similar a

las entrenadas, también el *cpu_per* estimado es el óptimo, debido a que el tiempo estimado para cada algoritmo del modelo tiene características similares. Por último, cuando la consulta evaluada no es conocida por el modelo y el tamaño de las columnas es muy dispar a las entrenadas por el mismo, las estimaciones obtenidas no coinciden con el óptimo, ya que el modelo no conoce lo que esta evaluando.

- Al generar un espacio de *cpu_per* extendido, es decir, con una mayor cantidad de *cpu_per* disponibles para la elección del optimizador HyPE, se observan pequeñas mejoras en los tiempos de ejecución obtenidos. Para ello se definieron nuevos algoritmos de *Join* híbrido concurrente cuyo *cpu_per* asociado $(0.10, 0.11, 0.12, \cdots, 0.19)$, se encuentre cercano al *cpu_per* óptimo obtenido (0.1 para el caso de prueba seleccionado) con el espacio de *cpu_per* original $(0, 0.1, 0.2, \cdots, 1)$.
- Variando las variables de aprendizaje (período de refinamiento, período de entrenamiento, largo de reentrenamiento) se logran mejoras significativas para aquellos casos de prueba en los cuales la elección del *cpu_per* era incorrecto. Se debe de realizar un correcto estudio a la hora de configurar estas variables para obtener los mejores desempeños sin perjudicar el funcionamiento del sistema en general.

5.2. Trabajo futuro

Como trabajo futuro, en el marco de este proyecto se consideró importante que la definición del espacio de *cpu_per* se genere de forma dinámica e inteligente teniendo en cuenta el hardware subyacente en el cual se está ejecutando. Actualmente el espacio se define manualmente pero se demostró que modificando el espacio se obtienen mejores resultados.

Se propone también paralelizar de manera híbrida concurrente suboperaciones de la consulta que no sean dependientes entre si, teniendo en cuenta el plan de consulta generado, como también extender el *Join* híbrido concurrente para que soporte otro tipo de implementaciones distintas a la del *Hash Join*.

Se esbozaron conclusiones al respecto pero quedaron fuera del alcance del proyecto el estudio y análisis del comportamiento para computadoras cuyos coprocesadores sean muy dispares, como también el comportamiento en casos para los cuales la memoria existente en la GPU no es suficiente para ejecutar

una consulta lo suficientemente grande.

Otro trabajo a futuro es lograr que la ejecución del *Join* híbrido concurrente ejecute sobre varias GPUs en simultáneo.

Anexo A

Bugs en CoGaDB

En esta sección se describen los bugs encontrados durante la realización del proyecto, junto con la solución encontrada para cada uno.

A.1. *Bugs* encontrados

Uso del *caché* en el *Join* híbrido

El *Join* híbrido implementado no utiliza el *Caché* provisto por CoGaDB.

Largo de reentrenamiento configurable

Por defecto el largo de reentrenamiento es 1, configurar la variable `HYPE_RETRAINING_LENGTH` no genera ningún cambio en el comportamiento por defecto.

A.2. Solución de los *bugs* encontrados

A continuación se describen las soluciones que fueron encontradas para resolver los *bugs* descriptos en la sección anterior.

Uso del *caché* en el *Join* híbrido

Para los algoritmos nativos CoGaDB previamente a realizar la copia a la GPU de las columnas involucradas, verifica en sus estructuras de *caché* si la copia es requerida. En el caso de los algoritmos híbridos este chequeo no se realizaba, y por lo tanto se copiaban las columnas en todos los casos. Por

Algorithm A.1: Función que realiza la copia de columna si no se encuentra en *caché*. De lo contrario, si la columna se encuentra en *caché*, se reajusta y retorna la porción utilizada por el algoritmo actual.

```
1  ColumnPtr copy_partition_if_required(ColumnPtr col, size_t
       limit, size_t offset, const hype::ProcessingDeviceMemoryID&
       mem_id) {
2          if (col->getMemoryID() == mem_id) {
3              return resize_reference(col, limit, offset, mem_id)
       ;
4          } else {
5              if (!isCPUMemory(mem_id)) {
6                  ColumnPtr cached = DataCacheManager::instance()
       .getDataCache(mem_id).getColumn(col);
7                  if ((cached) && (cached->size() == col->size())
       ) {
8                      return resize_reference(cached, limit,
       offset, mem_id);
9                  }
10             }else{
11                 ColumnPtr cached = DataCacheManager::instance()
       .getDataCache(col->getMemoryID()).getHostColumn(col);
12
13                 if(cached) {
14                    return resize_reference(cached, limit,
       offset, mem_id);
15                 };
16             }
17
18             return copy_partition(col, limit, offset, mem_id);
19         }
20     }
```

otra parte, para los algoritmos híbridos sucede que la columna en su totalidad puede estar en *caché*, pero al retornar la misma es necesario ajustar su tamaño ya que solo una porción de esta es utilizada en la ejecución del *Join*.

En los Algoritmos A.1, A.2, A.3 se presenta la solución implementada.

Largo de reentrenamiento configurable

La solución consiste en modificar la condición por la cual se desactiva la fase de reentrenamiento. En la versión de CoGaDB esta fase era desactivada cuando alguna de las siguientes condiciones evaluaba como verdadera:

- Si no se estaba en la fase de entrenamiento: esto evaluaba como verdadero en todos los casos ya que nunca se estaba en ambas fases en simultaneo. Se resolvió remover la condición.

Algorithm A.2: Método que reajusta el tamaño de la columna.

```
1 template<class T>
2 const ColumnPtr Column<T>::resize_reference(size_t limit,
    size_t offset, const hype::ProcessingDeviceMemoryID& mem_id)
     const {
3     ColumnPtr new_col = ColumnPtr(new Column<T>(*this, limit,
    offset, mem_id));
4
5     return new_col;
6 }
```

Algorithm A.3: Constructor que reajusta el tamaño de una columna existente.

```
1 template<class T>
2 Column<T>::Column(const Column<T> &x, size_t limit, size_t
    offset, const hype::ProcessingDeviceMemoryID& mem_id) :
    ColumnBaseTyped<T>(x.getName(), x.getType(),
    PLAIN_MATERIALIZED),
3                                         type_tid_comparator(),
    values_(0), num_elements_(0), buffer_size_(0),
4                                         mem_alloc(
    MemoryAllocator<T>::getMemoryAllocator(mem_id)) {
5
6   // reasigna la referencia
7   // al primer valor de la columna
8   values_ = x.values_ + offset;
9
10  // ajusta la cantidad de elementos
11  // de la nueva columna
12  num_elements_ = limit;
13
14  // utilizado para no liberar memoria
15  // para porciones de columnas
16  // ya que la liberacion ocurre siempre
17  // en la columna original
18  reference_to_values_ = true;
19 }
```

- Si la cantidad de reentrenamientos realizados es menor o igual a la configuración: la condición correcta es la opuesta, es decir, que no se debe desactivar la fase de reentrenamiento hasta no llegar al umbral de la configuración realizada.

En el Algoritmo A.4 se describe la solución planteada.

Algorithm A.4: Solución al largo de reentrenamiento configurable.

```
1 if(Runtime_Configuration::instance().getRetrainingLength() <
     retraining_length_++) {
2   is_in_retraining_phase_=false;
3   ...
4 }
```

Anexo B

Resultados detallados

A continuación se detallan las tablas con las ejecuciones realizadas para la evaluación experimental en la Sección 4.4.

B.1. Tablas de resultados obtenidos variando el tamaño de las consultas del *Join*

.

En esta sección se muestran las tablas con los resultados obtenidos al variar el tamaño de las consultas.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	3.91
0.1	0.69	4.32
0.2	0.39	3.94
0.3	0.39	3.83
0.4	0.38	3.92
0.5	0.37	3.89
0.6	0.39	4.01
0.7	0.40	3.90
0.8	0.41	3.93
0.9	0.39	4.04
1	**0.11**	**-**

Tabla B.1: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MS1 entrenado con la consulta Q_{s1} para ser evaluado con la consulta Q_{s1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	5.48
0.1	1.18	6.51
0.2	0.78	5.38
0.3	0.82	4.99
0.4	0.88	5.59
0.5	0.97	4.95
0.6	1.05	5.61
0.7	1.14	5.35
0.8	1.18	4.71
0.9	1.27	5.32
1	**0.97**	**-**

Tabla B.2: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MS1 entrenado con la consulta Q_{s2} para ser evaluado con la consulta Q_{s2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	**-**	**24.27**
0.1	27.38	21.89
0.2	34.09	20.25
0.3	34.33	17.94
0.4	38.25	16.09
0.5	45.71	14.61
0.6	48.33	12.62
0.7	53.52	11.95
0.8	61.08	10.07
0.9	69.48	9.28
1	68.88	–

Tabla B.3: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM1 entrenado con la consulta Q_{m1} para ser evaluado con la consulta Q_{m1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	54.47
0.1	**44.25**	**48.48**
0.2	54.64	44.48
0.3	72.50	40.47
0.4	78.47	34.60
0.5	90.37	29.37
0.6	103.65	25.59
0.7	124.95	21.53
0.8	130.64	16.26
0.9	137.75	10.34
1	162.87	–

Tabla B.4: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM1 entrenado con la consulta Q_{m2} para ser evaluado con la consulta Q_{m2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	210.47
0.1	141.72	194.64
0.2	**181.50**	**172.03**
0.3	246.45	155.27
0.4	281.31	130.49
0.5	333.42	109.12
0.6	368.56	87.64
0.7	414.04	66.75
0.8	448.99	47.14
0.9	527.44	26.51
1	541.32	–

Tabla B.5: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba ML1 entrenado con la consulta Q_{l1} para ser evaluado con la consulta Q_{l1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	821.84
0.1	523.428	753.71
0.2	**664.08**	**665.04**
0.3	826.33	615.21
0.4	985.34	533.43
0.5	1150.62	438.11
0.6	1317.13	354.15
0.7	1454.17	277.26
0.8	1618.91	193.28
0.9	1766.67	56.11
1	1800.05	–

Tabla B.6: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba ML1 entrenado con la consulta Q_{l2} para ser evaluado con la consulta Q_{l2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	102.35
0.1	**57.70**	**71.27**
0.2	80.83	65.33
0.3	100.17	57.69
0.4	121.00	50.98
0.5	142.88	43.40
0.6	171.49	37.84
0.7	176.54	29.63
0.8	191.03	23.39
0.9	222.65	13.57
1	216.91	–

Tabla B.7: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MR1 entrenado con la consulta Q_{s1}, Q_{m1} y Q_{l1} para ser evaluado con la consulta Q_{s1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	97.85
0.1	**51.09**	**67.12**
0.2	70.61	61.05
0.3	95.76	55.26
0.4	113.47	49.09
0.5	128.51	39.86
0.6	149.37	34.98
0.7	166.87	26.98
0.8	175.22	20.78
0.9	199.29	12.17
1	206.10	–

Tabla B.8: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MR1 entrenado con la consulta Q_{s1}, Q_{m1} y Q_{l1} para ser evaluado con la consulta Q_{m1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	95.75
0.1	**51.31**	**65.59**
0.2	68.88	59.81
0.3	84.80	52.68
0.4	100.17	46.27
0.5	119.20	38.88
0.6	140.25	33.35
0.7	148.20	26.08
0.8	164.79	19.21
0.9	181.85	12.11
1	187.01	–

Tabla B.9: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MR1 entrenado con la consulta Q_{s1}, Q_{m1} y Q_{l1} para ser evaluado con la consulta Q_{l1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	3.79
0.1	0.68	4.14
0.2	0.37	3.69
0.3	0.39	3.73
0.4	0.39	3.73
0.5	0.39	3.72
0.6	0.39	3.72
0.7	0.38	3.67
0.8	0.38	3.69
0.9	0.38	3.67
1	**0.11**	**-**

Tabla B.10: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MS2 entrenado con la consulta Q_{s1} para ser evaluado con la consulta Q_{s2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	4.11
0.1	0.98	4.45
0.2	0.66	3.68
0.3	0.69	3.66
0.4	0.76	3.66
0.5	0.85	3.66
0.6	0.90	3.71
0.7	0.94	3.63
0.8	1.02	3.53
0.9	1.09	3.42
1	**0.83**	**-**

Tabla B.11: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MS2 entrenado con la consulta Q_{s2} para ser evaluado con la consulta Q_{s1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	**-**	**25.26**
0.1	29.12	25.26
0.2	34.72	20.42
0.3	35.01	19.00
0.4	39.29	16.22
0.5	45.68	15.44
0.6	50.97	14.17
0.7	59.68	11.92
0.8	67.56	11.13
0.9	71.94	9.72
1	70.63	–

Tabla B.12: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM2 entrenado con la consulta Q_{m1} para ser evaluado con la consulta Q_{m2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	51.20
0.1	**42.09**	**47.41**
0.2	51.79	41.28
0.3	65.71	38.03
0.4	79.04	33.04
0.5	86.86	27.60
0.6	99.34	23.99
0.7	114.36	19.40
0.8	123.61	15.35
0.9	134.30	9.76
1	141.27	–

Tabla B.13: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM2 entrenado con la consulta Q_{m2} para ser evaluado con la consulta Q_{m1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	209.84
0.1	147.24	201.03
0.2	**191.62**	**176.77**
0.3	252.80	159.50
0.4	294.08	134.62
0.5	352.51	111.36
0.6	388.54	90.73
0.7	435.75	69.19
0.8	465.61	49.34
0.9	541.13	27.04
1	539.16	–

Tabla B.14: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba ML2 entrenado con la consulta Q_{l1} para ser evaluado con la consulta Q_{l2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	793.99
0.1	491.68	715.42
0.2	**615.06**	**643.58**
0.3	779.11	584.19
0.4	941.39	509.84
0.5	1110.95	423.50
0.6	1252.69	341.60
0.7	1422.25	262.17
0.8	1593.40	178.44
0.9	1719.31	53.54
1	1759.90	–

Tabla B.15: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba ML2 entrenado con la consulta Q_{l2} para ser evaluado con la consulta Q_{l1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	97.52
0.1	**53.94**	**68.32**
0.2	75.08	63.16
0.3	101.20	57.14
0.4	117.68	50.42
0.5	131.47	41.61
0.6	156.92	35.04
0.7	174.34	27.58
0.8	177.10	20.89
0.9	203.36	12.89
1	203.20	–

Tabla B.16: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MR2 entrenado con la consulta Q_{s1}, Q_{m1} y Q_{l1} para ser evaluado con la consulta Q_{s2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	107.90
0.1	**58.28**	**72.00**
0.2	79.15	62.89
0.3	107.32	61.33
0.4	123.71	49.60
0.5	147.39	42.43
0.6	171.12	39.80
0.7	169.33	29.08
0.8	192.85	21.54
0.9	214.01	15.02
1	198.87	–

Tabla B.17: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MR2 entrenado con la consulta Q_{s1}, Q_{m1} y Q_{l1} para ser evaluado con la consulta Q_{m2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	94.01
0.1	**48.87**	**65.94**
0.2	65.55	59.95
0.3	84.82	52.81
0.4	99.88	46.52
0.5	118.49	39.55
0.6	139.18	33.86
0.7	153.12	25.68
0.8	167.74	19.29
0.9	183.46	12.29
1	192.10	–

Tabla B.18: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MR2 entrenado con la consulta Q_{s1}, Q_{m1} y Q_{l1} para ser evaluado con la consulta Q_{l2} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	4.00
0.1	1.04	4.56
0.2	0.55	3.72
0.3	0.54	3.71
0.4	0.58	3.73
0.5	0.60	3.73
0.6	0.65	3.70
0.7	0.66	3.67
0.8	0.70	3.62
0.9	0.76	3.67
1	**0.46**	**-**

Tabla B.19: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MS3 entrenado con la consulta Q_{s1} y Q_{s2} para ser evaluado con la consulta Q_{m1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	4.11
0.1	1.09	4.69
0.2	0.54	3.80
0.3	0.55	3.81
0.4	0.59	3.80
0.5	0.61	3.85
0.6	0.65	3.81
0.7	0.67	3.78
0.8	0.73	3.74
0.9	0.75	3.69
1	**0.51**	**-**

Tabla B.20: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MS3 entrenado con la consulta Q_{s1} y Q_{s2} para ser evaluado con la consulta Q_{l1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	36.17
0.1	**33.11**	**32.67**
0.2	40.82	29.51
0.3	47.11	26.25
0.4	55.68	23.21
0.5	62.80	20.22
0.6	70.11	17.37
0.7	79.25	14.73
0.8	89.77	11.85
0.9	96.43	8.72
1	102.78	–

Tabla B.21: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM3 entrenado con la consulta Q_{m1} y Q_{m2} para ser evaluado con la consulta Q_{s1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	35.44
0.1	**34.20**	**33.17**
0.2	39.43	28.57
0.3	47.80	26.50
0.4	57.86	22.94
0.5	63.84	20.28
0.6	72.01	17.28
0.7	81.44	14.90
0.8	88.06	11.76
0.9	94.77	8.32
1	99.29	–

Tabla B.22: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba MM3 entrenado con la consulta Q_{m1} y Q_{m2} para ser evaluado con la consulta Q_{l1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	551.34
0.1	332.90	506.51
0.2	**440.27**	**448.40**
0.3	538.15	397.84
0.4	649.72	345.19
0.5	786.13	294.36
0.6	874.92	243.55
0.7	963.16	179.79
0.8	1059.83	127.05
0.9	1163.59	39.94
1	1160.86	–

Tabla B.23: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba ML3 entrenado con la consulta Q_{l1} y Q_{l2} para ser evaluado con la consulta Q_{s1} para los distintos *cpu_per*.

CPU_PER	**TIEMPO CPU**	**TIEMPO GPU**
0	–	502.43
0.1	308.039	454.84
0.2	**400.60**	**407.54**
0.3	510.30	371.41
0.4	608.75	322.62
0.5	725.57	268.95
0.6	811.35	217.37
0.7	940.11	165.47
0.8	1005.67	112.10
0.9	1101.15	36.79
1	1131.3	–

Tabla B.24: Tiempos promedios de ejecución (en milisegundos) para el caso de prueba ML3 entrenado con la consulta Q_{l1} y Q_{l2} para ser evaluado con la consulta Q_{m1} para los distintos *cpu_per*.

B.2. Tablas de resultados obtenidos variando el período de refinamiento.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
1	1	18	0.3
2	1	19	0.4
3	0.2	20	0.4
4	0.2	21	0.6
5	0.9	22	0.6
6	0.9	23	0.1
7	0.1	24	0
8	0.1	25	0.3
9	0.8	26	0.6
10	0.8	27	0.4
11	0.5	28	0
12	0.5	29	0.5
13	0.7	30	0.1
14	0.7	31	0.7
15	0	32	0.2
16	0	33	0.8
17	0.3	34	0

Tabla B.25: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 1.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
1	1	23	0.2
2	1	24	0.2
3	0.9	25	0.4
4	0.9	26	0.4
5	0.8	27	0.1
6	0.8	28	0.1
7	0.5	29	0
8	0.5	30	0
9	0.3	31	0.3
10	0.3	32	0.3
11	0.7	33	0.5
12	0.7	34	0.5
13	0.2	35	0.6
14	0.2	36	0.6
15	0.4	37	1
16	0.4	38	1
17	0.6	39	0.7
17	0.6	40	0.7
18	0.6	41	0.8
19	0.1	42	0.8
20	0.1	43	0.9
21	0	44	0.9
22	0	45	0

Tabla B.26: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 2.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-3$	1	$25-27$	0.1
$4-6$	0.9	$28-30$	0
$7-9$	0.3	$31-33$	0.7
$10-12$	0.8	$34-36$	0
$13-15$	0.2	$37-39$	0.1
$16-18$	0.4	$40-42$	0.2
$19-21$	0.6	43	0
$22-24$	0.5		

Tabla B.27: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 3.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-4$	1	$25-28$	0.7
$5-8$	0.9	$29-32$	0
$9-12$	0.8	$33-36$	0.1
$13-16$	0.6	$37-40$	0.4
$17-20$	0.3	$41-44$	0.2
$21-24$	0.5	45	0

Tabla B.28: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 4.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-5$	1	$31-35$	0.8
$6-10$	0.9	$36-40$	0.1
$11-15$	0.7	$41-45$	0.2
$16-20$	0.6	$46-50$	0.5
$21-25$	0.4	51	0
$26-30$	0.3		

Tabla B.29: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 5.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-6$	1	$37-42$	0.4
$7-12$	0.9	$43-48$	0.3
$13-18$	0.8	$49-54$	0.2
$19-24$	0.7	$55-60$	0.1
$25-30$	0.6	61	0
$31-36$	0.5		

Tabla B.30: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 6.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-7$	1	$43-49$	0.5
$8-14$	0.6	$50-56$	0.9
$15-21$	0.7	$57-63$	0.8
$22-28$	0.2	$64-70$	0.1
$29-35$	0.4	71	0
$36-42$	0.3		

Tabla B.31: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 7.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-8$	1	$49-56$	0.3
$9-16$	0.8	$57-64$	0.5
$17-24$	0.4	$65-72$	0.7
$25-32$	0.2	$73-80$	0
$33-40$	0.9	$81-88$	0.1
$41-48$	0.6	89	0

Tabla B.32: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 8.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-9$	1	$55-63$	0.7
$10-18$	0.9	$64-72$	0.8
$19-27$	0.2	$73-81$	0.1
$28-36$	0	$82-90$	0.5
$37-45$	0.6	$91-99$	0.3
$46-54$	0.4	100	0

Tabla B.33: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 9.

EJECUCIÓN	*CPU_PER*	EJECUCIÓN	*CPU_PER*
$1-10$	1	$61-70$	0.6
$11-20$	0.9	$71-80$	0.8
$21-30$	0.4	$81-90$	0.5
$31-40$	0.2	$91-100$	0
$41-50$	0.7	$101-110$	0.1
$51-60$	0.3	111	0

Tabla B.34: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de refinamiento con valor 10.

B.3. Tablas de resultados obtenidos variando el período de reentrenamiento.

EJECUCIÓN	*CPU_PER*
$1-8$	0
9	1

Tabla B.35: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de reentrenamiento con valor 2. El largo del reentrenamiento es fijado con valor 1.

EJECUCIÓN	*CPU_PER*
$1-12$	0
13	1

Tabla B.36: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de reentrenamiento con valor 3. El largo del reentrenamiento es fijado con valor 1.

EJECUCIÓN	*CPU_PER*
$1-16$	0
17	1

Tabla B.37: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de reentrenamiento con valor 4. El largo del reentrenamiento es fijado con valor 1.

EJECUCIÓN	*CPU_PER*
$1-20$	0
21	1

Tabla B.38: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el período de reentrenamiento con valor 5. El largo del reentrenamiento es fijado con valor 1.

B.4. Tablas de resultados obtenidos variando el largo del reentrenamiento.

EJECUCIÓN	*CPU_PER*
$1-4$	0
5	1

Tabla B.39: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el largo del reentrenamiento con valor 2. El período de reentrenamiento es fijado con valor 2.

EJECUCIÓN	*CPU_PER*
1	0
2	1

Tabla B.40: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el largo del reentrenamiento con valor 3. El período de reentrenamiento es fijado con valor 2.

EJECUCIÓN	*CPU_PER*
1	0
2	1

Tabla B.41: Listado de ejecuciones hasta la estabilización con el *cpu_per* seleccionado para el largo del reentrenamiento con valor 4. El período de reentrenamiento es fijado con valor 2.

Referencias bibliográficas

[1] David B Kirk and W Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, 2016.

[2] Sebastian Breß, Gunter Saake, Jens Teubner, and Kai-Uwe Sattler. Efficient Query Processing in Co-Processor-accelerated Databases. PhD thesis, Otto von Guericke University Magdeburg, 2015.

[3] Lesly Acuña y Valentina Parula. Uso de aceleradores de hardware en sistemas de Bases de Datos Relacionales. Proyecto de Grado 2015, Universidad De La República, Facultad De Ingeniería. `https://www.fing.edu.uy/biblioteca/documentacion/tesis/2541/2541.pdf`. (Accessed on 12/13/2018).

[4] TPC Benchmark TM H Standard Specification Revision 2.18.0. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`. (Accessed on 06/16/2019).

[5] Ram Elmasri and Shamkant B. Navathe. Fundamentals of database systems. Pearson-Addison-Wesley, 2011.

[6] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. The design and implementation of modern column-oriented database systems. Foundations and Trends® in Databases, 5(3):197–280, 2013.

[7] Goetz Graefe and Leonard D Shapiro. Data compression and database performance. In Applied Computing, 1991. Proceedings of the 1991 Symposium on, pages 22–27. IEEE, 1991.

[8] NVIDIA CUDA C programming guide. URL `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. (Accessed on 06/06/2019).

[9] Klaus Nordhausen. An Introduction to Statistical Learning—with Applications in R by Gareth James, Daniela Witten, Trevor Hastie Robert Tibshirani. International Statistical Review, 82, 04 2014.

[10] Tom M. Mitchell. Machine Learning. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

[11] Column-oriented GPU-accelerated DBMS. URL `http://cogadb.dfki.de/`.

[12] High Performance GPU Database for Big Data SQL - BlazingDB. URL `https://blazingdb.com/index.html`. (Accessed on 12/13/2018).

[13] Alenka: GPU database engine. `https://github.com/antonmks/Alenka`. (Accessed on 12/13/2018).

[14] Emily Furst, Mark Oskin, and Bill Howe. Profiling a GPU database implementation: a holistic view of GPU resource utilization on TPC-H queries. In Proceedings of the 13th International Workshop on Data Management on New Hardware, page 3. ACM, 2017.

[15] PG-Strom - Limit Breaker of PostgreSQL powered by GPU. URL `http://strom.kaigai.gr.jp/manual.html`. (Accessed on 12/13/2018).

[16] Pedram Ghodsnia. An In-GPU-Memory Column-Oriented Database for Processing Analytical Workloads. pages 54–59, 01 2012.

[17] Hybrid Query Processing Engine for Coprocessing in Database Systems: Documentation. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/hype/current/doc/`. (Accessed on 09/21/2019).

[18] Sebastian Breß and Gunter Saake. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. Proceedings of the VLDB Endowment, 6(12):1398–1403, 2013.

[19] Wenbin Fang, Bingsheng He, and Qiong Luo. Database Compression on Graphics Processors. Proc. VLDB Endow., 3(1-2):670–680, September 2010. ISSN 2150-8097.

[20] CoGaDB Reference Manual. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/0.3/refman.pdf`. (Accessed on 12/13/2018).

[21] Sebastian Breß, Henning Funke, and Jens Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 1891–1906, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7.