Instituto Politécnico Nacional

Centro de Investigación en Computación

Paralelización de un subconjunto de consultas SQL con unión natural utilizando una GPU

TESIS
QUE PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN
PRESENTA:

Angel Omar Cervantes Ramírez

Directores de tesis:
Dr. Gilberto Lorenzo Martínez Luna
Dr. Adolfo Guzmán Arenas

México, D. F., Junio de 2012

Agradecimientos

Agradezco a mi familia, por estar siempre a mi lado, por darme todo su apoyo, cariño y comprensión, en realidad me han ayudado mucho más de lo que ustedes creen. Gracias a mi papá, a mi mamá, a mi hermana y a mi pequeña sobrina, ustedes son todo para mí.

Agradezco infinitamente al Instituto Politécnico Nacional (IPN) mi alma máter, por darme la oportunidad de formar parte de su comunidad estudiantil, por enseñarme a sentir el orgullo de ser politécnico y a poner la técnica al servicio de la patria. Gracias POLI.

Agradezco al Centro de Investigación en Computación (CIC). A su personal docente y administrativo que me atendió durante mi estancia en esta maestría.

Agradezco a mis compañeros, por sus consejos y por su apoyo dentro y fuera de los salones de clase. Les deseo suerte en sus próximos proyectos.

Agradezco al CONACyT y a las instituciones que apoyan la ciencia y la investigación, realmente sería muy complicado estudiar un postgrado sin la ayuda que ustedes otorgan. Muchas gracias.

Contenido

Índice de Figuras i
Índice de Tablas ii
Índice de Gráficas iii
Capítulo 1 - Introducción 1
1.1 - Presentación 1
1.2 - Planteamiento 2
1.3 - Objetivos 3
1.3.1 - Objetivo General 3
1.3.2 - Objetivos Particulares 3
1.4 - Justificación 4
1.5 – Limitaciones y Alcances 5
Capítulo 2 - Marco Teórico 6
2.1 - GPU 6
2.1.1 – GPU y CPU 7
2.1.2 - GPGPU 8
2.1.3 – Arquitectura CUDA 10
2.2 - SQL 13
2.2.1 – Operaciones de Definición y Control de Datos 14
2.2.2 – Operaciones de Manipulación de Datos 14
2.2.2.1 - SELECT 16
2.2.2.2 – NATURAL JOIN 16
2.2.2.3 – WHERE 18
2.2.2.4 – Operaciones de Agregación 19
2.2.2.5 – Otras Operaciones 19
2.3 – Bases de Datos Distribuidas y Paralelas 20
2.3.1 – Bases de Datos Distribuidas 20
2.3.2 – Arquitecturas de Bases de Datos Paralelas 21
2.3.3 – Tipos de Paralelismo en Bases de Datos 23
2.4 - SQLite 24
2.4.1 – Arquitectura de SQLite 24
2.4.2 – OpCode de SQLite 26

2.5 – Estado del Arte 26
2.5.1 – Accelerating SQL Database Operations on a GPU with CUDA 26
2.5.2 – Relational Joins on Graphics Processors 28
2.5.3 – Relational Query Co-Processing on Graphics Processors 29
Capítulo 3 - Análisis y Diseño 31
3.1 – Análisis y Diseño General de la Aplicación 31
3.2 – Análisis y Diseño del Código de Operación de SQLite 34
3.3 – Diseño de Algoritmos Paralelos para Operaciones SQL 38
3.3.1 – Diseño de Operación SELECT 38
3.3.2 – Diseño de Paralelización para Operación WHERE 39
3.3.3 – Diseño de Paralelización para Operación NATURAL JOIN sin Índices 42
3.3.4 – Diseño de Paralelización para Operación NATURAL JOIN con Índices 44
3.3.5 – Diseño de Paralelización para operaciones de Agregación (MAX, MIN, AVG, COUNT, SUM) 46
3.3.5 – Análisis para otras operaciones SQL 47
Capítulo 4 - Desarrollo 48
4.1 – Plan de Desarrollo 48
4.2 – Interfaz de Línea de Comandos y Conexión con SQLite 50
4.3 – Uso de los Niveles de Memoria de GPU 51
4.4 – Implementación de Algoritmos Paralelos Diseñados 54
4.4.1 – WHERE paralelo sobre una GPU 55
4.4.2 – NATURAL JOIN Paralelo sin Índices sobre una GPU 61
4.4.3 – NATURAL JOIN Paralelo con Índices sobre una GPU 64
4.4.4 – Operaciones de Agregación Paralelas sobre una GPU 64
4.5 – Sincronización de Resultados 66
4.6 – Sistema en ejecución 68
Capítulo 5 - Pruebas y Resultados 72
5.1 – Información General de Pruebas 72
5.2 – Consultas sobre Una Tabla 75
5.2 – Consultas sobre Dos Tablas Indexadas 78
5.4 – Consultas sobre Tres y Cuatro Tablas 80
5.5 – Consultas sobre Dos Tablas sin Índices 82
5.6 – Consultas con Operaciones de Agregación 84
5.7 - Comparación de funcionalidad con otros trabajos relacionados 86
Capítulo 6 - Conclusiones 88

6.1 - Conclusiones 88
6.2 - Contribuciones 90
6.3 – Trabajo a Futuro 92
Anexo A 93
Anexo B 97
B.1 – Plan de ejecución para una consulta SQL sobre una sola tabla. 97
B.2 – Plan de ejecución para una consulta SQL sobre dos tablas sin índices. 98
B.3 – Plan de ejecución para una consulta SQL sobre dos tablas con índices. 99
B.4 – Plan de ejecución para una consulta SQL sobre más de dos tablas con índices. 101
B.5 – Plan de ejecución para una consulta SQL con operaciones de agregación. 102
Referencias 104

Índice de Figuras

Figura 1. Representación de cores de CPU y GPU. 7
Figura 2. Modelo de organización de hilos en CUDA. 10
Figura 3. Configuración GRID 2 dimensiones con BLOCKs de 3 dimensiones. 11
Figura 4. Configuración GRID 1 dimensión con BLOCKs de 3 dimensiones 11
Figura 5. Configuración GRID 1 dimensión con BLOCKs de 2 dimensiones 11
Figura 6. Configuración GRID 1 dimensión con BLOCKs de 1 dimensión 11
Figura 7. Niveles de memorias en arquitectura CUDA. 12
Figura 8. Subdivisiones de SQL. 13
Figura 9. Ejemplo gráfico de operación NATURAL JOIN. 17
Figura 10. Arquitectura de BDD de memoria compartida. 21
Figura 11. Arquitectura de BDD con discos compartidos. 22
Figura 12. Arquitectura de BDD sin compartir recursos 22
Figura 13. Arquitectura interna de SQLite. 24
Figura 14. Única tabla utilizada en trabajo relacionado 27
Figura 15. Módulos de SQLite y GPU 32
Figura 16. Interacción con el compilador de SQLite 34
Figura 17. Ejemplo de plan de ejecución SQLite 35
Figura 18. Tupla asignada al hilo X 38
Figura 19. Ejemplo de proyección de dos campos. 39
Figura 20. Repartición una tupla a cada hilo. 40
Figura 21. Repartición de tuplas por Round Robin. 41
Figura 22. Conjunto de tuplas asignadas por los OpCode: Rewind y Next 41
Figura 23. Un solo hilo compara cada tupla en A con cada tupla en B. 42
Figura 24. Hilos dirigidos a cada segmento de A con cada segmento de B formando una malla. 43
Figura 25. Relación A fragmentada. Mientras que B se encuentra ordenada. 44
Figura 26. Ejemplo ilustrativo de una búsqueda binaria, para encontrar el valor 13. 45
Figura 27. Ilustración del método de dos tiempos para operaciones de agregación. 46
Figura 28. Secuencia de implementación de módulos. 49
Figura 29. Solicitud de una consulta SQL y obtención de su plan de ejecución 51
Figura 30. Formas de uso de los diferentes niveles de memoria de la GPU. 52
Figura 31. GRID de una dimensión, con BLOCKs con una dimensión de hilos. 55
Figura 32. Ubicación de los OpCode: Rewind y Next dentro de un plan de ejecución 56
Figura 33. Representación de las variables internas de CUDA. 57
Figura 34. Tuplas asignadas a hilos mediante su Identificador Global 58
Figura 35. Fin de asignación por parte del OpCode: Next 59
Figura 36. Secuencia de proceso para un operador OR. 60
Figura 37. Secuencia de proceso para un operador AND. 60
Figura 38. Algoritmo propuesto y arquitectura de hilos en CUDA. 61
Figura 39. OpCodes Rewind y Next para dos tablas con identificadores 0 y 1. 63
Figura 40. Semi-join para un hilo (X,Y). 63
Figura 41. Ilustración de estrategia de trabajo por niveles, para la operación de agregación MAX. 65
Figura 42. Representación de función atómica para sincronización de escritura. 67
Figura 43. Secuencia de traslado de datos de memoria compartida hacia memoria global. 67
Figura 44. Tiempo consumido por la GPU para resolver una consulta. 68
Figura 45. Tiempo consumido por el CPU para resolver una consulta. 69
Figura 46. Resultados desplegados por SQLite-GPU 69
Figura 47. Resultados desplegados por SQLite original. 70
Figura 48. Tabla de datos dividida en las 4 sub-tablas utilizadas. 74
Figura 49. OpCode dentro de un plan de ejecución 93

Índice de Tablas

Tabla 1. Operaciones de Lenguaje de Definición de Datos (DCL). 14
Tabla 2. Operaciones del Lenguaje de Control de Datos (DDL) 14
Tabla 3. Lenguaje de Manipulación de Datos (DML). 15
Tabla 4. Operadores de comparación dentro de la cláusula WHERE. 18
Tabla 5. Operadores de conexión lógica. 18
Tabla 6. Operaciones de agregación de SQL. 19
Tabla 7. OpCodes de Control de Ejecución. 35
Tabla 8. OpCodes de Asignación de Valor a Registro 36
Tabla 9. OpCodes de Saltos Condicionales. 36
Tabla 10. OpCodes de Inicializador de Apuntador a Tabla después de una Búsqueda. 37
Tabla 11. OpCodes que resuelven Operaciones de Agregación. 37
Tabla 12. Variables propias de CUDA 56
Tabla 13. Plan de ejecución para un NATURAL JOIN sin índices. 62
Tabla 14. Algoritmo de búsqueda binaria. 64
Tabla 15. Valores tomados por cada campo de las tablas. 74
Tabla 16. Consultas sobre una Tabla 75
Tabla 17. Resultados para Consultas sobre una Tabla 76
Tabla 18. Consultas sobre dos Tablas indexadas 78
Tabla 19. Resultados para Consultas sobre dos Tablas indexadas 79
Tabla 20. Consultas sobre tres y cuatro Tablas 80
Tabla 21. Resultados para Consultas sobre tres y cuatro Tablas 81
Tabla 22. Consultas sobre Tablas sin índices. Método sin ordenar 82
Tabla 23. Resultados para Consultas sobre Tablas sin índices. Método sin ordenar. 83
Tabla 24. Consultas sobre Tablas sin índices. Método con ordenamiento 84
Tabla 25. Consultas con operaciones de agregación 84
Tabla 26. Resultados para Consultas con operaciones de agregación 85
Tabla 27. Comparación de Características Funcionales 86
Tabla 28. Descripción de OpCodes reprogramados en CUDA. 94
Tabla 29. Plan de ejecución - Consulta sobre una tabla. 97
Tabla 30. Plan de ejecución - Consulta sobre 2 tablas sin índices. 98
Tabla 31. Plan de ejecución - Consulta sobre 2 tablas con índices. 99
Tabla 32. Plan de ejecución - Consulta sobre más de 2 tablas. 101
Tabla 33. Plan de ejecución - Consulta con operaciones de agregación. 102

Índice de Gráficas

Gráfica 1. Aceleración de Consultas sobre una Tabla 76
Gráfica 2. Aceleración para Consultas sobre dos Tablas Indexadas 78
Gráfica 3. Aceleración para Consultas sobre tres y cuatro Tablas 81
Gráfica 4. Aceleración sobre Tablas sin Índices. Método sin ordenar 82
Gráfica 5. Aceleración sobre Tablas si Índices. Método con ordenamiento 83
Gráfica 6. Aceleración para Consultas con operaciones de agregación 85

Resumen

El cómputo paralelo siempre ha tenido una gran variedad de aplicaciones, lograr paralelizar tareas que en un principio son de ejecución secuencial representa un nuevo reto. Si se analiza el problema a resolver y se desarrollan estrategias correctas, la recompensa resulta ser satisfactoria, se obtienen tiempos de respuesta más cortos para el mismo trabajo, solo que ahora dicho trabajo es atendido por un conjunto de unidades de procesamiento trabajando al mismo tiempo. Hoy en día, existe una novedosa plataforma para desarrollar cómputo paralelo, aunque es una tecnología relativamente de reciente desarrollo ha tenido muchas aplicaciones con buenos resultados en áreas tan diversas que llama la atención y motiva a buscar nuevas nichos de aplicación. La tecnología en cuestión, es el GPU-Computing, o computo paralelo sobre una tarjeta de video.

En este trabajo aplicamos el GPU-Computing sobre el área de las bases de datos. Basándonos en el plan de ejecución real del manejador de bases de datos SQLite, las operaciones de resolución de dicho manejador fueron reprogramadas en la plataforma de desarrollo CUDA para poder procesar las mismas instrucciones dentro de una GPU. De esta forma, obtuvimos un motor de búsqueda capaz de resolver un subconjunto de operaciones del lenguaje SQL mediante cómputo paralelo (concretamente consultas unitabla, multitabla y operaciones de agregación). Este motor de búsqueda en paralelo se apoya en el manejador SQLite, pero obtiene tiempos de respuesta más rápidos para las consultas que es capaz de resolver. Además de reprogramar el plan de ejecución, se atendieron los problemas propios que el paralelismo implica, tales como, administración eficiente de la memoria, distribución equitativa de la carga de trabajo entre todas las unidades de procesamiento, y una coordinación controlada para la recuperación de los resultados.

Abstract

Parallel computing has always had a wide range of applications, achieve parallelize tasks that originally are sequential execution represents a new challenge. If we analyze the problem to solve and develop the right strategies, the reward is satisfactory, you get shorter response times for the same job, only now that work is served by a set of processing units. Today, there is a new platform for developing parallel computing, although it is a relatively newly developed technology has had many successful applications in such diverse areas that attracts attention and motivates to seek new niche applications. The technology in question is the GPU-computing, or parallel computing on a video card.

In this paper we apply the GPU-Computing on the databases area. Based on the actual execution plan engine SQLite databases, its resolution operations were reprogrammed in the CUDA development platform to process the same instructions but in a GPU. Thus, we obtained a search engine capable of solving a subset of SQL operations through parallel computing (specifically unitable queries, multitable queries and aggregation operations). This parallel search engine is based on the SQLite, but you get faster response times for queries that is able to solve. In addition to reprogramming the execution plan, were attended to the problems inherent parallelism implies, such as efficient memory management, equitable distribution of workload among processing units and controlled coordination for recovery results.

Capítulo 1 - Introducción

1.1 - Presentación

Los manejadores de bases de datos en su intento por conseguir mejores tiempos de respuesta sobre las consultas SQL solicitadas, han puesto a trabajar conjuntos de procesadores al mismo tiempo sobre las tareas que normalmente haría uno solo, así lo indica [1], un trabajo que nos habla de los diversos factores que se deben considerar dentro de las bases de datos paralelas, tales como la arquitectura del sistema, la distribución de los datos, los accesos a disco y memorias, etc.

Hasta ahora, la principal forma de conseguir que varios procesadores se dediquen a procesar una consulta SQL, es mediante la replica total de los datos en varios equipos de trabajo o la fragmentación de los datos y el almacenamiento de cada fragmento entre los distintos equipos del sistema distribuido. Sin embargo, como lo señala [2], la optimización de las consultas en sistemas distribuidos se complica al tratar de coordinar todas las unidades de procesamiento. Con la finalidad de maximizar el paralelismo, trabajos como [3] desarrollaron métodos para dividir las tareas de una consulta y poder asignar cargas de trabajo a todos los procesadores disponibles, así mientras más tareas puedan ser tratadas individualmente más unidades de trabajo podría trabajar simultáneamente. Pero, tanto [2] como [3] mencionan que dos factores primordiales en los sistemas distribuidos son: como distribuir la información y como reunir los resultados.

Las formas más comunes de fragmentar tablas son horizontal (por tuplas) y verticalmente (por campos), o inclusive técnicas hibridas como lo hacen en [4], donde no solamente utilizan ambas formas sino que también detallan cuando es conveniente tomar una, otra o un hibrido para lograr mejores desempeños en la paralelización de consultas. En otro trabajo, en [5] proponen extensiones al lenguaje SQL, para lograr que la fragmentación de datos sea transparente para el usuario.

No obstante, más allá de hacer una correcta fragmentación de datos, existe un problema mayor en los sistemas distribuidos, nos referimos a la comunicación entre los distintos equipos que conforman el sistema y que trabajarán juntos. El intercambio de datos de información entre todas las unidades de trabajo debe ser lo menor posible. En [6], mediante técnicas de reconocimiento de patrones, obtuvieron métricas de similitud de consultas, para que varias consultas sean atendidas y resueltas usando un solo envío de datos para todas, en lugar de resolver y enviar datos de una en una. Y en [7], inclusive plantean técnicas para determinar que unidades de trabajo van participar en una consulta, y así evitar la comunicación con unidades que no intervienen.

Hoy en día, tenemos a nuestra disposición, una nueva tecnología que es el GPU-Computing, la cual mediante tarjetas de video nos permite tener cientos de procesadores dentro de una misma computadora, y esto es un factor muy conveniente pensando en su aplicación sobre las bases de datos, ya que se evitan los problemas de comunicación e intercambio de información entre unidades de trabajo, y también, los problemas para la replicación y/o fragmentación de los datos, pues todo está dentro de una misma computadora..

1.2 - Planteamiento

Los manejadores de bases de datos se ven beneficiados de la cada vez mayor velocidad de cómputo de los nuevos procesadores, pero no dejan de ejecutar sus funciones de búsqueda de forma secuencial (es decir, un solo procesador evalúa tupla a tupla), desperdiciando tiempo que se podría aprovechar realizando algunas funciones de forma paralela.

Por otro lado, los SMBD distribuidos realizan consultas con sub-tareas repartidas entre distintas máquinas al mismo tiempo pero su velocidad de respuesta, al final, se ve limitada por la capacidad de comunicación entre los distintos equipos.

El hardware continúa en constante desarrollo. Existe una tecnología de hardware que aunque no es nueva, aún no ha sido explotada lo suficiente en todas las áreas de la computación, por ejemplo en las bases de datos. La tecnología en cuestión, son las GPU (Graphic Processing Unit), es decir las tarjetas de video, que nacieron con el objetivo particular de mejorar el desempeño de una computadora en aplicaciones de video o aplicaciones que requieren gran procesamiento y trabajo sobre imágenes. Su desarrollo ha sido tan grande que incluso algunos modelos llegan a superar el desempeño de los procesadores sobre aplicaciones específicas.

Tomando en cuenta la necesidad de obtener resultados más rápidos para una consulta a una base de datos, nuestro problema a resolver, es buscar una forma de aprovechar el procesamiento en paralelo que proporciona una GPU y adaptarlo a las funciones de búsqueda que desarrolla un manejador de bases de datos para obtener resultados más rápidos. La tarea no es sencilla pues las GPU no son dispositivos creados para desarrollar cualquier tipo de aplicaciones, ya que su objetivo son las aplicaciones gráficas, sin embargo, se han obtenido excelentes resultados usando estas tarjetas de video para atacar problemas de distintas áreas que no involucran precisamente imágenes o video, lo cual nos hace pensar que se puede obtener buenos resultados en un área tan distinta como son las bases de datos.

Una de las operaciones más tardadas para un manejador de bases de datos es la operación JOIN, la cual involucra cálculos sobre dos relaciones. JOIN es una operación computacionalmente desgastante para un solo procesador por el número de operaciones que realiza, y también una tarea complicada de paralelizar a la que por sí sola se le han dedicado trabajos enteros, tales como [8] y [9], donde dividen un JOIN en semi-JOINs y realizan permutaciones de los datos entre los equipos de trabajo que se encuentran formando una malla de nodos interconectados.

Particularmente nos enfocaremos a paralelizar el operador NATURAL JOIN, es decir, que este operador se beneficie del paralelismo masivo de una GPU y obtenga tiempos de búsqueda más rápidos.

1.3 - Objetivos

1.3.1 - Objetivo General

- Acelerar el proceso del operador NATURAL JOIN de SQL, utilizando el plan de ejecución propio de un motor de bases de datos y el paralelismo masivo del GPU-Computing.

1.3.2 - Objetivos Particulares

- Resolver consultas uni-tabla.
- Resolver consultas multi-tabla.
- Implementar un uso eficiente de los diferentes niveles de memorias de una GPU.
- Paralelizar las operaciones de agregación de SQL.

1.4 - Justificación

Aumentar la velocidad de procesamiento de una tarea, o en otras palabras, disminuir el tiempo de respuesta de la misma, siempre será beneficioso, cualquiera que sea la tarea en cuestión.

En el caso de las bases de datos, el tiempo de respuesta para las consultas SQL ha disminuido gracias a que los procesadores de la máquinas que se usan, son cada vez mejores, más no por el hecho de que se ejecuten nuevos métodos, es decir, las técnicas utilizadas siguen siendo secuenciales y no paralelas como se tiende a ser ahora con nuevos algoritmos.

Una idea clara para acelerar una tarea, es repartir su carga de trabajo entre varios núcleos o cores de trabajo, para ello se pretende utilizar una GPU. Si bien es cierto que los SMBD distribuidos, ya reparten la tarea entre las máquinas que componen el sistema, también es cierto que su desempeño se ve limitado por la velocidad de comunicación entre sus miembros. Esta limitante de velocidad de comunicación es omitida en una GPU, ya que la base de datos nunca se divide físicamente, sino que la distribución de registros a trabajar se hace virtualmente para cada hilo de ejecución, además de que la información nunca sale de la misma máquina.

Utilizar una tarjeta de video también tiene la ventaja de que no se requiere un gran número de equipos para trabajar de forma paralela, sino que basta con un solo equipo con una tarjeta de video para poder funcionar, esto también se ve reflejado inmediatamente en el costo económico, pues quizás una GPU no tenga un precio barato en el mercado, sin embargo es más barato que comprar un clúster o varios equipos para trabajar de forma distribuida. Incluso el costo por mantenimiento también se reduce considerablemente utilizando GPU, y el uso de la misma es transparente al usuario, es decir, no hay necesidad de ser un experto en su uso y configuración.

Paralelizar la operación NATURAL JOIN de SQL es una buena forma de acelerar la ejecución de consultas, tomando en cuenta que si se busca reducir tiempo, a las operaciones más tardadas son también a las que se les puede obtener mayor provecho si se paralelizan de forma eficiente. Además al paralelizar la operación NATURAL JOIN aseguramos que nuestro trabajo sea de utilidad, pues es una de las operaciones más utilizadas en SQL tomando en cuenta que las consultas multi-tabla tienen gran uso en bases de datos relacionales.

Por el momento nos enfocaremos a mejorar el NATURAL JOIN, y se dejan otras operaciones como ORDER BY, GROUP BY u otras como opciones de trabajo a futuro, esto, ya que operaciones de ordenamiento o agrupamiento al momento de ser paralelizadas son altamente co-dependientes entre los hilos de trabajo. Es decir, por ejemplo, un hilo puede lograr un ordenamiento parcial para un subconjunto de datos asignado, pero para obtener un ordenamiento global de los datos, necesita saber los resultados de otros hilos (incluso trabajos como [10] y [11] están dedicados únicamente a la búsqueda de algoritmos paralelos de ordenamiento); caso similar pasa con el agrupamiento, pero no sucede con un NATURAL JOIN, ya que un hilo no requiere conocer los resultados de otro(s) hilo(s).

1.5 – Limitaciones y Alcances

Limitaciones:

- La limitante más importante para el desarrollo de este trabajo, es la memoria que ofrecen actualmente las GPU, ya que las tablas a trabajar deberán estar residentes dentro de ella para poder ser accedidos sin problemas, y el espacio total es cerrado, es decir, es delimitado por las características de nuestra memoria GPU, por lo tanto se debe tener precaución con el tamaño de los datos a la hora de ser enviados a la GPU.

Alcances

- Paralelizar solo un subconjunto de operaciones de SQL. La idea no es generar todo un manejador paralelo, quizás eso se podría ver como proyecto a largo plazo, pero por ahora vamos a centrarnos principalmente la operación NATURAL JOIN, la cual implícitamente debe ser acompañada por SELECT, FROM y WHERE, otros operadores pueden ser vistos como extensiones a futuro.

- Conjuntos de tamaño medio. Este alcance viene de la mano con nuestra primer limitante, no podemos trabajar con tablas demasiados grandes que no quepan en memoria, pero no por eso nos vamos a limitar a trabajar con tablas demasiados pequeñas, sino que trataremos de usar el mayor espacio disponible para tener tablas de un tamaño medio.

- Consultas hechas sobre un mismo conjunto de datos. Las consultas que soporta nuestra aplicación deberán ser aquellas que se hagan sobre los datos que se encuentren residentes, de lo contrario la consulta no podrá ser atendida.

- Tipos de datos numéricos o cadenas de longitud corta. Se trabajará principalmente sobre datos de tipo numérico, ya que las GPU se desenvuelven mejor con tipos de dato entero o de punto flotante. Adicionalmente, podemos trabajar con cadenas de no más de 20 caracteres, el reducido tamaño de las cadenas es para tener espacio y poder cargar una mayor cantidad de tuplas.

Capítulo 2 - Marco Teórico

En este capítulo se presentan los temas o herramientas que se utilizan en este trabajo y que tomarlos en cuenta es conveniente para una mejor comprensión del desarrollo de este proyecto.

En primer lugar se aborda lo que es una GPU, como se ve frente a un CPU y revisamos su arquitectura para tener a grandes rasgos un mejor entendimiento de este tópico tan basto.

Después nos ponemos a revisar el lenguaje SQL y nos adentramos en revisar las funciones que realizan los operadores que serán paralelizados en esta tesis.

Continuamos repasando un poco a cerca de las bases de datos distribuidas, y así también, sobre las bases de datos paralelas.

Enseguida, hablamos del motor de bases de datos transaccional SQLite, presentamos su arquitectura interna, y se explica brevemente la razón de escoger este motor, la cual se centra primordialmente en el plan de ejecución que genera para cada consulta.

Finalmente, cerramos este capítulo discutiendo el estado del arte, analizamos algunos trabajos relacionados directamente con este que se presenta, es decir, esencialmente trabajos que involucran bases de datos y cómputo en GPU, los cuales son en conjunto un campo relativamente nuevo.

2.1 - GPU

La GPU es por sus siglas en inglés Graphics Processing Unit o bien una Unidad de Procesamiento Gráfico en español, hay quienes optan por referirlas simplemente como tarjetas de video o aceleradoras de gráficos. En fin, una GPU es una tarjeta de expansión para computadora, encargada de procesar datos provenientes desde la CPU (Central Processing Unit o Unidad de Procesamiento Central) y transformarlos en información comprensible y representable en un dispositivo de salida como un monitor.

2.1.1 – GPU y CPU

La GPU es un chip diseñado para llevar a cabo todos los cálculos necesarios en la generación de gráficos. La GPU es un hardware cuyo desarrollo fue impulsado principalmente por la industria del entretenimiento y el diseño digital, es decir, aplicaciones tales como los videojuegos, software profesional de dibujo o diseño asistido por computadora, reproductores de video, animaciones digitales, etcétera, las cuales requieren un gran número de operaciones para ejecutarse en pantalla y dicho número de cálculos llega a ser demasiado trabajo para un solo procesador creado para ejecutar operaciones aritméticas y lógicas, que además debe atender a otras aplicaciones al mismo tiempo. Por lo tanto, empresas como nVIDIA o ATI, desarrollaron hardware específico que pudiera dedicarse a atender aplicaciones gráficas y ayudar a desahogar la carga de trabajo del CPU.

CPU y GPU son dos chips independientes pero que trabajan en forma conjunta, es decir, su colaboración les permite atender procesos específicos y aligerar sus colas de tareas mutuamente, por ejemplo, la CPU cede a la GPU las operaciones que tengan que ver con cálculos en vértices, pixeles y salidas a pantalla, y mientras tanto CPU se puede encargar de atender otras tareas.

Una GPU podemos verla como un conjunto de procesadores en un mismo chip que trabajan en forma paralela. Como ya se dijo anteriormente, CPU y GPU trabajan en colaboración y ninguna pretende sustituir a la otra, ya que sus propósitos son diferentes. Hoy por hoy el rendimiento de una tarjeta de video ha mejorado demasiado, al grado de que tener una buena aceleradora de gráficos mejora notablemente el desempeño de cualquier computadora en aplicaciones gráficas, digamos que una tarjeta de video ya no es un simple accesorio de la computadora.

La alta especialización de una GPU le permite tener una gran potencia de procesamiento y su desarrollo y mejoramiento ha crecido a pasos acelerados, en otras palabras, al estar pensadas para desarrollar una sola tarea, es posible dedicar más silicio en su diseño para llevar a cabo sus procesos de forma más eficiente.

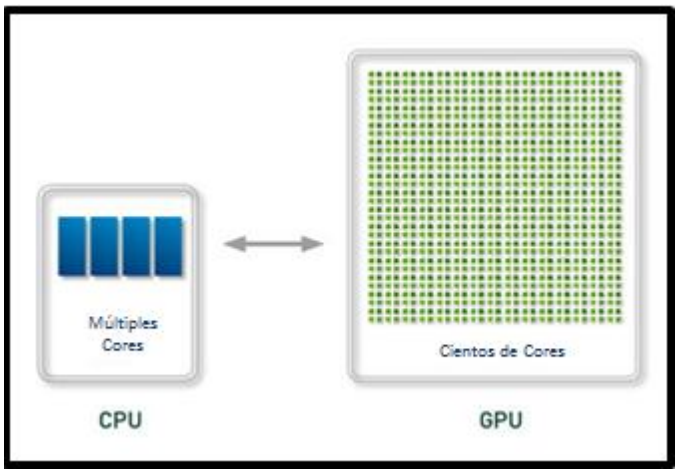


Figura 1. Representación de cores de CPU y GPU.

Ahora, si bien es cierto que el poder de procesamiento de un núcleo de GPU no se compara con el procesamiento de un core de procesador moderno, también es cierto que por la gran cantidad de procesadores que conforman una GPU y la cantidad de hilos (o threads) que pueden ser ejecutados en forma simultanea o paralela, existen tareas que pueden desarrollarse dentro de la tarjeta gráfica y obtener un mejor desempeño en cuanto a tiempo de respuesta.

En la Figura 1, tenemos una representación gráfica de un CPU y una GPU; quizás un core o núcleo de GPU es más pequeño (en capacidad de procesamiento) frente a su equivalente de CPU, pero la ventaja es que ahora tenemos cientos y en algunos casos hasta miles de cores que nos permiten realizar un paralelismo masivo.

2.1.2 - GPGPU

El hecho de que una tarjeta gráfica pueda lanzar cientos o miles de hilos al mismo tiempo, es lo que le permite trabajar sobre muchos pixeles a la vez en imágenes y video, pero quizás esta característica no sea únicamente beneficiosa para tratar aplicaciones gráficas. A la idea de aprovechar dicha característica y usar su poder de procesamiento en paralelo en otras áreas no necesariamente gráficas se le llama GPGPU (en inglés General-Purpose Computing on GPU, o bien Computo de Propósito General sobre una GPU, en español) [30].

De acuerdo con nVIDIA [30], GPU computing, o GPGPU, es el uso de la GPU (unidad de procesamiento gráfico) para realizar operaciones de cálculo científico o técnico de propósito general.

El modelo empleado para esta tecnología se basa en el uso combinado de una CPU y una GPU en un sistema de coprocesamiento heterogéneo. La parte secuencial de la aplicación se ejecuta en la CPU y las partes de mayor carga computacional (siempre que estas sean paralelizables) se aceleran en la GPU.

Tradicionalmente las GPUs se dedicaban exclusivamente a acelerar el procesamiento de imágenes que se despliegan en pantalla, desde su representación en forma de datos hasta su correcta visualización en forma de pixeles en el monitor. Esta tarea la efectúan eficientemente, pues en esencia ese es el propósito de su creación. Sin embargo, el modelo y evolución de esta importante pieza de hardware la ha llevado a ser una máquina con muy buen rendimiento en tareas de ejecución paralela, frente al trabajo serial de un procesador. Esto se debe a que actualmente una tarjeta gráfica puede llegar a tener cientos de procesadores (en algunos modelos inclusive más) y a esto todavía podemos agregar que es posible utilizar múltiples tarjetas gráficas en un mismo sistema, con lo cual el paralelismo se incrementa a un nivel mayor y los tiempos de respuesta también se reducen.

El cómputo de propósito general (GPGPU) busca sacar provecho de las cualidades de las GPUs en tareas no netamente gráficas, no obstante, debemos siempre recordar cual es propósito de estas tarjetas, cual es su finalidad (es decir, estas tarjetas nacieron específicamente con el objetivo de procesar imágenes, dedicarse únicamente a ello y no entrometerse con tareas que si son para el

CPU) y a partir de ello, considerar que no todas las tareas pueden ser ejecutadas en una GPU y obtener mejores resultados que en una CPU.

La CPU sigue siendo mejor para muchas tareas tan habituales como el acceso aleatorio a la memoria o ejecutar pasos en orden secuencial (y muchas más) dado que para eso fueron diseñados y construidos a diferencia de las GPU.

Entonces es importante analizar que es lo se va a trabajar, identificar primero que tan paralelizable es cada proceso o algoritmo que se vaya a emplear, estudiar si es viable llevar su implementación a la tarjeta de video y rediseñar el algoritmo para una ejecución paralelamente masiva con fines de optimizar todos los recursos de la GPU.

Recientemente se han estado haciendo trabajos con GPGPU obteniendo desempeños muy satisfactorios en áreas tan diversas tales como, modelación molecular, química computacional, bioinformática, diagnósticos clínicos por imagen, cálculo financiero, servicios de información geográfica, y un largo etcétera, muchos de ellos campos en los que quizás nunca se llego a imaginar o pensar en utilizar una tarjeta aceleradora de gráficos podría ayudarles a acelerar procesos y mejorar tiempos de respuesta.

2.1.3 – Arquitectura CUDA

NVIDIA, una marca muy reconocida en la fabricación de tarjetas aceleradoras de gráficos, desarrolló CUDA, lo cual es una plataforma de cómputo paralelo y un modelo de programación, que puede incrementar el rendimiento computacional utilizando la potencia de cálculo de una unidad de procesamiento gráfico [31].

CUDA son las siglas de Compute Unified Device Architecture (o en español, Arquitectura de Dispositivos de Cómputo Unificado), es una tecnología de desarrollo computacional de reciente creación, cuyo primer SDK o kit de desarrollo fue publicado entre finales de 2006 y principios de 2007, que ofrece además de una extensión al lenguaje C/C++, un compilador y una arquitectura de computo paralelo para generar funciones que se ejecuten dentro de la GPU [31].

Para poder entender el desarrollo del trabajo que se presenta en este documento, es importante conocer la arquitectura de CUDA, si bien no hay necesidad de tener un dominio a fondo, si es conveniente tener presentes los conceptos básicos de esta nueva tecnología para lograr un mejor entendimiento en general.

Parte de la filosofía de CUDA es trabajar en conjunto con la CPU, como si se tratase de un sistema heterogéneo [14]. Por ello es importante saber como se efectúa la comunicación entre ambos dispositivos (CPU y GPU) y la manera en como se les conoce dentro de la arquitectura. La Figura 2 ilustra el modelo utilizado por la arquitectura CUDA.

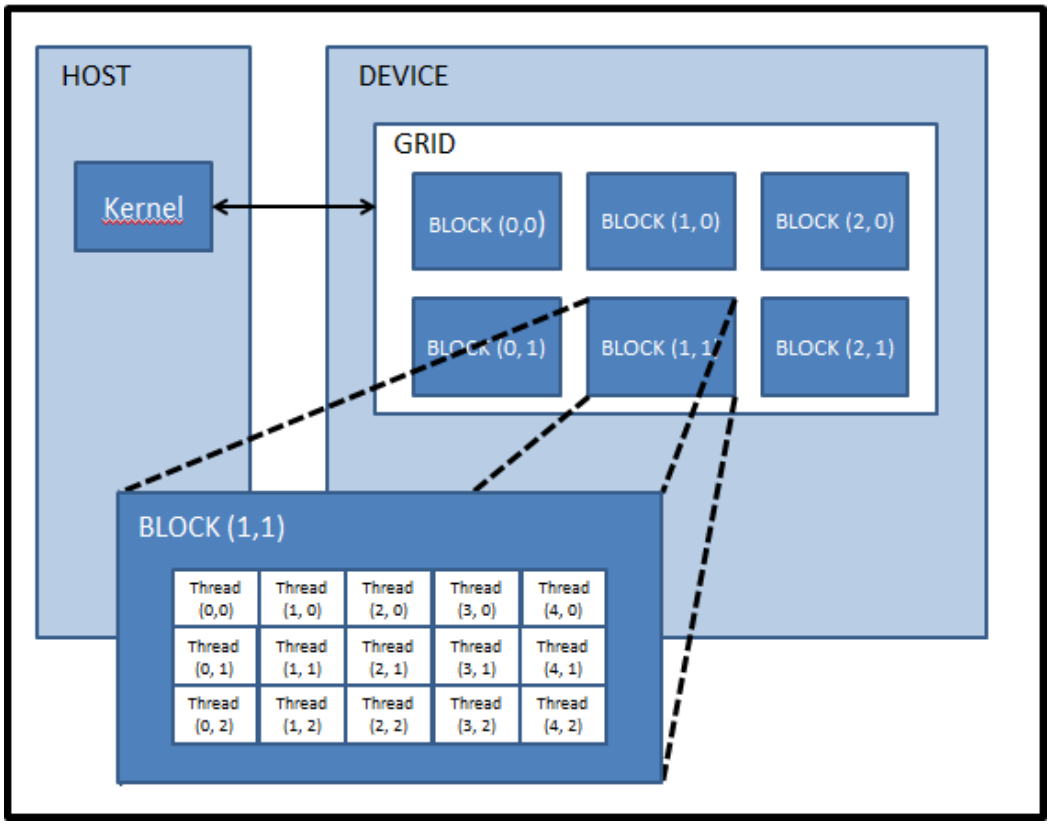


Figura 2. Modelo de organización de hilos en CUDA.

Como se puede apreciar en la Figura 2, el diagrama se compone de dos bloques principales [14]. El bloque del lado izquierdo representa CPU el cual a partir de este momento nombraremos como HOST, y del lado derecho se muestra la GPU desde una perspectiva más interna de su composición y a quien nombraremos ahora simplemente como DEVICE.

Un DEVICE está compuesto por al menos una GRID las cuales son ejecutados desde una función kernel del lado del HOST. La traducción de una GRID en español sería "rejilla" y es que tanto de forma visual como de forma lógica se le puede comprender como una reja o una malla de BLOCKs de una o hasta dos dimensiones. A su vez, cada BLOCK está compuesto de THREADs o hilos, los cuales pueden estar ordenados de una, dos o hasta tres dimensiones.

En las siguientes figuras se pueden ver distintas configuraciones de los THREADs y BLOCKs que conforman una GRID.

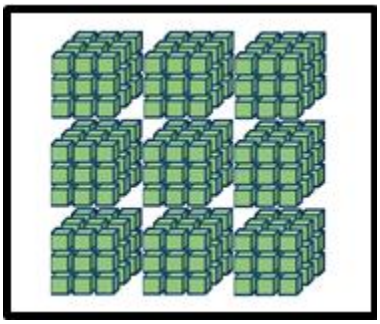


Figura 3. Configuración GRID 2 dimensiones con BLOCKs de 3 dimensiones.

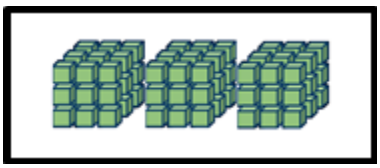


Figura 4. Configuración GRID 1 dimensión con BLOCKs de 3 dimensiones.

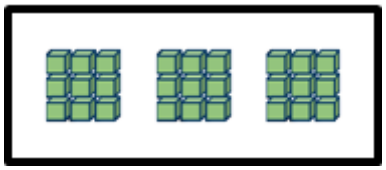


Figura 5. Configuración GRID 1 dimensión con BLOCKs de 2 dimensiones.

Figura 6. Configuración GRID 1 dimensión con BLOCKs de 1 dimensión.

En las Figuras 3, 4, 5 y 6 cada cubo pequeño representa un THREAD, estos se acomodan en un BLOCK de 1, 2 o 3 dimensiones, que a su vez componen la GRID de 1 o 2 dimensiones. La configuración que se debe utilizar es a elección del desarrollador y la que mejor se acople a cada problema [13].

La forma en como podemos indicar o trabajar con un hilo en particular es mediante constantes globales de cada THREAD. Estas constantes son: threadIdx (con sus respectivas dimensiones x, y, z) y blockIdx (con sus respectivas dimensiones x, y, o en algunos modelos de GPU hasta z). Y el tamaño de cada dimensión de GRID y BLOCK será dado por gridDim y blockDim respectivamente (y su correspondiente dimensión x, y, z) [12].

Ahora exploremos otro aspecto muy importante a considerar dentro de esta arquitectura CUDA, son los distintos tipos y niveles de memoria. Mediante el siguiente diagrama (Figura 7) podemos visualizarlo de mejor manera [14].

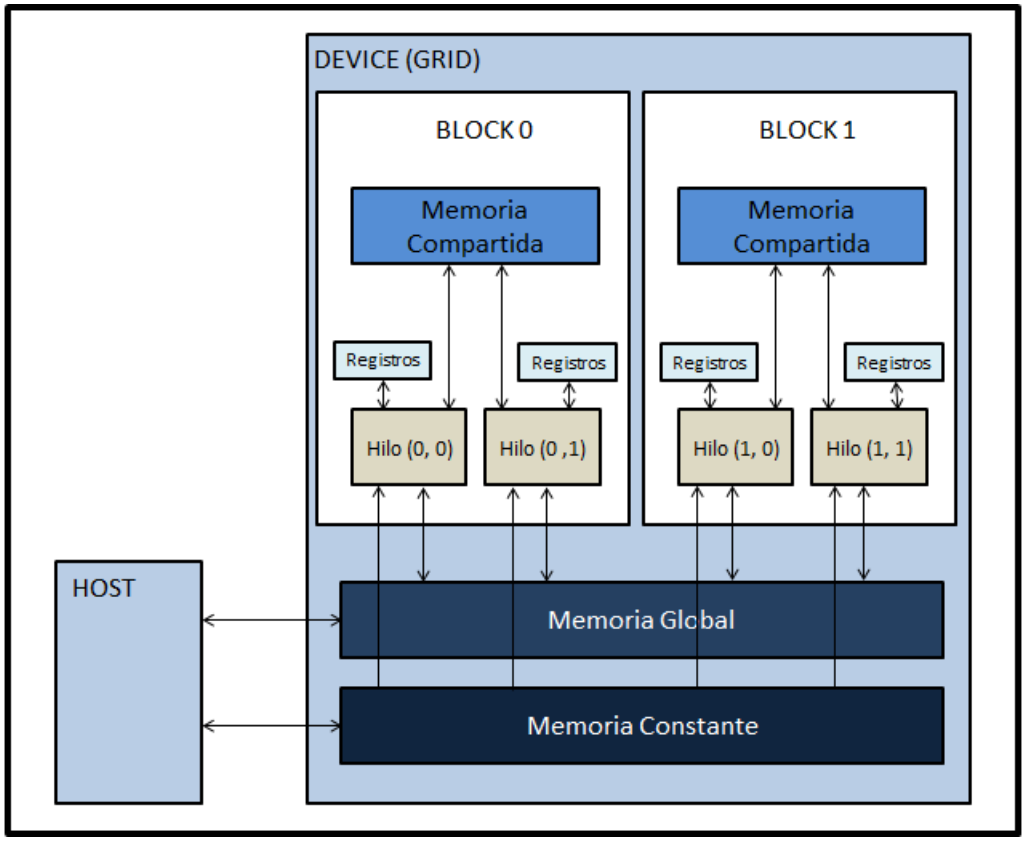


Figura 7. Niveles de memorias en arquitectura CUDA.

En primer lugar tenemos que la Global Memory (o memoria global) es el medio de comunicación entre HOST y DEVICE (o bien CPU y GPU), cualquier dato o información que se desee procesar dentro de la GPU debe ser enviado a memoria global (o también a memoria constante que se explicará más adelante). La memoria global puede ser accedida por todos y cada uno de los Threads sin importar su posición o bloque al que correspondan, y también puede ser leída y escrita desde el HOST.

Enseguida tenemos la Constant Memory (o memoria constante), esta al igual que la memoria global puede ser accedida mediante lectura por todos los hilos, la diferencia esta en que nadie puede escribir ni cambiar sus valores. Los datos de la memoria constante se determinan desde el HOST y sus datos permanecen siempre fijos.

Después tenemos la Shared Memory (o memoria compartida), que como su nombre lo indica es una memoria compartida pero solamente por los hilos de un mismo bloque, y que puede ser leída y escrita únicamente por los hilos miembros a dicho bloque.

Finalmente, están los Registers (o registros) que no son más que variables propias de cada hilo. Es decir registros que pertenecen a un solo hilo y que pueden ser leídos y escritos solo por el THREAD correspondiente.

2.2 - SQL

SQL (Structured Query Language o Lenguaje de Consulta Estructurado) es un lenguaje estándar de acceso a bases de datos. Por lo tanto, al ser estándar o normalizado, nos permite trabajar en cualquier tipo de lenguaje de programación y comunicarnos con una base de datos relacional en cualquier SMBD (Sistema Administrador de Bases de Datos o Database Management System).

Existen diversos SMBD, algunos libres (MySQL, SQLite) y otros más de licencia (Oracle, SQL Server, DB2). Un SMBD es un software especializado, que controla la organización, almacenamiento, recuperación, seguridad e integridad de toda la información en una base de datos. Algunos SMBD, proporcionan más herramientas o funciones que otros, pero todos deben ser capaces de ejecutar al menos las indicadas por la versión actual de SQL que es la SQL: 2008, cuyo estándar es validado por ANSI (American National Standars Institute) e ISO (International Standar Organization).

Las operaciones soportadas por SQL se agrupan en 3 categorías [19]:

DDL – Data Definition Language o Lenguaje de Definición de Datos.
DCL – Data Control Language o Lenguaje de Control de Datos.
DML – Data Manipulation Language o Lenguaje de Manipulación de Datos.

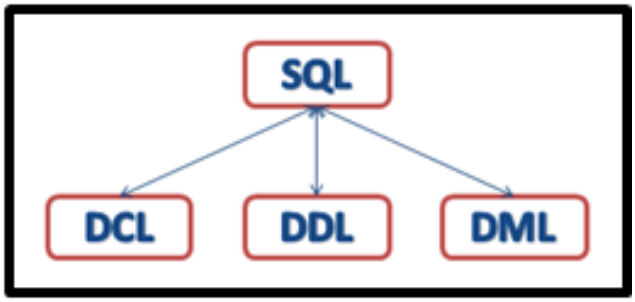


Figura 8. Subdivisiones de SQL.

2.2.1 – Operaciones de Definición y Control de Datos

Las operaciones para definir y controlar datos son básicas dentro de SQL. Este tipo de operaciones no representan demasiado trabajo ni desgaste para el procesador, ya que la cantidad de registros que modifican y operaciones que conllevan no suelen ser demasiados.

Las sentencias que engloba DDL (Lenguaje de Definición de Datos) son aquellas que nos permiten la creación de objetos en la base de datos tales como tablas, vistas, store procedures [19], etc. Dentro de ellas tenemos las siguientes (Tabla 1):

Tabla 1. Operaciones de Lenguaje de Definición de Datos (DCL).

Operaciones del Lenguaje de Definición de Datos	
CREATE	Crea una base de datos o un objeto.
ALTER	Modifica la estructura de una base de datos o un objeto.
DROP	Elimina una base de datos o un objeto.

Después de utilizar las sentencias DDL para crear objetos, la siguiente tarea es asegurarlos con las sentencias DCL (Lenguaje de Control de Datos), las cuales sirven para asignar permisos a usuarios en específico o roles de bases de datos, y dentro de esta tenemos las siguientes (Tabla 2):

Tabla 2. Operaciones del Lenguaje de Control de Datos (DDL).

Operaciones del Lenguaje de Control de Datos	
GRANT	Asigna permisos a usuarios o roles para poder acceder a una base de datos o a alguno de sus objetos.
DENY	Evita que un usuario o rol ingrese a una base de datos o a alguno de sus objetos.
REVOKE	Remueve un permiso que se le ha signado a un usuario o a un rol.

Las tareas sobre las cuales se pueden conceder o denegar permisos son básicamente todas las que pertenecen al DCL.

2.2.2 – Operaciones de Manipulación de Datos

Finalmente, tenemos el tercer grupo de sentencias, las del DML (Lenguaje de Manipulación de Datos). Este grupo de comandos merece un apartado especial, porque es sobre estas sentencias sobre las cuales se enfoca el trabajo del presente documento, por lo tanto es necesario saber cuales son dichas sentencias y sobre cuales vamos a trabajar en particular.

Las operaciones SQL engloban todas aquellas que pertenecen a DDL, DCL y DML, aunque es muy común confundir y pensar solamente en sentencias DML cuando se habla de operaciones SQL en general. Las sentencias DML nos permiten trabajar con los objetos creados en una base de datos y dentro ellas tenemos las siguientes (Tabla 3):

Tabla 3. Lenguaje de Manipulación de Datos (DML).

Operaciones del Lenguaje de Manipulación de Datos	
INSERT	Agrega uno o más registros a una y solo una tabla dentro de una base de datos relacional.
UPDATE	Es utilizada para modificar los valores de un conjunto de registros existentes dentro de una tabla.
DELETE	Borra uno o más registros existentes en una tabla.
SELECT	Despliega información contenida dentro de una o varias tablas.

Las 3 primeras sentencias, implican una escritura directa a disco ya que las operaciones de agregar, modificar o borrar datos, deben dejar sus resultados permanentes y disponibles para cualquier consulta posterior. Y en el presente trabajo, lo que se busca paralelizar son precisamente las consultas SELECT que se hacen con los datos contenidos en una base de datos, es decir, no se pretende acelerar la forma en como se guardan o escriben los datos en el disco duro, sino que se intenta agilizar los métodos de búsqueda que se necesitan para devolver al usuario los datos resultantes de un SELECT, apoyándonos para ello en la utilización del cómputo paralelo masivo que nos brinda una GPU.

Es importante entonces revisar y analizar cuales son las operaciones que se pueden hacer dentro de un SELECT así como también medir su complejidad y el número de operaciones que le exigen al CPU.

A partir de este momento nos referiremos indistintamente a una consulta SELECT simplemente como consulta.

La consulta más básica es la siguiente:

```
SELECT * FROM relacion1;
```

Este consulta solamente nos esta pidiendo que despleguemos todos los datos de la relación relacion1 sin ninguna condición de filtro. Esta consulta la llamamos básica porque contiene únicamente las dos cláusulas esenciales para una consulta: SELECT y FROM, y únicamente implica una sola relación. Esta consulta puede sufrir modificaciones hasta alcanzar una consulta mucho más compleja para obtener resultados más específicos de acuerdo a las necesidades del usurario.

Ahora analizaremos cada una de las cláusulas y factores que pueden aumentar la complejidad de una consulta.

2.2.2.1 - SELECT

La cláusula SELECT nos sirve para hacer proyecciones de una consulta. Se usa para listar los atributos deseados del resultado de una consulta [19].

El operador * (asterisco) que va después del SELECT sirve para hacer la petición de que el usuario requiere todos los campos implicados en una consulta, es decir, todos los campos de la relación o relaciones involucradas estarán en el resultado final de la consulta.

Por otro lado, si solo se requiere de uno o más campos en particular como resultado entonces dichos campos deben ser listados inmediatamente después del SELECT. El operador * es sustituido por el nombre de los campos requeridos, en caso no haber necesidad de obtener todos los campos.

SELECT campo1, campo2, campo_n FROM relacion1;

La complejidad computacional de un SELECT no aumenta ni disminuye ya que el número de filas en el resultado sigue siendo el mismo, la diferencia esta en que la cantidad de atributos desplegados será menor o mayor, pero no representa trabajo extra.

El verdadero trabajo de una consulta radica en encontrar todos aquellos registros que cumplan los criterios de búsqueda, y una vez encontrados esos registros, el hecho de mostrar todos los campos participes o solamente algunos no involucra mayor ni menor procesamiento en los tiempo de búsqueda.

2.2.2.2 – NATURAL JOIN

En la consulta básica, se hace mención a dos palabras reservadas, SELECT (que se describe anteriormente) y FROM. Ésta última, será quien nos indique el número de relaciones que se involucraran en la consulta.

SELECT * FROM relacion1, relacion2, relacion_x;

Para una búsqueda con una sola relación la complejidad computacional es de n, donde n es el número de tuplas que contiene dicha relación. Ahora, para una consulta que involucra más de una relación, la complejidad aumenta considerablemente.

El presente trabajo, aborda la cláusula NATURAL JOIN. Ésta cláusula es un operador que funciona sobre 2 relaciones (una al lado izquierdo y la otra al lado derecho del operador).

SELECT * FROM relación_izq NATURAL JOIN relación_der;

Una consulta con dos relaciones tendrá una complejidad de n * m, donde n es el número de tuplas en relación_izq y m el número de tuplas en relación_der. Si la consulta consta de 3 relaciones, la complejidad será de n * m * p, donde p será el número de tuplas para la tercera relación, y así sucesivamente.

La razón por la cual aumenta de esta forma la complejidad al agregar una relación más a la consulta, se debe a que para cada tupla de la primer relación se debe hacer un emparejamiento (o match) con cada tupla de la segunda relación. Es decir, se harán m matches (recordemos que m es el número de tuplas en relación_der) para cada tupla de relación_izq, y dado que relación_izq entonces el número total de matches estará dado por m * n.

Por otro lado, la diferencia entre utilizar el operador NATURAL JOIN entre dos relaciones o utilizar el operador ‘,’ (coma), es la siguiente, el operador ‘,’ lo único que hace es emparejar tupla con tupla de ambas relaciones sin hacer ningún filtro. Por ejemplo, si ambas relaciones tuvieran un millón de tuplas, entonces el como resultado se obtendrían un billón de tuplas (n * m), lo cual es un resultado poco práctico. A esta operación se le llama producto cruz.

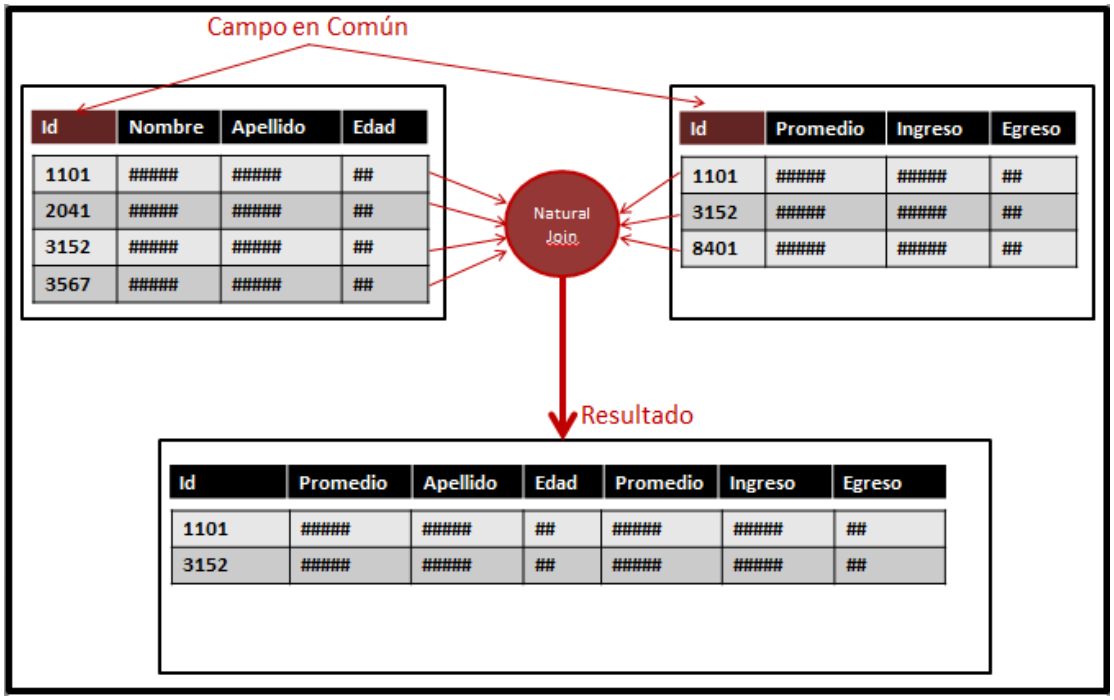


Figura 9. Ejemplo gráfico de operación NATURAL JOIN.

El operador NATURAL JOIN (Figura 9) hace un match entre las tuplas de ambas relaciones pero con un filtro y regresará como resultado un número de tuplas mucho menor al de un producto cruz. Dicho filtro consiste en comparar un campo en común dentro de ambas relaciones y desechar todos los matches que no tengan el mismo valor en el campo correspondiente. Es decir, ambas relaciones deberán tener un campo con el mismo nombre (campo en común), se evaluarán los valores de dicho campo en las dos relaciones y se filtrarán solo aquellos matches que tengan idéntico valor sobre el campo en cuestión.

El número de atributos que se regresan después de un NATURAL JOIN es de a+b-1, donde ‘a’ es el número de atributo de la primera relación y ‘b’ los atributos de la segunda, el ‘-1’ se debe a que el campo en común se repite y el operador NATURAL JOIN lo omite una vez.

2.2.2.3 – WHERE

La cláusula WHERE sirve para filtrar los resultados de una tupla. Por ejemplo, dada una consulta, se requiere que solo regrese tuplas cuyo valor en campo1 sea mayor x, o cuyo valor en campo2 sea diferente a y, etc.

La funcionalidad de WHERE va acompañada con operadores de comparación y lógicos (Tablas 4 y 5).

Tabla 4. Operadores de comparación dentro de la cláusula WHERE.

Operador de Comparación	Uso
<	Menor que
>	Mayor que
< > (o !=)	Diferente de
<=	Menor o igual que
>=	Mayor o igual que
= (o ==)	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores

Tabla 5. Operadores de conexión lógica.

Operador Lógico	Uso
AND	Es el 'y' lógico. Evalúa dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Es el 'o' lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

Las operaciones de la cláusula WHERE se ejecutan después del matching de relaciones (si es que hay más de una) y reducen o mantienen el número de tuplas regresadas como resultado dependiendo cuantas de ellas pasen los filtros o condiciones de búsqueda del WHERE.

Ejemplos de consultas incluyendo cláusula WHERE.

SELECT campo1, campo2 FROM relacion1 WHERE campo1 > A and campo2!= Y;

SELECT * FROM relacion1 NATURAL JOIN relacion2 WHERE campo_n BETWEEN (X, Y);

2.2.2.4 – Operaciones de Agregación

Las operaciones de agregación son básicamente 5 las que se incluyen en SQL: 2008 (Tabla 6), aunque existen algunos SMBD que suelen ofrecer otras más.

Las funciones de agregación (o de agregado) se usan dentro de la cláusula SELECT para devolver un único valor que se aplica a un grupo de registros.

Tabla 6. Operaciones de agregación de SQL.

Operación Agregación	Descripción
MAX	Devuelve el valor más alto en un campo especificado.
MIN	Devuelve el valor más bajo en un campo especificado.
AVG	Calcula el promedio de los valores de un campo.
COUNT	Devuelve el número de tuplas del resultado de una selección.
SUM	Devuelve la suma de todos los valores de un campo.

Ejemplos de consultas con operaciones de agregación:

SELECT AVG (edad) FROM alumnos;

SELECT COUNT (*) FROM relacion1 WHERE campo1 > A and campo2!= Y;

2.2.2.5 – Otras Operaciones

Existen otras operaciones dentro del SQL que no abordarán más en este trabajo, ya que por delimitación del problema fueran dejadas para un trabajo a futuro del mismo.

GROUP BY – Agrupa un número de tuplas mediante un campo especificado.
HAVING – Permite utilizar funciones de agregación dentro del WHERE.
ORDER BY – Ordena los resultados obtenidos por un campo en especificado.
OUTER JOIN – Variante del NATURAL JOIN.

2.3 – Bases de Datos Distribuidas y Paralelas

Con la evolución de las bases de datos se han desarrollado nuevas arquitecturas para manejarlas. Para obtener resultados más rápidos se busca utilizar más de una computadora que se encarguen de procesar las consultas requeridas, ya sea distribuyendo los datos o bien repartiendo la carga de trabajo entre varios procesadores.

Repasemos ahora, en que consisten las bases de datos distribuidas y las bases de datos paralelas.

2.3.1 – Bases de Datos Distribuidas

Un sistema de computación distribuida consiste en un conjunto de computadoras (que no necesariamente tienen que ser homogéneos), que están interconectados entre sí formando una red, y que cooperan para realizar una determinada tarea. Un sistema de computación distribuida parte un problema grande en pequeñas piezas, y soluciona cada una de ellas eficientemente de una manera coordinada.

Podemos definir una base de datos distribuida (BDD o DDB: Distributed Database System) como aquella cuyos datos están repartidos entre más de una máquina interconectadas por una red de comunicación, y un sistema manejador de bases de datos distribuidas (SMBDD o DDBMS: Distributed Database Management System) como el software que administra una base de datos distribuida haciendo que la distribución de los datos sea transparente al usuario [17].

En la arquitectura distribuida el SMBD y la base de datos no están asociados a una computadora determinada, sino a una red cuyos nodos se reparten las funciones. Una base de datos distribuida es vista por las aplicaciones igual que si fuera centralizada. Es el DDBMS el que se encarga de preservar la integridad y coherencia de la base de datos.

Los usuarios acceden a la base de datos distribuida a través de aplicaciones. Estas aplicaciones se pueden clasificar en aquellas que no requieren datos de otras computadoras (aplicaciones locales) y aquellos que requieren datos de otras computadoras (aplicaciones globales).

El principio fundamental de las bases de datos distribuidas es el siguiente:

“Desde el punto de vista del usuario, un sistema distribuido deberá ser idéntico a un sistema no distribuido” [20, 21, 22].

Esta regla fundamental da lugar a las siguientes doce reglas [20, 21, 22]:

1.- Autonomía local
2.- No dependencia de un sitio central
3.- Operación continua.
4.- Independencia con respecto a la localización

5.- Independencia con respecto a la fragmentación.
6.- Independencia de réplica
7.- Procesamiento distribuido de consultas.
8.- Manejo distribuido de transacciones.
9.- Independencia con respecto al equipo.
10.- Independencia con respecto al sistema operativo.
11.- Independencia con respecto a la red.
12.- Independencia con respecto al SMBD.

2.3.2 – Arquitecturas de Bases de Datos Paralelas

El procesamiento en paralelo consiste en dividir una tarea en subtareas más sencillas que puedan ser gestionadas concurrentemente por múltiples unidades de trabajo. Los sistemas paralelos están constituidos por un conjunto de componentes (procesadores, memoria y discos) y pueden comunicarse entre sí a través de una red de interconexión.

De acuerdo a los recursos que comparten, podemos distinguir cuatro arquitecturas de sistemas paralelos [19]:

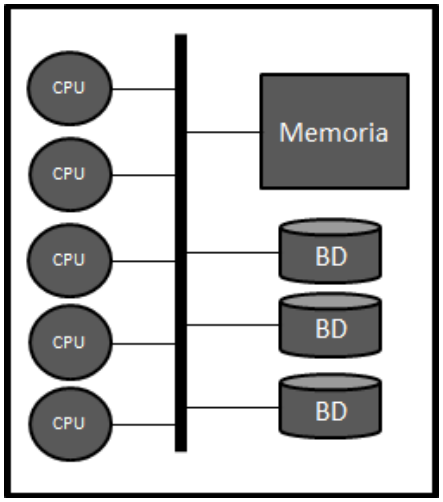


Figura 10. Arquitectura de BDD de memoria compartida.

• De memoria compartida (Figura 10): Todos los procesadores comparten una memoria común. El beneficio de la memoria compartida es la extremada eficiencia en cuanto a la comunicación entre procesadores; cualquier procesador puede acceder a los datos de la memoria compartida sin necesidad de la intervención del software. Un procesador puede enviar mensajes a otros procesadores utilizando escrituras en la memoria de modo que la velocidad de envío es mucho mayor que la que se alcanza con un mecanismo de comunicación.

Las arquitecturas de memoria compartida suelen dotar a cada procesador de una memoria caché grande para evitar las referencias a la memoria compartida siempre que sea posible. No obstante, en la caché no podrán estar todos los datos y no podrá evitarse el acceso a la memoria compartida. El mantenimiento de la coherencia de la caché aumenta la sobrecarga cuando aumenta el número de procesadores.

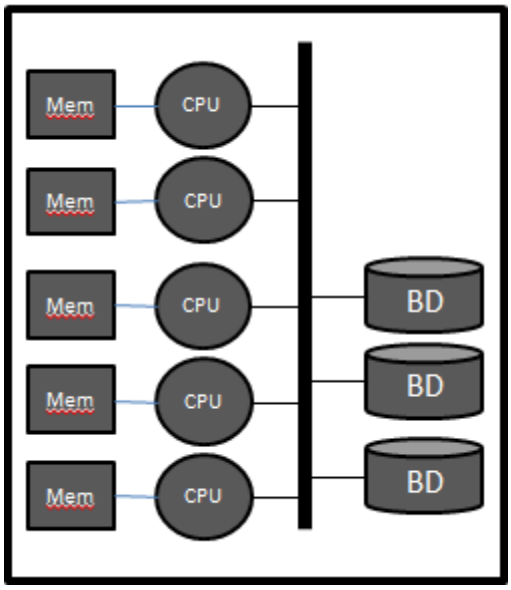


Figura 11. Arquitectura de BDD con discos compartidos.

• De discos compartidos (Figura 11): Todos los procesadores comparten un conjunto de discos en común, pero tienen memorias privadas. Las arquitecturas de disco compartido ofrecen dos ventajas respecto de las de memoria compartida. Primero, el bus de la memoria deja de ser un cuello de botella, ya que cada procesador dispone de memoria propia. Segundo, esta arquitectura ofrece una forma barata para proporcionar una cierta tolerancia ante fallos: si falla un procesador los de más procesadores pueden hacerse cargo de sus tareas, ya que la base de datos reside en los discos, a los cuales tienen acceso todos los procesadores.

La arquitectura de disco compartido tiene aceptación en bastantes aplicaciones. El problema principal de los sistemas de discos compartidos es, de nuevo, la ampliabilidad. Aunque el bus de la memoria no es cuello de botella muy grande, la interconexión con el subsistema de discos es ahora el nuevo cuello de botella; esto es especialmente grave en situaciones en las que la base de datos realiza un gran número de accesos a los discos. Los sistemas de discos compartidos pueden soportar un mayor número de procesadores en comparación con los sistemas de memoria compartida, pero la comunicación entre los procesadores es más lenta, ya que se realiza a través de una red de interconexión.

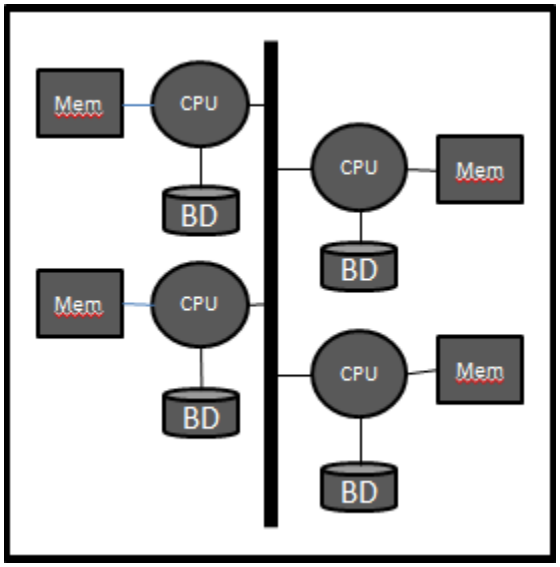


Figura 12. Arquitectura de BDD sin compartir recursos.

• Sin compartir recursos (Figura 12): Los procesadores no comparten ni memoria ni disco. Los procesadores de un nodo pueden comunicarse con un procesador de otro nodo utilizando una red de interconexión de alta velocidad. Un nodo funciona como el servidor de los datos almacenados en los discos que posee.

El modelo sin compartimiento salva el inconveniente de requerir que todas las operaciones de E/S vayan a través de una única red de interconexión, ya que las referencias a los discos locales son servidas por los discos locales de cada procesador; solamente van por la red las peticiones, los accesos a discos remotos y las relaciones de resultados. Es más, habitualmente las redes de interconexión para los sistemas sin compartimiento se diseñan para ser ampliables por lo que su capacidad de transmisión crece a medida que se añaden nuevos nodos.

Como consecuencia, las arquitecturas sin compartimiento son más ampliables y pueden soportar con facilidad un gran número de procesadores. El principal inconveniente de los sistemas sin compartimiento es el coste de comunicación y de acceso a discos remotos, coste que es mayor que el que se produce en las arquitecturas de memoria o disco compartido, ya que el envío de datos provoca la intervención del software en ambos extremos.

También existen modelos híbridos que combinan varias de las técnicas anteriores, muchos de los cuales suelen trabajar con jerarquías, y en cada nivel de la jerarquía manejan un modelo distinto.

2.3.3 – Tipos de Paralelismo en Bases de Datos

Los sistemas de bases de datos con arquitectura paralela deben asegurar que dos procesadores no trabajen simultáneamente los mismos datos de manera independiente, por ello es recomendable atender en paralelo aquellas consultas que requieran solo lecturas a disco y no las que necesiten escribir para actualizar información.

Los tipos de paralelismo que se pueden trabajar en bases de datos al momento de atender consultas son [16]:

- Paralelismo InterQuery: Más de una consulta a la base de datos son atendidas al mismo tiempo.

- Paralelismo InterOperación: Sobre una misma consulta, más de una operación pueden ser procesadas simultáneamente.

- Paralelismo IntraOperación: Una operación puede ser dividida en sub-operaciones más pequeñas si los datos que se procesan están particionados.

La principal métrica para evaluar el desempeño en los resultados de un sistema en paralelo, es la aceleración, y esta se refiere a la mejora en tiempo obtenida en la ejecución de una tarea al incrementar el grado de paralelismo. La aceleración es una medida típica para medir el desempeño en consultas de solo lectura.

La aceleración se puede obtener simplemente dividiendo el tiempo empleado para procesar una consulta en un sistema uni-procesador, entre el tiempo empleado para la misma consulta en un sistema multi-procesador [18].

2.4 - SQLite

SQLite es un motor de bases de datos embebido, desarrollado en C. El código fuente de SQLite es de dominio público y es libre para uso de cualquier propósito, comercial o privado [33].

2.4.1 – Arquitectura de SQLite

SQLite tiene una arquitectura modular para la administración de bases de datos relacionales. Consiste en ocho módulos separados agrupados en tres subsistemas mayores (Figura 13). Estos módulos dividen el procesamiento de una consulta en tareas discretas que trabajan como una línea de ensamble, donde la parte superior compila la consulta, la parte intermedia la ejecuta, y la parte más baja es la que se encarga del almacenamiento y la interacción con el sistema operativo [15].

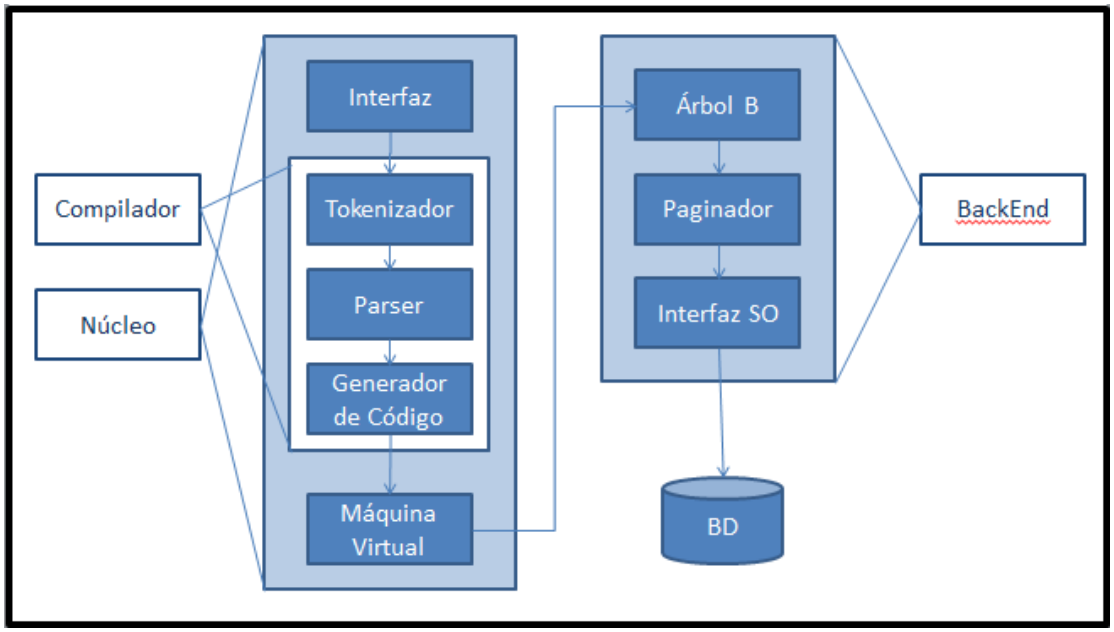


Figura 13. Arquitectura interna de SQLite.

Interfaz

En la parte alta encontramos la Interfaz, la cual consiste en un API de C. Es el medio por el cual los programas, scripts o librerías por igual interactúan con SQLite. Literalmente, es donde el usuario llámese desarrollador, administrador, estudiante etc., se comunica con SQLite.

Compilador

El proceso de compilación empieza con el tokenizador y el parser. Ellos trabajan juntos para tomar una sentencia SQL en forma de texto, validar su sintaxis y entonces convertirla en una estructura de datos jerárquica que pueda ser manipulada más fácilmente por las siguientes capas.

Una vez que la sentencia ha sido dividida en tokens, evaluada y reorganizada en forma de árbol, esto se envía al generador de código.

El generador de código traduce el árbol de parseo en un tipo de lenguaje ensamblado específico para SQLite. Este lenguaje ensamblado consiste en instrucciones que son ejecutadas por la máquina virtual.

Máquina Virtual

Al centro del sistema se encuentra la máquina virtual también llamada motor virtual de bases de datos (VDBE por las siglas en inglés de Virtual Database Engine). El código de VDBE consiste en más de cien posibles tareas conocidas como OpCode, las cuales están centradas sobre operaciones de bases de datos. El VDBE es diseñado específicamente para el procesamiento de información. Cada instrucción dentro del conjunto de OpCodes logra una operación específica de bases de datos o realiza la manipulación necesaria para preparar dicha operación. Todas estas instrucciones juntas y en un orden correcto pueden satisfacer cualquier comando SQL.

BackEnd

El BackEnd consiste de un árbol-B, un paginador y una interface con el sistema operativo. El árbol-B y el paginador trabajan juntos como agentes de información, y manejan información tal como los registros, los campos, índices, etc. Ni el árbol-B ni el paginador tienen conocimiento de su contenido, únicamente mueven y ordenan las páginas sin importar lo hay adentro.

El trabajo del árbol-B es ordenar, mantiene muchas relaciones complejas entre las páginas, las cuales deja conectadas, fáciles de localizar y organizadas dentro de una estructura de árbol. El paginador, trabaja junto con el árbol-B alimentándolo de páginas, transfiere páginas entre el disco duro y el árbol-B. Además el paginador intenta acelerar el proceso dejando páginas frecuentemente utilizadas dentro de la memoria cache y así minimizar el número de veces que lee a disco.

Finalmente, cosas tales como el bloqueo de archivo a menudo se emplean de diferente forma en cada sistema operativo, la interface con el sistema operativo se encarga de proporcionar una capa de abstracción que oculta estos detalles a los demás bloques del sistema, el resultado final es que los demás módulos verán una interface única y consistente para trabajar sin tener que preocuparse por la plataforma en la que se trabaja.

2.4.2 – OpCode de SQLite

La principal razón por la cual en este trabajo se decidió utilizar SQLite, es porque éste es un manejador de bases de datos que nos permite visualizar y entender el plan de ejecución que genera para resolver una consulta SQL.

El plan de ejecución que genera SQLite podemos obtenerlo anteponiendo el comando "explain" a una sentencia SQL cualquiera. La diferencia de ejecutar el comando "explain" en SQLite y en otro manejador, es que otros manejadores nos dan una descripción muy simple del procedimiento que siguen para una consulta, mientras que SQLite nos muestra paso a paso como trabajará la misma consulta, y si estudiamos y analizamos que es lo que hace cada OpCode de SQLite [32], podemos darnos una idea de como atenderá la consulta, no solo eso, sino que también podemos emular cada OpCode para obtener el mismo resultado pero con otros medios.

En capítulos posteriores, se hablará más a detalle de lo que hace cada OpCode, así como también se indicará como es que fueron emulados para trabajar adentro de una GPU y obtener el mismo resultado utilizando el cómputo paralelo.

2.5 – Estado del Arte

Como resultado de la investigación realizada, se enlistan a continuación una serie de trabajos relacionados con el GPU-Computing y Bases de Datos. Desafortunadamente no se encontraron demasiados trabajos que hablen en específico sobre GPUs y bases de datos, quizás se deba a que GPU-Computing es una tecnología de reciente creación, pero aun así, los trabajos que a continuación se mencionan fueron sin duda alguna de mucha utilidad y sirvieron como referencia para el desarrollo de esta tesis.

2.5.1 – Accelerating SQL Database Operations on a GPU with CUDA

En primer lugar hablaremos de [25], este trabajo es el de mayor relevancia para nuestro fin, pues sirvió como principal referencia para esta tesis. El artículo fue publicado en 2010 y fue desarrollado en la Universidad de Virginia, Estados Unidos. La contribución de esta publicación radica en que trabajos anteriores sobre aceleración de operaciones en bases de datos particularmente usando GPUs, muestran aceleraciones muy buenas pero solamente utilizan primitivas que no son parte del lenguaje convencional de SQL.

Este trabajo utiliza el plan de ejecución del manejador SQLite, se enfoca en resolver las consultas SELECT sobre una sola tabla y atienden únicamente datos de tipo entero y flotante, además de las funciones de agregación propias de SQL, tales como MAX, MIN, SUM, COUNT y AVG.

El tipo de consultas que es capaz de resolver y las cuales son mostradas en su documento son como la siguiente:

SELECT id, uniformi, normali5 FROM test WHERE uniformi > 60 AND normali5 < 5;

La estructura del resto de las consultas que se muestran es similar, pudiéndolas describir como la proyección de 3 campos de una tabla en donde se agregan dos o en ocasiones tres filtros por medio de la cláusula WHERE.

En los resultados de las pruebas que realizaron, podemos notar que las aceleraciones para las consultas procesadas en GPU van desde 36x hasta 59x en comparación con la ejecución de la mismas en el CPU, pero además también muestran resultados donde toman en cuenta el tiempo por transferir los resultados de GPU a CPU, y con esta nueva consideración, las aceleraciones van de 28x hasta 43x, es claro apreciar que la aceleración disminuye pues se suma el tiempo procesado más el tiempo de transmisión de resultados.

Las grandes aceleraciones que se obtienen en este trabajo, se deben en gran parte a que el número de accesos a la memoria global se reducen al mínimo, pues cada hilo solo realiza una sola lectura a dicha memoria. Esto se logra por el hecho de que solo atienden consultas sobre una tabla, para esto cada hilo se encarga de procesar una y solamente una tupla, cuando el hilo termina de analizar su tupla correspondiente entonces su trabajo ha finalizado. Esto no sucedería para una consulta con dos tablas en donde las lecturas a memoria global son más numerosas.

El artículo fue de mucha importancia para el desarrollo de esta tesis, pues de ahí se retomó la idea de usar el plan de ejecución de SQLite, pero siempre con la idea de mejorar y superar lo ya hecho, planteando para ello la resolución de consultas más elaboradas que incluyan más de una tabla. A continuación también se presenta el esquema de la tabla que fue utilizada por el trabajo en cuestión [25], el cual podemos considerar como nuestro antecesor directo, ya que su trabajo consiste en paralelizar el también el plan de ejecución de SQLite pero tiene menos características en su funcionalidad.

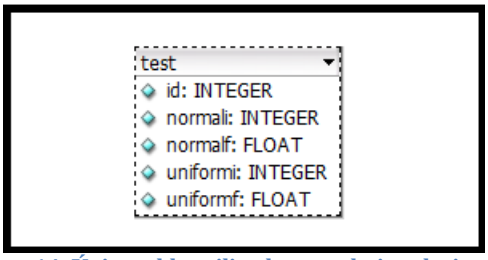


Figura 14. Única tabla utilizada en trabajo relacionado.

La tabla en la Figura 14 muestra cinco campos, el primer campo es un id, pero no contiene índice (ya que, el trabajo en cuestión no soporta índices), además de dos campos de tipo INTEGER y otros dos de tipo FLOAT (no da soporte para tipos de dato VARCHAR). Los datos que contiene la tabla fueron generados de forma aleatoria y toman valores de entre -99.9 y 99.9.

Como podemos ver, solo requiere de una sola tabla, ya que no es capaz de resolver consultas con más de una tabla. Y finalmente el número de tuplas utilizadas es de 5 millones. El hardware que fue descrito en ese mismo trabajo ([25]), fue un procesador Intel Core 2 Duo (Quad Core) con 2GB en RAM, con una sola tarjeta de video NVIDIA Tesla C1060.

2.5.2 – Relational Joins on Graphics Processors

El trabajo [24], procesa o emula la operación JOIN de las bases de datos relacionales y además lo hace sobre GPUs. Este trabajo no resuelve consultas propias de SQL pues no realiza proyecciones o filtros, sin embargo realizan un extenso análisis del procesamiento del operador JOIN sobre una GPU. En este trabajo, implementaron y analizaron cuatro formas de procesar un JOIN.

La primera es simplemente, evaluar tupla por tupla de ambas relaciones, ninguna incluye índices. Esta forma de procesar un JOIN es computacionalmente muy desgastante, pues requiere hacer un gran número de evaluaciones, sin embargo al paralelizar dichas evaluaciones con miles de hilos de ejecución en una GPU, se lograron aceleraciones de hasta 7.0x, en comparación a hacer la misma operación en el CPU.

La segunda forma, es la contraparte de la primera, pues evalúan dos relaciones pero estas ya tienen índices. Sabemos que la búsqueda dentro de un conjunto ordenado siempre será más rápida que la búsqueda dentro de un conjunto desordenado, pues en este último caso se recurre a hacer una búsqueda de miembro a miembro dentro del conjunto. Sin embargo, tener índices es beneficioso en el trabajo de búsqueda tanto para el GPU como para el CPU, la aceleración de las búsquedas sobre relaciones indexadas es de 6.1x a favor del GPU.

El tercer método, es una combinación de los dos primeros, ninguna relación cuenta con índice pero, lo que cambia, es que antes de procesar el JOIN, se crea un estructura de árbol para cada relación ordenándolas mediante merge-sort, así obtienen las tuplas indexadas. Ordenar de forma paralela un conjunto siempre implica tener gran coordinación entre las unidades de trabajo, por lo que el tiempo empleado en realizar la ordenación debe ser sumado al tiempo que se tarda en realizar la búsqueda, con esto es lógico comprender que la aceleración disminuye también, aunque sigue siendo a favor del GPU con 2.4x contra el tiempo que del CPU.

Y finalmente, parecida a la anterior, la cuarta forma inicia sin índices en las relaciones, y antes de procesar el JOIN son indexadas, pero esta vez en lugar de crear un árbol, ahora se acomodan las relaciones dentro de tablas hash, para conseguir índices. De la misma forma que en la anterior, crear el índice requiere de cierto tiempo, tiempo que debe ser sumado al tiempo total, y finalmente se obtienen aceleraciones de 1.9x con la GPU.

Este trabajo, es un muy buen análisis de la operación JOIN en GPUs, rápidamente salta a la vista que las aceleraciones para los dos primero casos (sin índices y con índices) son muy altas, y la diferencia entre una y otra es pequeña. Por otro lado, los últimos dos casos, aquellos en que las relaciones no tienen índices pero les son fabricados unos mediante técnicas distintas, la aceleración es menor, evidentemente el tiempo necesario para crear los índices es bastante alto, por lo que tiempo total requerido aumenta considerablemente.

2.5.3 – Relational Query Co-Processing on Graphics Processors

En [23], podemos encontrar el trabajo más completo sobre procesamiento de consultas sobre una GPU. Este trabajo es de la misma autoría que el anterior, y su idea de trabajo es similar. En este artículo no solo se enfocan a la operación JOIN en particular, sino que van más allá y ahora son capaces de resolver verdaderas consultas SQL.

Desafortunadamente, este trabajo no paraleliza el plan de ejecución propio de un manejador, sino que desarrolla sus propias primitivas para resolver cada una de las operaciones que se pueden presentar en una consulta.

Las primitivas que ellos desarrollaron son las siguientes:

Map: Dado un arreglo de tuplas de datos y una función, la primitiva Map aplica dicha función a cada tupla dentro del arreglo.

Scatter and Gather: escriben y leen dentro de los índices de una relación.

Split: Divide una relación en un número de particiones disjuntas de acuerdo a una función de particionamiento dada.

Sort: Transforma un arreglo de datos desordenados en un arreglo ordenado.

Filter: Selecciona un subconjunto de elementos de una relación, y descarta el resto.

Con estas primitivas, el sistema es capaz de procesar las consultas SQL solicitadas. Otra cualidad a resaltar dentro de este trabajo, es un modelo que propone para obtener el tiempo empleado para resolver una consulta, el modelo de costo es el siguiente:

$$T_{overall} = T_{mm\_dm}(I) + T_{GPU} + T_{dm\_mm}(O)$$

El costo total es la suma de los siguientes componentes:

$T_{mm\_dm}(I)$: Es el tiempo para copiar los datos de entrada desde la memoria principal a la memoria del dispositivo (GPU). (I) denota el conjunto de datos de entrada.

$T_{GPU:}$ Es el tiempo para evaluar la consulta, dados los datos de entrada ya en el dispositivo. Los datos de salida son almacenados en la memoria de video.

$T_{dm\_mm}(O)$: Es el tiempo de copiar los resultados de salida desde la memoria de video hacia la memoria principal. (O) denota el conjunto de datos de salida.

Capítulo 3 - Análisis y Diseño

En este capítulo se detalla el diseño general de la aplicación, se explica como es que interactúa con SQLite, y así también como se logra emular el funcionamiento de dicho manejador.

Analizamos los OpCodes de SQLite para determinar cuáles son los que se ocupan en el tipo de consultas SQL que vamos se van a tratar, y se diseña como será la repartición de la carga de trabajo entre los hilos de procesamiento paralelo de la GPU.

Finalmente se muestran los algoritmos que fueron diseñados para poder trabajar las operaciones SQL, todos ellos pensados desde una perspectiva de cómputo paralelo.

3.1 – Análisis y Diseño General de la Aplicación

Primeramente se tiene que analizar con que arquitectura se va a trabajar, considerando que trabajaremos sobre una GPU debemos pensar entonces que tendremos un sistema de bases de datos paralelo con memoria compartida.

Es importante tomar en cuenta que no es lo mismo trabajar con varios CPU que trabajar con una sola GPU, decimos que tenemos una arquitectura de memoria compartida porque todos los núcleos de procesamiento dentro de la GPU tienen acceso a una memoria en común que es la VRAM (o memoria de video), a través de la cual se pueden comunicar todos los hilos en ejecución.

Se debe destacar, que en este trabajo se operará con una base de datos no distribuida, es decir, con una base de datos centralizada, en el sentido de que no se necesitan particiones físicas (la base de datos esta almacenada en una sola máquina) y tampoco se requieren copias o replicas. Este último aspecto es una característica en el tiempo presente, ya que en un futuro podríamos hablar de trabajar con un arreglo de GPUs y entonces sí se requerirían replicas en todos las máquinas que vayan a operar, pero por el momento basta con tener la información concentrada en un solo equipo.

Ahora, es necesario definir que tipo de paralelismo se va a trabajar, sabemos que existen dos formas de aplicar el cómputo paralelo dentro de SQL, una es paralelismo entre consultas, la cual consisten a grandes rasgos en dedicar un procesador a atender una consulta distinta de forma

individual, así se pueden atender tantas consultas como procesadores se cuenten al mismo tiempo; por otro lado, tenemos el paralelismo en consultas**,** el cual difiere en que se dedican todos los procesadores a trabajar sobre una única consulta. En este trabajo se aborda la segunda opción ya que se emplean todos los GPU-cores de procesamiento para resolver en forma conjunta una misma consulta a la vez. Dicho de otra forma, procesamos una consulta dedicándole todos los hilos o flujos de trabajo de forma exclusiva.

Es momento, de analizar como es que nuestra aplicación trabaja con SQLite, para que se requiere dicho manejador y como se interactúa con él.

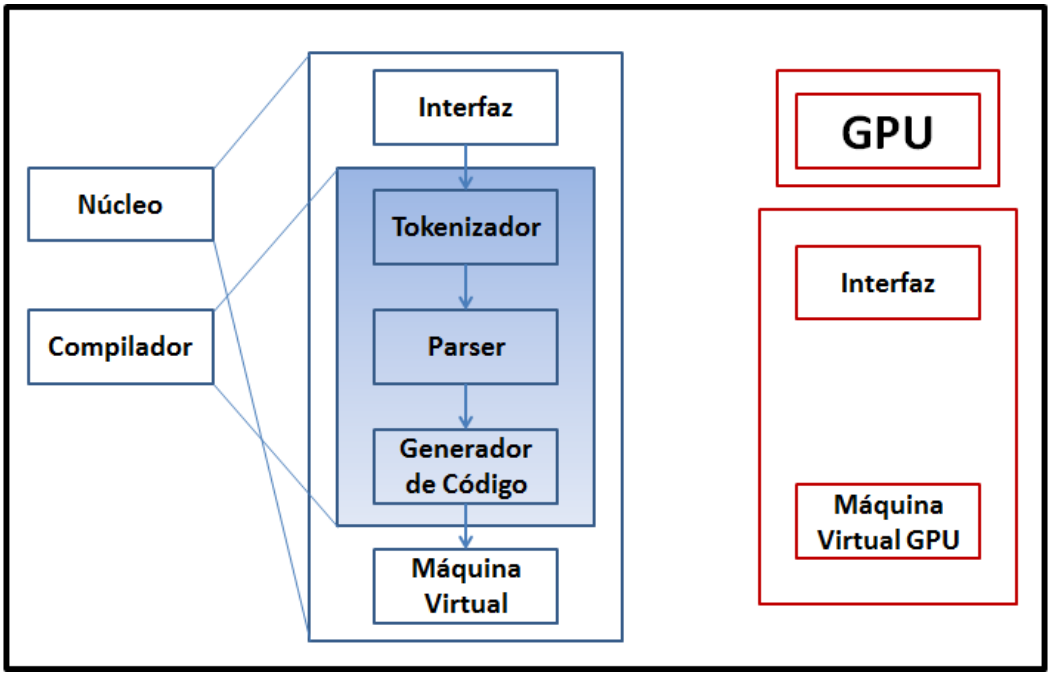


Figura 1. Módulos de SQLite y GPU

De acuerdo con la Figura 15, por un lado tenemos el núcleo de SQLite y por otro tenemos la GPU. Recordemos que no es posible ejecutar el mismo código de SQLite en una GPU tal cual, pero si podemos emular su funcionamiento. El lenguaje de programación que se utiliza es CUDA, él cual se basa sobre el lenguaje C y hace uso también de su compilador, para permitirnos realizar programas que se ejecuten en tarjetas de video (específicamente nVIDIA), sin embargo, pese al gran poder de procesamiento paralelo que nos ofrece la GPU también nos restringe en algunas particularidades de programación tales como hacer uso de apuntadores a funciones, utilizar recursividad y otras, lo cual complica un poco la tarea de igualar el funcionamiento de cada OpCode.

Pero dentro del núcleo de SQLite, debemos identificar los módulos que se requieren para nuestra aplicación. En primer lugar encontramos el módulo de interfaz, este es un módulo que podemos sustituir, ya que solo ofrece la interacción con el usuario y no es más que un prompt en una terminal para capturar las consultas SQL solicitadas por el usuario, por lo tanto, es conveniente desarrollar nuestra propio interfaz y desde ahí dirigir las consultas hacia nuestra aplicación dentro de la GPU para que el contacto entre usuario y el nuevo sistema sea transparente.

Después de la interfaz vienen tres módulos que son el tokenizador, el parser y el generador de código, éstos tres en un conjunto forman lo que es el compilador. El compilador será el factor principal de SQLite que vamos a requerir y con quien interactuará directamente nuestro sistema. La razón de por que es importante utilizar el compilador de SQLite, se debe a la idea de reutilizar su función, es decir, el compilador cumple con la tarea de evaluar una sentencia SQL, revisar su correcta formulación, verificar que no tiene errores de sintaxis o semántica y que es una instrucción válida. Por lo tanto, aprovecharemos esta utilidad del compilador, sin embargo, más allá de validar la sentencia SQL, lo que realmente nos ayuda en este trabajo es el tercer módulo dentro del compilador, la parte final llamado generador de código.

El generador de código, se encarga de elaborar el plan de ejecución para resolver cada consulta, de tal forma que convierte las instrucciones SQL a secuencias de códigos de operación que son comprendidas por SQLite, es decir desarrolla un algoritmo de resolución.

A la salida del generador de código termina también la tarea del compilador, para dar entrada a la máquina virtual, la cual no es más que el motor que dará interpretación al plan de ejecución generado y regresará como resultado las tuplas obtenidas al final de la búsqueda correspondiente a la consulta SQL.

Nuestro trabajo consistirá en analizar los OpCode, adaptar el funcionamiento que cada uno tiene y la manera en como es ejecutado desde el procesador, para ahora ejecutarlo en la GPU tomando en cuenta sus características particulares.

A diferencia del CPU que procesa el plan de ejecución de forma secuencial, la GPU habrá de ejecutarlo de forma paralela, y para ello se deben hacer cálculos para distribuir de forma correcta la carga de trabajo entre todos los hilos, de forma tal que ningún hilo repita o interrumpa las tareas de otros hilos, y finalmente sincronizar la escritura de los resultados para que ningún hilo los pueda sobrescribir.

Finalmente la arquitectura de nuestro sistema (Figura 16), iniciará como un prompt como interfaz de interacción con el usuario, al recibir una consulta SQL esta es analizada completamente por el compilador de SQLite hasta obtener el código de operación correspondiente, mismo que será enviado hacia la máquina virtual-GPU (o VM-GPU), su nombre se debe a que realiza el mismo trabajo que la máquina virtual de SQLite pero procesada dentro de la GPU.

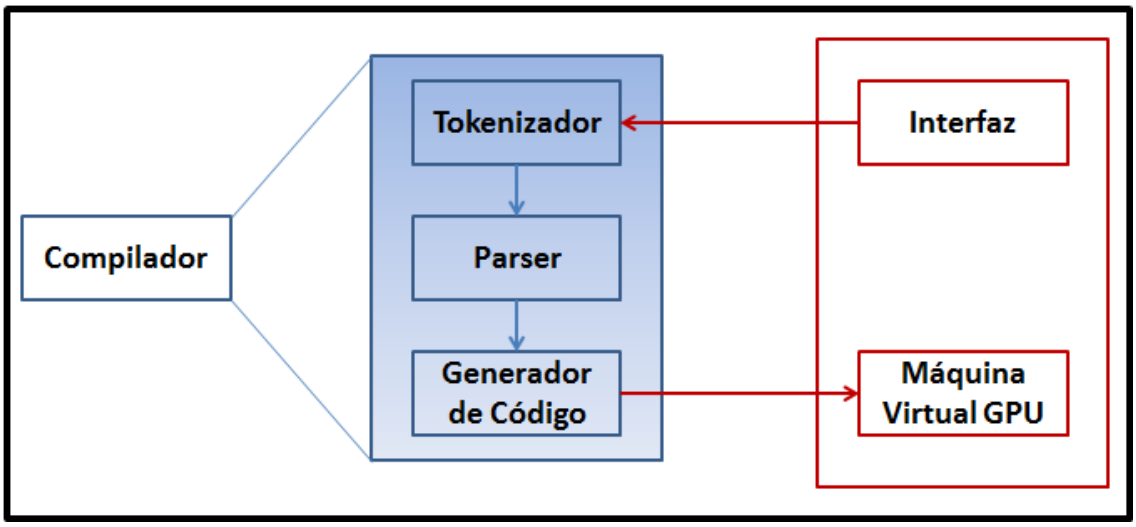


Figura 2. Interacción con el compilador de SQLite

3.2 – Análisis y Diseño del Código de Operación de SQLite

Como se ha mencionado, la parte central de sistema es sustituir el trabajo de la máquina virtual (o motor de búsqueda) por otra que se ejecute de la misma forma pero dentro de la GPU. Es entonces importante entender que es lo que hace cada uno de los OpCode para poder emular su función de manera los más semejante posible.

Cabe mencionar que SQLite tiene un total de 146 OpCodes, de los cuales muchos son de tareas muy específicas y solo aparecen en determinadas sentencias de SQL, por lo tanto, no es necesario mencionarlos todos, sino que es suficiente trabajar con solo aquellos que aparecen en las instrucciones para las cuales esta dedicado este trabajo. Los 45 OpCodes que fueron reprogramados en CUDA se muestran en el anexo A del presente trabajo.

Recordemos que para visualizar el algoritmo de resolución o plan de ejecución para una consulta dentro del prompt de SQLite, basta con colocar el comando “explain” antes de la consulta SQL y enseguida se despliega la secuencia de OpCodes. (Tal como se ve en la figura 17)

```
sqlite> explain select * from prueba where edad > 18;
0   Trace          0    0    0                    00
1   Integer        18   1    0                    00
2   Goto           0    14   0                    00
3   OpenRead       0    2    0    3               00
4   Rewind         0    12   0                    00
5   Column         0    2    2                    00
6   Le             1    11   2    collseq(BINARY) 6c
7   Column         0    0    4                    00
8   Column         0    1    5                    00
9   Column         0    2    6                    00
10  ResultRow      4    3    0                    00
11  Next           0    5    0                    01
12  Close          0    0    0                    00
13  Halt           0    0    0                    00
14  Transaction    0    0    0                    00
15  VerifyCookie   0    1    0                    00
16  TableLock      0    2    0    prueba          00
17  Goto           0    3    0                    00
sqlite>
sqlite>
```

Figura 3. Ejemplo de plan de ejecución SQLite

El plan de ejecución para resolver una consulta SQL se compone de una secuencia ordenada de códigos de operación. A continuación nombramos brevemente algunos OpCodes que fueron utilizados en este trabajo, una descripción más técnica y precisa se encuentra en el anexo A, así mismo en el anexo B se muestran algunos planes de ejecución para distintas consultas SQL.

Los primeros OpCode que veremos (Tabla 7) son de operadores de control de ejecución, ya que sus funciones en general, son para iniciar o terminar la ejecución, o bien mover el índice de instrucción dentro de los operadores ordenados. La mayoría de estos OpCode aparecen prácticamente en todos los planes de ejecución de las consultas que podemos atender. Dentro de estos operadores se encuentran los siguientes:

Tabla 1. OpCodes de Control de Ejecución.

OpCode de Control de Ejecución	
OpCode	**Función**
Trace	Señal de inicio.
Transaction	Inicia la transacción.
TableLock	Obtiene el bloqueo de lectura de una tabla en particular.
Halt	Fin de la ejecución.
OpenRead	Abre una tabla para ser leída.
VerifyCookie	Verifica que la BD este actualizada.
Goto	Un salto incondicional a una instrucción específica.
Rewind	Apunta a la primer tupla de una tabla.
Next	Avanza a la siguiente tupla a evaluar.

Ahora vienen los operadores de asignación de valores a registro (Tabla 8), tanto variables como fijos. Un registro dentro del plan de ejecución de SQLite es una variable que almacena valores numéricos o cadenas de caracteres, para su uso dentro del plan de ejecución

En cuanto a valor fijo, nos referimos a aquellos valores que son dados por el usuario, en otras palabras, aquellos valores que pueden encontrarse textualmente dentro de la sentencia SQL dada por el usuario. Y valor variable, son aquellos que cambian continuamente duran la ejecución.

Tabla 2. OpCodes de Asignación de Valor a Registro

OpCode de Asignación de Valor Registro	
OpCode	**Función**
String	Almacena cadenas de carácter dentro de un registro
Real	Almacena valores de punto flotante dentro de un registro.
Int64	Almacena valores numéricos enteros dentro de un registro.
MustBeInt	Evalúa que un registro contenga un valor entero.
Null	Escribe un tipo de dato nulo dentro de un registro.
RealAffinity	Convierte un valor numérico de entero a punto flotante.
Column	Extrae un dato desde una tupla y lo almacena en un registro.
Rowid	Extrae un dato desde un índice y lo almacena en un registro.
Copy	Copia un dato desde un registro a otro registro.
ResultRow	Indica los registros que contienen una tupla resultado.
IdxRowid	Escribe en un registro la última posición de un índice.
Count	Escribe en un registro el número de registros que hay de dentro de una tabla.

También tenemos operadores que realizan saltos dentro de las instrucciones del plan de después de evaluar una condición (Tabla 9). Dependiendo de que la condición se cumpla o no, se determina cual será la siguiente instrucción a ejecutar.

Tabla 3. OpCodes de Saltos Condicionales.

OpCode de Saltos Condicionales	
OpCode	**Función**
Eq	Salta si los valores que compara son iguales.
Ne	Salta si los valores que compara son diferentes.
Ge	Salta si el segundo valor es mayor o igual al primero.
Gt	Salta si el segundo valor es mayor al primero.
Le	Salta si el segundo valor es menor o igual al primero.
Lt	Salta si el segundo valor es menor al primero.
If	Salta si el valor que evalúa es verdadero.
IfNot	Salta si el valor que evalúa es falso.
IfPos	Salta si el valor que evalúa es mayor que cero.
IfNeg	Salta si el valor que evalúa es menor que cero.
IfZero	Salta si el valor que evalúa es igual que cero.

IsNull	Salta si el valor que evalúa es nulo.
NotNull	Salta si el valor que evalúa es diferente de nulo.
IdxGe	Salta si el segundo valor es mayor o igual al segundo (dentro de un índice).
IdxLt	Salta si el segundo valor es menor al segundo (dentro de un índice)
Function	Compara dos cadenas de caracteres

Ahora, llegamos a los operadores que posicionan el apuntador dentro de una tabla, después de realizar una búsqueda (Tabla 10). Inicializar el apuntador a tabla no necesariamente debe ser desde el inicio, también puede ser inicializado desde un valor intermedio y para ello debe ser buscado para saber si existe ese valor, o si existe uno mayor o menor.

Tabla 4. OpCodes de Inicializador de Apuntador a Tabla después de una Búsqueda.

OpCode de Inicializador de Apuntador a Tabla después de Búsqueda	
OpCode	**Función**
NotExists	Busca la posición de un valor dentro de un índice, salta en caso de no encontrarlo.
Seek	Posiciona el apuntador en la tupla que contiene el valor idéntico al que se busca.
SeekGe	Posiciona el apuntador en la tupla que contiene un valor mayor o igual al que se busca.
SeekGt	Posiciona el apuntador en la tupla que contiene un valor mayor al que se busca.
SeekLe	Posiciona el apuntador en la tupla que contiene un valor menor o igual al que se busca.
SeekLt	Posiciona el apuntador en la tupla que contiene un valor menor al que se busca.

Finalmente, están los operadores que ayudan a resolver las operaciones de agregación (MAX, MIN, AVG, COUNT, SUM). (Tabla 11)

Tabla 5. OpCodes que resuelven Operaciones de Agregación.

OpCode que resuelven Operaciones de Agregación	
OpCode	**Función**
AggStep	Actualiza el registro temporal que contiene en resultado en cada iteración.
AggFinal	Obtiene el resultado final correspondiente a la operación solicitada (MAX, MIN, AVG, COUNT, SUM).

Hasta aquí, todos los OpCode mencionados tuvieron que ser programados en CUDA para poder emular el desarrollo del mismo plan de ejecución, pero ahora sobre la GPU. La lista estos mismos operadores con una descripción más técnica se encuentran en el anexo A.

3.3 – Diseño de Algoritmos Paralelos para Operaciones SQL

Ahora se explicarán los algoritmos paralelos diseñados para resolver las operaciones SQL que atenderá nuestro sistema.

3.3.1 – Diseño de Operación SELECT

La operación SELECT nos permite realizar proyecciones, recordemos que proyectar es la operación que nos permite seleccionar uno o varios campos dentro de los campos pertenecientes a la tabla o tablas involucradas en una consulta.

Básicamente esta operación no se paraleliza, sino que su acción va implícita dentro de la operación WHERE, es decir, pensemos en una tupla resultado, entonces aquel hilo de ejecución que haya encontrado dicha tupla resultado será el encargado también de regresar como resultado solo los campos solicitados con los valores correspondientes de la tupla que encontró.

Ejemplo (ver Figura 18), pensemos que el hilo X de ejecución encontró una tupla resultado, evidentemente el hilo tomó la tupla completa con todos sus campos.

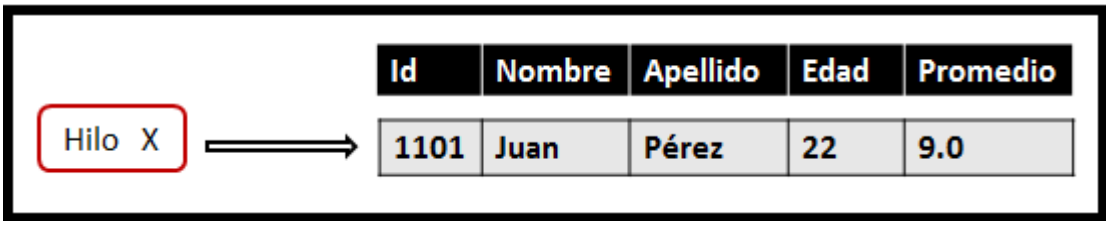


Figura 4. Tupla asignada al hilo X

Pero que pasa si el usuario únicamente solicitó que se proyectaran los campos Nombre y Apellido,

SELECT Nombre, Apellido from tabla;

Aquí es entonces cuando entre en escena uno de los OpCode de SQLite, el operador ResultRow está presente en todos los planes de ejecución y su labor es la de arrojar las tuplas resultado. ResultRow indica cuantos y cuales son los campos de la tulpa respuesta que deberá regresar al usuario (ver Figura 19).

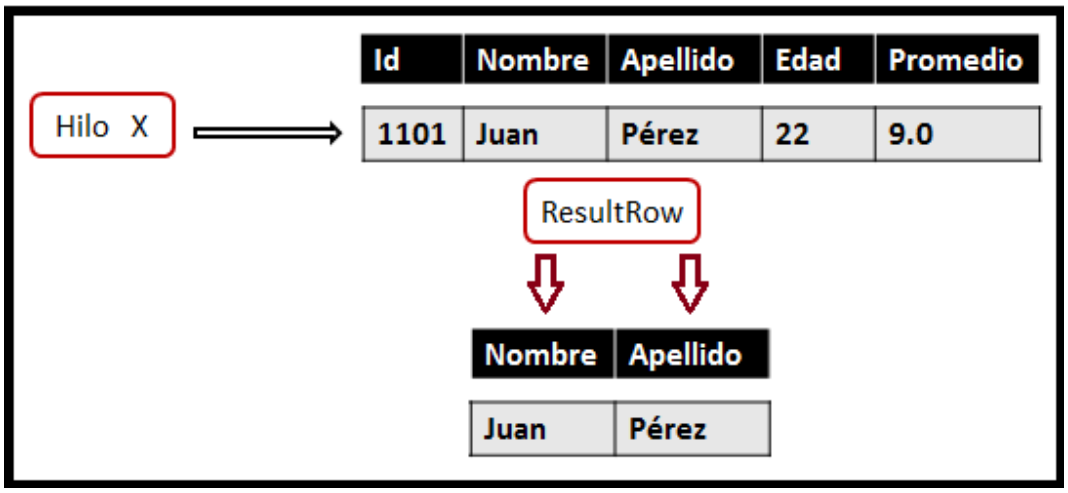


Figura 5. Ejemplo de proyección de dos campos.

Por lo tanto la operación SELECT seguirá realizándose tal cual, pero son los hilos quienes tendrán un trabajo menor, ya que solamente trabajan sobre aquellas tuplas que superan los filtros del WHERE.

3.3.2 – Diseño de Paralelización para Operación WHERE

Recordemos que la tarea de la operación WHERE es realizar filtros y evaluar cuales son las tuplas que cumplen con las condiciones de búsqueda dadas por el usuario. Dentro de las operaciones que pueden existir en la cláusula WHERE están las operaciones aritméticas, lógicas o desigualdades.

Para la operación WHERE, se hace una asignación de tuplas para cada hilo. Una asignación ideal sería, una tupla a cada hilo. En una asignación de tipo tupla-hilo basta con conocer el identificador global del hilo para saber cual es la tupla que le corresponde y así el hilo evaluaría si aprueba o no las condiciones de búsqueda.

Al hilo 0 le correspondería evaluar la tupla 0, el hilo 1 haría lo propio con la tupla 1, así sucesivamente hasta llegar al último hilo H-1 quien evaluaría la tupla H-1 (como en la Figura 20).

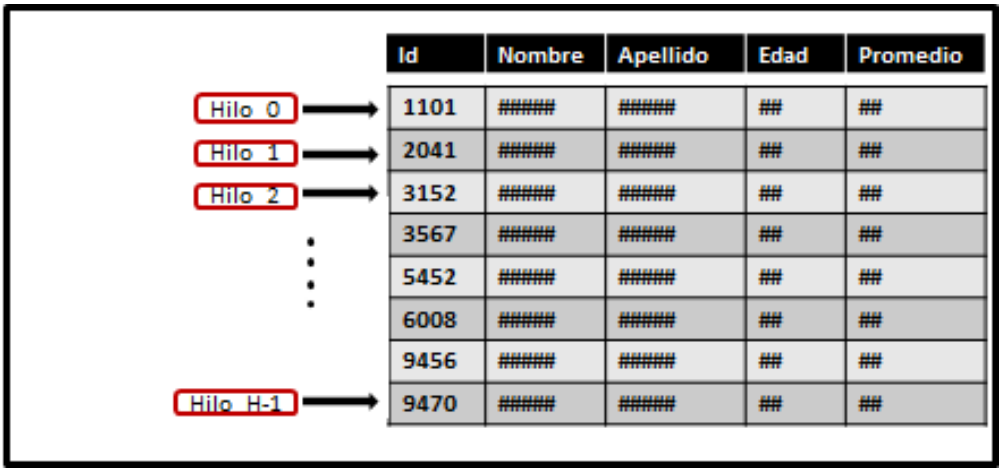


Figura 6. Repartición una tupla a cada hilo.

Cuando existen más hilos que tuplas, no hay ningún problema, solo existirían hilos que no evaluarían ninguna tupla, simplemente no trabajan. Pero del otro lado, cuando hay más tuplas que hilos, se ocasionarían problemas ya que quedarían tuplas sin ser asignadas a algún hilo y por lo tanto quedarían sin ser evaluadas. Para evitar este tipo de conflictos, debemos repartir varias tuplas a cada hilo, de forma que habrá hilos que evalúan más de una tupla si es necesario.

La forma en cómo se asignan tuplas a un hilo es mediante el algoritmo de round robin (Figura 21). El método de round robin, consiste en repartir las tuplas por rondas, de tal forma que en la primera ronda se distribuye una tupla a cada hilo hasta agotar los hilos, al término de esa ronda, si sobran tuplas nuevamente se distribuye una tupla a cada hilo, y así sucesivamente hasta que no queden tuplas. Por supuesto, en la última ronda muy probablemente algunos hilos queden sin una tupla asignada, esto es normal que suceda cuando el número de tuplas no es un múltiplo del número de hilos.

De cualquier forma, la diferencia en cuanto a número de tuplas asignadas a cada hilo será a lo sumo de una tupla, con lo cual tenemos una repartición de lo más justa y equitativa posible.

Ahora, los OpCode que trabajan en la asignación son prácticamente dos, primeramente el OpCode Rewind es quien logra una asignación inicial para comenzar a trabajar, y otro OpCode, el Next se encarga de avanzar a la próxima tupla por procesar.

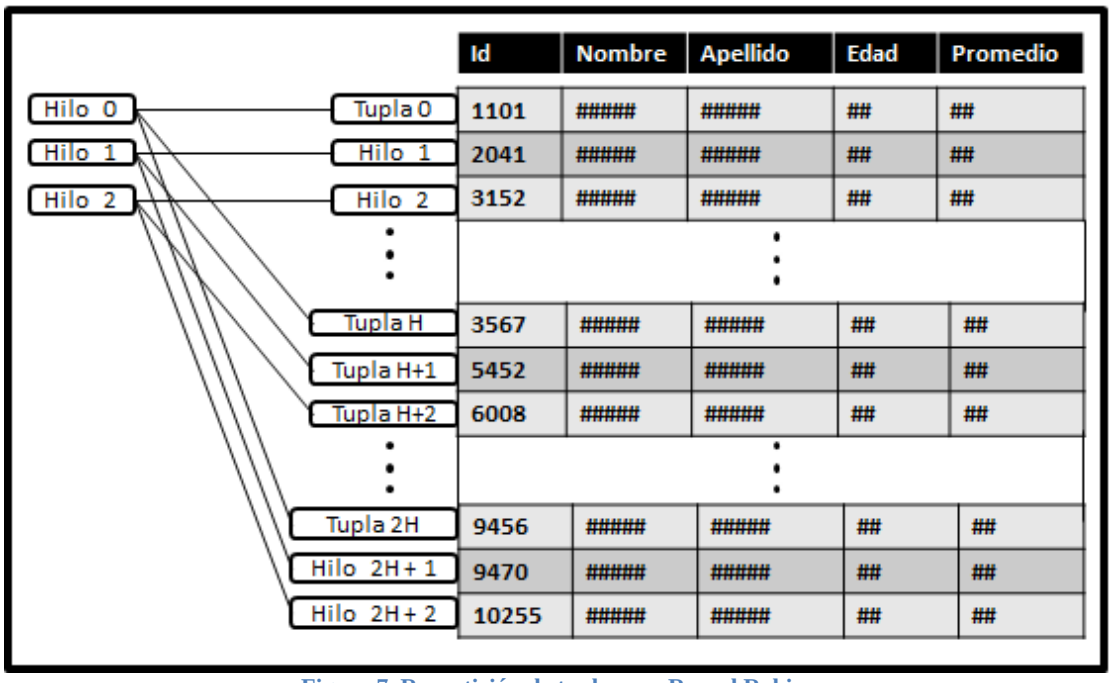


Figura 7. Repartición de tuplas por Round Robin.

En el caso del Rewind, se asigna al hilo 0 la tupla 0, al hilo 1 la tupla 1, hasta llegar al hilo H-1 (donde H es el número total de hilos) que le corresponde la tupla H-1. Rewind trabaja en conjunto con el OpCode Next, este operador indicará a cada hilo cual es la siguiente tupla que le corresponde de acuerdo al método round robin, así al hilo 0 le corresponderán las tuplas H+0, 2H+0, 3H+0 hasta el número de tupla sea menor que N (donde N es el número de renglones en la tabla), para el hilo 1 las siguientes tuplas serían H+1, 2H+1, 3H+1, y el mismo procedimiento para el resto de los hilos (Figura 22).

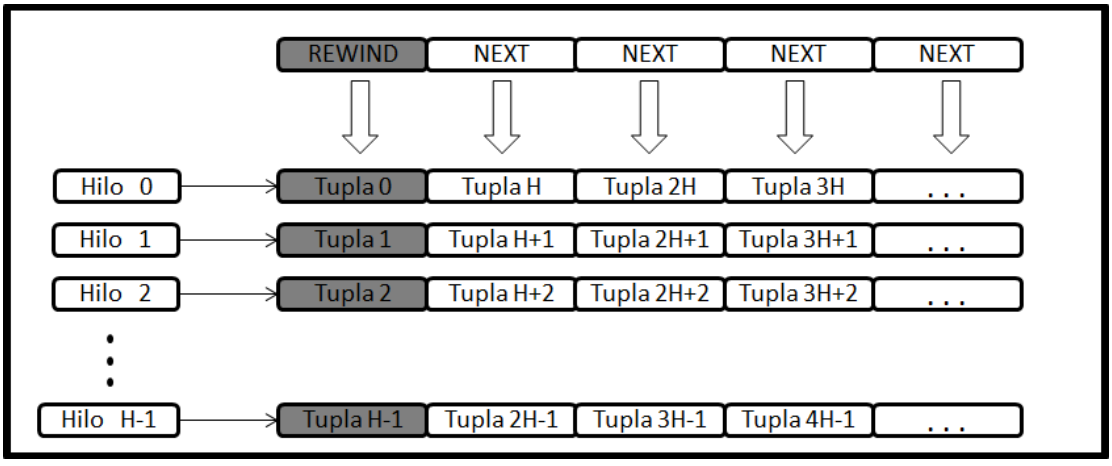


Figura 8. Conjunto de tuplas asignadas por los OpCode: Rewind y Next.

3.3.3 – Diseño de Paralelización para Operación NATURAL JOIN sin Índices

En una operación NATURAL JOIN, sabemos que se trabaja sobre dos relaciones, ambas relaciones tienen un campo en común, y lo que se hace es buscar aquellos valores que dentro del campo común se encuentren tanto en una relación como en la otra. Por lo tanto para cada renglón de la primer relación debemos buscar si existe o no su correspondiente en la segunda relación.

Para esta operación se realiza de igual forma que en el WHERE una asignación mediante el procedimiento de round robin. Sin embargo lo interesante de este diseño de algoritmo, es ver como se distribuyen ambas tablas participantes entre todos los hilos de ejecución concurrente.

Partimos del hecho de que tenemos dos relaciones (A y B) como en la Figura 23, y que la forma secuencial de comparar los valores dentro del campo en común de ambas relaciones es hacerlo tupla a tupla.

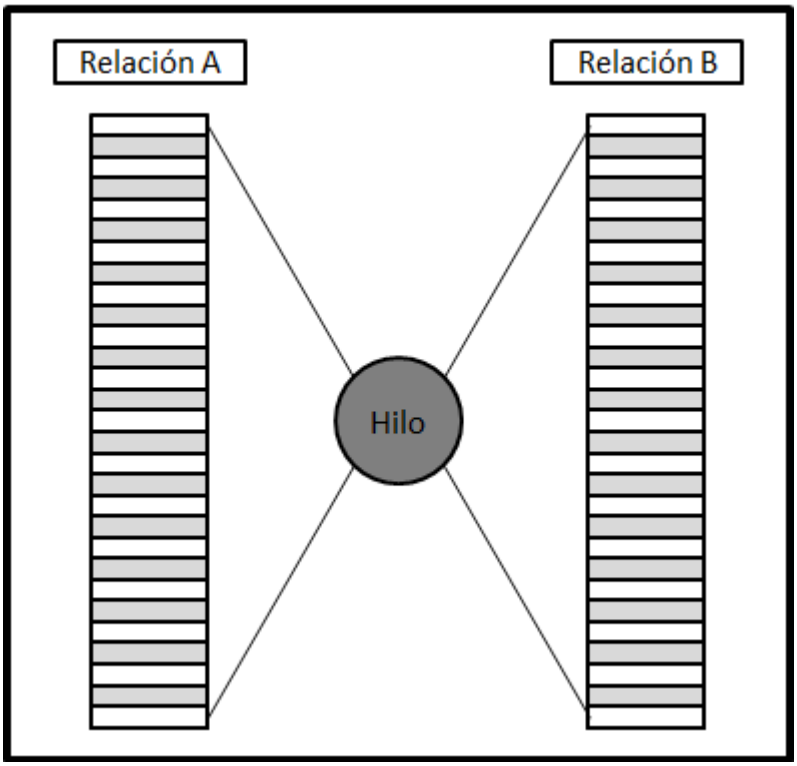


Figura 9. Un solo hilo compara cada tupla en A con cada tupla en B.

Lo primero que vamos a hacer es fragmentar tanto la relación A como la relación B en varios segmentos. Es evidente que se necesitará un hilo de ejecución para que compare cada fragmento de A contra cada fragmento de B. Finalmente, vamos a tener una estructura como muestra la Figura 24.

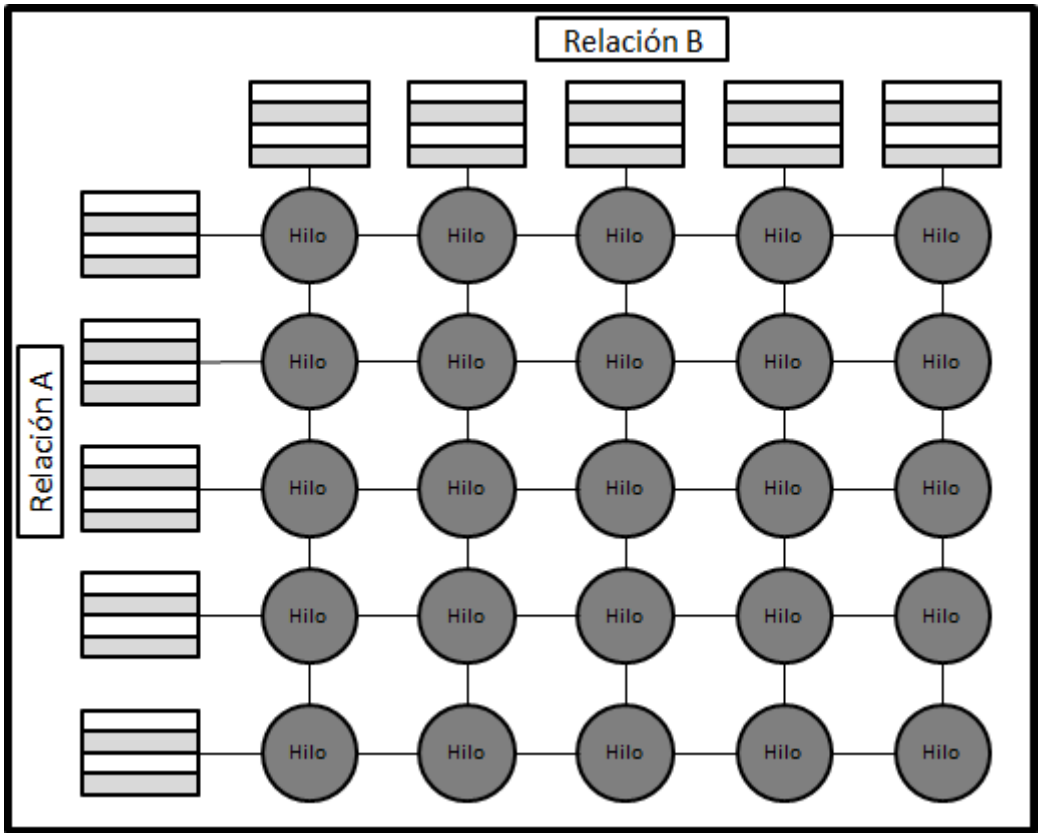


Figura 10. Hilos dirigidos a cada segmento de A con cada segmento de B formando una malla.

Podemos ver que el total de hilos que se deben lanzar estará dado por el número obtenido de multiplicar el número de segmentos de A por el número de segmentos de B. Este algoritmo es así porque debemos tomar en cuenta que las relaciones no tienen índices y no están ordenadas, por lo que hay revisar tupla por tupla, y esta es una buena técnica paralela adaptable a las GPU.

3.3.4 – Diseño de Paralelización para Operación NATURAL JOIN con Índices

Ahora, ¿Qué pasa cuando las tablas tienen índices?, bueno, los índices son una característica propia de las bases de datos de gran utilidad al momento de realizar búsquedas. Los índices nos permiten tener los datos ordenados y con esto sabemos que los tiempos de búsqueda pueden ser más rápidos dentro de un conjunto ordenado.

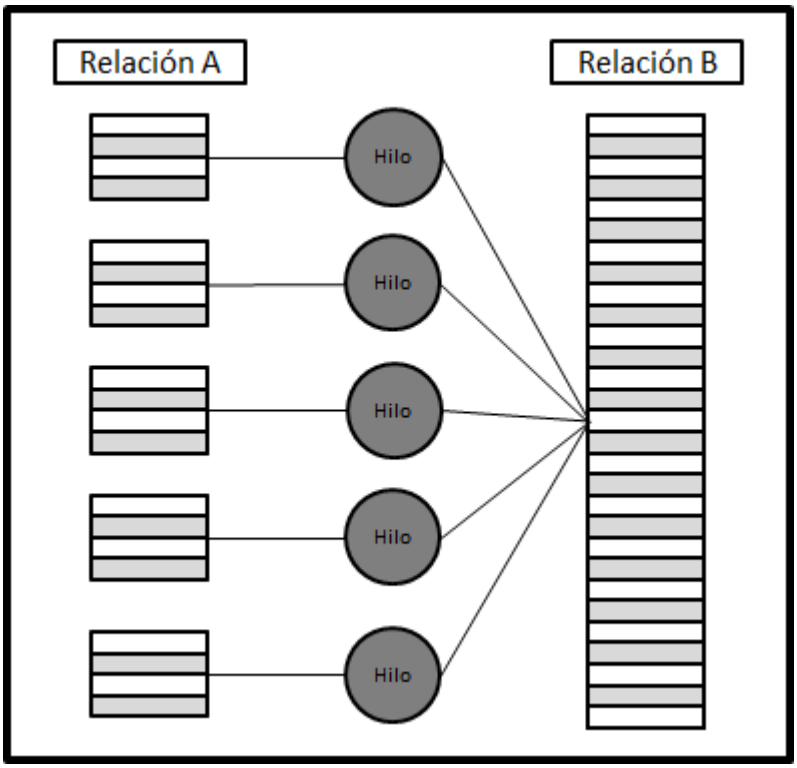


Figura 11. Relación A fragmentada. Mientras que B se encuentra ordenada.

La estrategia que seguiremos para este caso es la siguiente. Tal como se puede ver en la Figura 25, vamos a fragmentar únicamente la relación A. Ahora podemos dedicar todos los hilos para tener más fragmentos de A (mediante Round Robin), y la razón para no fragmentar la relación B, es porque al estar ordenada, podemos utilizar una búsqueda binaria, ya que como sabemos, la complejidad de este tipo de búsqueda es logarítmica y en pocos pasos podemos saber si el valor que se busca está o no presente en el conjunto.

En la Figura 26, tenemos una representación sencilla del funcionamiento de la búsqueda binaria para encontrar el valor 13 (simple ejemplo). Este algoritmo es iterativo y en cada iteración compara el valor que se busca contra el valor que se tiene en la posición central del arreglo, dependiendo de si el valor que se busca es menor o mayor, los limites inferior o superior respectivamente se van moviendo, reduciendo el espacio de búsqueda en cada pasada.

Naturalmente, en el peor de los casos se obtiene la respuesta en $\log_2(N)$ iteraciones, donde N es el número total de elementos dentro del arreglo de búsqueda o número de tuplas de la relación B. Por lo que el trabajo para cada hilo está dado por: $(M/h)\log_2(N)$, donde M es el número de tuplas en la relación A y h el número de hilos lanzados.

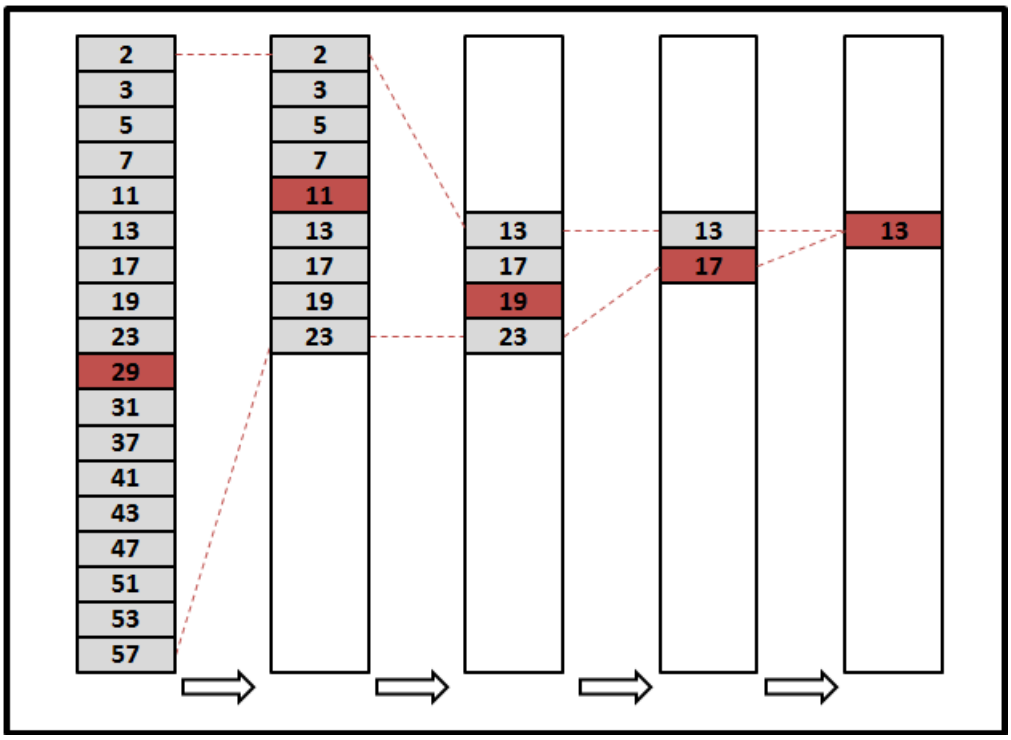


Figura 12. Ejemplo ilustrativo de una búsqueda binaria, para encontrar el valor 13.

3.3.5 – Diseño de Paralelización para operaciones de Agregación (MAX, MIN, AVG, COUNT, SUM)

La forma para atender las operaciones de MAX, MIN, AVG, COUNT, SUM, es sencilla, cada hilo llevará a cabo normalmente el plan de ejecución, cada uno obtendrá un máximo local (o mínimo, o la operación que haya sido solicitada).

Al término del plan de ejecución, para obtener el máximo total (o según la función en cuestión), se realizará un proceso de dos tiempos. En el primer tiempo, los hilos de un mismo bloque serán organizados para obtener quien de ellos tiene el máximo grupal. Y en el segundo tiempo, mediante un hilo representante de cada bloque, se organizarán para ver quién de ellos tiene el máximo total. Un ejemplo ilustrativo se ve en la Figura 27.

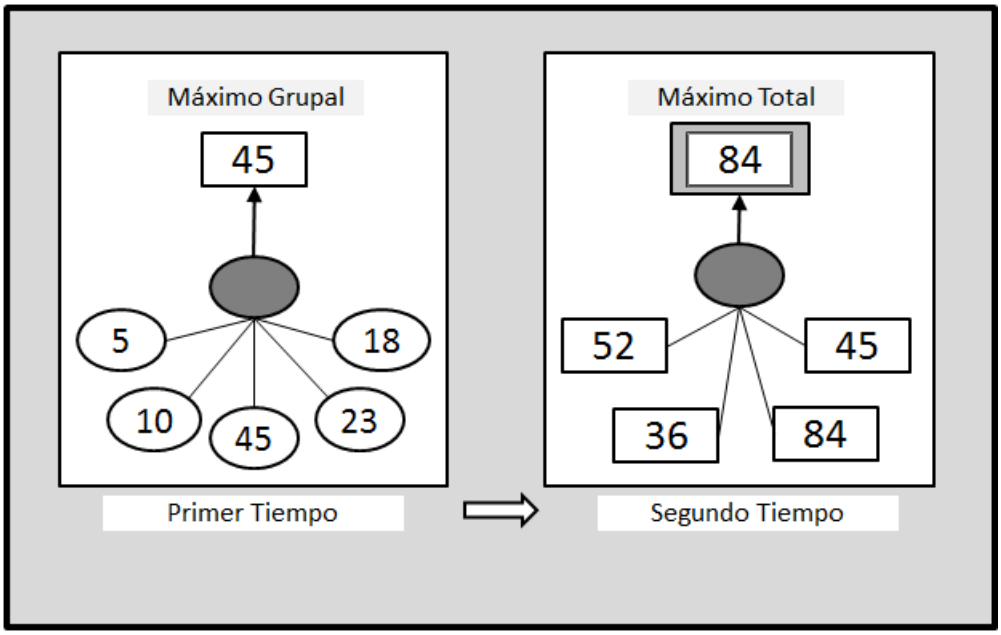


Figura 13. Ilustración del método de dos tiempos para operaciones de agregación.

Para el caso de la operación MIN, el proceso es similar simplemente tomaremos el valor mínimo en cada tiempo; para COUNT y SUM, el operación realizada será acumulativa en cada tiempo, mientras que para AVG, se llevan a cabo las dos operaciones anteriores, y al final se hace una división de SUM / COUNT.

3.3.5 – Análisis para otras operaciones SQL

Existen otras operaciones dentro del SQL que no se abordan en este trabajo (GROUP BY, ORDER BY, entre otras), ya que su resolución con algoritmos actuales y el plan de ejecución de SQLite son incompatibles, son operaciones que se tienen que plantear y diseñar para ser atendidas desde una forma externa y ajena al plan de ejecución que se está utilizando en este trabajo, y por el momento se dejan como un trabajo a futuro.

No existe una métrica para evaluar qué operación puede ser o no paralelizable, pero hay algunas operaciones que al paralelizarlas sus resultados son codependientes, es decir, los resultados de los hilos dependen de otros hilos mutuamente.

Hasta ahora, las operaciones detalladas en los apartados anteriores, son operaciones perfectamente paralelizables, pues su resolución puede ser distribuida entre varios hilos o unidades de proceso, que pueden trabajar libremente al mismo tiempo, y al final sus resultados locales son también resultados totales.

En cambio, operaciones como ORDER BY (ordenación) o GROUP BY (agrupamiento), bien se pueden distribuir y obtener resultados locales, es decir, un hilo de trabajo puede obtener una ordenación correcta de los resultados que haya obtenido el mismo, pero al final, para conseguir una ordenación global, el mismo hilo necesita saber cuales fueron los resultados de los demás hilos, lo cual implica una sincronización exhaustiva.

Actualmente, ha habido trabajos que se dedican completa y exclusivamente a desarrollar métodos paralelos de ordenamiento, tales como [10] y [11], sin embargo, estos métodos no son compatibles con el plan de ejecución actual de SQLite, ya que no hay que perder de vista que a final de cuentas es un plan de ejecución cuyos operadores (algunos) han sido adaptados para ser ejecutados de forma paralela en este trabajo, pero que en esencia, no deja de ser un plan de ejecución pensado y diseñado para ejecutarse de forma secuencial.

Capítulo 4 - Desarrollo

Ya entendidos los módulos que conformarán el sistema, ahora se procede a explicar la forma en como fueron implementados y como se compaginan dichos módulos.

Si bien los algoritmos previamente diseñados y explicados en el capítulo anterior son paralelos, ahora en este capítulo se detalla como fueron desarrollados dentro del contexto de las GPU, pues más allá de tener un algoritmo paralelo, las GPU tienen características peculiares muy importantes a tomar en cuenta que difieren ligeramente de desarrollar el mismo algoritmo en un CPU.

Los módulos fueron desarrollados en orden de tal forma que se pudieran ir probando aquellas consultas simples y gradualmente agregar cierta complejidad y resolverlas mediante la implementación de módulos conforme fueran requeridos, por lo tanto, se explican los módulos de acuerdo a la secuencia en como fueron desarrollados.

4.1 – Plan de Desarrollo

El sistema fue pensado para desarrollarse de forma modular (ver Figura 28), es decir que se le pueda ir agregando funcionalidades para atender consultas cada vez más precisas dentro de una base de datos. Esta forma de trabajar ayudó a que desde el momento en que se desarrolló el primer módulo de funcionalidad (SELECT), se pudieran ver resultados a la hora de ejecutar el programa, al mismo tiempo se realizaban pruebas y se refinaban aquellos errores de programación que se presentaban.

Si bien es claro notar que aún faltan funciones propias del lenguaje SQL por implementar, la ventaja de haber trabajado con esta arquitectura modular es que nos permite agregar dichas funciones en un trabajo a futuro, así la ausencia de ciertas operaciones o clausulas SQL no impide seguir trabajando sobre consultas que si son soportadas por nuestro sistema, y además si en un futuro se pretende sumar módulos que atiendan las operaciones faltantes, su inclusión no entorpecerá el trabajo ya realizado, logrando de esta forma que la escalabilidad funcional sea hasta cierto punto transparente.

Por otro lado, aunque en este trabajo se habla de módulos, es necesario recordar que se utiliza el plan de ejecución del manejador SQLite y que dentro de dicho plan no existen módulos sino OpCode (o códigos de operación), y esta premisa es importante conocerla, ya que no existen reglas ni información que indiquen que cierto OpCode es suficiente y necesario para la clausula X o que otro OpCode lo es para la clausula Y.

El concepto de módulo no existe dentro de SQLite y tampoco hay un OpCode exclusivo para X o Y cláusula, por lo tanto es necesario ubicar que subconjunto de OpCodes son los que están presentes en determinadas tareas, es decir, que existen OpCodes que son necesarios para más de un tarea, en cambio existen otros que únicamente son de utilidad para una sola tarea.

El orden de implementación de módulos, no atiende un ranking de importancia ni tampoco una dependencia de un módulo con su predecesor o sucesor.

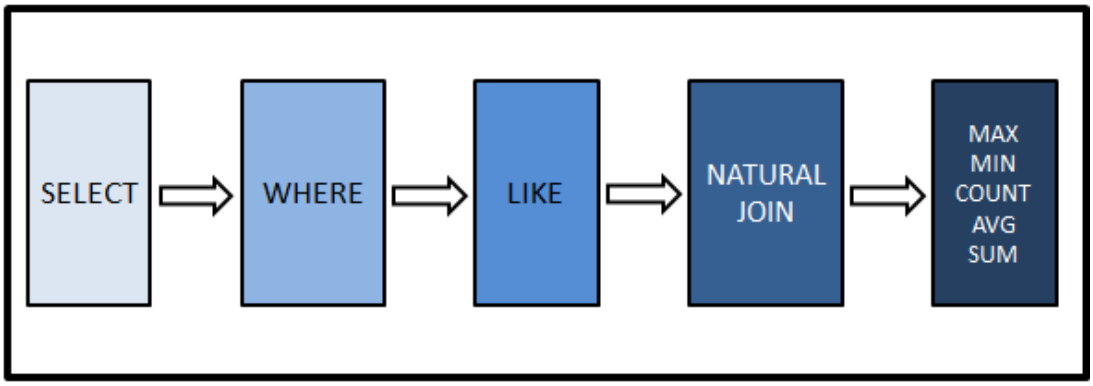


Figura 1. Secuencia de implementación de módulos.

El primer módulo en ser desarrollado fue SELECT, ya se mencionó que la aparición de los módulos no tiene ningún grado de importancia, más sin embargo, SELECT sí tiene un grado más alto de jerarquía, y la razón es simple, SELECT está presente en todas las consultas, sin un SELECT simple y llanamente no existe consulta, podemos tener una consulta sin cláusula WHERE, LIKE o sin funciones de agregación, si solamente involucra una tabla no se requiere el NATURAL JOIN, pero definitivamente no podemos formular consultas sin un SELECT.

Aunado al SELECT va la palabra reservada FROM, si hasta el momento no ha sido mencionada es porque su uso es un tanto obvio, ya que siempre que hay un SELECT debe haber un FROM, ambos van tomados de la mano, FROM hace referencia a al menos un tabla, en otras la palabras, no es posible hacer un consulta sin indicar sobre que tabla o tablas se va a trabajar.

Después se encuentra la cláusula WHERE que como se explico en capítulos anteriores sirve para filtrar resultados y realizar consultas más precisas. En otro módulo separado se colocó la cláusula LIKE, si bien esta es un operador dentro del WHERE, la intención de ponerla aislada es para darle más énfasis a dicha operación, puesto que trabajos relacionados han excluido a este operador a diferencia de nuestro sistema el cual si lo incluye.

Enseguida viene el módulo de NATURAL JOIN, este módulo podría considerarse el más relevante dentro de este trabajo, pues es la característica más sobresaliente en comparación con otros trabajos. El trabajo existente más parecido que incluye bases de datos y GPU, solo realiza consultas sobre una única tabla, por lo tanto, al implementar el operador NATURAL JOIN, nuestro sistema da la posibilidad de procesar consultas que involucren dos o incluso más tablas, con esto podemos resolver consultas más complejas que no son resueltas por otros trabajos anteriores.

Finalmente llegamos al módulo de las operaciones de agregación, la característica de estas operaciones es que requieren de una sincronización de todos los hilos para obtener el resultado total final.

Hasta aquí terminan las funciones que son atendidas por el sistema, y siguiendo la misma idea, podemos agregar módulo que en un futuro atiendan operaciones SQL que no han sido incluidas aún, tales como GROUP BY o ORDER BY.

4.2 – Interfaz de Línea de Comandos y Conexión con SQLite

La interacción entre el sistema y el usuario es simplemente mediante una línea de comandos. En la terminal de texto existe un prompt, siempre a la espera de nuevas consultas a resolver.

Este prompt (SQLite-GPU) sustituye al original de SQLite, así el contacto se hace directamente con nuestro sistema y el uso de SQLite es transparente.

Al momento de iniciar nuestro sistema, internamente se hace una conexión con SQLite, cada consulta ingresada a nuestra línea de comandos, se envía a SQLite para ser evaluada (ver Figura 29). Utilizamos SQLite para comprobar que la consulta ha sido bien formulada, que es válida y que no tiene errores sintácticos ni semánticos.

Los errores que puedan resultar al evaluar la consulta son mostrados al usuario en la misma terminal de texto. Por otro lado, si la consulta es una orden válida, entonces procedemos a trabajarla.

Como en cualquier conexión de un programa con un manejador de bases de datos, las consultas le son enviadas para que devuelva los resultados. En este caso, el procedimiento es similar, lanzamos la consulta al manejador, pero lo que nos interesa no son las tuplas resultado sino su plan de ejecución que utiliza para resolver la misma consulta.

Ya existen funciones predefinidas en las librerías de SQLite para obtener su plan de ejecución mediante código de programación en lenguaje C, dicho plan es atrapado dentro de una estructura.

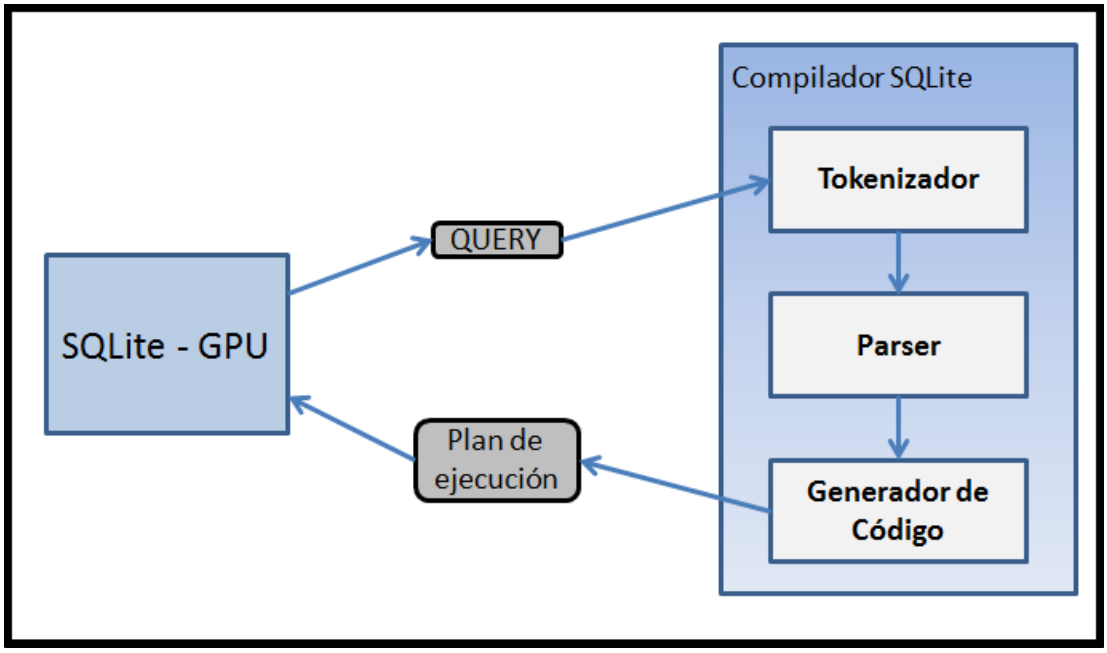


Figura 2. Solicitud de una consulta SQL y obtención de su plan de ejecución.

Para resolver la consulta desde la GPU es necesario enviarle los datos. Dentro del plan de ejecución tenemos un operador de nombre OpenRead, este operador aparece tantas veces como tablas se utilizan en la consulta, un operador por cada tabla, y al mismo tiempo le asigna un identificador a cada tabla con el cual se referirá a lo largo de las instrucciones del plan de ejecución total.

4.3 – Uso de los Niveles de Memoria de GPU

La información que estará dentro de la GPU tiene distintos propósitos y es necesario comprenderlos para hacer un uso lo más eficiente de las memorias dentro de la GPU.

Para detallar las características de cada nivel de memoria de la GPU y el uso de nuestra información almacenada, se muestra la siguiente imagen que ilustra cada caso particular.

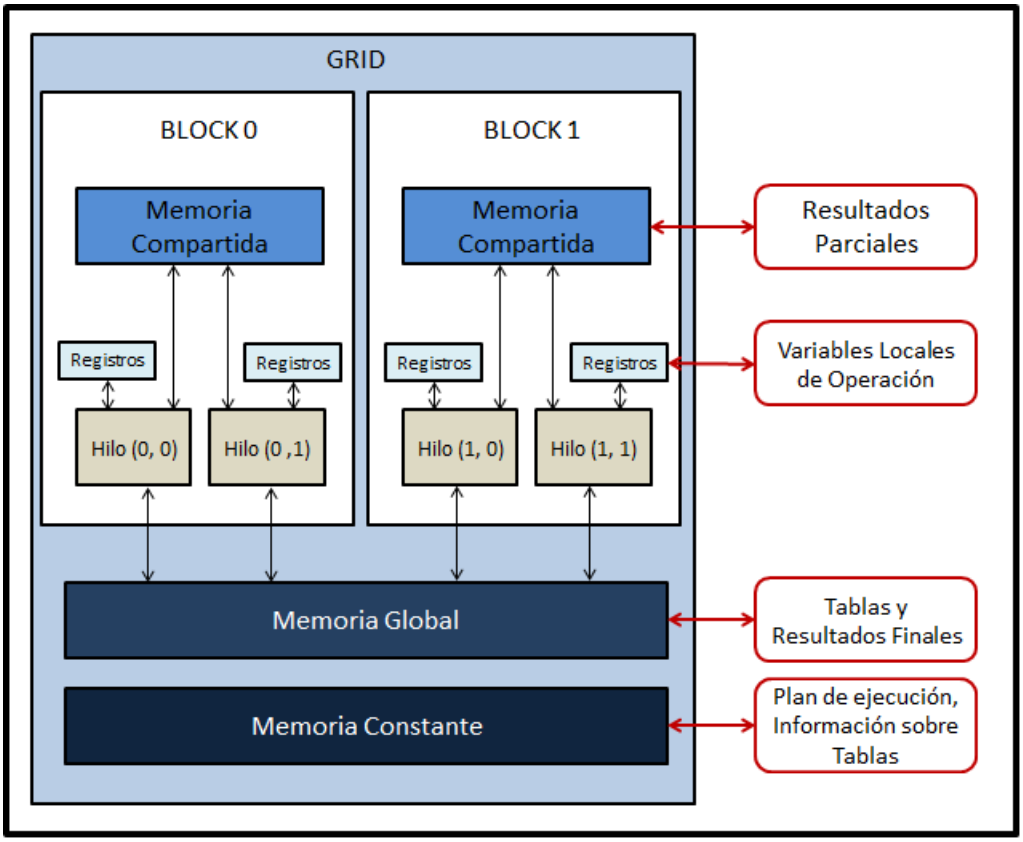


Figura 3. Formas de uso de los diferentes niveles de memoria de la GPU.

La Figura 30 muestra una GRID con dos BLOQUES y cada uno de ellos con dos HILOS, son pocos elementos pues la intención es meramente ilustrativa.

En la parte inferior se encuentra la memoria constante, y su nombre se debe a que la información que contiene no cambia durante la ejecución del programa, esta memoria se escribe solamente desde el HOST (o sea el CPU), pero dentro de la GPU solo es leíble, todos los hilos pueden leerla pero ninguno puede sobrescribirla, a diferencia de la memoria global, la memoria constante es sumamente veloz. Estas características, hacen que la memoria constante sea ideal para guardar datos de uso general, información sobre las tablas tal como número de registros, número de campos y sus tipos o identificador asignado durante el plan de ejecución, así como también el plan de ejecución mismo. Todos los hilos sin excepción requieren en algún momento información sobre las tablas y en repetidas ocasiones están accediendo al plan de ejecución sin embargo todos estos datos nunca deben ser modificados puesto que al hacerlo se pone en riesgo la conclusión exitosa de la búsqueda, por lo que la memoria constante es perfecta para guardar esa información, además como frecuentemente se solicita leer las instrucciones del plan de ejecución, la memoria constante ofrece rapidez para accesos simultáneos.

Yendo hacia arriba en la imagen, encontramos la memoria global, y su nombre radica en su alcance o nivel de visibilidad, en otras palabras, es accesible para todos los hilos en general. A diferencia de la memoria constante, la memoria global si es tanto leíble como escribible desde el CPU y desde dentro de la GPU, y esta característica la hace única pues además de servir para almacenar información sirve como puente de comunicación entre ambos dispositivos HOST y DEVICE (CPU y GPU).

Veamos ahora lo que se guarda en la memoria global y su porque. La memoria global para nuestro uso podemos comprenderla en dos secciones, en una se guardan los datos de ida, o sea los que el CPU envía a la GPU, y en la otra se guardan los datos de vuelta, los que la GPU regresa al CPU.

En los datos de ida, tenemos a las tablas de la base de datos, las tablas deben ser accesibles por absolutamente todos los datos, el acceso es de lectura y nunca de escritura. Ahora, si las tablas están solamente para ser leídas y no modificarse, cumplen con las características para ser almacenadas en memoria constante sin embargo está es muy pequeña y el tamaño de las tablas de datos es demasiado grande que no cabe en dentro de ella. Por otro lado la memoria global sí es lo suficientemente grande como para dar cabida a tablas de tamaño considerable y efectuar las consultas.

El tamaño de la memoria global con la que se trabaja es de 1 gigabyte, por el momento el tamaño de la VRAM o video RAM es la principal limitante de nuestro sistema, ya que solamente podemos procesar tablas que en conjunto no rebasen 1 gigabyte de espacio ya que los datos se cargan a la GPU y la información que excede este tamaño ya no es maniobrable, sin embargo una de las extensiones a futuro del sistema es agregar un paginador que pueda leer las tablas por bloques e ir cambiando los bloques ya procesados por nuevos bloques sin procesar.

En la otra sección de la memoria global se encuentran los datos de vuelta, es específico los resultados, por lo tanto se reserva un espacio de la memoria para depositar las tuplas resultado. El espacio que se debe reservar para los resultados se obtiene evaluar el número de columnas y el número de registros que se esperan. Desde antes de empezar a procesar la consulta, ya sabemos cuantas columnas serán devueltas, ello lo sabemos mediante el OpCode ResulRow, evaluando sus parámetros P1 y P2, el número final de columnas es de P1 + P2 – 1.

Para determinar el número de tuplas que serán registradas, no es posible saber cual es la cantidad exacta pero si se sabe el máximo de tuplas posibles. Si se trata de una sola tabla, el máximo tuplas devueltas es el mismo del número de registros que contiene la tabla. Si se tratan de dos o más entonces el máximo esta dado por el número de registros que contiene la tabla más grande, sabemos esto porque un requisito para las consultas que se evalúan es que el campo llave o llave primaria corresponda al campo sobre el cual se hace el NATURAL JOIN, así conocemos cual será el máximo.

Dado que la memoria global es accesible desde afuera como dentro de la GPU, el CPU bien puede leer los resultados ahí escritos por la GPU.

Otro tipo o nivel de memoria es la compartida, se llama así porque es una memoria a la que tienen acceso los hilos de un mismo bloque, y aquellos hilos externos al mismo no pueden

acceder a los datos allí guardados. La memoria compartida es de lectura y escritura, es veloz pero es pequeña por lo que constantemente hay que vaciarla.

La memoria compartida se utiliza para almacenar temporalmente resultados parciales. Explicándolo a detalle, los hilos de un mismo bloque van depositando sus resultados en la memoria compartida, cuando está por llenarse es vaciada por algún hilo para que pueda seguir siendo almacenando resultados sin alterar los anteriores.

El encargado de vaciar la memoria compartida es el hilo 0 de cada bloque, cuando la memoria esta por llenarse, los hilos se sincronizan mediante la función __syncthreads (), todos los hilos aguardan, el hilo 0 copia los resultados que permanecían en memoria compartida hacia la memoria global, y después de esto, ya no hay problema de sobre escritura en la memoria compartida. La razón por la cual no se almacenan los resultados desde el principio dentro de memoria global, es porque el acceso a ella es lento, y al guardar los resultados temporalmente en memoria compartida reducimos el número de accesos hacia la global.

Finalmente, tenemos la memoria local, los normalmente llamados registros. Los registros son variables de uso exclusivo para cada hilo, absolutamente ningún otro hilo puede entrometerse con los registros de otro hilo, sea cual sea. Se hace mención de esta memoria, aunque la explicación de su uso dentro de esta aplicación requiere mucho detalle, quizás ahondar internamente en el código fuente. Imaginemos que la aplicación corre de forma secuencial, la aplicación requerirá variables de control de avance dentro del plan de ejecución, variables de almacenamiento temporal de datos, etc. Por lo que al correr de forma paralela cada entidad de ejecución requiere de esas mismas variables para si mismo. Esas variables son los registros de GPU también nombrada memoria local, única para cada hilo.

4.4 – Implementación de Algoritmos Paralelos Diseñados

Para implementar nuestros algoritmos en una arquitectura de CUDA-GPU, se tomó en cuenta que no es necesario utilizar una grid de dos o tres dimensiones, para las características de nuestro problema a resolver, sino que basta con una sola dimensión de bloques.

Ahora, dentro de los bloques, también es suficiente trabajar con una sola dimensión de hilos. El hecho de utilizar uno, dos o tres dimensiones de hilos no entorpecen el desempeño de la aplicación, sino que se debe utilizar la configuración que mejor se ajuste a la resolución de cada problema. Estas características se muestran en la Figura 31.

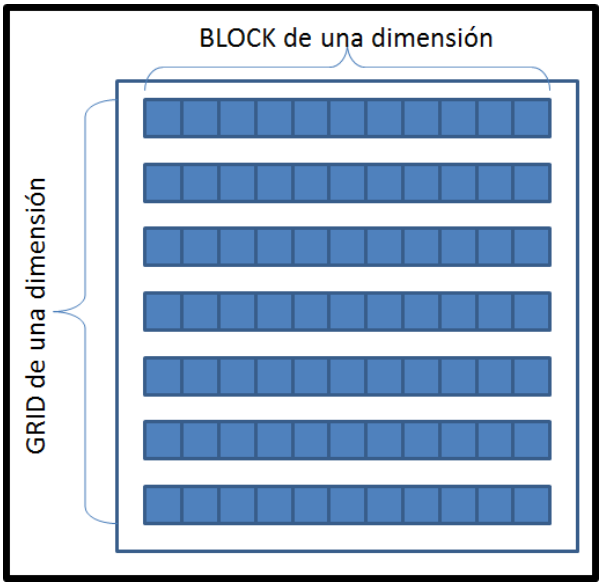


Figura 4. GRID de una dimensión, con BLOCKs con una dimensión de hilos.

4.4.1 – WHERE paralelo sobre una GPU

Partamos desde la consulta más simple:

SELECT * FROM tabla;

A esta consulta la llamamos consulta base, por el hecho de que los OpCode que aparecen en su plan de ejecución están presentes también para consultas más elaboradas. Si se agregan más operaciones a la consulta, más OpCode van apareciendo, pero siempre se encuentran al menos los que componen la secuencia de resolución para la consulta base.

```
sqlite> EXPLAIN SELECT * FROM tablaPrueba;
ad  opcode        p1    p2    p3    p4           p5    comment
--  ------------  ----  ----  ----  -----------  ----  ----------
0   Trace         0     0     0                  00
1   Goto          0     12    0                  00
2   OpenRead      0     2     0     4            00
3   Rewind        0     10    0                  00
4   Rowid         0     1     0                  00
5   Column        0     1     2                  00
6   Column        0     2     3                  00
7   Column        0     3     4                  00
8   ResultRow     1     4     0                  00
9   Next          0     4     0                  01
10  Close         0     0     0                  00
11  Halt          0     0     0                  00
12  Transaction   0     0     0                  00
13  VerifyCookie  0     1     0                  00
14  TableLock     0     2     0     tablaPrueba  00
15  Goto          0     2     0                  00
```

Figura 5. Ubicación de los OpCode: Rewind y Next dentro de un plan de ejecución.

Si bien, la consulta base no contiene la cláusula WHERE aún, el procedimiento para resolver consultas sobre una tabla con y sin WHERE es el mismo. Ya se ha mencionado anteriormente todos los OpCode que fueron reprogramados para conseguir su misma función, pero hay dos de ellos que son la clave para lograr la paralelización de forma ordenada y sin tener traslape en la carga de trabajo de los hilos, estos OpCode son: Rewind y Next (ver Figura 32).

Dentro de CUDA tenemos 4 variables propias de su lenguaje que están presentes en cualquier aplicación y que ayudan a identificar los hilos, estas se ven en la Tabla 12 y en la Figura 33:

Tabla 1. Variables propias de CUDA

Variables dentro de CUDA	
threadIdx	Es un número personal de un hilo para identificarlo dentro de un bloque, ya sea en sus componentes x, y o z cuando son necesarias.
blockIdx	Es un número personal de un bloque para identificarlo dentro de la grid, ya sea en sus componentes x, y o z cuando son necesarias.
blockDim	Es un número de acceso general, que indica el tamaño (o magnitud en hilos) de los bloques en sus dimensiones x, y.
gridDim	Es un número de acceso general, que indica el tamaño (o magnitud en bloques) de la grid en sus dimensiones x, y.

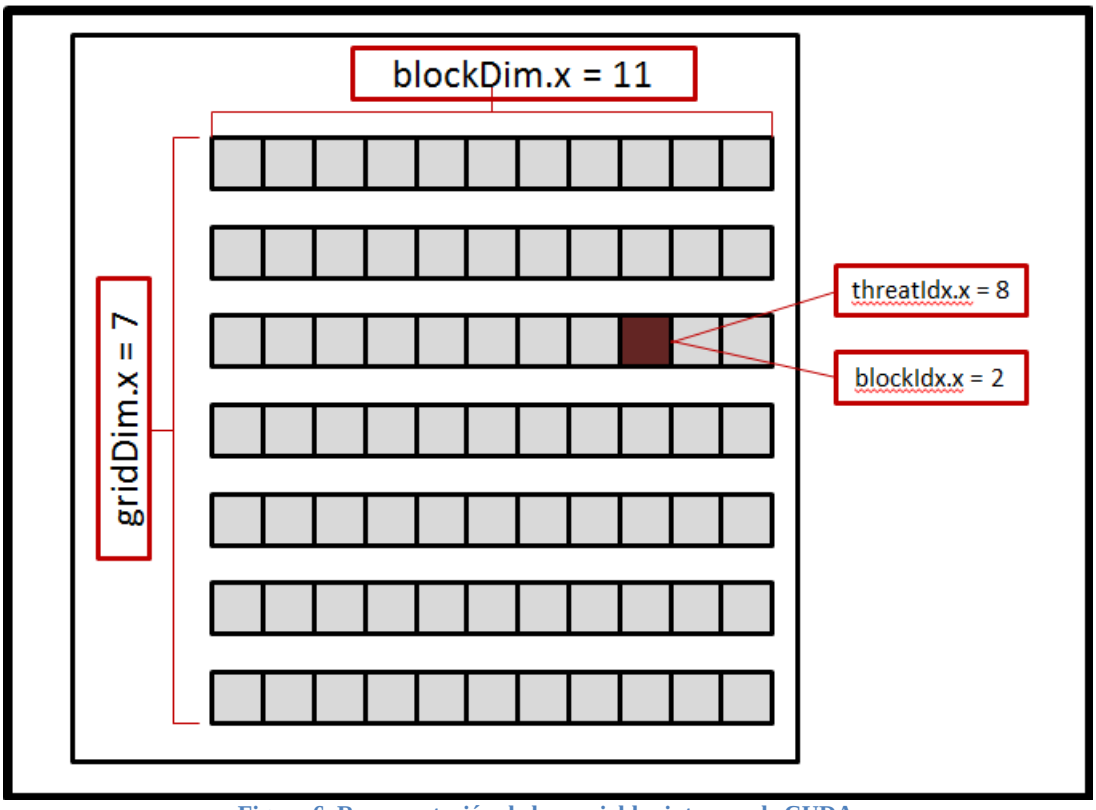


Figura 6. Representación de las variables internas de CUDA.

La idea de identificar a un hilo es para que se pueda hacer una asignación correcta de trabajo sin que repitan el mismo proceso entre sí.

El OpCode Rewind tiene la tarea de reposicionar el flujo de posición a la tupla de inicio en la tabla que corresponde a la instrucción actual. Cuando se habla de una aplicación secuencial solo hay un flujo de ejecución, por lo tanto Rewind siempre nos posiciona a la tupla cero, ya que es la primera. Pero cuando se trabaja sobre una aplicación en paralelo, más de un flujo de ejecución.

Cuando tenemos muchos flujos de ejecución, no podemos permitir que Rewind posicione a cada hilo a la tupla cero, porque todos ellos estarían repitiendo el trabajo sobre la misma tupla, y la intención es que solo un hilo trabaje sobre cada tupla. Entonces para lograr que cada hilo tenga su Rewind sobre tuplas sin repetir, asignamos la tupla que en número corresponde al identificador global de cada hilo.

El identificador global de un hilo es un número irrepetible entre ellos, que no es ni el identificador dentro de un bloque (threadIdx), ni el identificador de su bloque dentro de la grid, pero sí que lo podemos obtener a partir de ellos dos. El identificador global entonces esta dado por:

idGlobal = threadIdx.x + blockIdx.x * blockDim.x;

Solamente usamos la componente en el eje x, pues como recordamos tenemos una grid de una sola dimensión de bloques, los cuales también están conformados por una única dimensión de

hilos. Si tuviéramos más dimensiones, el proceso para obtener el identificador global sería similar pero agregando la complejidad de incluir los eje y y/o z, pero este no es el caso.

De esta forma el operador Rewind, asigna la tupla 0 solo a aquel hilo cuyo identificador global sea 0, al hilo 1 la tupla 1, y así sucesivamente para los demás, por lo tanto nunca más habrá repetición (ver Figura 34).

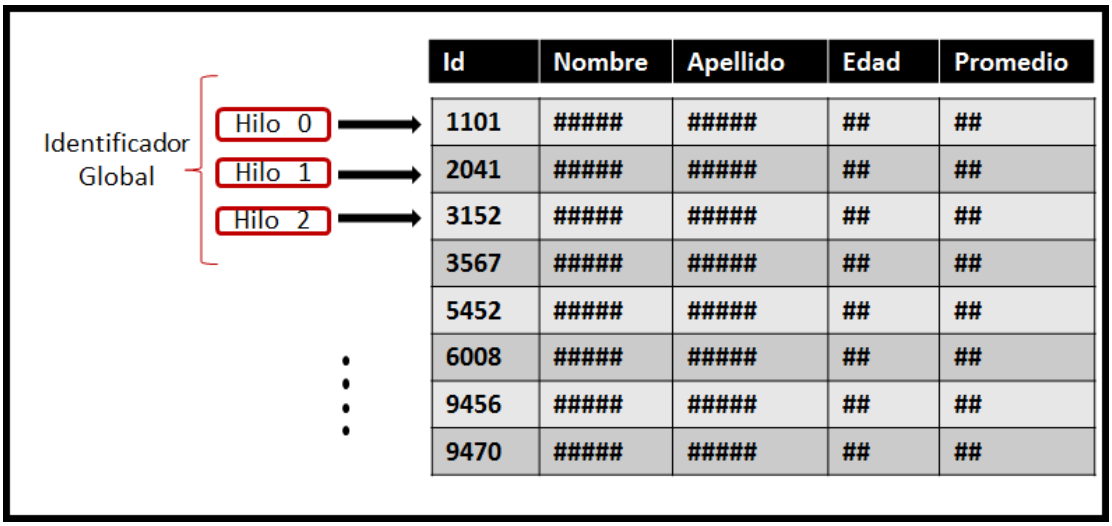


Figura 7. Tuplas asignadas a hilos mediante su Identificador Global

El operador Next, tiene la tarea de avanzar el apuntador hacia la siguiente tupla a analizar dentro de la tabla correspondiente. En un flujo único y secuencial, la siguiente tupla es la inmediatamente posterior, basta con avanzar en una unidad el apuntador a partir de la posición actual.

En nuestro caso, al tener muchos flujos corriendo en paralelo no es posible que Next asigne al hilo quien hizo el llamado la tupla inmediata, ya que seguramente esa tupla ya habrá sido asignada a algún hilo, por ejemplo, el hilo 0 analizó en primera instancia a la tupla 0, y al pasar por el OpCode Next no es posible asignarle la tupla inmediata, porque con toda certeza la tupla 1 ya fue asignada a otro hilo, precisamente hablamos del hilo 1. De la misma forma tampoco se le puede asignar la tupla 2 ni sus posteriores.

Se eligió la técnica de repartición round robin para lograr que Next no tuviera este tipo de problemas, es decir, para que Next haga una correcta asignación, no se suma una unidad a la posición actual del apuntador, sino que suma N, donde N es el número total de hilos. El número total de hilos lo podemos obtener a base de dos de las variables reservadas de CUDA:

N = gridDim.x * blockDim.x;

Simplemente se multiplica la cantidad de bloques que tiene la grid en su componente x (gridDim.x) por la cantidad de hilos que tiene cada bloque en su componente x. Así la tupla asignada por Next para un hilo, es igual a la posición actual de su apuntador incrementado en N unidades, por ejemplo, al hilo 0 le es asignada la tupla 0+N en la primera vez que pase por Next,

para la próxima ocasión Next le asigna la tupla N+N ya que el apuntador actualmente se encuentra en la tupla N y partiendo de ahí se desplaza N posiciones hacia adelante.

Con el operador Next, cada hilo analiza la siguiente tupla que se le asignó siempre y cuando sea una tupla válida. La forma para saber si una tupla es válida o no, es muy sencilla, si Next agrega N unidades al valor actual del apuntador y ese nuevo valor es más pequeño que el número registros de la tabla es entonces un asignación válida, de lo contrario si el valor del apuntador rebasa el tamaño de la tabla, significa que se asignó una tupla inexistente, por lo que es un movimiento no válido, ignorando con ello la instrucción Next y continuando con la siguiente instrucción del plan de ejecución (ver Figura 35).

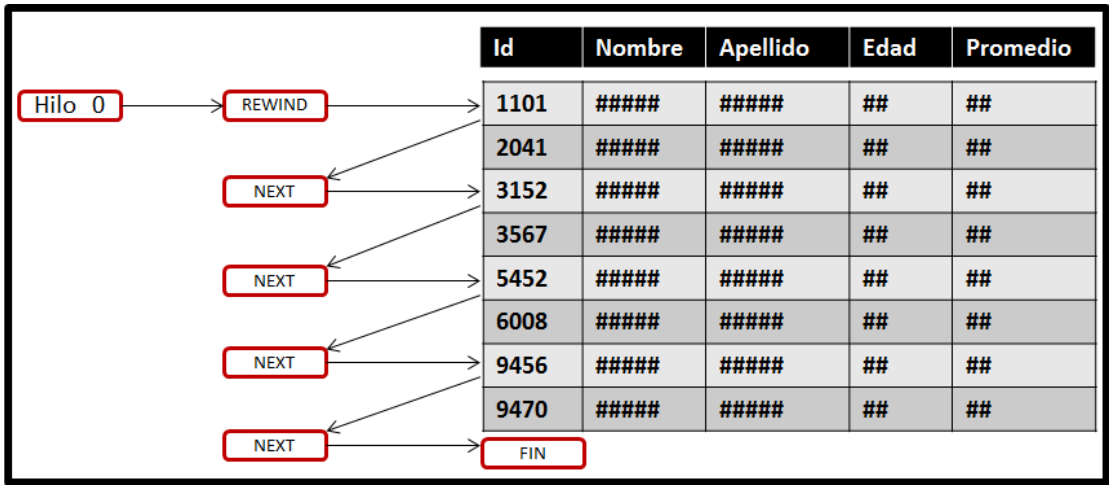


Figura 8. Fin de asignación por parte del OpCode: Next

Ahora sí, cuando queremos agregar filtros de resultados a la consulta, se incluye entonces la cláusula WHERE.

Para trabajar los campos numéricos utilizamos los operadores:

Eq (=), Ne (¡=), Le (<=), Lt (<), Ge (>=), Gt (>) y Between.

Como ya se explicó su análisis en el capítulo anterior, estos operadores saltan hacia la instrucción marcada por P2 si no se cumple con la condición que cada uno evalúa o bien se continúa con la instrucción inmediata en caso de cumplirse.

Por otro lado para trabajar sobre campos de cadenas se utiliza el operador LIKE, y con él, aparecen dentro del plan de ejecución los OpCode siguientes:

Function, If e IfNot

El OpCode Function es quien se encarga de evaluar dos cadenas, ya sea la operación de verificar si las dos son iguales o si una es subcadena contenida dentro de la otra. Como sabemos, las operaciones sobre cadenas tienen la particularidad de que se tiene que revisar carácter por carácter para saber si la operación solicitada es cierta o falsa.

La operación Function regresa como resultado un dato booleano con valor True si se cumple exitosamente la operación y False en otro caso. Con la respuesta de Function, los OpCodes If e IfNot saltan hacia el valor de P2 si el resultado es positivo o negativo correspondientemente, caso contrario continúan sin saltar, con la instrucción próxima en el plan de ejecución.

Finalmente, los operadores AND y OR dentro de una consulta SELECT no tienen un OpCode correspondiente sino que su efecto para unir dos condiciones viene implícito en el plan de ejecución.

Por ejemplo, con el operador OR (Figura 36) basta con que se cumpla una de las dos condiciones que involucra a cada uno de sus lados, para que el plan de ejecución salte hasta el OpCode ResultRow, el cual es el encargado de devolver las tuplas resultado.

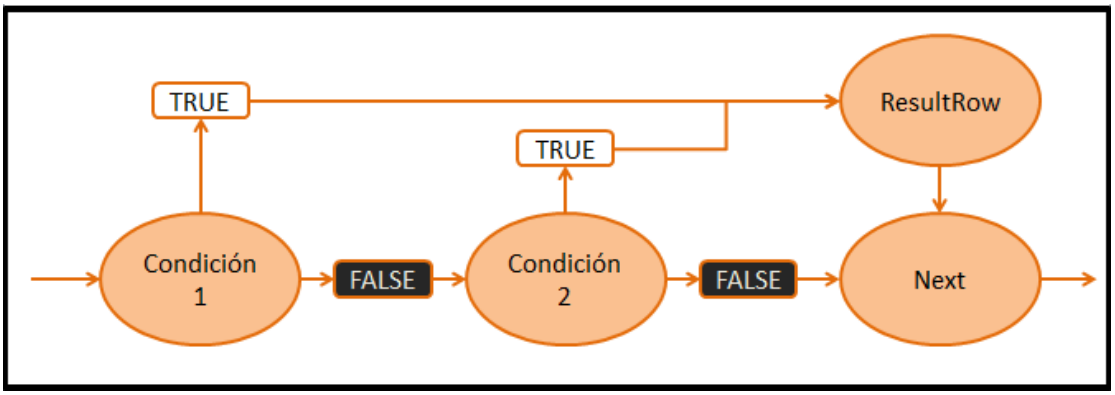


Figura 9. Secuencia de proceso para un operador OR.

Por otro lado, para el operador AND (Figura 37) basta con que una de las dos condiciones que involucra a sus lados no se cumpla, para dejar de evaluar la tupla actual y pasar a la siguiente con el OpCode Next.

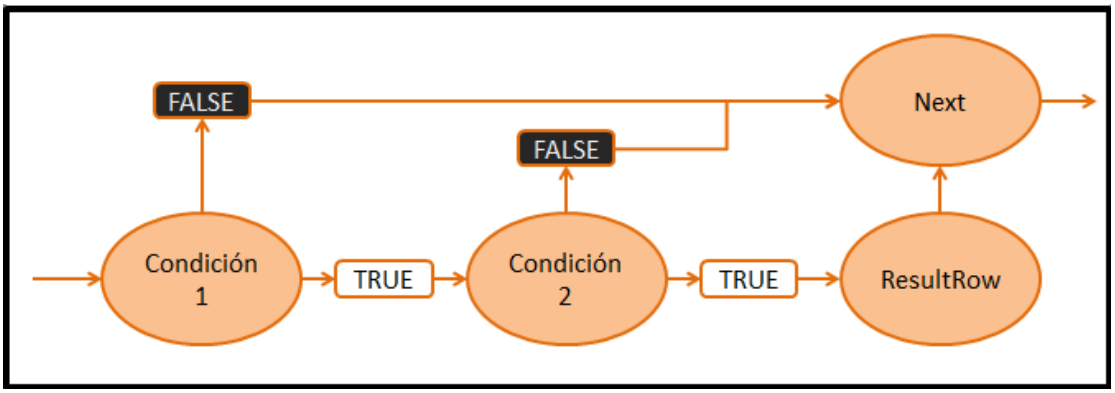


Figura 10. Secuencia de proceso para un operador AND.

4.4.2 – NATURAL JOIN Paralelo sin Índices sobre una GPU

Recordemos que el algoritmo para NATURAL JOIN propuesto y explicado en el capítulo anterior tiene una forma de rejilla o una retícula de hilos. Recordemos también la configuración que le dimos a la GPU de una grid de una dimensión de bloques conformados por una dimensión de hilos. Si empalmamos la imagen del algoritmo propuesto junto con la imagen de la grid de la GPU, podemos observar que embonan perfectamente (ver Figura 38).

Tal como ya se ha explicado anteriormente, el operador NATURAL JOIN involucra dos tablas, por lo tanto, se tienen que hacer recorridos por dos tablas y no por una, como hasta el momento hemos venido haciendo.

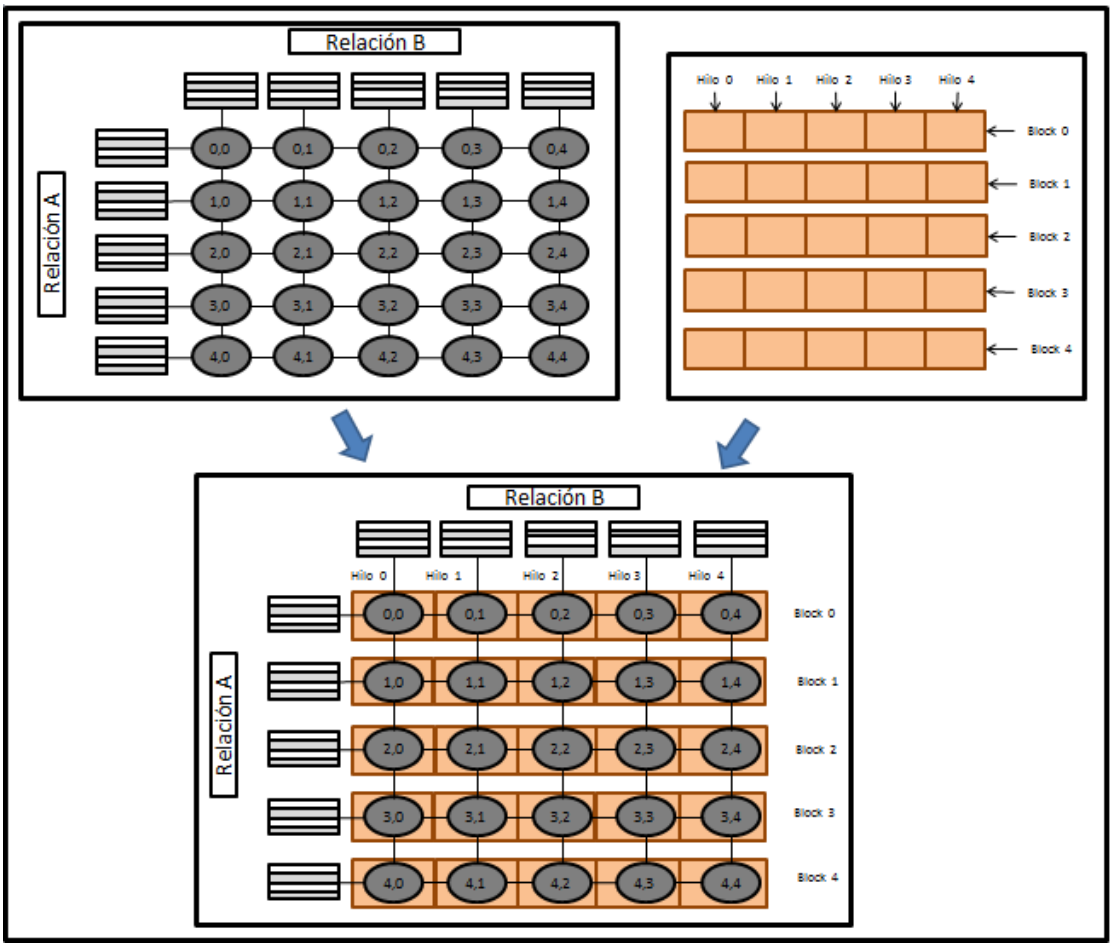


Figura 11. Algoritmo propuesto y arquitectura de hilos en CUDA.

El hecho de tener dos tablas implica que en el plan de ejecución tendremos dos OpCode Rewind y también dos OpCode Next, una pareja de ellos por cada tabla.

Ahora es importante saber cómo es que se mueven los apuntadores de cada tabla, y para ello el funcionamiento de Rewind y Next cambia ligeramente de acuerdo al trabajo que realizaban y al nuevo trabajo más específico que ahora realizan.

Para empezar tenemos el OpCode Rewind. Ya sabemos que Rewind ubica al apuntador a una tabla en la posición inicial sobre la cual comienza a trabajar un hilo determinado. Hasta ahora, nuestro sistema asigna la tupla inicial a cada hilo mediante su número identificador global, sin embargo este método solo es óptimo para consultas con una sola tabla, pero no para las que involucran más de una tabla. Así como tampoco el avance dado por Next es ideal para este caso.

La forma en cómo trabaja nuestro sistema el NATURAL JOIN se muestra en la Tabla 13.

Tabla 2. Plan de ejecución para un NATURAL JOIN sin índices.

Plan de ejecución para un NATURAL JOIN sin índices			
Addr	OpCode	P1	Acción
0	OpenRead	0	Abre la Tabla 0
1	OpenRead	1	Abre la Tabla 1
2	Rewind	0	Apuntador_0 = threadIdx.x
3	Rewind	1	Apuntador_1 = blockIdx.x
4	Column	0	//
5	Column	1	//
6	Ne		//
7	Column		//
8	Column		//
9	ResultRow		//
10	Next	0	Apuntador_0 = Apuntador_0 + blockDim.x
11	Next	1	Apuntador_1 = Apuntador_1 + gridDim.x
12	Close	0	Cierra la Tabla 0
13	Close	1	Cierra la Tabla 1

Lo que sucede, es que cada hilo tendrá dos apuntadores a renglón o tupla, el Apuntador_0 para la tabla que haya sido abierta primero, y Apuntador_1 para la tabla que haya sido abierta en segundo lugar. Pero analicemos que es lo que hacen los OpCode Rewind y Next.

Rewind inicializa a Apuntador_0 en la posición threadIdx.x y el Apuntador_1 en la posición blockIdx.x, así todos los hilos del mismo bloque empiezan a trabajar sobre la misma tupla de la tabla 1, pero su posición inicial para con la tabla 0 estará dado por su número de identificación dentro del bloque, estas dos posiciones no se repiten para ningún hilo y así no tenemos traslape de trabajo.

Por otro lado, el OpCode Next, se encarga de dar el avance para cada apuntador. Next para el Apuntador_0 avanza a partir de la posición actual tantas posiciones como sea el tamaño de la dimensión de bloque (+blockDim.x), es decir, si los bloques son de tamaño N, entonces el apuntador avanzará N unidades, así no hay repetición de tuplas ya procesadas por algún otro hilo.

De la misma forma para el Apuntador_1 pero ahora este avanza a partir de su posición actual tantas posiciones como sea el tamaño de la grid (+gridDim.x), es decir, si la grid es tamaño M, entonces el apuntador avanzará M posiciones y con esto también evitamos repetir trabajo sobre tuplas ya procesadas.

En la Figura 39, vemos como efectivamente, los identificadores de hilo y de bloque no se repiten nunca para dos hilos, siempre cambia uno o cambia el otro, pero nunca son los mismos.

Finalmente, el sub-join generado para cada hilo queda como se ve en la Figura 40. El hilo (X,Y) es aquel donde X es el índice del hilo dado por threadIdx.x y Y es el índice del bloque dado por blockIdx.x.

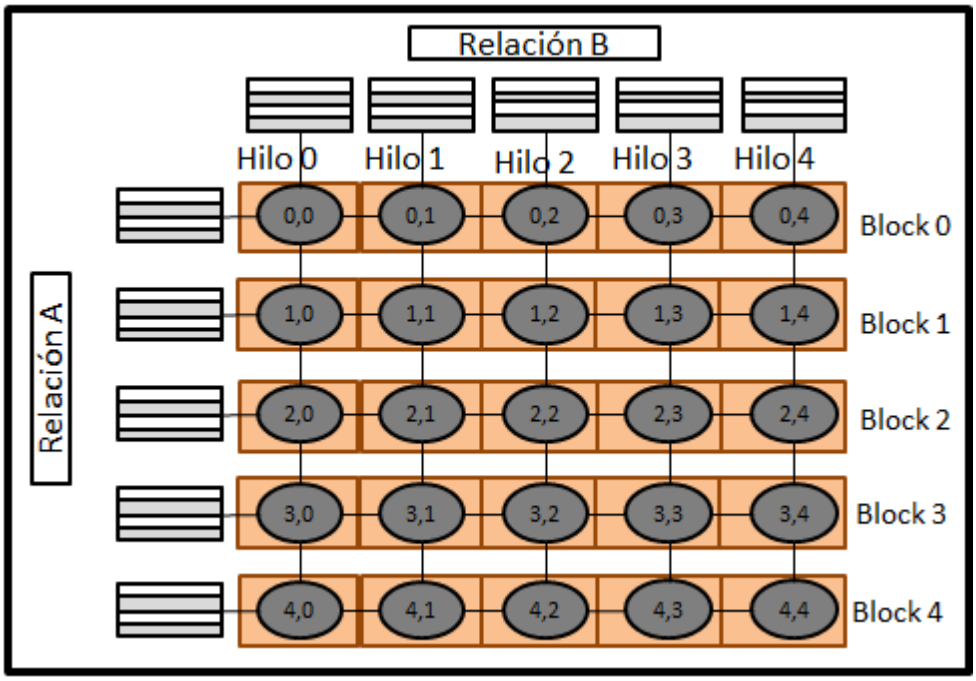


Figura 12. OpCodes Rewind y Next para dos tablas con identificadores 0 y 1.

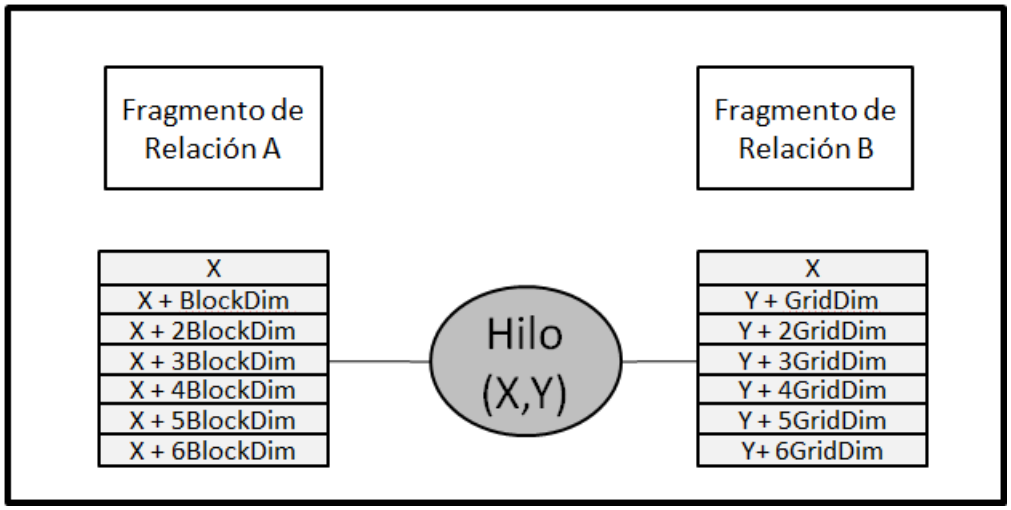


Figura 13. Semi-join para un hilo (X,Y).

4.4.3 – NATURAL JOIN Paralelo con Índices sobre una GPU

Como ya se ha mencionado en el capítulo anterior, los índices nos permiten realizar búsquedas mucho más rápidas, pues al tener los datos ordenados, se ha implementado una búsqueda binaria, la cual nos evita el desgaste de recorrer toda una tabla en la búsqueda de un dato.

Para resolver este caso, dentro del plan de ejecución de SQLite, existe el OpCode "Seek", el cual mediante un método de búsqueda, averigua si un valor solicitado está o no dentro de una tabla. El método implementado en esta tesis para emular el trabajo de "Seek" es el algoritmo de búsqueda binaria y se muestra a continuación en la Tabla 14:

Tabla 3. Algoritmo de búsqueda binaria.

	Algoritmo de Búsqueda Binaria
0	**Búsqueda_binaria(arreglo[], valor_buscado){**
1	lim_inferior = inicio //donde inicio es la posición cero de la tabla o arreglo
2	lim_superior = fin //donde fin es la última posición de la tabla o arreglo
3	posición = 0
4	mientras (lim_inferior <= lim_superior){
5	posición = (lim_inferior + lim_superior) / 2
6	Si (arreglo[posición] es igual a valor_buscado)
7	Termina y regresa posición
8	Si (arreglo[posición] es mayor que valor_buscado)
9	lim_superior = posición – 1
10	Sino
11	lim_inferior = posición + 1
12	}
13	Termina y regresa -1
15	}

4.4.4 – Operaciones de Agregación Paralelas sobre una GPU

Sabemos que en GPU tenemos tres unidades o niveles de trabajo, el primero es el hilo, después tenemos al bloque, y finalmente encontramos la grid. Entendiendo esto, a continuación se describe cómo resolvemos las operaciones de agregación mediante un ejemplo.

Veamos la Figura 41, aquí suponemos que vamos a realizar la operación MAX, es decir, que nos han pedido el valor máximo dentro de un campo. Vamos a necesitar tres variables para almacenar el resultado, una de ellas en cada nivel de memoria de la GPU, en otras palabras, una variable en memoria local (para cada hilo), otra en memoria compartida (para cada bloque), y finalmente una más en memoria global (para el resultado final).

Se resuelven las operaciones de agregación de la siguiente forma: todos los hilos respetan el plan de ejecución y lo desarrollan tal cual. Hasta aquí, cada hilo tendrá su valor máximo (MAX) de aquel grupo de tuplas que le hayan sido asignadas, dicho valor lo mantienen almacenado dentro de su variable local.

El siguiente paso es sincronizar los hilos de un mismo bloque y utilizar una función atómica. Las funciones atómicas son aquellas que, permiten hacer una operación pero solamente por un hilo a la vez (Figura 42), es decir, rutinas que de alguna forma cuando un hilo hasta haciendo uso de ellas, excluye a los demás hilos y nadie más puede entrar hasta que el primero termina, la desocupa y vuelve a entrar otro que lo mantendrá ocupado nuevamente.

La función atómica hace que solamente un hilo a la vez pueda almacenar su valor dentro de la variable de memoria compartida propia de su bloque. Al final de este paso tenemos el valor más alto de todos los hilos pertenecientes al bloque, podríamos decir que tenemos el máximo del bloque. Pero aquí no acaba la operación.

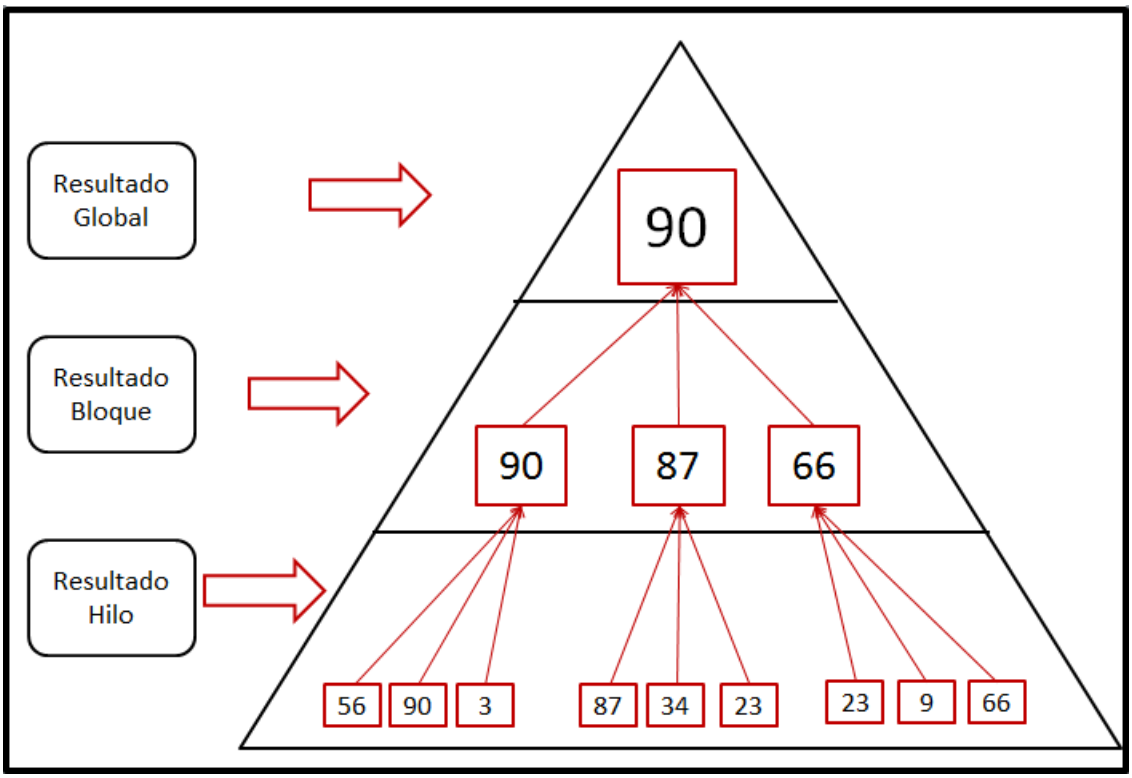


Figura 14. Ilustración de estrategia de trabajo por niveles, para la operación de agregación MAX.

Para el último paso, se utiliza otra función atómica, pero esta vez el resultado se guarda en la variable de memoria global (que mencionamos al inicio de este apartado), solo que ahora solamente entra a la función atómica un líder de cada bloque, es decir, solo el hilo 0 de cada bloque podrá almacenar su valor en la variable global si y solo si, su valor es más alto que el que pueda tener previamente la variable global.

Al término de la operación, tenemos en la variable global el resultado final, es decir, el máximo de todos los valores (ver Figura 41). En el caso de las operaciones MIN, SUM y COUNT, el procedimiento es idéntico, pero claro con su respectiva operación. Pero para el caso de la operación AVG (promedio), no solo se reserva una variable en cada nivel de memoria, sino que se reservan dos, en una se guarda la sumatoria de todos los valores, y en la otra, se guarda el contador de las tuplas que participaron. Y al final, únicamente el hilo cuyo identificador global se a 0, se encarga de dividir el valor de la sumatoria global entre el valor del contador global.

Se eligió esta estrategia de trabajo por niveles, porque es más práctico y se crea un cuello de botella mucho menor al que se crearía si se permitiera que todos los hilos sin excepción entren a la función atómica de la variable global. Lo que se quiere dar a entender, es que efectivamente, existen dos funciones atómicas (una a nivel de bloque y otra a nivel de grid), y con ello, pues claramente dos cuellos de botella (o embudos), pero la ventaja es que en ambos, el número de participantes que puede intervenir, es mucho menor al que se consigue si se deja una sola función atómica en el nivel de grid.

4.5 – Sincronización de Resultados

Como ya se mencionó durante la explicación de los distintos niveles de memoria de una GPU, los resultados son colocados momentáneamente en la memoria compartido por todos los miembros de un mismo bloque.

Para evitar que incluso los miembros del mismo bloque modifiquen los resultados puestos por algún otro hilo en la memoria compartida, se tuvo la necesidad de buscar una solución la cual impidiera la sobre escritura de datos, y para ello, se decidió echar mano de las funciones atómicas (ver Figura 42).

Las funciones atómicas son aquellas que, permiten hacer una operación pero solamente por un hilo a la vez (Figura 42), es decir, rutinas que de alguna forma cuando un hilo hasta haciendo uso de ellas, excluye a los demás hilos y nadie más puede entrar hasta que el primero termina, la desocupa y vuelve a entrar otro que lo mantendrá ocupado nuevamente.

La operación atómica en cuestión es una simple suma, la cual viene ya implementada (addAtom) dentro de las librerías de programación de CUDA. La suma atómica sirve para que cada hilo sume una unidad a una variable compartida por todos los hilos del bloque, si el valor obtenido luego de la suma es menor al máximo de resultados que se pueden contener en memoria compartida, el hilo entonces puede escribir su resultado,

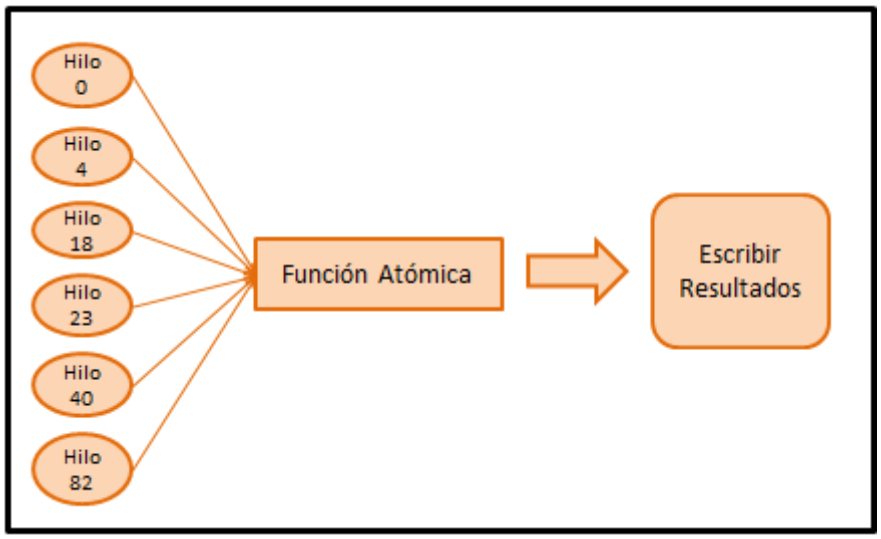


Figura 15. Representación de función atómica para sincronización de escritura.

Por otro lado, si se excede la capacidad de la memoria compartida, entonces, se pone una barrera de sincronización para que antes de que alguien siga escribiendo, la memoria sea vaciada hacia la memoria global que es más grande. Una vez que la memoria compartida está vacía, los hilos pueden seguir escribiendo en ella nuevamente (Figura 43).

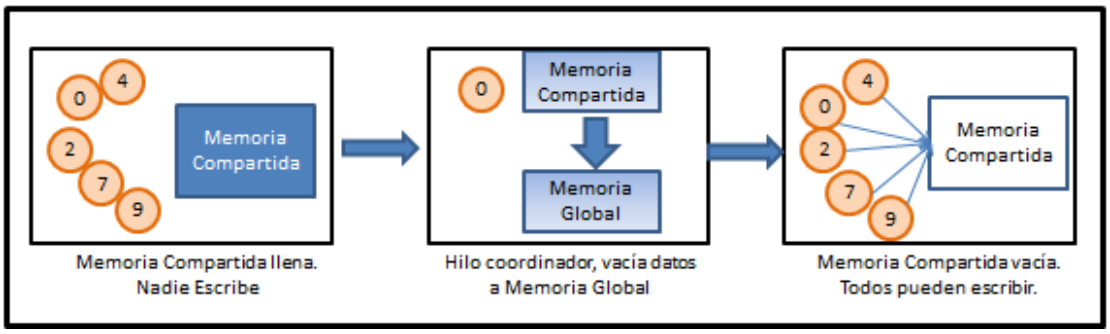


Figura 16. Secuencia de traslado de datos de memoria compartida hacia memoria global.

Pero antes de vaciar la memoria compartida, se ha puesto otra función atómica pero esta es a nivel global. La razón de esta nueva función atómica es para controlar los hilos que vacían datos en memoria global, si bien es cierto que solo un hilo de cada bloque escribirá, aun así debemos sincronizar dichas escrituras. Nuevamente basta con usar una suma atómica para que cada hilo representante de un bloque sepa en qué posición le corresponde depositar sus resultados.

Es mejor hacerlo así que buscar la opción de que cada hilo escriba siempre sus resultados directamente en memoria global, ya que mientras menos accesos para escribir en memoria global, hay un mejor desempeño por parte de la GPU.

4.6 – Sistema en ejecución

A continuación se muestran algunas capturas de pantalla del sistema, donde se muestran los tiempos obtenidos tanto por el motor de búsqueda en paralelo desarrollado en este trabajo, como por el manejador de bases de datos original SQLite.

La consulta que se resuelve en las pantallas fue formulada al azar y es simplemente representativa. La consulta de prueba fue la siguiente:

SELECT t1.id, age, education, occupation FROM T1 NATURAL JOIN t2 WHERE
age BETWEEN 40 AND 60 AND education LIKE “Bachelors” AND occupation NOT LIKE “Sales”;

La Figura 44, muestra el prompt de nuestro sistema desde una línea de comandos siempre a la espera de recibir una nueva consulta. Se decidió llamar al prompt “SQLite-GPU ->”, haciendo referencia a que usamos el mismo plan de ejecución que el manejador SQLite pero que este se resuelve dentro de una GPU.

Después de resolver la consulta de ejemplo, el sistema muestra el número de renglones que se encontraron y que cumplen con las condiciones de búsqueda, así también muestra el tiempo que le tomó resolverla, el cual es de 0.341831 segundos. Este tiempo incluye el tiempo de envío de datos a la GPU, el tiempo de procesamiento de la consulta y el tiempo de recepción de resultados desde la GPU a memoria principal.

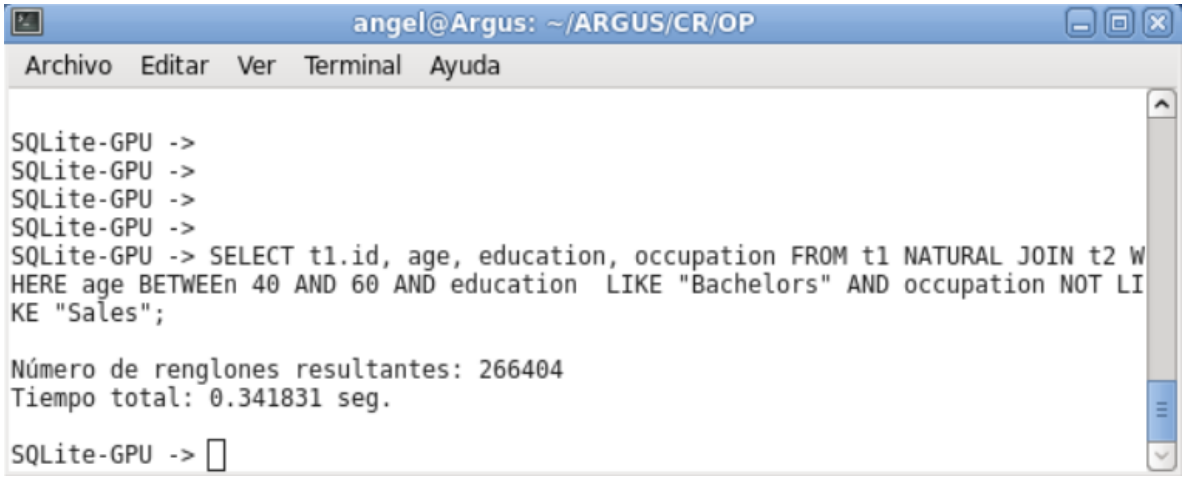


Figura 17. Tiempo consumido por la GPU para resolver una consulta.

La Figura 45 muestra el tiempo que le toma a SQLite original resolver la consulta, el tiempo medido es de 6.604793 segundos. Para obtener este tiempo, se desarrolló un pequeño programa que lo único que hace es conectarse al manejador SQLite, lanza la consulta y cronometra el tiempo de respuesta sin tomar en cuenta el tiempo de imprimir en pantalla.

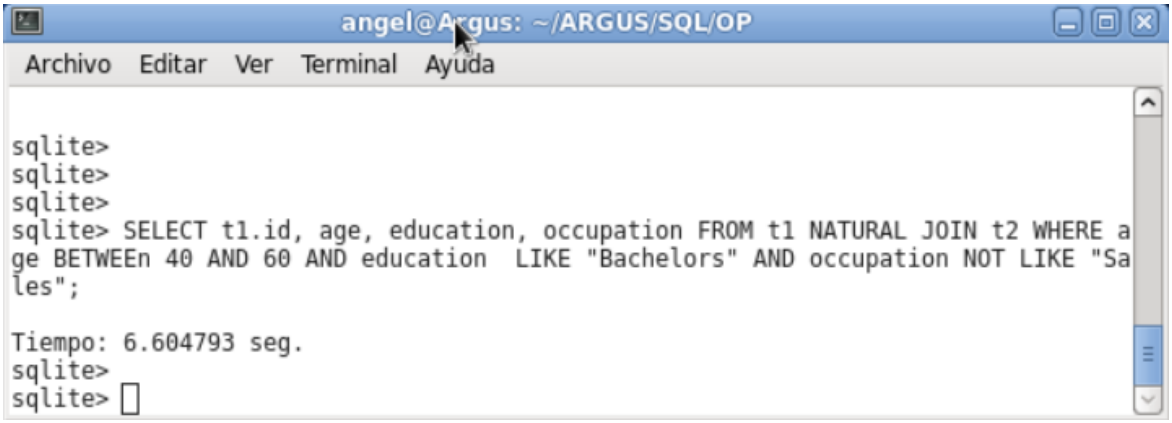


Figura 18. Tiempo consumido por el CPU para resolver una consulta.

Ahora, nuestro sistema, tambien nos permite visualizar los renglones que encontró imprimiendolos en la misma terminal de ejecución. Para el caso particular de la consulta que se probó, se obtuvieron 266404 renglones respuesta, de los cuales por cuestiones de espacio solo se muestran los últimos que se alcanzan a ver en la terminal.

La Figura 46 nos muestra los renglones que obtuvo nuestro motor de búsqueda paralelo, podemos identificarlo por el prompt “SQLite-GPU ->”, mientras que la imagen 46 muestra los renglones que obtuvo el manejador SQLite original, podemos identificarlo por su prompt “sqlite>”.

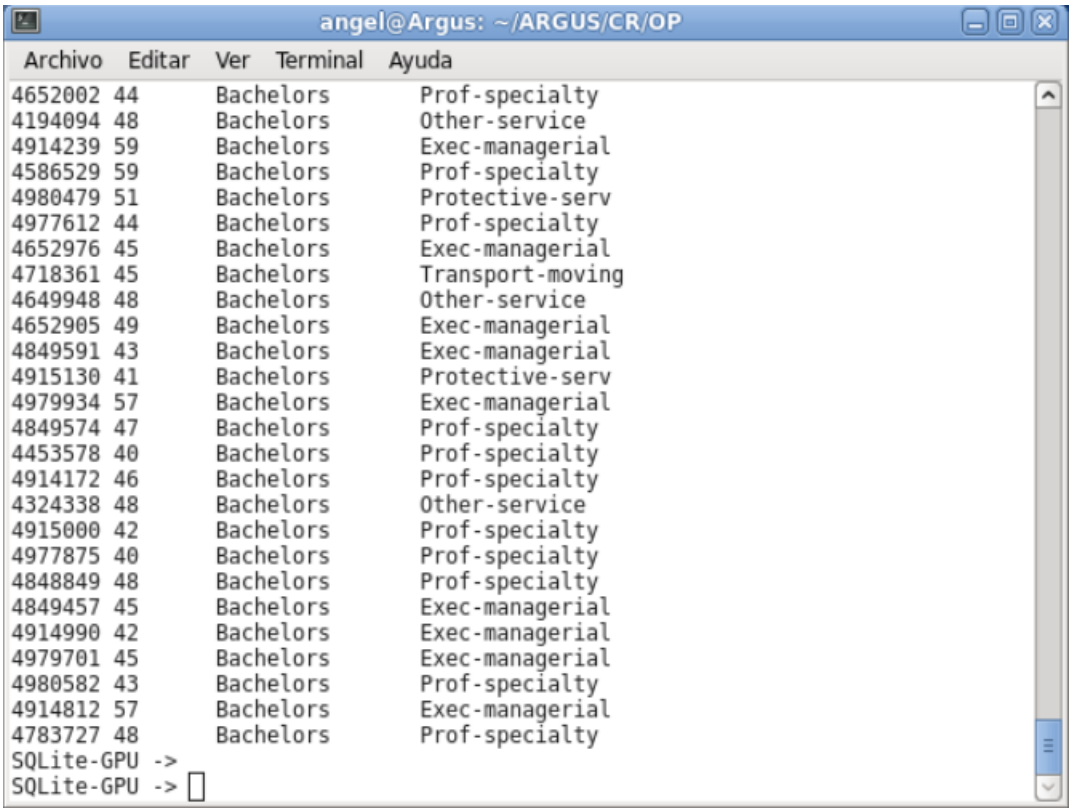


Figura 19. Resultados desplegados por SQLite-GPU

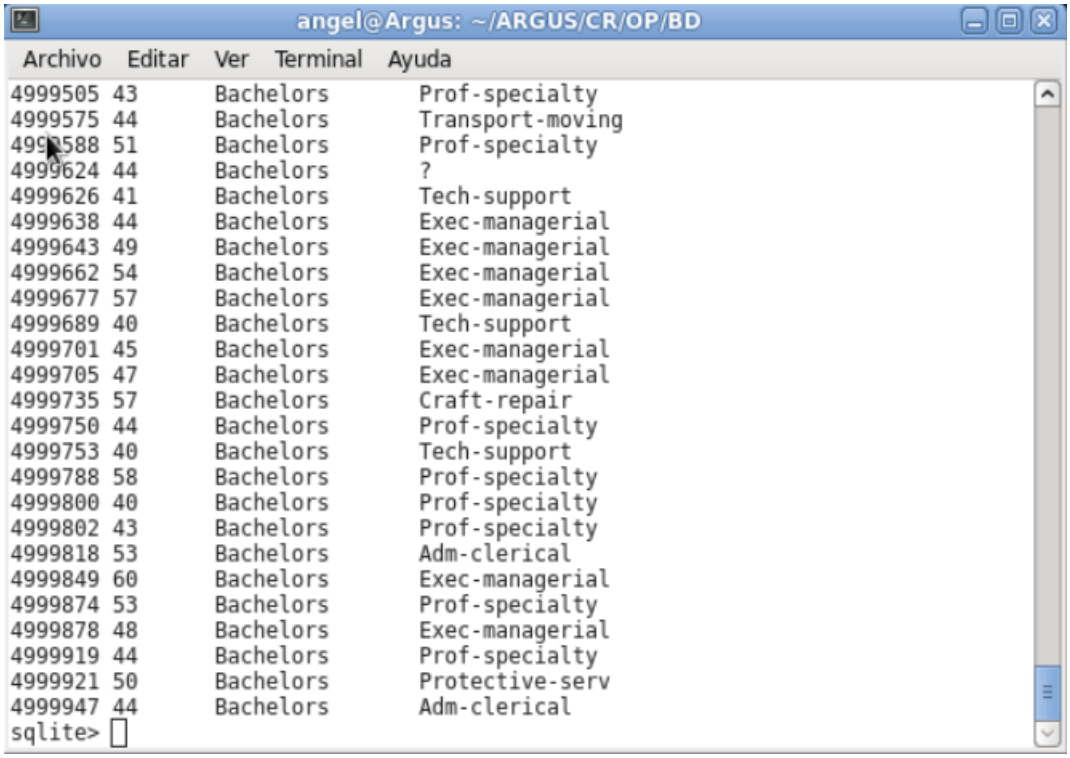


Figura 20. Resultados desplegados por SQLite original.

En ambas figuras (46 y 47) podemos apreciar 4 columnas, las cuales corresponden a los campos id, age, education y occupation respectivamente.

Notamos que efectivamente todos los renglones cumplen con los filtros especificados en la consulta SQL que probamos, ya que la columna que pertenece a age (la segunda) tiene valores de "entre 20 y 60", la columna que pertenece a education (la tercera) siempre tiene valor igual a "Bachelors", y la columna que pertenece a occupation (la última) sus valores siempre son distintos de "Sales". Todos los renglones cumplen los criterios de búsqueda solicitados.

Sin embargo, la única diferencia entre una y otra imagen es el orden en que aparecen los renglones, esto podemos apreciarlo si miramos la primer columna, la cual pertenece al campo id, pero ello tiene una explicación simple. En la terminal correspondiente al manejador SQLite original, los renglones aparecen ordenados de forma ascendente por el campo id, esto se debe a que al realizar una ejecución secuencial solamente una tupla puede ser evaluada a la vez, por lo que los resultados se imprimen conforme se va avanzando dentro de las tablas, y así el orden en que se recorren las tablas marca el orden en que se imprimen los resultados.

Por otro lado, nuestro motor de búsqueda en paralelo, al ser atendido por una gran cantidad de unidades de trabajo al mismo tiempo, cada hilo evalúa sus tuplas asignadas sin importarle el estado avance de los demás, por lo que cada hilo devuelve resultados conforme los obtiene de forma individual, lo que ocasiona que no haya un orden a la hora de entregar los renglones resultado, ya que es imposible determinar que hilo terminará primero y/o que hilo terminará después.

Finalmente, los resultados son los mismos únicamente cambia el orden en que estos son presentados. Pensando en un trabajo a futuro, la operación ORDER BY podría ayudar a omitir la diferencia en cuanto al orden de los resultados.

Claramente aún faltan operaciones SQL por paralelizar en nuestro sistema. Cuando un usuario incluye una consulta con un operador ORDER BY o GROUP BY aparecen automáticamente OpCodes de SQLite dentro del plan de ejecución que no son identificados por nuestro motor de búsqueda. En estos casos, nuestro sistema al ubicar un OpCode no reconocido, simplemente advierte al usuario que dicha consulta por el momento no podrá ser atendida o procesada dentro de la GPU, pero le ofrece la opción de resolverla usando el manejador SQLite de forma normal y transparente desde el CPU.

Capítulo 5 - Pruebas y Resultados

En este capítulo se detallan las pruebas realizadas, así como su respectivo análisis y evaluación para cada tipo de prueba.

5.1 – Información General de Pruebas

Las pruebas se realizaron no solo sobre una tarjeta GPU, sino que se utilizó un equipo que cuenta con dos tarjetas GPU dentro de una misma computadora. Es decir, aprovechamos la tecnología SLI (Scalable Link Interface), la cual nos permite aumentar aún más el rendimiento del equipo mediante la conexión de dos o más tarjetas de video que producen una sola señal de salida [34]. El modelo de tarjetas que se utilizaron fue el GeForce GTX 580.

Las pruebas que se realizaron, básicamente consisten en ejecutar varias consultas SQL tanto en el manejador SQLite que se ejecuta en el CPU, como en el sistema que se desarrolló para esta tesis, el cual corre dentro de la GPU.

El factor a evaluar no radica en comparar quien es más rápido, si el CPU o la GPU, sabemos de antemano que la GPU al contar con los datos en VRAM, ya tiene una ventaja considerable y por lo tanto sería una comparación injusta.

Lo que se va a evaluar es la funcionalidad de nuestra aplicación y el número de veces que se acelera el tiempo de respuesta para una consulta en la GPU

Los tiempos que se están tomando en cuenta, son los mismos que se utilizaron en [23], en dicho trabajo publican la siguiente fórmula para medir el tiempo que gasta la GPU para resolver una consulta SQL solicitada.

$$T_{overall} = T_{mm\_dm}(I) + T_{GPU} + T_{dm\_mm}(O)$$

El costo de tiempo total ($T_{overall}$) es la suma de los siguientes componentes:

$T_{mm\_dm}(I)$: Es el tiempo para copiar los datos de entrada desde la memoria principal a la memoria del dispositivo (GPU). (I) denota el conjunto de datos de entrada.

$T_{GPU:}$ Es el tiempo para evaluar la consulta, dados los datos de entrada ya en el dispositivo. Los datos de salida son almacenados en la memoria de video.

$T_{dm\_mm}(O)$: Es el tiempo de copiar los resultados de salida desde la memoria de video hacia la memoria principal. (O) denota el conjunto de datos de salida.

Como se puede apreciar el tiempo a considerar va desde el momento en que se envían los datos de entrada hasta el momento en que se devuelven los resultados, por ahora no se toma en cuenta el tiempo requerido para desplegar los resultados en pantalla.

Para medir el tiempo del CPU, se realizó también un pequeño programa, que lo único que hace es conectarse con el manejador SQLite, lanzar la consulta SQL requerida y cronometrar el tiempo que tarda en responder pero sin imprimir los resultados en pantalla.

Las consultas que se ejecutaron fueron agrupadas en base al número de tablas que ocupan. En los siguientes apartados se muestran y se describen consultas hechas sobre una, dos, tres y hasta cuatros tablas. También se realizó una evaluación para consultas sobre tablas sin índices. Finalmente se evalúan los resultados obtenidos para consultas que utilizan operaciones de agregación (MAX, MIN, AVG, SUM, COUNT). Los tiempos que se muestran son el resultado de haber ejecutado 5 veces cada consulta, y haber promediado los tiempos obtenidos.

Información de los Datos y Equipo Utilizados

El hardware utilizado para las pruebas realizadas fue una computadora con procesador Intel Core i7 a 3.33 GHz con 6 GB en RAM. Aunque realmente, lo más importante para nuestro sistema es la GPU, ya que es la tarjeta de video quien efectúa el proceso de resolución de las consultas, de hecho, tal como se explicó anteriormente, el tiempo que se está midiendo va desde el momento en que se envían los datos a la GPU hasta el momento en que la misma devuelve los resultados obtenidos. Utilizamos dos tarjetas de video, el modelo de ambas tarjetas es GeForce GTX 580, es decir, aprovechamos la tecnología SLI (Scalable Link Interface), la cual nos permite aumentar aún más el rendimiento del equipo mediante la conexión de dos o más tarjetas de video que producen una sola señal de salida [34].

Los datos utilizados para las pruebas fueron descargados de la página www.kdnuggets.com, son datos de prueba dentro de una sola tabla. Esta tabla fue fragmentada en 4 sub-tablas y unidas simplemente por un campo identificador (id) que funciona como campo llave, esto con la finalidad de poder ejecutar consultas que involucren más de una tabla. La cantidad de registros o tuplas existentes en la página mencionada es de 32,561, pero fueron multiplicados para tener un número mayor, así, una vez repetidos logramos tener hasta diez millones de registros.

La Figura 48 muestra el esquema del resultado de dividir la tabla de datos en cuatro tablas más pequeñas, se muestran los nombres de las tablas resultantes (t1, t2, t3 y t4), sus campos y tipo de dato para cada uno de ellos.

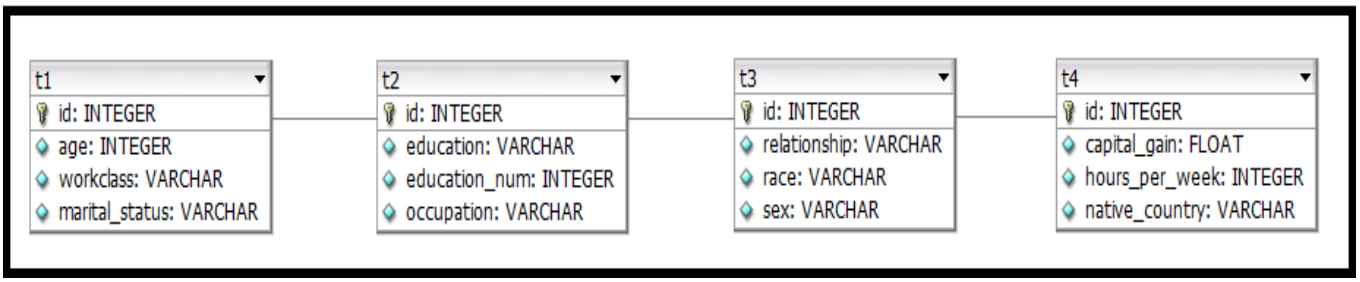


Figura 1. Tabla de datos dividida en las 4 sub-tablas utilizadas.

Los valores que toman cada uno de los campos se muestran en la Tabla 15.

Tabla 1. Valores tomados por cada campo de las tablas.

Campo	Valores
age	17 – 90
workclass	Federal-gov, Local-gov, Never-worked, Private, Self-emp, State-gov, Without-pay, ?
marital_status	Divorced, Married, Never-Married, Separated, Widowed
education	10th, 11th, 12th, 1st-4th, 5th-6th, 7th-8th, 9th, Assoc-acdm, Assoc-voc, Bachelors, Doctorate, HS-grad, Masters, Preschool, Prof-school, Some-college
education_num	1 – 16
occupation	Adm-clerical, Armed-Forces, Craft-repair, Exec-managerial, Farming-fishing, Handlers-cleaners, Machine-op-inspct, Other-service, Priv-house-serv, Prof-specialty, Protective-serv, Sales, Tech-suport, Transport-moving, ?
relationship	Husband, Not-in-family, Other-relative, Own-child, Unmarried, Wife
race	Asian, Black, Indian, Other, White
sex	Female, Male
capital_gain	0.0 – 99999.9
hours_per_week	1 – 99
native_country	Cambodia, Canada, China, Columbia, Cuba, Dominican-Republic, Ecuador, El-Salvador, England, France, Germany, Greece, Guatemala, Haiti, Holans, Honduras, Hungary, India, Iran, Ireland, Italy, Jamaica, Japan, Laos, Mexico, Nicaragua, Outlying-US, Peru. Philippines, Poland, Portugal, Puerto-Rico, Scotland, SouthAfrica, Taiwan, Thailand, Trinidad&Tobago, United-States, Vietnam, Yugoslavia

5.2 – Consultas sobre Una Tabla

Las primeras consultas que se analizaron fueron hechas utilizando una sola tabla. Se utilizó la tabla t1 del esquema presentado en la figura 48 con 10,000,000 de registros, y campos tipo integer y varchar.

Tabla 2. Consultas sobre una Tabla

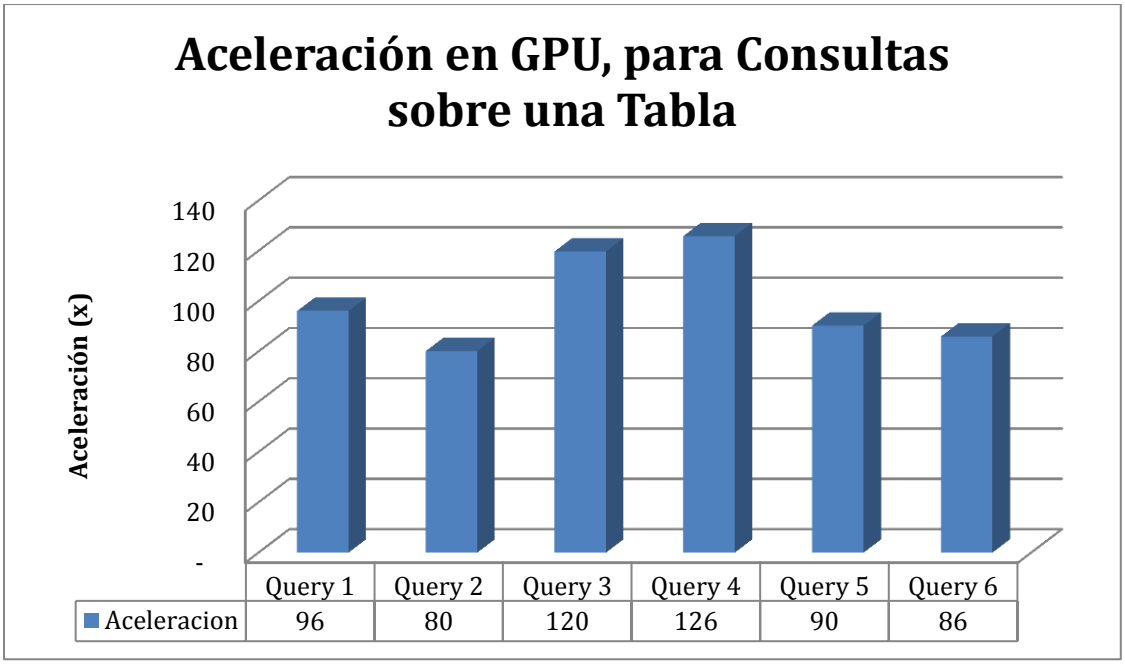
#	Consulta
Q1	SELECT * FROM t1 WHERE age = 40;
Q2	SELECT * FROM t1 WHERE age BETWEEN 20 AND 60;
Q3	SELECT * FROM t1 WHERE workclass LIKE "Private";
Q4	SELECT * FROM t1 WHERE workclass LIKE "Private" AND marital_status NOT LIKE "Married";
Q5	SELECT id, age FROM t1 WHERE workclass LIKE "Private" AND marital_status NOT LIKE "Married" AND age > 60;
Q6	SELECT id, age FROM t1 WHERE workclass LIKE "Private" OR marital_status NOT LIKE "Married" OR age > 60;

Las consultas realizadas se pueden ver en la Tabla 16. Se muestran 6 consultas:

La primer y segunda consultas realizan filtros mediante la cláusula WHERE sobre un campo tipo integer. Una evalúa que el valor del campo sea idéntico a un número dado (en este caso 40), y la otra evalúa que el valor se encuentre dentro de un rango de números (específicamente entre 20 y 60), estas pruebas se hicieron así para analizar el desempeño de búsqueda sobre un campo de tipo numérico y comparar con las consultas Q3 y Q4 que se hacen sobre campos de tipo cadena o varchar.

Como bien ya se dijo, las consultas Q3 y Q4 son la contraparte de las anteriores, pues estas realizan filtros en la cláusula WHERE pero ahora utilizando campos de tipo cadena. La consulta Q3 tiene solo un operador LIKE y busca solo valores idénticos a los solicitados. La consulta Q4 tiene dos operadores LIKE, el primero busca valores idénticos y el segundo busca valores distintos a los que se le indican.

Finalmente, las consultas Q5 y Q6 son similares, pues realizan proyecciones en el SELECT, tienen filtros numéricos y de cadena, pero la diferencia entre una y otra son los conectores lógicos que utilizan, en la primera se usan solo operadores AND y en la segunda operadores OR, naturalmente como es de suponerse, la consulta que contiene operadores OR tendrá muchos más resultados que la otra.



Gráfica 1. Aceleración de Consultas sobre una Tabla

Los resultados en cuanto a aceleración de tiempos de respuesta se muestran en la Gráfica 1 y su correspondiente tabla de resultados se muestra continuación en Tabla 17. Los campos que contiene la tabla de resultados, son los tiempos de CPU y GPU para cada consulta, el número de veces que se acelera, y finalmente un campo que es de mucha utilidad para hacer el análisis, nos referimos al # de tuplas devueltas. Veremos la razón.

Tabla 3. Resultados para Consultas sobre una Tabla

Consulta #	Tiempo CPU (seg.)	Tiempo GPU (seg.)	Aceleración (x)	# Tuplas devueltas
Q1	3.7974	.03953	96	243851
Q2	8.6081	.10741	80	8774918
Q3	11.0722	.09245	120	6970267
Q4	12.5417	.09965	126	3884060
Q5	10.9893	.12174	90	152019
Q6	10.9588	.12752	86	8607224

En las primeras dos consultas, que como sabemos realizan evaluaciones sobre un campo numérico, tenemos aceleraciones de 96x y 80x respectivamente. Si miramos el tiempo que gasta la GPU, podemos apreciar que la segunda consulta tardó mucho más tiempo que la primera, pero la razón de esos tiempos pueden ser resueltos si miramos el campo de # de Tuplas devueltas. Como podemos ver a mayor número de tuplas devueltas mayor es el tiempo de respuesta de la GPU, esto se debe a dos factores:

Primero, que cuando hay muchas tuplas resultado, hay también una gran cantidad de hilos queriendo escribir sus resultados al mismo tiempo, y esto crea un efecto de embudo o cuello de botella, ya que los hilos deben formarse y esperar su turno para poder escribir sus resultados en memoria, uno a la vez para que no haya sobre escritura y no se alteren los datos previamente guardados.

Segundo y más importante, es que implica más tiempo regresar datos desde VRAM hasta memoria principal cuando hay más información que transmitir, pues naturalmente la cantidad de información es proporcional al número de tuplas devueltas.

Para los resultados de las consultas Q3 y Q4, sucede algo curioso. Vemos que Q4 entrega aproximadamente el doble de registros que Q3 y sin embargo, el tiempo de GPU y los números de aceleraciones son muy similares. Esto se debe a que las evaluaciones sobre campos de tipo cadena, son más tardadas que evaluaciones sobre datos numéricos, por lo tanto, el tiempo de transmisión de resultados no se ve impactado en el número de tuplas devueltas cuando se utiliza el operador LIKE, puesto que el tiempo de procesamiento domina al tiempo de transmisión.

Finalmente, los resultados de Q5 y Q6 evidencian de forma más drástica lo que se comentó en el párrafo anterior, vemos que Q5 entrega un número mucho, pero mucho menor de tuplas que las que devuelve Q6, y sin embargo, la cantidad de aceleraciones no varía demasiado. Esto se debe nuevamente a que se hacen evaluaciones sobre campos varchar, por lo que el número de tuplas devueltas tiene ciertamente un ligero impacto pero aun así el tiempo total es dominado por el tiempo de procesamiento.

Desde aquí podemos empezar a concluir que el tiempo de procesamiento siempre es el más dominante, aunque es mucho más notorio cuando se aparece el operador LIKE en la consulta, ya que las evaluaciones de cadena, implican una comparación de caracter por caracter, a diferencia de cuando solo se usan campos numéricos, donde la comparación es directa. Cuando solo se usan campos numéricos el tiempo de transmisión si tiene una presencia más evidente en el tiempo total dependiendo la cantidad de registros que se tienen que enviar.

En comparación con [25], dicho trabajo no sería capaz de resolver nuestras consultas desde el momento en que no da soporte para trabajar con datos de cadenas de caracteres.

5.2 – Consultas sobre Dos Tablas Indexadas

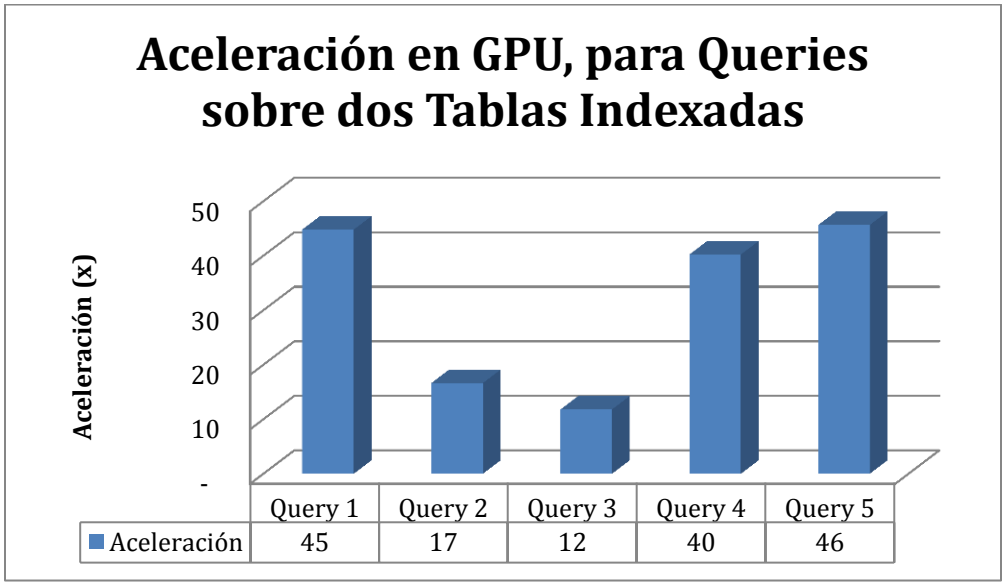
Para poder utilizar el operador NATURAL JOIN, se trabajó con dos tablas (concretamente las tablas t1 y t2 del esquema presentado en la Figura 48) de 5,000,000 de tuplas cada una, unidas mediante un campo en común llamado id.

Las consultas que sirvieron de modelo se muestran en la siguiente Tabla 18:

Tabla 4. Consultas sobre dos Tablas indexadas

#	Consulta
Q1	SELECT * FROM t1 NATURAL JOIN t2;
Q2	SELECT * FROM t1 NATURAL JOIN t2 WHERE age BETWEEN 20 AND 30 AND education_num > 10;
Q3	SELECT id, age, occupation FROM t1 NATURAL JOIN t2 WHERE workclass LIKE "Federal-gov" AND occupation LIKE "Armed-Forces";
Q4	SELECT id, age, occupation FROM t1 NATURAL JOIN t2 WHERE workclass NOT LIKE "Federal-gov" AND occupation NOT LIKE "Exec-managerial";
Q5	SELECT id, age, workclass, education FROM t1 NATURAL JOIN t2 WHERE workclass LIKE "Private" AND occupation LIKE "Sales" OR education LIKE "Masters" AND marital_status LIKE "Divorced";

En la siguiente Gráfica 2 se muestran el número de aceleraciones que se obtuvieron al ejecutar las consultas tanto en CPU como en GPU, ahora podemos apreciar que hay una mayor diferencia en aceleraciones para una y otra consulta.



Gráfica 2. Aceleración para Consultas sobre dos Tablas Indexadas

Tabla 5. Resultados para Consultas sobre dos Tablas indexadas

Consulta #	Tiempo CPU (seg.)	Tiempo GPU (seg.)	Aceleración (x)	# Tuplas devueltas
Q1	15.8899	.3547	45	5000000
Q2	5.4504	.3276	17	391291
Q3	3.5137	.2980	12	1382
Q4	17.0006	.4227	40	4255878
Q5	17.4622	.3826	46	485761

Ya sabemos cómo funciona el operador NATURAL JOIN sobre tablas indexadas, entonces podemos pasar a analizar cada resultado obtenido.

En primer lugar Q1 es la consulta base, simplemente actúa el operador NATURAL JOIN sin realizar una actividad extra.

En el caso para Q2, además del NATURAL JOIN, se realizan dos filtros en WHERE pero ambos son sobre campos numéricos, mientras que para Q3, se realizan también dos filtros pero sobre campos de tipo varchar. El número de tuplas resultado es un factor que influye, se escogieron intencionalmente consultas que trajeran pocos resultados, podemos ver que Q2 devuelve prácticamente 30 veces más el número de resultados de Q3, por lo que el tiempo de recepción se ve reflejado en el tiempo GPU, y el número de aceleraciones para una y otra es similar.

Para los casos de Q4 y Q5, ya podemos ver proyecciones en las consultas y vemos también filtros sobre campos pertenecientes a las dos tablas involucradas. En estos casos podemos observar que tanto CPU como GPU tardan más en evaluar estas consultas que en evaluar la consulta base o Q1, esto se debe a como se había advertido anteriormente a que se hacen filtros sobre campos de tipo varchar.

Si comparamos las consultas de este apartado podemos notar un gran incremento en el tiempo de GPU con respecto a las consultas del apartado anterior. El aumento en el tiempo se debe a dos factores relevantes:

El primero es que, efectivamente el operador NATURAL JOIN realiza una actividad más desgastante pues las consultas involucran al menos dos tablas, a diferencia de cuando se ejecutan consultas sobre una sola tabla.

Y el segundo factor, es que para este apartado si se toma en cuenta el tiempo de enviar los datos de memoria principal a memoria de video. Este tiempo es el mismo para todas las consultas, y resulta ser dominante para aquellas consultas que entregan pocos resultados.

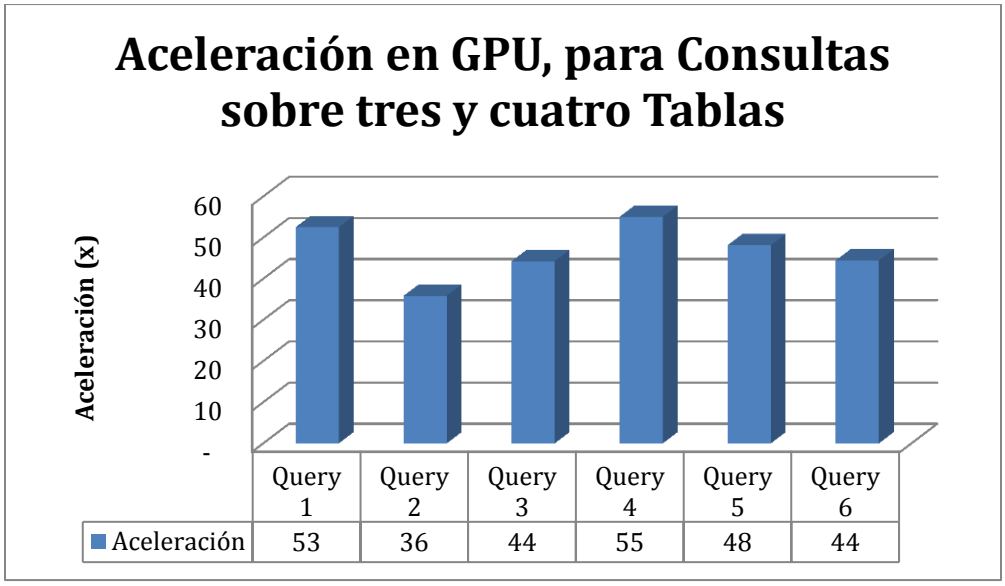
5.4 – Consultas sobre Tres y Cuatro Tablas

Ahora mostramos consultas que fueron utilizadas para poner a prueba nuestro sistema y mostrar que no solo se pueden atender consultas sobre 2 tablas, sino que estamos preparados para consultas multitabla. Por cuestiones de espacio de VRAM, vamos a probar consultas sobre 3 tablas donde cada tablas tiene 3,300,000 de tuplas, y también consultas sobre 4 tablas donde cada una de ellas contiene 2,400,000 de tuplas. Las consultas ejecutadas son las siguientes (Tabla 20) (se utilizaron todas las tablas mostradas en el esquema de la Figura 48):

Tabla 6. Consultas sobre tres y cuatro Tablas

#	Consulta
Q1	SELECT * FROM t1 NATURAL JOIN t2 NATURAL JOIN t3;
Q2	SELECT t1.id, workclass, race, sex FROM t1 NATURAL JOIN t2 NATURAL JOIN t3 WHERE workclass LIKE "Private" AND race LIKE "White" AND sex LIKE "Male";
Q3	SELECT t1.id, age, education, race, sex FROM t1 NATURAL JOIN t2 NATURAL JOIN t3 where age > 40 or education LIKE "HS-grad" AND race NOT LIKE "Asian" AND sex LIKE "Male";
Q4	SELECT * FROM t1 NATURAL JOIN t2 NATURAL JOIN t3 NATURAL JOIN t4;
Q5	SELECT t1.id, age, race FROM t1 NATURAL JOIN t2 NATURAL JOIN t3 NATURAL JOIN t4 WHERE native_country NOT LIKE "United-States" AND capital_gain BETWEEN 8000 AND 16000 OR education LIKE "Some-college" OR occupation NOT LIKE "Exec-managerial";
Q6	SELECT t1.id, education, capital_gain FROM t1 NATURAL JOIN t2 NATURAL JOIN t3 NATURAL JOIN t4 WHERE hours_per_week >= 40 AND capital_gain > 10000 OR race NOT LIKE "White";

Nuevamente mostramos, en gráficas los resultados obtenidos para las consultas señaladas y de la misma forma mostramos la tabla de resultados (Gráfica 3 y Tabla 21). Solamente resta comentar que Q1, Q2 y Q3 son consultas que involucran 3 tablas, mientras que Q4, Q5 y Q6 utilizan 4 tablas.



Gráfica 3. Aceleración para Consultas sobre tres y cuatro Tablas

Tabla 7. Resultados para Consultas sobre tres y cuatro Tablas

Consulta #	Tiempo CPU (seg.)	Tiempo GPU (seg.)	Aceleración (x)	# Tuplas devueltas
Q1	17.6699	.3363	53	3300000
Q2	13.9617	.3894	36	1329991
Q3	16.4364	.3713	44	1777493
Q4	18.1485	.3296	55	2400000
Q5	18.521	.3845	48	2165890
Q6	15.413	.3466	44	394458

El análisis para este apartado es similar al del anterior. Las consultas Q1 y Q4, podemos llamarlas como consultas base para 3 y 4 tablas respectivamente, ya que no incluyen alguna otra operación más allá de los operadores NATURAL JOIN para dichos números de tablas.

Para las consultas Q2, Q3, Q4 y Q5, ya podemos ver proyecciones y filtros sobre los campos pertenecientes a las tablas involucradas. En todas las consultas de este apartado, podemos ver que el número de aceleraciones son muy similares y también lo son así los tiempos de GPU, sin importar demasiado el número de tuplas devueltas. Esto significa que en una búsqueda sobre tres o cuatro tablas, el tiempo de procesamiento es ampliamente dominante.

Estas consultas no pueden ser resueltas por otros trabajos, ya que ninguno de los trabajos semejantes al de esta tesis, ha demostrado poder resolver consultas multitabla. Por el momento esto podemos verlo como una característica a favor de nuestro trabajo, y además el número de aceleraciones sobrepasa las 40x en la mayoría de los casos, por lo que tenemos realmente buenos resultados.

5.5 – Consultas sobre Dos Tablas sin Índices

Ahora, también se realizaron pruebas al operador NATURAL JOIN usando tablas que no tienen índices. Existen dos formas de trabajar un NATURAL JOIN sobre dos tablas sin índices, la primera es comparar tupla a tupla de cada tabla, y la segunda y más eficiente, es primero ordenarlas tablas antes de comenzar la búsqueda.

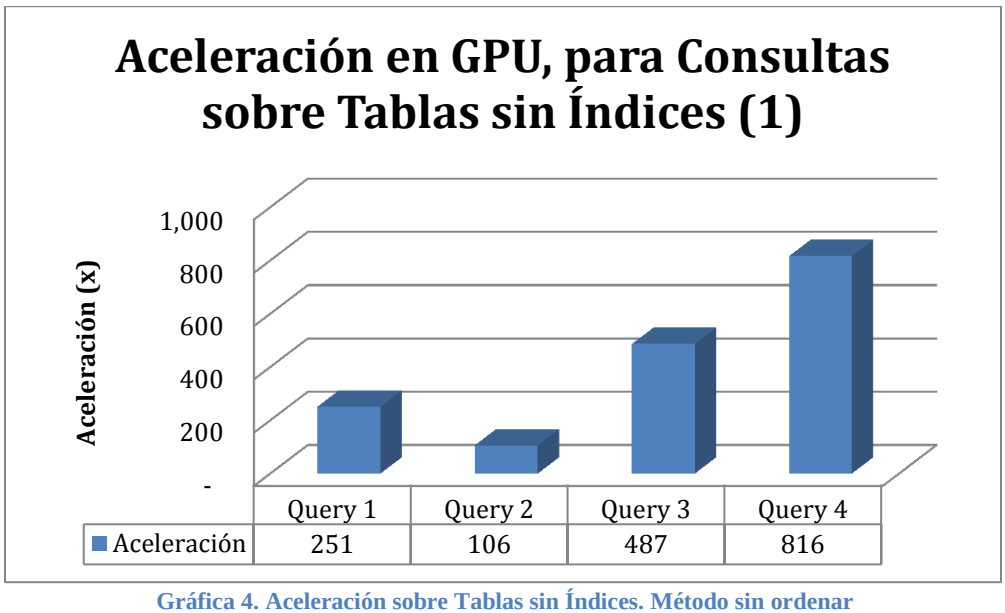
La siguiente Tabla 22 muestra las consultas que se ejecutaron para ambos métodos.

Tabla 8. Consultas sobre Tablas sin índices. Método sin ordenar

#	Consulta
Q1	SELECT * FROM m1 NATURAL JOIN m2;
Q2	SELECT id, age, occupation FROM m1 NATURAL JOIN m2 WHERE workclass LIKE "Federal-gov" AND occupation LIKE "Armed-Forces";
Q3	SELECT id, age, occupation FROM m1 NATURAL JOIN m2 WHERE workclass NOT LIKE "Federal-gov" AND occupation NOT LIKE "Exec-managerial";
Q4	SELECT id, age, workclass, education FROM m1 NATURAL JOIN m2 WHERE workclass LIKE "Private" AND occupation LIKE "Sales" OR education LIKE "Masters" AND marital_status LIKE "Divorced";

Primero se muestran los resultados que se obtuvieron mediante el primer método (sin ordenar).

Realmente este método se encuentra en desuso por los manejadores de bases de datos modernos, y por esto nos vimos en la necesidad recurrir a una versión anterior de SQLite, para poder obtener los resultados que se muestran en la siguiente Grafica 4 y su respectiva Tabla 23.



Gráfica 4. Aceleración sobre Tablas sin Índices. Método sin ordenar

Tabla 9. Resultados para Consultas sobre Tablas sin índices. Método sin ordenar.

Consulta #	Tiempo CPU (seg.)	Tiempo GPU (seg.)	Aceleración (x)	# Tuplas devueltas
Q1	217.44	.8651	251	32561
Q2	91.2	.8584	106	9
Q3	421.95	.8665	486	27715
Q4	704.31	.8626	816	3163

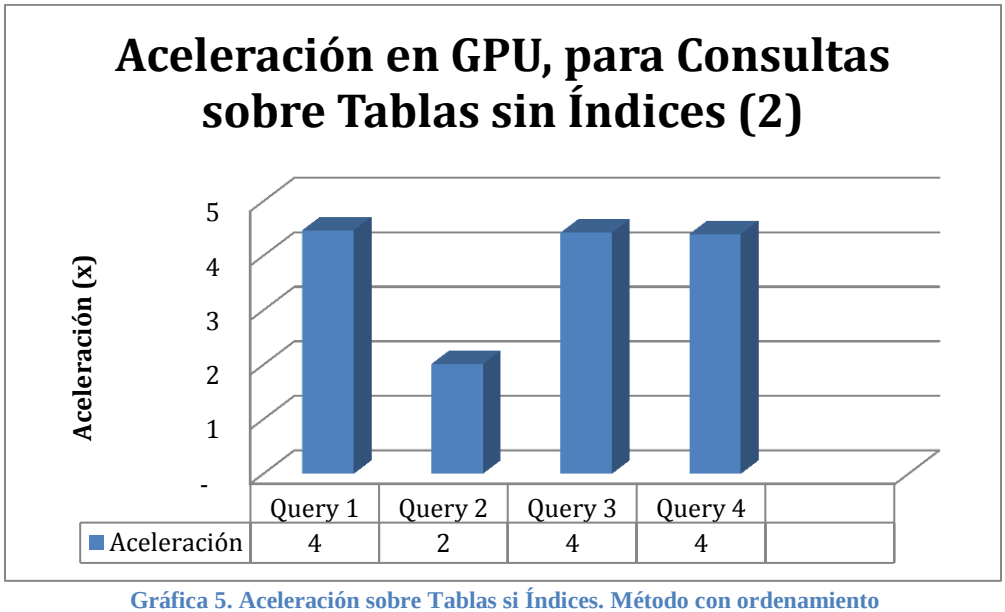
En realidad, el manejador SQLite en una versión anterior que sí ejecuta el operador NATURAL JOIN sin recurrir a una ordenación previa, tarda un tiempo exageradamente alto para resolver las consultas, por lo que se optó por recortar el número de registros y dejar que trabajara solamente con los datos originales de las tablas que son 32,561 tuplas.

Los número de aceleraciones obtenidos parecieran demasiado elevados, sin embargo, debemos tomar que el manejador SQLite tiene que hacer en total (32,561) * (32,561) comparaciones y esto le demoraba varios minutos.

Por otro lado, el GPU al repartir el mismo número de operaciones entre los hilos que ejecuta paralelamente, el trabajo es mucho menor para cada hilo, pues se lanzaron 65536 hilos de trabajo y entonces el número de comparaciones es mucho menor al que realiza CPU de forma secuencial.

De cualquier forma se admite que este es un método prácticamente obsoleto por los manejadores modernos.

La siguiente Gráfica 5 con su respectiva Tabla 24, nos muestran los resultados de haber ejecutado las mismas consultas sobre tablas sin índices, pero esta vez utilizando la versión más reciente del manejador SQLite, la cual ordena las tablas antes de iniciar cualquier operación.



Gráfica 5. Aceleración sobre Tablas si Índices. Método con ordenamiento

Tabla 10. Consultas sobre Tablas sin índices. Método con ordenamiento

Consulta #	Tiempo CPU (seg.)	Tiempo GPU (seg.)	Aceleración (x)	# Tuplas devueltas
Q1	26.9149	6.0249	4	5000000
Q2	12.3427	6.1353	2	1382
Q3	28.9246	6.5399	4	4255878
Q4	27.9862	6.3648	4	485761

Este es sin duda el punto débil de nuestra aplicación, pues tuvimos que comparar un algoritmo que ordena las tablas antes de procesar la búsqueda contra nuestro algoritmo que ejecuta un método que realiza todas las comparaciones sin ninguna ordenación previa.

Aunque se obtienen aceleraciones de entre 2x y 4x, esta forma de trabajar no es óptima, dado que aún no hemos implementado una función para ordenar. Quizás ese debería de ser uno de los primero pasos para un trabajo a futuro, ya que si se lograran ordenar los datos, podríamos obtener aceleraciones semejantes a las que se obtuvieron en el apartado 5.3. Definitivamente la estrategia utilizada, aunque funciona, no fue la mejor para este caso particular.

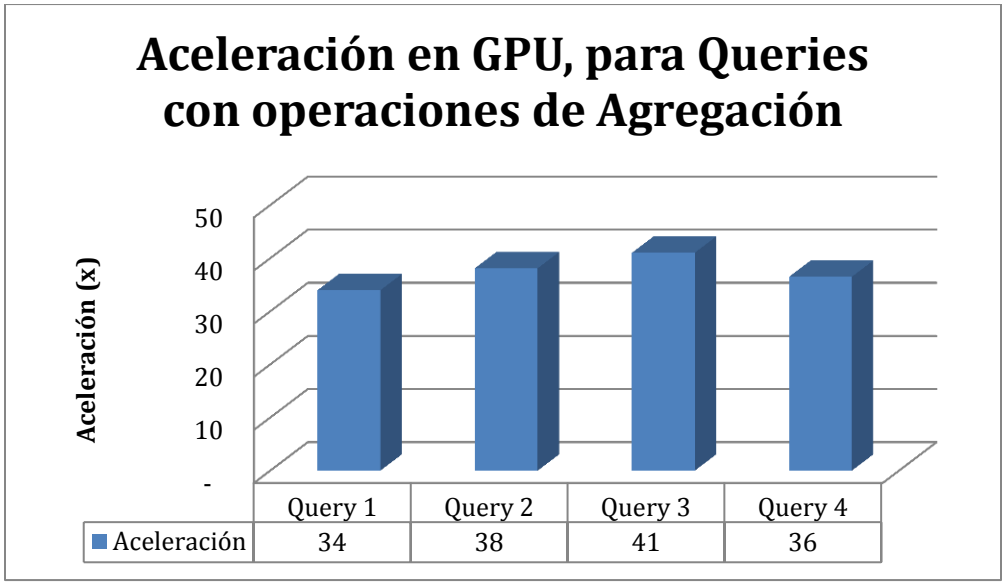
5.6 – Consultas con Operaciones de Agregación

Finalmente, se muestran consultas sobre dos tablas indexadas que incluyen funciones de agregación (MAX, MIN, AVG, COUNT, SUM), la Tabla 25 nos indica cuales fueron:

Tabla 11. Consultas con operaciones de agregación

#	Consulta
Q1	SELECT max(age) FROM t1 WHERE workclass LIKE "Private" AND marital_status NOT LIKE "Married";
Q2	SELECT min(education_num) FROM t1 NATURAL JOIN t2 WHERE workclass LIKE "Federal-gov" OR occupation LIKE "Exec-managerial";
Q3	SELECT count() FROM t1 NATURAL JOIN t2 WHERE workclass LIKE "Private" AND occupation LIKE "Sales" OR education LIKE "Masters" AND marital_status LIKE "Divorced";
Q4	SELECT avg(age) FROM t1 NATURAL JOIN t2 WHERE workclass LIKE "Private" AND occupation LIKE "Sales" AND education LIKE "Masters" AND marital_status LIKE "Divorced";

Para este tipo de pruebas, nos comparamos nuevamente con [25], y podemos ver la aceleración en la Gráfica 6 con su Tabla 26.



Gráfica 6. Aceleración para Consultas con operaciones de agregación

Tabla 12. Resultados para Consultas con operaciones de agregación

Consulta #	Tiempo CPU (seg.)	Tiempo GPU (seg.)	Aceleración (x)	# Tuplas devueltas
Q1	2.0054	.0590	34	1
Q2	5.2284	.1372	38	1
Q3	6.5081	.1586	41	1
Q4	2.2852	.0626	36	1

En este tipo de pruebas, el tiempo total de la ejecución es prácticamente 100% del tiempo de procesamiento, pues como claramente vemos, solo devuelven una tupla resultado, por lo que el tiempo de transmisión de resultados se vuelve despreciable.

Este tipo de consultas no podrían ser resueltas por ninguno de los trabajos relacionados, ya que en el caso de [25] no es capaz de resolver consultas sobre 2 o más tablas, y por otro lado, los trabajos [23] y [24] no son aptos para atender operaciones de agregación. Por lo que nuestro trabajo toma delantera con esta funcionalidad.

5.7 - Comparación de funcionalidad con otros trabajos relacionados

Si bien es cierto que nuestro sistema obtiene mayor número de aceleraciones frente a otros trabajos relacionados, también es verdad que dicha comparación es un tanto injusta si tomamos en cuenta que ningún trabajo fue probado bajo las mismas condiciones de hardware ni sobre los mismos datos. Aun así, los resultados obtenidos hasta el momento son esperanzadores, pues si pensamos en un caso hipotético donde aun con nuestro hardware se obtuvieran resultados inferiores, iguales o muy ligeramente superiores a los otros trabajos, entonces eso sí sería un grave indicador que algo se está haciendo mal, afortunadamente, este no es el caso.

Sin embargo algo que si podemos comparar de manera justa, es la funcionalidad de nuestro sistema frente a trabajos previos. Hasta el momento ningún otro trabajo se encuentra en condiciones de resolver todas las consultas SQL que puede resolver el presente trabajo, ya que la cobertura de funciones SQL por parte de nuestro sistema es más amplio. La Tabla 27 muestra las características funcionales soportadas tanto por nuestro sistema como por otros similares, y más abajo se ofrece un análisis de dichas características.

Tabla 13. Comparación de Características Funcionales

Característica Funcional	Presente Trabajo	BAKKUM, 2010 [25]	YANG, 2008 [23]	YANG, 2009 [24]
Paraleliza el plan de ejecución	X	X	-	-
Soporta datos numéricos	X	X	X	X
Soporta cadenas de caracteres	X	-	X	X
Soporta índices en tablas de datos	X	-	X	X
Realiza proyecciones (SELECT)	X	X	-	X
Realiza filtros (WHERE)	X	X	-	X
Realiza JOIN sobre dos tablas	X	-	X	X
Realiza JOIN sobre más de dos tablas	X	-	-	-
Realiza operaciones de agregación	X	X	-	-
Aprovecha tecnología SLI (2 o más GPU)	X	-	-	-
Realiza ordenación (ORDER BY)	-	-	-	X
Implementa un paginador de memoria	-	-	-	-

La primera característica a favor de nuestro sistema, es que se ha paralelizado un plan de ejecución real, mientras que algunos otros trabajos fabricaron primitivas (funciones ajenas a un motor de bases de datos y adaptadas) para tratar de obtener los mismos resultados que un motor de bases de datos. Quizás el GPU-Computing sea un desarrollo reciente, pero los motores de bases de datos ya llevan mucho tiempo desarrollándose y perfeccionando sus algoritmos de respuesta, por lo que hay fundamentos para pensar que el plan de ejecución de un motor es más eficiente que otros métodos que intentan resolver consultas SQL.

Otra característica es la capacidad de trabajar tanto datos numéricos como cadenas de texto, ya que esto ofrece la posibilidad de trabajar con datos más reales que trabajar solo con números (enteros y flotantes) sin mucho sentido.

Además nuestro sistema soporta trabajar con índices, que como sabemos, son una estructura dentro de las bases de datos que mantiene de alguna forma los datos ordenados, esto con el firme propósito de que las búsquedas se realicen en tiempos más cortos.

Podemos realizar filtros (WHERE) y proyecciones (SELECT), operaciones esenciales en las bases de datos. Pero otra operación también muy importante dentro del modelo relacional de bases de datos es el operador JOIN, ya que sin él, veríamos a cada tabla de datos como si se tratasen de bases de datos individuales y separadas entre sí, por lo tanto se requiere al operador JOIN, para realizar consultas sobre varias tablas y poder realizar consultas más complejas. Existen trabajos que han logrado paralelizar el JOIN de dos tablas, pero nadie hasta el momento ha mostrado ser capaz de resolver consultas con más de dos tablas, y eso es algo que nuestro sistema sí logra hacer.

Nuestro sistema también realiza operaciones de agregación tales como MAX, MIN, SUM, COUNT, AVG, cosa que no todos los demás trabajos anteriores pueden realizar.

Aprovechamos también la tecnología SLI que nos permite trabajar sobre dos GPUs o más como si se tratara de solo una. Ningún otra trabajo utiliza SLI.

Una operación SQL que no hemos implementado aun, y que otro trabajo ya lo hizo, es el ORDER BY. Sin embargo, aunque es una operación útil, no es crítica para tener resultados correctos, ya que a final de cuentas las tuplas obtenidas son las mismas, es decir, la forma en que se presentan es diferente, pero los resultados correctos permanecen ahí, no hay más ni menos resultados de los que debe haber.

Finalmente una característica que no ha sido implementada por absolutamente ningún trabajo (incluyendo el nuestro), es un paginador de memoria. Hasta el momento la capacidad de VRAM es la principal limitante en común para todos los trabajos existentes, pero ese es tal vez un buen tema a desarrollar a futuro que ayudará a trabajar con bases de datos mucho más grandes sin necesidad de preocuparse demasiado por la VRAM.

Hasta aquí terminamos con la comparación de funcionalidad. Salta a la vista que las características funcionales desarrolladas en nuestro sistema superan de forma individual a cada uno de los trabajos anteriores, resaltando como principales contribuciones el manejo de datos numéricos, cadenas de caracteres e índices de tablas. Podemos utilizar NATURAL JOIN no solo para dos tablas sino inclusive más. Trabajamos sobre dos GPU sin ningún problema.

Capítulo 6 - Conclusiones

6.1 - Conclusiones

Las constantes mejoras a las unidades de procesamiento gráfico (GPU), les han convertido en una gran alternativa para realizar tareas que demandan demasiados cálculos para un solo procesador.

El GP-GPU o GPU-Computing es una tecnología relativamente nueva y que tiene un gran futuro, hablando desde un punto de vista muy personal. Una de las áreas del cómputo que ha permanecido un tanto alejada al desarrollo de esta nueva tecnología, son las bases de datos. Existen pocos, realmente pocos trabajos hasta el momento que aborden el paralelismo de bases de datos desde una tarjeta de video. Este aislamiento podría comprenderse hasta cierto punto obvio, si tomamos en cuenta que hablamos de una técnica de reciente creación y quizás también, poco podría esperarse que un hardware especializado en video pudiera ayudar en un tema tan ajeno y distante a simple vista como las bases de datos.

El presente trabajo, demuestra que bases de datos y unidades de procesamiento gráfico pueden coexistir y trabajar de forma más estrecha si se buscan las formas en como puedan adaptarse la una a la otra.

A diferencia de otros trabajos que desarrollan sus propias funciones para resolver consultas, en esta tesis, paralelizamos directamente el plan de ejecución de un motor de bases de datos (SQLite) sin necesidad de recurrir a funciones externas.

Además este trabajo supera en funcionalidad a sus antecesores, pues ahora ya tenemos la posibilidad de operar consultas a bases de datos que involucren 2 o más tablas. Esta es una operación muy importante dentro de la bases de datos relacionales, ya que es la operación que nos permite relacionar dos tablas y formular consultas más complejas. Además podemos trabajar sobre campos de tipo varchar, y también utilizar campos indexados, cosas que tampoco puede realizar el trabajo [25] el cual fue tomado como principal punto de referencia.

Para consultas sobre una sola tabla demostramos tener aceleraciones en promedio de 80x, para consultas sobre dos tablas, tuvimos aceleraciones de hasta 49x. Además, ningún otro trabajo ha demostrado ser capaz de resolver consultas con más de dos tablas.

El punto débil de este trabajo son las relaciones sobre tablas que no tienen índices, pues aunque nuestras pruebas demuestran que tenemos buenos tiempos, también se reconoce que la estrategia tomada para estos casos no es la óptima, nuestros tiempos todavía pueden ser mejorados.

Es claro que existen operaciones o tareas que no convienen ser ejecutadas en una GPU, porque son poco paralelizables o porque su ejecución de forma secuencial en el CPU es difícilmente superable, pero eso está bien, pues la idea principal no es sustituir al procesador central, sino que únicamente restarle trabajo, ayudarle a procesar trabajos que si pueden ser acelerados mediante cómputo paralelo.

Existen otras tecnologías de cómputo paralelo, como por ejemplo los clúster, empero su adquisición resulta económicamente más difícil de conseguir que una tarjeta de video de uso científico, es decir, si bien es verdad que una tarjeta de video altamente veloz es cara, resulta aún más caro comprar todo un conjunto de computadoras para armar un clúster, además su configuración y mantenimiento pueden también resultar más complicados que teniendo una sola máquina con una buena GPU.

Finalmente es importante señalar, que en general llevar un proceso secuencial a una ejecución en paralelo, puede dar como resultado un mejor desempeño en cuanto a velocidad de procesamiento, siempre y cuando se sepa identificar que procesos son paralelizables y diseñar las estrategias adecuadas para tener buenos resultados.

No importa que tan extraño suene la idea de adaptar o compaginar dos campos totalmente separados (al menos a primera vista) como los que se abordan en este trabajo, si se entiende bien el problema que se pretende resolver y se razonan bien las herramientas que se tienen a la mano, junto con sus alcances y límites, se pueden lograr resultados no solo innovadores sino que también alentadores para seguir trabajando en un futuro.

El cómputo paralelo, es un campo que personalmente resulta fascinante en cualquiera de sus variantes, realmente me gustaría ahondar más y más en estos temas, pues considero que tiene un presente muy amplio y en auge, lo cual augura también un futuro prometedor.

6.2 - Contribuciones

La contribución de este trabajo es la realización de un motor de búsqueda de bases de datos que realiza un procesamiento en paralelo sin necesidad de contar con un gran número de equipos interconectados, sino que trabaja en uno solo utilizando solamente su(s) tarjeta(s) de video. Este motor obtiene aceleraciones por más de 50x en comparación con el mismo motor que se ejecuta en el CPU de forma secuencial.

La funcionalidad de nuestro motor paralelo de búsqueda permite atender las siguientes operaciones o características del lenguaje de bases de datos SQL:

- Soporta trabajar con datos tanto numéricos (enteros y flotantes) como con cadenas de caracteres.

- Realiza proyecciones (SELECT), filtros (WHERE) y funciones de agregación (MAX, MIN, AVG, SUM COUNT).

- Atiende consultas sobre una sola tabla (unitabla) y sobre 2 tablas o incluso más (multitabla). Esta última característica es de suma importancia en las bases de datos relacionales.

- Soporta trabajar tablas indexadas, una característica que aligera enormemente el proceso de búsqueda.

- Aprovecha la capacidad de SLI, que nos permite trabajar con dos o más GPU al mismo de forma transparente, incrementando la capacidad de procesamiento.

Todas las características anteriores no son ofrecidas en su totalidad por ningún otro trabajo desarrollado previamente.

Con este motor de búsqueda, se evitan problemas que presentan arquitecturas de cómputo distribuido, tales como hacer una correcta distribución de los datos entre distintos equipos pues ahora la información permanece completa y en un solo lugar. Tampoco hay que preocuparse por los tiempos de comunicación entre los equipos interconectados, ya que estos simplemente no existen.

Trabajos más parecidos al que se presenta en este documento, se encuentran en universidades de Estados Unidos, incluso se admite que este trabajo surgió a partir de la revisión de un artículo que aplicaba la tecnología del GP-GPU a las bases de datos, aunque también es cierto que hasta el momento no se han vuelto a tener noticias de que dicho trabajo continúe en desarrollo o que hayan salido nuevas publicaciones de trabajos similares.

Este trabajo, detalla además un uso eficiente de las características propias de una GPU, tal como es el uso de sus distintos niveles de memoria, ya que este es un factor en ocasiones determinante para tener un buen desempeño.

Si bien es cierto que aún faltan operaciones SQL por desarrollar, los resultados obtenidos hasta el momento, hacen pensar que se está siguiendo un camino correcto en un campo poco explorado aún, por lo tanto, este trabajo contribuye además para alentar y motivar a que siga la investigación y el desarrollo de herramientas paralelas sobre las bases de datos y tecnologías de la información.

En ocasiones es difícil imaginar o idear nuevas oportunidades de desarrollo en campos de la computación como las bases de datos, pero ahora con las GPU surge una nueva alternativa por explorar y que quizás pueda llegar a impactar de rebote en otras áreas como la minería de datos y el análisis de información en textos.

6.3 – Trabajo a Futuro

Hay muchas cosas por realizar aun, y realmente todo apunta y va dirigido hacia crear un manejador de bases de datos completo que corra totalmente sobre una GPU, algo que hasta el momento no existe.

La primera tarea que salta a la vista, sin duda es implementar los operadores SQL que faltan por paralelizar, tales como GROUP BY, ORDER BY y otros. Ya existen trabajos que están desarrollando algoritmos para hacer ordenaciones en paralelo, sería cuestión de analizar y evaluar que tan viable sería llevar dichos métodos hacia el plan de ejecución para bases de datos.

Sin embargo, otra actividad que desde un punto de vista personal, tendría un impacto mucho mayor, sería la realización de un paginador de memoria, es decir, un módulo dentro del sistema que pueda agilizar la extracción de los datos desde el disco duro, y que pueda sustituir las tuplas o registros ya procesados por nuevos registros sin procesar. Este módulo permitiría trabajar tablas muchos más grandes, y de esta forma la VRAM ya no sería una limitante en el desempeño de nuestro - de búsqueda.

Otras tareas, que podrían desarrollarse serían implementar un paralelismo aun mayor, utilizando arreglos de GPU y no solamente una. O también, paralelizar funciones de otras áreas como la minería de datos y agregar más funcionalidades al motor de búsqueda desarrollado.

Referencias

[1] KHAN, M. F., PAUL, R., GHAFOOR, A. (1999) "Intensive Data Management in Parallel System: A Survey", Journal: Distributed and Parallel Database, vol. 7, no. 4, pp. 383-414.

[2] SUBRAMANIAN, D. K., SUBRAMANIAN, K. (1998) "Query Optimization in Multidatabase Systems", Journal: Distributed and Parallel Database, vol. 6, no. 2, pp. 183-210.

[3] EVRENDILEK, C., DOGAC, A., OZCAN, F. (1997) "Multidatabase Query Optimization", Journal: Distributed and Parallel Database, vol. 5, no. 1, pp. 77-114.

[4] BAIAO, F., MATTOSO, M., ZAVERUCHA, G. (2004) "A Distribution Design Methodology for Object DBMS", Journal: Distributed and Parallel Database, vol. 16, no. 1, pp. 45-90.

[5] COSTA, R. L., FURTADO, P. (2011) "Quality of Experience in Distributed Database", Journal: Distributed and Parallel Database, vol. 29, no. 5, pp. 361-396.

[6] PAPADOPOULOS, A. N., MANOLOPOULOS, Y. (2001) "Distributed Processing of Similarity Queries". Journal: Distributed and Parallel Database, vol. 9, no. 1, pp. 67-92.

[7] AHMAD, I., KARLAPALEM, K., KWOK, Y. K. (2002) "Evolutionary Algorithms for Allocating Data in Distributed Database Systems", Journal: Distributed and Parallel Database, vol. 11, no. 1, pp. 5-32.

[8] CHEN, S. D., SHEN, H., TOPOR, R. (2002) "Permutation-Based Range-Join Algorithms on N-Dimensional Meshes", IEEE Transactions on Parallel and Distributes Systems, vol. 13, no. 4, pp. 413-431.

[9] TRAN, T. M., LEE, B. S. (2010) "Distributed Stream Join Query Processing with Semijoins". Journal: Distributed and Parallel Database, vol. 27, no. 3, pp. 211-254.

[10] PETERS, H., HILDEBRANT O. S., LUTTENBERG, N. (2009) "Fast in Place Sorting with CUDA Based on Bitonic Sort". Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I, pp. 403-410

[11] SINORN, E., ASSARSSON U. (2008) "Fast Parallel GPU-sorting using a Hybrid Algorithm", Journal of Parallel and Distributed Computing , Vol. 68, Issue 10, pp. 1381-1388.

[12] SANDERS, J., KANDROT, E. (2011) "CUDA by Example. An Introduction to General-Purpose GPU Programming". Boston, MA: Addison-Wesley.

[13] FARBER, R. (2011) "CUDA Application Design and Development". Waltham, MA: Morgan Kaufmann – Elsevier.

[14] KIRK, D. B., HWU, W. W. (2010) "Programming Massively Parallel Processors. A Hands-on Approach". Burlington, MA: Morgan Kaufmann – Elsevier.

[15]ALLEN, G., OWENS, M. (2010) "The Definitive Guide to SQLite". United States of America: Appress.

[16] ABDELGUERFI, M., WONG, K. (1998) "Parallel Database Techniques". United States of America: IEEE Computer Society.

[17] TAMER ÖZSU, M., VALDURIEZ, P. (2010) "Principles of Distributed Database Systems". United States of America: Springer.

[18] TANIARD, D., LEUNG, C. H., RAHAYU, W., GOEL, S. (2008) "High-Performance Parallel Database Processing and Grid Database". United States of America: Wiley.

[19] SILBERSCHATZ, A, KORTH, H. F., SUDARSHIAN, S. (2005) "Database System Concepts". Boston, MA: Mc Graw-Hill.

[20] PRATT, P. J., ADAMSKI, J. J. (2011) "Concepts of Database Management". United States of America: Course Technology CENGAGE Learning.

[21] DATE, C. J. (1987) "Twelve Rules for a Distributed Database", Computer World 21.23, June 8, 1987.

[22] DATE, C. J. (1990) "An Introduction to Database Systems". Boston, MA: Addisson-Wesley.

[23] HE, B., YANG, K., YANG, K., FANG, R. (2009) "Relational Query Coprocessing on Graphics Processors", ACM Transactions on Database Systems, Vol. 34, Issue 4, Article 21.

[24] HE, B., YANG, K., FANG, R., LU, M. (2008) "Relational Joins on Graphics Processors", Proceedings of the 2008 ACM SIGMOD international conference on Management of Data, pp. 511-524.

[25] BAKKUM, P., SKADRON, K. (2010) "Accelerating SQL Database Operations on a GPU with CUDA", Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Unit, pp. 94-103.

[26] HERMANN, E., RAFFIN, B., FAURE, F., GAUTIER, T., ALLARD, J. (2010) "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations". Euro-Par 2010 – Parallel Processing, pp. 235 – 246.

[27] HARDAVELLAS, N., PANDIS, I., JHONSON, R., MANCHERIL, N. G. (2007) "Database Servers on Chip Multiprocessors: Limitations and Opportunities". Proceedings of the 3rd International Conference on Innovative Data Systems Research, pp. 79-87.

[28] SU, S. Y. W., RANKA, S., He, X. (2000) "Performance Analysis of Parallel Query Processing Algorithms for Object-Oriented", IEEE Transactions on Knowledge and Data Engineering, vol. 12, no. 6, pp. 979-997.

[29] FANG, W., LAU, K. K., LU, M. (2008) "Parallel Data Mining on Graphics Processors", Technical Report HKUST-CS08-07, Oct 2008.

[30] "What is GPU Computing?", Abril 2012, http://www.nvidia.com/object/GPU_Computing.html

[31] "CUDA, Parallel Programing Made Easy", Abril 2012, http://www.nvidia.com/object/cuda_home_new.html

[32] "SQLite Virtual Machine OpCodes", Abril 2012, http://www.sqlite.org/opcode.html

[33] "About SQLite", Abril 2012, http://www.sqlite.org/about.html

[34] "NVIDIA SLI ", Abril 2012, http://www.geforce.com/hardware/technology/sli

Anexo A

En este anexo, se enlistan y se describe la función que realizan todos y cada uno de los operadores u OpCodes de SQLite que durante la elaboración de este trabajo fueron reprogramados en CUDA para poder emular su resultado y poder ser ejecutados dentro de una GPU.

Recordemos que para visualizar el algoritmo de resolución o plan de ejecución para una consulta dentro del prompt de SQLite, basta con colocar el comando "explain" antes de la consulta SQL y enseguida se despliega la secuencia de OpCodes.

Antes de estudiar los OpCodes, observemos como se compone un OpCode. Un OpCode está constituido por 7 campos (ver la Figura 49), el primero de ellos es el "Addr", que no es más que el índice o posición que le corresponde a la instrucción en cuestión dentro de la secuencia de OpCodes que en conjunto y de forma ordenada van a resolver la consulta SQL.

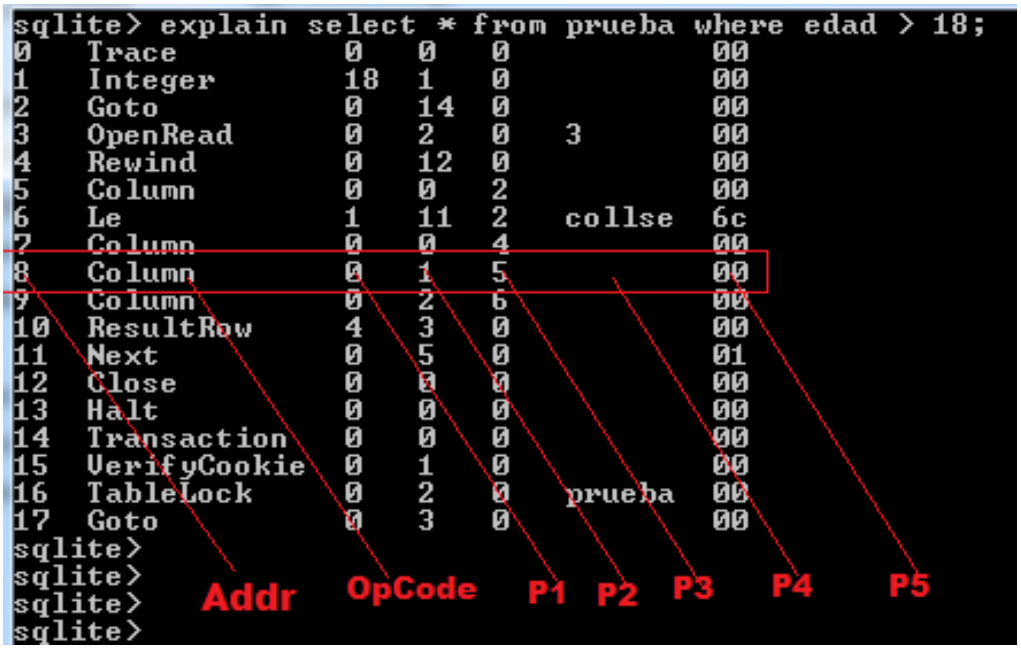


Figura 1. OpCode dentro de un plan de ejecución.

El segundo campo es el "OpCode" y es simplemente un mnemónico que ayuda a identificar de qué operación estamos hablando. SQLite despliega el mnemónico para que el usuario logre reconocerlo más rápido, pero internamente dentro del código fuente del programa, no es más que un número.

Posteriormente, vienen cinco campos, de los cuales los primeros tres: P1, P2 y P3, están presentes prácticamente en todos los OpCodes, y en cada uno tienen un significado distinto. P4 aparece solo en algunos OpCode y finalmente P5 que aunque existe en todos los OpCode en la mayoría su valor es cero, por lo que su presencia resulta irrelevante muchas veces.

Ahora sí, sabemos que P1, P2 y P3 son los campos más importantes para los OpCode, y que P4 y P5 tienen poca relevancia en la mayoría de ellos, sin embargo si están presentes en algunos y son importantes en dichos casos.
Los OpCode que fueron programados en CUDA para su ejecución en la GPU se muestran en la Tabla 28.

Tabla 1. Descripción de OpCodes reprogramados en CUDA.

OpCode	Descripción
AggFinal	Finaliza la función de agregación (max, min, sum, count, avg). El resultado final queda en P1.
AggStep	Ejecuta un paso de la función de agregación (max, min, sum, count, avg).
Close	Cierra el apuntador indicado por P1 previamente abierto.
Column	P1 apunta a una tupla dentro de una tabla. Extrae el valor de la columna P2 de la tupla apuntada por P1. Si no existe valor, extrae NULL. El valor extraído es almacenado en el registro P3.
Copy (SCopy)	Hace una copia del registro P1 hacia el registro P2.
Count	Almacena en P2 el número de tuplas en la tabla o índice que señala P1.
Eq	Salta a la dirección P2 si el valor de los registros P1 y P3 es igual.
Function	Invoca la función indicada por P4 (comparación de cadenas), y almacena resultado en P3.
Ge	Salta a la dirección P2 si el valor del registro P3 es mayor o igual al del registro P1.
Goto	Un salto incondicional a la dirección de P2. La siguiente operación ejecutada será la indicada por P2.
Gt	Salta a la dirección P2 si el valor del registro P3 es mayor al del registro P1.
Halt	Fin y salida inmediata. Todos los apuntadores se cierran.
IdxGe	P1 es un apuntador al índice de la tupla actual de una tabla. P3 es un valor llave. Compara el valor de P3 con el valor del índice de P1. Si P1 es mayor o igual que P3, salta a P2. Si no entonces continúa con la siguiente instrucción. If P5 no es cero la comparación debe ser: P1 mayor a P3.
IdxLt	P1 es un apuntador al índice de la tupla actual de una tabla. P3 es un valor llave. Compara el valor de P3 con el valor del índice de P1. Si P1 es menor que P3, salta a P2. Si no entonces continúa con la siguiente instrucción. If P5 no es cero la comparación debe ser: P1 menor o igual a P3.
IdxRowid	Escribe dentro del registro P2 un entero, el cual es la última entrada dentro del

	índice de apuntado por P1.
If	Salta a P2 si el valor del registro P1 es verdadero. Se considera verdadero si es numérico y distinto de cero.
IfNeg	Si el valor de P1 es menor que cero, salta a P2.
IfNot	Salta a P2 si el valor del registro P1 es falso. Se considera falso si tiene un valor numérico de cero.
IfPos	Si el valor de P1 es uno o más grande, salta a P2.
IfZero	Si al sumar el valor de P1 con el valor de P3 el resultado es cero, salta a P2.
Integer (Int64)	El valor entero en P4 es almacenado en el registro de P2.
IsNull	Salta a P2 si el valor de P1 es NULL.
Le	Salta a la dirección P2 si el valor del registro P3 es menor o igual al del registro P1.
Lt	Salta a la dirección P2 si el valor del registro P3 es menor al del registro P1.
MustBeInt	Evalúa que el valor de P1 sea entero. Si no es entero entonces salta a la dirección P2.
Ne	Salta a la dirección P2 si el valor de los registros P1 y P3 no es igual.
Next	Avanza a la siguiente tupla del apuntador P1. P1 apunta a una tupla dentro de una tabla real, no a una temporal. Si no hay más tuplas en la tabla, entonces continua con la siguiente instrucción. Pero si el avance del apuntador es exitoso, salta inmediatamente a la instrucción indicada por P2.
NotExist	P1 es un apuntador al índice de una tabla. P3 es un valor llave. Busca el valor que se encuentra en el registro P3 dentro del índice de la tabla P1, si el valor no existe entonces salta a P2. Si existe continua con la siguiente instrucción inmediata.
NotNull	Salta a P2 si el valor de P1 no es NULL.
Null	Escribe NULL en el registro P2. Si P3 es más grande que P2, entonces escribe NULL en todos los registros entre P2 y P3.
OpenRead	Abre una tabla para ser leída. P2 es el índice de la tabla a abrir. P3 el apuntador a la BD. El nuevo apuntador a tupla dentro de la tabla abierta será puesto en P1. Si P1 es negativo significa un error al momento de abrir la tabla.
Real	El valor de punto flotante en P4 es almacenado en el registro P2.
RealAffinity	Si el valor de P1 es un valor entero, lo convierte a un valor flotante.
ResultRow	Los registros desde P1 hasta P1+P2-1, contienen una tupla de resultados.
Rewind	Apunta hacia la primer tupla dentro de la tabla indicada por P1. Si la tabla esta vacía y P2 > 0, entonces salta inmediatamente a P2. Si P2 = 0 o la tabla no es vacía, continua entonces con la siguiente instrucción.
RowId	Almacena en P2 el valor del índice de la tupla actual dentro de la tabla P1.
Seek	P1 es un apuntador a una tabla abierta. P2 es un valor llave. Reposiciona el apuntador P1 hacia el valor de entrada dado por P2.
SeekGe	P1 es un apuntador a una tabla abierta.

	P3 es un valor llave. Reposiciona el apuntador P1 hacia el valor de entrada inmediatamente mayor o igual dado por P3. Si no existe dicha entrada en el índice, salta a P2.
SeekGt	P1 es un apuntador a una tabla abierta. P3 es un valor llave. Reposiciona el apuntador P1 hacia el valor de entrada inmediatamente mayor dado por P3. Si no existe dicha entrada en el índice, salta a P2.
SeekLe	P1 es un apuntador a una tabla abierta. P3 es un valor llave. Reposiciona el apuntador P1 hacia el valor de entrada inmediatamente menor o igual dado por P3. Si no existe dicha entrada en el índice, salta a P2.
SeekLt	P1 es un apuntador a una tabla abierta. P3 es un valor llave. Reposiciona el apuntador P1 hacia el valor de entrada inmediatamente menor dado por P3. Si no existe dicha entrada en el índice, salta a P2.
String	La cadena de caracteres en P4 es almacenado en el registro P2.
TableLock	Obtiene el bloqueo de una tabla en particular. P1 se refiere al identificador de la BD en uso. P2 es el identificador de la tabla que será bloqueada. P3 en valor 0 significa bloqueo de lectura, en valor 1 es bloqueo de escritura. P4 tiene una cadena con el nombre de la tabla en cuestión.
Trace	Señal de inicio. Sus valores P1, P2, P3 no tienen significado. Todos los planes de ejecución comienzan siempre con este OpCode.
Transaction	Inicia la transacción. P1 es el índice del archivo de base de datos a utilizar, generalmente su valor es 0 y se refiere a la BD principal, otro valor se refiere a una BD temporal. P2 tiene valor 0 cuando se busquen bloqueos de lectura en la BD, en cualquier otro cado serán de escritura.
VerifyCookie	Verifica que la BD este actualizada, debe usar la última versión con los cambios guardados.

Anexo B

En este anexo se presentan los planes de ejecución para algunas consultas representativas que puede resolver nuestro motor paralelo de búsqueda.

B.1 – Plan de ejecución para una consulta SQL sobre una sola tabla.

SELECT id, workclass, marital_status, age FROM t1 WHERE workclass LIKE "Private" AND marital_status NOT LIKE "Married" AND age > 60;

Tabla 2. Plan de ejecución - Consulta sobre una tabla.

#	OpCode	P1	P2	P3	P4	P5
0	Trace	0	0	0		00
1	Integer	60	1	0		00
2	Goto	0	23	0		00
3	OpenRead	0	2	0	4	00
4	Rewind	0	21	0		00
5	String	0	3	0	Private	00
6	Column	0	2	4		00
7	Function	1	3	2	Like(2)	02
8	IfNot	2	20	1		00
9	String	0	3	0	Married	00
10	Column	0	3	4		00
11	Function	1	3	2	Like(2)	02
12	If	2	20	1		00
13	Column	0	1	2		00
14	Le	1	20	2		6c
15	Rowid	0	6	0		00
16	Column	0	2	7		00
17	Column	0	3	8		00
18	Column	0	1	9		00
19	ResultRow	6	4	0		00
20	Next	0	5	0		01
21	Close	0	0	0		00
22	Halt	0	0	0		00
23	Transaction	0	0	0		00
24	VerifyCookie	0	8	0		00
25	TableLock	0	2	0	t1	00
26	Goto	0	3	0		00

En este plan de ejecución (Tabla 29), podemos apreciar que los principales operadores que resuelven esta consulta se encuentran entre las líneas 4 y 20. El proceso resolución entra en un ciclo iniciado por la línea 5 (después de Rewind) y la instrucción Next de la línea 20 se encarga de repetir dicho procedimiento tantas veces como tuplas o registros haya en la tabla de datos, todos los OpCode que se encuentran entre ambas líneas se repetirán una y otra vez. Para paralelizar este tipo de consultas, la asignación de trabajo entre los hilos se realiza tal y como se explica en el apartado 4.4.1, de este documento.

B.2 – Plan de ejecución para una consulta SQL sobre dos tablas sin índices.

SELECT id, age, occupation FROM m1 NATURAL JOIN m2 WHERE workclass NOT LIKE "Federal-gov" AND occupation NOT LIKE "Exec-managerial";

Tabla 3. Plan de ejecución - Consulta sobre 2 tablas sin índices.

#	OpCode	P1	P2	P3	P4	P5
0	Trace	00	00	00		00
1	Goto	0	26	0		00
2	OpenRead	0	6	0	3	00
3	OpenRead	1	9	0	4	00
4	Rewind	0	23	0		00
5	String	0	2	0	Federal-gov	00
6	Column	0	2	3		00
7	Function	1	2	1	Like(2)	02
8	If	1	22	1		00
9	Rewind	1	22	0		00
10	String	0	2	0	Exec-managerial	00
11	Column	1	3	3		00
12	Function	1	2	1	Like(2)	02
13	If	1	21	1		00
14	Column	0	0	1		00
15	Column	1	0	4		00
16	Ne	4	21	1		6b
17	Column	0	0	5		00
18	Column	0	1	6		00
19	Column	1	3	7		00
20	ResultRow	5	3	0		00
21	Next	1	10	0		01
22	Next	0	5	0		01
23	Close	0	0	0		00
24	Close	1	0	0		00
25	Halt	0	0	0		00
26	Transaction	0	0	0		00
27	VerifyCookie	0	8	0		00

28	TableLock	0	6	0	m1	00
29	TableLock	0	9	0	m2	00
30	Goto	0	2	0		00

Para consultas sobre dos tablas sin índices como la que se muestra en esta sección, podemos ver dentro de su plan de ejecución (Tabla 30) que los operadores que se encargan de resolverla están entre las líneas 4 y 22. Por los operadores Next (líneas 21 y 22), podemos ver que existen dos ciclos anidados. El primer ciclo va desde la línea 5 (después de su Rewind) hasta el Next de la línea 22. El segundo ciclo es más pequeño y va desde la línea 10 (efectivamente después del segundo Rewind) hasta el Next de la línea 21. Para paralelizar este tipo de consultas, la asignación de trabajo entre los hilos se realiza tal y como se explica en el apartado 4.4.2, de este documento.

B.3 – Plan de ejecución para una consulta SQL sobre dos tablas con índices.

SELECT id, age, occupation FROM t1 NATURAL JOIN t2 WHERE workclass NOT LIKE "Federal-gov" AND occupation NOT LIKE "Exec-managerial";

Tabla 4. Plan de ejecución - Consulta sobre 2 tablas con índices.

#	OpCode	P1	P2	P3	P4	P5
0	Trace	0	0	0		00
1	Goto	0	2	4		00
2	OpenRead	0	2	0	3	00
3	OpenRead	1	3	0	4	00
4	Rewind	0	21	0		00
5	String	0	2	0	Federal-gov	00
6	Column	0	2	3		00
7	Function	1	2	1	Like(2)	02
8	If	1	20	1		00
9	Rowid	0	1	0		00
10	MustBeInt	1	20	0		00
11	NotExists	1	20	1		00
12	String	0	2	0	Exec-managerial	00
13	Column	1	3	3		00
14	Function	1	2	4	Like(2)	02
15	If	4	20	1		00
16	Rowid	0	5	0		00
17	Column	0	1	6		00
18	Column	1	3	7		00
19	ResultRow	5	3	0		00

20	Next	5	3	0		00
21	Close	0	0	0		00
22	Close	1	0	0		00
23	Halt	0	0	0		00
24	Transaction	0	0	0		00
25	VerifyCookie	0	8	0		00
26	TableLock	0	2	0	t1	00
27	TableLock	0	3	0	t2	00
28	Goto	0	2	0		00

Este plan de ejecución (Tabla 31) corresponde a una consulta sobre dos tablas con índices. En este plan, las instrucciones que resuelven la consulta se encuentran entre las líneas 4 y 20, y a pesar de estar involucradas dos tablas, encontramos solamente un ciclo que va desde la línea 5 (precisamente después del único Rewind presente) hasta la línea 20 (Next). La razón se debe a que solamente una tabla debe ser recorrida completamente, mientras que la otra tabla al estar indexada, podemos realizar una búsqueda dentro de ella (en este trabajo se eligió una búsqueda binaria), y dicha búsqueda ocurre en la línea 11, es decir, el OpCode NotExists se encarga de buscar cada valor llave de la primer tabla dentro de la tabla indexada. El procedimiento para paralelizar consultas de este tipo, y para realizar la búsqueda binaria se explican en la sección 4.4.3 de este documento.

B.4 – Plan de ejecución para una consulta SQL sobre más de dos tablas con índices.

SELECT t1.id, age, occupation, race FROM t1 NATURAL JOIN t2 NATURAL JOIN t3 where age > 40 and occupation LIKE "Exec-managerial" AND race NOT LIKE "Asian";

Tabla 5. Plan de ejecución - Consulta sobre más de 2 tablas.

#	OpCode	P1	P2	P3	P4	P5
0	Trace	0	0	0		00
1	Integer	40	1	0		00
2	Goto	0	32	0		00
3	OpenRead	0	2	0	2	00
4	OpenRead	1	3	0	4	00
5	OpenRead	2	4	0	3	00
6	Rewind	0	28	0		00
7	Column	0	1	2		00
8	Le	1	27	2	Collseq(BINARY)	00
9	Rowid	0	2	0		00
10	MustBeInt	2	27	0		00
11	NotExist	1	27	2		00
12	String	0	4	0	Exec-managerial	00
13	Column	1	3	5		00
14	Function	1	4	3	Like(2)	00
15	IfNot	3	27	0		00
16	MustBeInt	2	27	0		00
17	NotExists	2	27	2		00
18	String	0	4	0	Asian	00
19	Column	2	2	5		00
20	Function	1	4	6	Like(2)	00
21	If	6	27	1		00
22	Rowid	0	7	0		00
23	Column	0	1	8		00
24	Column	1	3	9		00
25	Column	2	2	10		00
26	ResultRow	7	4	0		00
27	Next	0	7	0		01
28	Close	0	0	0		00
29	Close	1	0	0		00
30	Close	2	0	0		00
31	Halt	0	0	0		00
32	Transaction	0	0	0		00
33	VerifyCookie	0	8	0		00
34	TableLock	0	2	0	t1	00
35	TableLock	0	3	0	t2	00
36	TableLock	0	4	0	t3	00
37	Goto	0	3	0		00

En la Tabla 32 se muestra un plan de ejecución para casos en los que la consulta involucra más de dos tablas, la forma de resolverlas es similar a las anteriores, recordemos que el operador JOIN trabaja sobre dos tablas a la vez, entonces la tercera tabla tiene que esperar los resultados del JOIN entre la primera y la segunda tabla. En este plan de ejecución, los operadores que resuelven la consulta están entre las líneas 6 y 27. El único ciclo que vemos va desde la línea 7 hasta la 27 y se encarga de recorrer completamente la primera tabla. La segunda tabla al estar indexada se hace una búsqueda binaria dentro de ella con el operador NotExists de la línea 11, y solamente si se encuentra el valor llave, se procede a utilizar la tercera 3 con un nuevo operador NotExists en la línea 17, realizándole también una búsqueda binaria.

B.5 – Plan de ejecución para una consulta SQL con operaciones de agregación.

SELECT avg(age) FROM t1 NATURAL JOIN t2 WHERE workclass LIKE "Federal-gov" OR occupation LIKE "Exec-managerial";

Tabla 6. Plan de ejecución - Consulta con operaciones de agregación.

	OpCode	P1	P2	P3	P4	P5
0	Trace	0	0	0		00
1	Null	0	2	0		00
2	Null	0	1	0		00
3	Goto	0	27	0		00
4	OpenRead	0	2	0	3	00
5	OpenRead	1	3	0	4	00
6	Rewind	0	21	0		00
7	Rowid	0	3	0		00
8	MustBeInt	3	20	0		00
9	NotExists	1	20	3		00
10	String	0	5	0	Federal-gov	00
11	Column	0	2	6		00
12	Function	1	5	4	Like(2)	02
13	If	4	18	0		00
14	String	0	5	0	Exec-managerial	00
15	Column	1	3	6		00
16	Function	1	5	4	Like(2)	02
17	IfNot	4	20	1		00
18	Column	0	1	5		00
19	AggStep	0	5	1	avg(1)	01
20	Next	0	7	0		01

21	Close	0	0	0		00
22	Close	1	0	0		00
23	AggFinal	1	1	0	avg(1)	00
24	SCopy	1	7	0		00
25	ResultRow	7	1	0		00
26	Halt	0	0	0		00
27	Transaction	0	0	0		00
28	VerifyCookie	0	8	0		00
29	TableLock	0	2	0		t1
30	TableLock	0	3	0		t2
31	Goto	0	4	0		00

Para el plan de consultas con operaciones de agregación (Tabla 33), aparecen dos OpCode que se encargan de obtener el resultado de la operación en cuestión (MAX, MIN, AVG, SUM, COUNT). Para el caso particular del plan de ejecución de la consulta mostrada, el primer operador se encuentra en la línea 19, AggStep se encarga de almacenar el resultado temporal, es decir, en cada iteración del ciclo entre las líneas 7 y 20, dicho OpCode actualiza su resultado. Mientras que en la línea 23, aprovechamos este operador para realizar el algoritmo de dos tiempos que se explica en la sección 4.4.4, la cual detalla como se paralelizaron las operaciones de agregación en el presente trabajo.