Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Optimización de consultas en Hive-MapReduce

Tesis que presenta

Alexis de la Cruz Toledo

para obtener el Grado de

Maestro en Ciencias en Computación

Director de tesis:

Dr. Jorge Buenabad Chávez

México,DF **Diciembre del 2012**

Resumen

MapReduce es un modelo y ambiente de programación desarrollado por Google para procesar grandes volúmenes de datos (Exabytes) en paralelo en un clúster compuesto de computadoras de uso general. *MapReduce* no requiere programación paralela del usuario, es tolerante a fallas y capaz de balancear la carga de trabajo de manera transparente al usuario. Las *aplicaciones mapreduce* consisten de pares de funciones map y reduce secuenciales. *Hive* es un Datawarehouse, desarrollado por Facebook, que brinda una infraestructura de base de datos sobre *Hadoop*(una versión libre y abierta de *MapReduce*) y un compilador de SQL (con algunas diferencias menores) que compila consultas SQL a *trabajos mapreduce*. A pesar de que el compilador de *Hive* tiene un módulo de optimización, consultas complejas del tipo OLAP (*On-Line Analytical Processing*) no se optimizan adecuadamente.

Está tesis presenta un análisis de posibles optimizaciones adicionales para mejorar el desempeño de tales consultas, y el diseño de dos optimizaciones específicas para el compilador de *Hive*. Estas dos optimizaciones se enfocan a consultas SQL que involucran subconsultas, y/o funciones de agregación y agrupación. En las consultas que involucran subconsultas que son similares, o iguales, se busca eliminar tantas subconsultas como sean posibles y, consecuentemente, reducir el número total de *trabajos mapreduce* a ejecutar. En las consultas que involucran funciones de agregación y agrupación se busca eliminar *trabajos mapreduce* innecesarios que crea *Hive* actualmente. Ambas optimizaciones tienden a reducir el número de operaciones de entrada/salida y la cantidad de datos a enviar por la red.

Extendimos el compilador de *Hive* versión 0.8 con nuestras optimizaciones y las evaluamos con varias consultas OLAP típicas, algunas del *benchmark* TPC-H, ejecutándose en un clúster de 20 nodos. Nuestras optimizaciones mejoran el desempeño de *Hive* hasta en un 30 %. Todavía es posible mejorar al compilador de *Hive*.

Abstract

MapReduce is a programming model and execution environment developed by Google to process very large amounts of data (Exabytes) in parallel in clusters configured with off-the-shelf computer hardware. *MapReduce* does not require parallel programming, is fault tolerant and balances the workload transparently to the programmer. *Mapreduce applications* consist of pairs of sequential *map and reduce functions*. *Hive* is a *Datawarehouse*, developed by *Facebook*, that provides a database infrastructure atop *MapReduce*, and a compiler of SQL (with a few minor differences) that compiles SQL queries to *mapreduce jobs*. Although Hive's compiler has an optimization module, complex queries such as OLAP (On Line Analytical Processing) queries are not optimized properly.

This thesis presents an analysis of other possible optimizations to improve the performance of such queries, and the design of two specific optimizations for the *Hive* compiler. These two optimizations are targeted at SQL queries involving subqueries and/or aggregate functions. For queries that involve subqueries that are similar or identical, our optimization seeks to eliminate as many instances of such subqueries as possible, thereby reducing the total number of *mapreduce jobs* to run. For queries that involve aggregate functions, our optimization seeks to remove unnecessary *mapreduce jobs* that *Hive* currently generates. Both optimizations tend to reduce the number of input/output operations and the amount of data send over the network.

We extended the compiler of *Hive* version 0.8 with our optimizations and evaluate them with several typical OLAP queries, some of the TPC-H benchmark, running on a cluster of 20 nodes. Our optimizations improve performance by up to 30 %. It is still possible to improve the *Hive* compiler.

Agradecimientos

Quiero agradecer a *DIOS* por darme la oportunidad de alcanzar esta meta y sueño de mi vida, por poner en mi camino a personas maravillosas que me apoyaron en cada momento de esta etapa.

A mis padres, Conrado de la Cruz García y Cristina Toledo Arévalo por su incondicional apoyo, por educarme en mis primeros pasos y hacer de mí lo que ahora soy. Gracias por apoyarme en cada una de mis decisiones y pasos de mi vida, pero sobre todo gracias por regalarme la vida.

A mis hermanos Ivan y Felix porqué cuando más los necesitaba siempre estaban ahí para apoyarme y brindarme palabras de ánimo, porqué siempre me alentaban a seguir adelante.

A mis tíos, en especial a mi tío Filiberto, mi tía Virginia y mi tío Elpidio quiénes me han impulsado ha salir adelante y siempre han estado al pendiente de mí y mi familia, siento su cariño y aprecio, muchas gracias.

A mi asesor, el Dr. Jorge Buenabad Chávez por la confianza brindada, gracias por sus enseñanzas, observaciones, dedicación y apoyo en este trabajo de tesis. Gracias por su paciencia y su disponibilidad para hacer las cosas. Pero sobre todo, gracias por ser un amigo que estuvó conmigo siempre en las buenas y en las malas.

A mis revisores de tesis, el Dr. José Guadalupe Rodríguez García y el Dr.Renato Barrera Rivera por tomarse el tiempo de revisar mi tesis y sus valiosos comentarios.

A mis amigos, gracias por acompañarme en este camino, me han hecho sentir lo que es una verdadera amistad, a todos los llevó en el corazón. Gracias por estar en los momentos difíciles y alegres de mi vida. A mis amigos Dorian Argueta, Laura

Granados, Cinthya Duplan, Heriberto Cruz, Tonantzin Guerrero, Alejandra Moreno y demás amigos, a todos gracias por ser mis mejores amigos. En especial, quiero agradecer a Viridiana Ponce y Paulina León por su incomensurable amistad, gracias por hacerme parte de sus vidas como ustedes son en la mía, gracias por tener las palabras correctas que siempre me alentaban a ser mejor y a superarme día a día, gracias por escucharme y levantarme en mis momentos de desaliento, gracias por estar ahí siempre para mi, saben que cuentan conmigo. También quiero agradecer a mi amiga Gabriela de León y a sus tíos por su apoyo en mi llegada a la Ciudad de México, muchas gracias por su apoyo.

También agradezco, al Dr. Héctor Peralta Cortés por impulsarme a tomar este camino de la investigación, gracias Dr. Héctor por enseñarme los primeros pasos.

Agradezco a Sofia Reza por su amabilidad y consejos, gracias por apoyarme en cada uno de los trámites que necesite y por regalarme una sonrisa cada vez que nos veíamos. También a César Nabor por su disponibilidad y amabilidad cada vez que visitaba la biblioteca. Ambos demuestran su gusto y placer por su trabajo.

A CONACYT (Consejo Nacional de Ciencia y Tecnología) por la confianza y el apoyo económico que me brindó durante mi formación académica en la maestría.

Al CINVESTAV-IPN (Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional) por permitirme ser parte de ésta importante familia de investigación, gracias por el apoyo económico que se me otorgó para la finalización de mi tesis y las facilidades brindadas durante toda mi estancia. Gracias a cada uno de mis profesores del departamento de Computación que me brindaron sus conocimientos y experiencias, contribuyendo con mi formación profesional.

Índice general

Agradecimientos vii

Índice de figuras xiii

Índice de tablas xix

1. Introducción **1**

1.1. Organización de la Tesis 6

2. MapReduce **7**

2.1. Modelo de programación de *MapReduce* 7

2.2. Ambiente de ejecución 9

2.3. Hadoop 12

2.3.1. Arquitectura de *Hadoop* 14

2.3.2. Configuración de *Hadoop* 17

2.4. Ejemplos 20

2.4.1. *Programa mapreduce escrito* en *Java* 21

2.4.2. *Programa mapreduce* escrito en *Python* 25

2.5. Resumen 27

3. Hive **29**

3.1. Arquitectura de Hive 31

3.2. Estructuras del datawarehouse 32

3.3. HiveQL 34

3.3.1. *Lenguaje de Definición de Datos (DDL)* 35
3.3.2. *Lenguaje de Manipulación de Datos (DML)* 35
3.4. Compilador de Hive 37
3.5. Optimizador de Hive 38
3.5.1. Optimizaciones actuales en Hive 40
3.5.2. Componentes de una optimización 49
3.6. Motor de ejecución de Hive 53
3.7. Resumen 56

4. Otras posibles optimizaciones para consultas *HiveQL* 59
4.1. Introducción 59
4.1.1. Consultas de Chatziantoniou 60
4.1.2. Consultas del estudio TPC-H 61
4.2. Eliminación de *trabajos mapreduce* de operadores de agregación y agrupación (GroupByOperator) 63
4.3. Eliminación de secuencia de operadores *Hive* redundantes en un DAG 67
4.3.1. Eliminación de operadores redundates entre ramas simples . . 68
4.3.2. Eliminación de operadores redundantes entre ramas compuestas 72
4.4. Optimización del operador JoinOperator 83
4.5. Eliminación de *trabajos mapreduce* innecesarios en una misma rama de un *DAG*. 86
4.6. Resumen 89

5. Nuestras Optimizaciones realizadas a consultas *HiveQL* 91
5.1. Introducción 91
5.2. Eliminación de *trabajos mapreduce* asociados a operadores de agregación y agrupación (*GroupByOperator*) 93
5.2.1. Rule 94
5.2.2. GraphWalker 95
5.2.3. Dispatcher 97

5.2.4. Processor 97
5.3. Eliminación de operadores redundantes entre ramas simples o compuestas 101
5.3.1. GraphWalker 102
5.3.2. Dispatcher 105
5.3.3. Processor 108
5.4. Trabajo relacionado 112
5.5. Resumen 118

6. Evaluación experimental y resultados 119
6.1. Plataforma experimental 119
6.1.1. Hardware 119
6.1.2. Software 120
6.1.3. Aplicaciones 121
6.2. Organización de experimentos y resultados 127
6.2.1. Experimento 1: Hive vs HiveC 128
6.2.2. Experimento 2: Variando el número de nodos 133
6.2.3. Experimento 3: Variando el número de *tareas reduce* 138
6.3. Resumen 145

7. Conclusiones y trabajo futuro 147
7.1. Trabajo a futuro 151

A. Eliminación de operadores redundantes entre ramas simples o compuestas en un *DAG* 153
A.1. Estructuras de datos de Hive 154
A.2. Estructuras de datos propias de nuestra optimización 155
A.3. GraphWalker 156
A.4. Dispatcher 163
A.5. Processor 171

B. Arquitectura del optimizador de sentencias *HiveQL* 177

C. Número de *trabajos mapreduce* que generó *Hive* por defecto para cada consulta de Chatziantoniou y TPC-H que se utilizó para evaluar nuestras optimizaciones. 185

Bibliografía 191

Índice de figuras

1.1. Resultados obtenidos de las consultas 1, 2 y 3 en Hadoop, Pig y Hive con un log de 1GB. 4

2.1. Ambiente de ejecución de MapReduce [1]. 10
2.2. Flujo de datos en *MapReduce* [2]. 11
2.3. Componentes y subproyectos de Hadoop [2]. 13
2.4. Topología de un clúster *Hadoop* [3]. 15
2.5. Interacción entre el *JobTracker* y *TaskTracker* [3]. 16
2.6. Interacción de *NameNode* y DataNodes en HDFS [3]. Los números entre paréntesis corresponden a los *ids* de los *DataNodes*. 17

3.1. Arquitectura de hive [4]. 31
3.2. Representación física de las estructuras de *Hive* en el sistema de archivos *HDFS*. 33
3.3. DAG de la *consulta facebook*. 41
3.4. DAG y plan físico de la *consulta facebook* con la optimización de reducción de la escritura a HDFS por medio de agregaciones parciales en *tareas reduce*. 45
3.5. DAG y plan físico de la *consulta facebook* con la optimización de sesgo de datos en la operación GroupBy. 47
3.6. Ejemplo de recorrido de un árbol con el algoritmo *DFS*. 51
3.7. Diagrama de actividad de una optimización en *Hive*. 52

3.8. Formas de identificar una tarea map. 54
3.9. DAG y plan físico de la *consulta facebook*. 55
3.10. Formas de identificar una tarea reduce. 55

4.1. Esquema relacional del estudio TPC-H 62
4.2. Plan físico de la consulta 1 de Chatziantoniou con las optimizaciones actuales de Hive 65
4.3. Plan físico de la consulta 1 de Chatziantoniou con la optimización propuesta 66
4.4. DAG de la consulta 2 de Chatziantoniou 69
4.5. Plan físico de la consulta 2 de Chatziantoniou con las optimizaciones actuales de Hive 70
4.6. Plan físico de la consulta 2 de Chatziantoniou con la optimización propuesta 71
4.7. DAG de la consulta 3 de Chatziantoniou 74
4.8. Plan físico de la consulta 3 de Chatziantoniou con las optimizaciones actuales de Hive 75
4.9. Plan físico de la consulta 3 de Chatziantoniou con la optimización propuesta 76
4.10. DAG de la consulta 11 de TPC-H 79
4.11. Plan físico de la consulta 11 de TPC-H con las optimizaciones actuales de Hive 80
4.12. Plan físico de la consulta 11 de TPC-H con la optimización propuesta 81
4.13. Plan físico de la consulta 3 de Chatziantoniou con la optimización del operador *Join* propuesta 84
4.14. Plan físico de la consulta q1c que resulta del *DAG* que se optimizó después aplicar nuestra optimización de eliminar *trabajos mapreduce* asociados a operadores de agregación. 87

4.15. Plan físico de la consulta q1c que resulta del *DAG* después de aplicar la optimización propuesta para eliminar *trabajos mapreduce* innecesarios en una misma rama de un *DAG*. 88

5.1. *DAG* de la consulta 1 de Chatziantoniou 96
5.2. *Plan físico* de la consulta 1 de Chatziantoniou. 98
5.3. *Plan físico* de la consulta 1 de Chatziantoniou después de aplicar nuestra optimización al *DAG* de la consulta. 100
5.4. Nuestras estructuras de datos utilizadas en la entidad *GraphWalker* de la optimización para eliminar secuencia de operadores *Hive* redundante en un DAG. Los valores corresponden a la ejecución de la entidad *GraphWalker* en el *"DAG optimizado"* que entregan las optimizaciones actuales de *Hive* para la consulta 3 de Chatziantoniou. 102
5.5. *DAG* de la consulta 3 de Chatziantoniou 104
5.6. *DAG* de la consulta 3 de Chatziantoniou después de eliminar la primera secuencia de operadores redundantes. 110
5.7. *DAG* de la consulta 3 de Chatziantoniou después de eliminar la segunda secuencia de operadores redundantes. 111
5.8. *Plan físico* de la consulta 3 de Chatziantoniou con el *DAG* optimizado que se forma con las optimizaciones actuales de *Hive*. 113
5.9. *Plan físico* de la consulta 3 de Chatziantoniou con el *DAG* optimizado que se forma después de aplicar nuestra optimización de eliminar operadores redundantes entre ramas. 114
5.10. Posible abstración de la correlación de entrada de *YSMART* en un plan físico de *Hive* 116
5.11. Posible abstración de las correlaciones de transición y flujo de trabajo de *YSMART* en un plan físico de *Hive* 117

6.1. Esquema relacional del estudio TPC-H 124
6.2. Tiempo de ejecución del 1er. Grupo de experimentos con 4GB. 131

6.3. Tiempo de ejecución del 1er. Grupo de experimentos con 8GB. 131

6.4. Tiempo de ejecución del 1er. Grupo de experimentos con 16GB. . . . 132

6.5. Tiempos de ejecución de la consulta q1c procesando 16GB de datos en 8, 16 y 20 nodos. 135

6.6. Tiempos de ejecución de las consultas q2c y q3c procesando 16GB en 8, 16 y 20 nodos. Obsérvese como se disminuye el tiempo de ejecución de las consultas de acuerdo al nivel de paralelismo. 136

6.7. Tiempos de ejecución de las consultas q2t y q3t procesando 16GB de datos en 8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución de la tabla q2t no se disminuye conforme se aumentan los nodos, esto se debe a que la consulta procesa tablas con pocos datos y al parecer *MapReduce* no es escalable con pocos datos. Por lo otro lado, los tiempos de ejecución de la consulta q3t si se reducen conforme se aumentan nodos debido a que involucra tablas con una mayor cantidad de datos. 136

6.8. Tiempos de ejecución de las consultas q11t y q13t procesando 16GB de datos en 8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución de ambas consultas no se disminuye conforme aumentan los nodos, esto se debe a que ambas consultas procesan tablas con pocos datos y al parecer *MapReduce* no es escalable con pocos datos. 137

6.9. Tiempo de ejecución de la consulta q1c con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 142

6.10. Tiempo de ejecución de la consulta q2c con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 142

6.11. Tiempo de ejecución de la consulta q3c con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 143

6.12. Tiempo de ejecución de la consulta q2t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 143

6.13. Tiempo de ejecución de la consulta q3t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 144

6.14. Tiempo de ejecución de la consulta q11t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 144

6.15. Tiempo de ejecución de la consulta q13t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*. 145

A.1. *DAG* y plan físico de la consulta 2 de Chatziantoniou sin nuestra optimización. 158

A.2. Datos en las estructuras de datos que se utilizan en la entidad *GraphWalker* para la consulta 2 de Chatziantoniou. 159

A.3. *DAG* y plan físico de la consulta 11 de tpch sin nuestra optimización. 160

A.4. Datos en las estructuras de datos que se utilizan en la entidad *GraphWalker* para la consulta 11 del estudio TPC-H. 161

A.5. Datos en las estructuras de datos que se utilizan en la entidad *Dispatcher* para la consulta 2 de Chatziantoniou. 165

A.6. Datos en las estructuras de datos que se utilizan en la entidad *Dispatcher* para la consulta 11 del estudio de mercado TPC-H. 165

A.7. *DAG* y plan físico de la consulta 2 de Chatziantoniou con nuestra optimización. 174

A.8. *DAG* y plan físico de la consulta 11 del estudio de mercado TPC-H con nuestra optimización. 176

B.1. Diagrama de paquetes del optimizador de consultas *HiveQL*. 178

B.2. Diagrama de clases que muestra las interfaces y clases base de cualquier optimización de consultas *HiveQL* en *Hive*. 179

B.3. Diagrama de clases que muestra las clases que representan a los operadores *Hive*. 180

B.4. Diagrama de clases que muestra algunas de las clases auxiliares que se consideran durante la compilación de una sentencia *HiveQL* a *trabajos mapreduce*. 181
B.5. Diagrama de clases de algunas optimizaciones de *Hive*. 182

Índice de tablas

3.1. Estructuras y sentencias *DDL* soportadas en *HiveQL*. 36

3.2. Relación entre *las cláusulas de una sentencias HiveQL* y *operadores Hive*. 38

3.3. Ejemplo de resultado de la vista *subq1*. El alias de la tabla *status_updates* es 'a' y el alias de la tabla *profiles* es 'b'. 40

3.4. Ejemplo de resultado de la consulta facebook. 40

6.1. Configuraciones específicas del primer grupo de experimentos. 128

6.2. Distribución de los datos en las 8 tablas del estudio *TPC-H*. 129

6.3. Configuraciones específicas del segundo grupo de experimentos 133

6.4. Configuraciones específicas del tercer grupo de experimentos 138

C.1. Número de *trabajos mapreduce* por cada consulta de Chatziantoniou para 4GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas. 187

C.2. Número de *trabajos mapreduce* por cada consulta de Chatziantoniou para 8GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas. 187

C.3. Número de *trabajos mapreduce* por cada consulta de Chatziantoniou para 16GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas. 187

C.4. Número de *trabajos mapreduce* por cada consulta del estudio TPC-H para 4GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas. 188
C.5. Número de *trabajos mapreduce* por cada consulta del estudio TPC-H para 8GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas. 189
C.6. Número de *trabajos mapreduce* por cada consulta del estudio TPC-H para 16GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas. 190

Capítulo 1

Introducción

Estamos inmersos en la era de los datos. Esto ha sido posible gracias a los avances de las tecnologías de información y comunicación que permiten colectar y almacenar información de actividades comerciales, científicas y cotidianas de manera fácil y económica. Según la Corporación de Datos Internacional (del inglés *International Data Corporation* o IDC), en el año 2011 el "Universo Digital" superó los 1.8 Zettabytes [1] de datos y se espera que para el año 2015 alcance cerca de 8 Zettabytes [5] [6].

El procesamiento y análisis de datos de actividades comerciales es indispensable para identificar tendencias de mercado, detectar fraude en tiempo real, búsqueda de páginas web, análisis de redes sociales, etcétera. Por otra parte, el análisis de datos científicos como el ADN, fenómenos meteorológicos, movimientos de cuerpos espaciales, entre otros son esenciales para el avance de la ciencia [7].

Una manera de almacenar y procesar grandes volúmenes de datos es a través de un Datawarehouse, donde el análisis de los mismos se lleva acabo con consultas *OLAP (del inglés, On-Line Analytical Processing*). Según Bill Inmon (padre del *Datawarehousing*), *"Datawarehouse, es un conjunto de datos integrados, históricos, variantes en el tiempo y unidos alrededor de un tema específico"*[8]. Es usado por la

[1] Un Zettabyte es equivalente a mil Exabytes, un millón de Petabytes, un billón de Terabytes y a un trillón de Gigabytes

gerencia para la toma de decisiones.

Empresas como Netezza [9], Teradata [10], AsterData [11], Greenplum [12], Oracle [13], entre otras, ofrecen servicios y productos (hardware y software) Datawarehouse para procesar y analizar grandes volúmenes de datos con tiempo y detalle adecuado. Sin embargo, los costos de instalación y mantenimiento de este tipo de *Datawarehouse* son excesivos para pequeñas y medianas empresas por utilizar software privado y hardware de propósito específico.

Otra manera de procesar grandes volúmenes de datos es a través de base de datos relacionales, de las cuales existen un tipo que opera en un solo nodo llamadas tradicionales y las que operan en un ambiente distribuido conformado por un clúster. Procesar grandes volúmenes de datos en base de datos tradicionales como *MySQL y Postgres* no es óptimo: el procesamiento puede tardar días. Además, las bases de datos tradicionales no son escalables. Por otra parte, las base de datos que operan en un clúster procesan los datos de manera paralela y distribuida logrando un mejor desempeño, alta disponibilidad, rendimiento y escalabilidad. Sin embargo, el costo de estas bases de datos es alto. Por ejemplo, la licencia anual de *MySQL clúster* cuesta 10,000 doláres [14].

Por tal motivo, Jeffrey Dean y Sanjay Ghemawat fundadores de la empresa *Google* desarrollaron *MapReduce* en el 2004. *MapReduce* es un modelo de programación y ambiente de ejecución para procesar grandes cantidades de datos de manera paralela y distribuida en un clúster conformado por nodos de propósito general. Los usuarios *MapReduce* solo tienen que especificar pares de *funciones map y reduce* secuenciales que constituyen un *trabajo mapreduce*, y el ambiente *MapReduce* replicá las *funciones map y reduce* en los nodos del clúster y las ejecuta en paralelo. *Google* utiliza *MapReduce* para ordenar datos, mineria de datos, aprendizaje de máquina, y en su servicio de búsqueda [1].

Hadoop es una implementación de software libre de *MapReduce* desarrollado por Doug Cutting en Yahoo. Se puede utilizar en un clúster propio o en la nube (por ejemplo, a través de servicios web de Amazon) [15]. Cuenta con el soporte técnico de

Cloudera [16]. Actualmente, se utiliza en compañias como Yahoo, Facebook, Twitter, Last.FM, Amazón, LinkedIn, entre otras. De hecho, las compañias como Netteza, Teradata, Asterdata, Oracle, han integrado *Hadoop* en sus respectivas tecnologías.

Las ventajas de *MapReduce* son [4]:

- Mayor escalabilidad (soporta miles de nodos).
- Tolerancia a fallas a gran escala.
- Flexibilidad en el manejo de datos no estructurados.

Las desventajas de *MapReduce* son [4]:

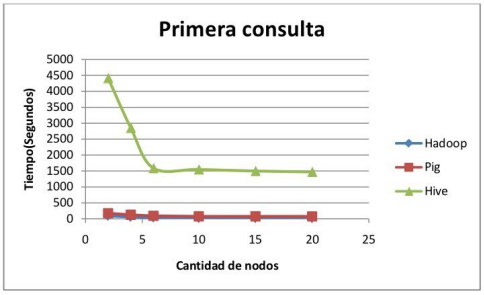
- El desarrollo de programas en *MapReduce* no es fácil, especialmente para los usuarios que no están familiarizados con *funciones map y reduce*, por tal motivo, la productividad de los desarrolladores se ve disminuida.
- Los programas generados son dificiles de mantener y adaptar a otros proyectos.

Por tal motivo en el 2010, *Facebook* desarrolló un framework que se ejecuta sobre *Hadoop* llamado *Hive*. *Hive* es un *Datawarehouse* distribuido de código abierto sin costo. Permite el procesamiento de datos estructurados y no estructurados a través de un lenguaje de consultas parecido al SQL llamado *HiveQL*, que además de soportar consultas SQL, permite incrustar código *MapReduce* como parte de la misma consulta *HiveQL*. *Hive* compila las sentencias *HiveQL* a una serie de *trabajos mapreduce* que se ejecutan en *Hadoop*.

Sin embargo, el rendimiento de *Hive* aun no es óptimo. En el año 2010, se realizó un estudio donde se compara el rendimiento de *Hive* con *Hadoop* y *Pig* [17], otro framework desarrollado sobre *Hadoop* para el procesamiento de grandes volúmenes de datos.

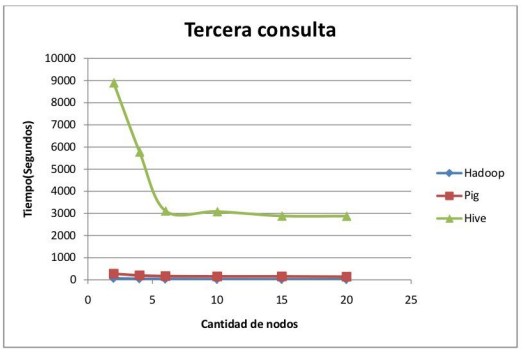
El estudio tuvó como propósito comparar tres consultas implementadas en *Hadoop, Pig y Hive* para procesar logs de un servidor web. Las consultas fueron:

- Primera consulta: Determinar la cantidad de veces que aparece cada dirección IP en el log.



(a) Consulta 1

(b) Consulta 2



(c) Consulta 3

Figura 1.1: Resultados obtenidos de las consultas 1, 2 y 3 en Hadoop, Pig y Hive con un log de 1GB.

- Segunda consulta: Conocer la hora a la que se ha generado la mayor cantidad de errores en el servidor.

- Tercera consulta: Obtener la página o recurso que más veces ha generado errores en el servidor y saber a qué hora este ha producido la mayor cantidad de errores.

El tamaño del log es de 1GB. Las versiones utilizadas fueron: *Hadoop* 0.18, *Pig* 0.5 y *Hive* 0.4.0. Se evaluaron en un cluster con el servicio EC2 (Elastic Computing Cloud) de Amazon Web Service con 2, 4, 6, 10, 15 y 20 nodos. Los resultados obtenidos se observan en las figuras 1.1a, 1.1b, 1.1c respectivamente.

Como se observa en las figuras 1.1a, 1.1b, 1.1c el rendimiento de *Hive* en todas las consultas es menor con respecto a *Hadoop y Pig*. Esto se debe en parte a que las consultas *HiveQL* son consultas *OLAP* y en su mayoría involucran varias subconsultas. El problema es que el compilador de *Hive* compila cada subconsulta en un *trabajo mapreduce* y cuando la consulta involucra subconsultas idénticas o

iguales, el compilador de *Hive* no se da cuenta y genera *trabajos mapreduce* similares o repetidos reduciendo el rendimiento de *Hive*. Además, en ocasiones el compilador está generando *trabajos mapreduce* innecesarios para otro tipo de consultas, por lo que se plantea el problema de **¿Cómo identificar y eliminar *trabajos mapreduce* innecesarios y repetidos generados para una consulta *HiveQL*?, con el objetivo de mejorar el desempeño de consultas *OLAP* en *Hive***.

Una consulta *HiveQL* pasa por 4 fases para compilarse en una serie de *trabajos mapreduce*: Un análisis léxico donde se crea un Árbol Sintáctico Abstracto (AST) como representación gráfica de la sentencia *HiveQL*; después pasa por un análisis sintáctico y semántico donde se crea un *DAG*[2] como representación interna de la consulta *HiveQL* en *Hive*; después pasa por una fase de optimización del *DAG*; y por último se contruyen los *trabajos mapreduce* a partir del *DAG* optimizado.

Esta tesis tuvó como objetivo mejorar el optimizador de consultas de *Hive*, para esto se hizó un análisis de los DAG's que generaron un grupo de consultas *OLAP* utilizadas ampliamente en estudios de Datawarehouse y bases de datos. En base al análisis se llegó a la conclusión que las optimizaciones actuales de *Hive* están pensadas para optimizar la ejecución de una consulta en cada *trabajo mapreduce* que se construye, tomando en cuenta las condiciones del ambiente *MapReduce*. Sin embargo, cuando las consultas involucran subconsultas similares o iguales, *Hive* no se da cuenta de ello y duplica operaciones en el *DAG* de tal modo, que al transformarse el *DAG* a *trabajos mapreduce* se duplican *trabajos mapreduce* optimizados. Por supuesto, esto no es conveniente debido a que cada *trabajos mapreduce* implica un costo de lectura/escritura, un costo de red y un costo de procesamiento en cada nodo del clúster. Así mismo, nos dimos cuenta que en consultas que involucran funciones de agregación y agrupación (*sum(), avg(), max()*, entre otras), en algunas ocasiones están creando un *trabajo mapreduce* innecesario por cada función de agregación y agrupación implicada. Se realizó un análisis de como realiza las optimizaciones internamente *Hive* y se agregó dos optimizaciones al compilador de *Hive* que buscan solucionar

[2]Un DAG es un grafo dirigido que no contiene ciclos.

los problemas planteados. Posteriormente, se realizó un análisis comparativo entre el *Hive* versión 0.8 y el *Hive* con las optimizaciones que se han realizado. El análisis involucró un conjunto de consultas OLAP ampliamente utilizadas en análisis de rendimiento de Datawarehouse y base de datos. El resultado fue que el *Hive* con nuestras optimizaciones tuvó un mejor desempeño que el *Hive* versión 0.8 disminuyendo los tiempos de ejecución hasta en un 30 %.

1.1. Organización de la Tesis

La organización de la tesis es la siguiente:

El Capítulo 2 describe el modelo de programación de *MapReduce*, como se ejecutan los *programas mapreduce* en un clúster y la arquitectura de *Hadoop*.

El capítulo 3 describe la arquitectura y los componentes de *Hive*. También explica cómo se construyen los *trabajos mapreduce* correspondientes a las sentencias *HiveQL*, cuáles son y en que consisten las optimizaciones actuales de *Hive*, y que elementos se deben de considerar para agregar una nueva optimización.

El capítulo 4 presenta un análisis de posibles optimizaciones a realizar en el *DAG* optimizado que brinda *Hive* actualmente. Se describen los problemas que se han encontrado, las optimizaciones propuestas y las ventajas que se tendría al aplicar nuestras optimizaciones.

El capítulo 5 describe las optimizaciones que realizamos en el optimizador de *Hive*.

El capítulo 6 presenta nuestra plataforma experimental. Describe el hardware utilizado, las configuraciones de software y los casos de estudios utilizados para evaluar el desempeño de *Hive* con nuestras optimizaciones en comparación con el *Hive* versión 0.8. Además, muestra los resultados obtenidos.

El capítulo 7 muestra las conclusiones obtenidas del trabajo realizado y algunas ideas para realizar trabajo a futuro.

Capítulo 2

MapReduce

MapReduce es un modelo de programación y ambiente de ejecución desarrollado por Google para procesar grandes cantidades de información del orden de Gigabytes, Terabytes, Petabytes o Zettabytes, de manera paralela y distribuida en un clúster conformado por nodos de uso general.

MapReduce es utilizado por muchas compañias para llevar acabo tareas de inteligencia de negocios (del inglés *Business Intelligence*), como tendencias de mercado, introducción de un nuevo producto, minería de datos, etcétera. *Hadoop* es una versión libre y gratuita de *MapReduce*.

2.1. Modelo de programación de *MapReduce*

La programación paralela y distribuida es compleja, más aun si se requiere especificar balance de carga y tolerancia a fallas. MapReduce es un *middleware* que se encarga de estos aspectos. La programación en *MapReduce* es secuencial, el usuario solo debe de especificar un *programa mapreduce* constituido por al menos una *función map* y una *función reduce* [2] [3]. El ambiente *MapReduce* replica cada función en varios nodos y ejecuta las réplicas en paralelo, realizando balance de carga y tolerancia a fallas de ser necesario; todo esto de manera transparente al programador.

El código 2.1 muestra el pseudocódigo de un programa *MapReduce* que cuenta las

```
1  map (String clave, String valor){
2          // clave: desplazamiento dentro del archivo.
3          // valor: línea de archivo a procesar
4
5      linea = valor;
6      palabras[] = obtener_palabras_de_la_linea(linea);
7
8      i = 0;
9      while (i < palabras.length){
10       palabra = palabras[i];
11       emit(palabra, 1);
12       i++;
13     }
14 }
15
16 reduce (String clave, Iterator valores){
17         // clave: palabra
18         // valores: lista de 1's.
19
20     sum = 0;
21     for each v in valores{
22                 sum += v;
23     }
24
25     emit (clave, sum);
26 }
```

Código 2.1: Pseudoalgoritmo para contar las veces que aparece cada palabra en un archivo.

veces que aparece cada palabra en uno o más archivos. En este ejemplo, el usuario solo especifica una función map y una función reduce. En el ejemplo, la función map se invoca (ejecuta) de manera iterativa sobre los datos de entrada, recibiendo una linea (o registro) del archivo de entrada por cada invocación. El procesamiento de cada línea consiste en obtener cada una de las palabras en la línea, y por cada palabra emitir (imprimir) una línea compuesta de la palabra misma y un 1. El ambiente Mapreduce organiza toda la salida de la función map de tal manera que todos los 1s de una palabra conforman una lista. Entonces se invoca la función reduce de manera iterativa por cada palabra y su lista correspondiente de 1's. El procesamiento de cada lista de 1's por la función reduce es sumarlos e imprimir la palabra y la suma de 1's correspondiente.

Conceptualmente, las *funciones map y reduce* se pueden representar como se

muestra a continuación:

```
map(k1,v1) -> list (k2, v2)
reduce(k2,list(v2)) -> list (k3, v3)
```

Los datos de entrada se procesan por medio de una función map que lee el archivo de entrada de manera iterativa a través de un par de parámetros llamados clave y valor respectivamente (*k1,v1*). La clave y valor depende del tipo de archivo que se lee. Por ejemplo, para un archivo de texto, por cada invocación de la función map se lee una línea de datos y a la función se le envía como clave el desplazamiento en el archivo correspondiente a la línea leída, y como valor la línea misma. En la sección *Input Formats* del libro [2] se explican los diferentes tipos de clave y valor que se manejan según el tipo de archivo de entrada. Por cada invocación de la función map se realiza el procesamiento especificado y al final se puede emitir un par $< clave, valor >$ que formará parte de la lista *list (k2, v2)*. El ambiente *MapReduce* se encarga de agrupar en una lista todos los valores asociados con una misma clave (k2, list(v2)) e invoca la función reduce por cada clave diferente. Por cada invocación de la función reduce se realiza el procesamiento correspondiente y al final se puede emitir un par $< clave, valor >$ formando la lista de pares *list (k3,v3)* que es el resultado final del procesamiento.

Nótese, que el dominio de las *claves y valores* de entrada de la *función map* pueden ser de diferente al dominio de las *claves y valores* de salida de la *función reduce*. Así mismo, las *claves y valores* de salida de la *función map* son del mismo dominio que las *claves y valores* de entrada de la *función reduce* [1].

2.2. Ambiente de ejecución

Las *funciones map y reduce* se ejecutan de manera paralela y distribuida en un clúster. El ambiente *MapReduce* replica las *funciones map y reduce* en los nodos del clúster de tal manera, que las réplicas de ambas funciones se ejecutan al mismo tiempo en nodos distintos (ver figura 2.1). Los datos de entrada a las réplicas de la *función map*

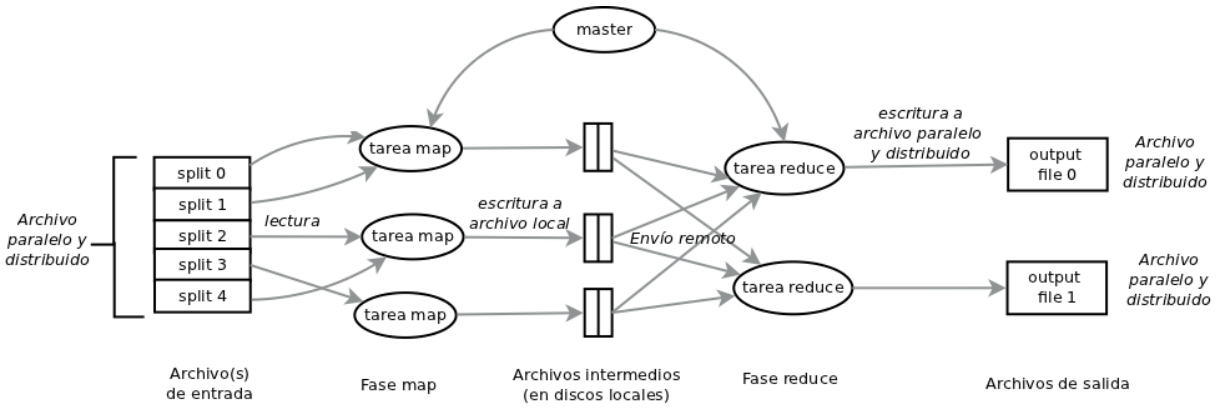


Figura 2.1: Ambiente de ejecución de MapReduce [1].

se encuentran almacenados en un archivo paralelo y distribuido: datos distintos se almacenan en nodos distintos para que sean procesados simultáneamente y así reducir el tiempo total de acceso a datos. Los datos de salida de las réplicas de la *función reduce* también se escriben en el sistema de archivos paralelo y distribuido.

Google File System (GFS) es el sistema de archivos paralelo y distribuido de Google. Se encarga de particionar un archivo en M partes de tamaño fijo llamados *"splits"*. Estos *splits* se distribuyen y se replicán en los nodos del clúster con el objetivo de brindar balance de carga y tolerancia a fallas. El tamaño de un split típicamente es de 16 o 64 MB [18].

En principio, el número de réplicas de la *función map*, o *tareas map*, creadas por el ambiente *MapReduce* es como máximo el número de *splits* (M) que componen los datos de entrada a procesar. Cada *tarea map* lee y procesa uno o varios splits. El procesamiento de cada split se realiza de manera local sin utilizar el recurso de red, y la salida de una *tarea map* (par <clave, valor>) se almacena en memoria, pero si ésta se satura dicha salida se ordena y se almacena en el sistema de archivos local. Esto se debe a que la salida de las *tareas map* son resultados intermedios y no es necesario mantenerlos después de que el *programa mapreduce* ha finalizado [19] [1].

Por otro lado, el número de réplicas de la *función reduce*, o *tareas reduce*, puede ser especificado por el usuario y es independiente del número de *splits* que componen los datos de entrada o salida.

Cuando se han terminado de ejecutar al menos todas las *tareas map* que emiten

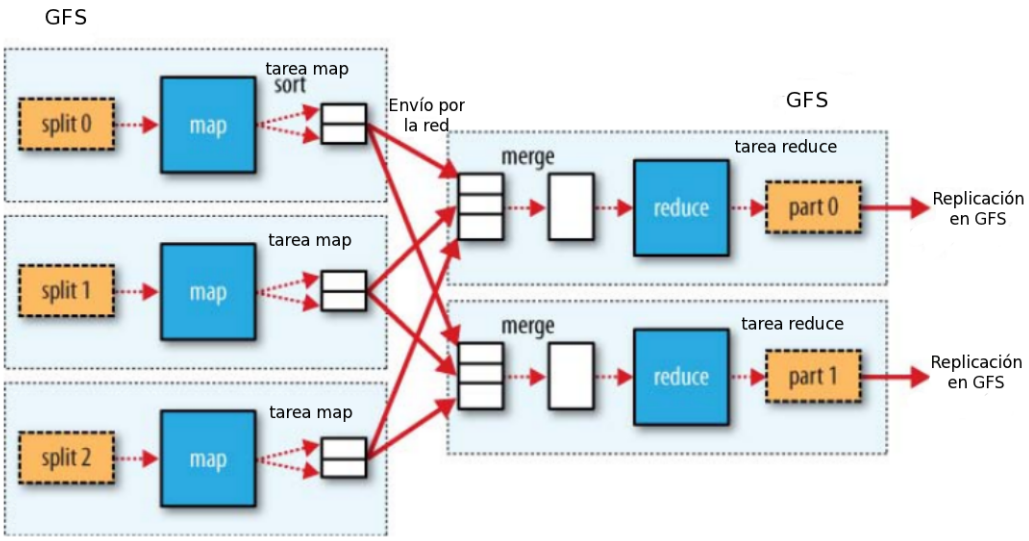


Figura 2.2: Flujo de datos en *MapReduce* [2].

pares intermedios $< clave, valor >$ con la misma clave, entonces se ejecuta una *función de partición* que se encarga de enviar a través de la red, todos los pares intermedios $< clave, valor >$ con una misma clave a una sola *tarea reduce*. Con esto se logra que las *tareas reduce* se empiecen a ejecutar tan pronto se tenga pares intermedios $< clave, valor >$ con una misma clave, sin necesidad de esperar a que todas las *tareas map* terminen de ejecutarse. Una *tarea reduce*, como se observa en la figura 2.2, está constituida por una operación *"merge"* y una réplica de la *función reduce*. Como se observa en la figura 2.1 y en la figura 2.2, una *tarea reduce* recibe los datos de diferentes nodos utilizando el recurso de red, entonces la operación *"merge"* se encarga de crear la lista de valores asociada con una misma clave y ejecuta la *función reduce*. Por último, la salida de cada *función reduce* se escribe en el sistema de archivos distribuido (*GFS*). Posteriormente, si la *tarea reduce* es asignada a otro conjunto de pares intermedios $< clave, valor >$ asociados con una misma clave, entonces los recibe y realiza el mismo procedimiento, en caso contrario finaliza su ejecución. El número de conjuntos de pares intermedios $< clave, valor >$ con diferente clave que procesa cada *tarea reduce* depende del número de réplicas de *tareas reduce* que se especifique [1].

La coordinación de las *tareas map y reduce* se lleva acabo a través de un proceso

llamado *master* (ver figura 2.1). Este proceso se encarga de coordinar el balanceo de carga y tolerancia a fallas. El *master* realiza el balanceo de carga de tal forma que los nodos más rápidos (con un mejor CPU y/o mayor cantidad de memoria) ejecutan más *tareas map y reduce* que los nodos más lentos. Por otra parte, la tolerancia a fallas lo realiza de la siguiente manera: Una *tarea map o reduce* puede estar en dos estados: en ejecución o finalizado. Si una *tarea map o reduce* está en ejecución y el nodo donde se está ejecutando falla (es decir no responde al master), entonces se vuelve a ejecutar la tarea respectiva en otro nodo. Si una *tarea map* está en estado finalizado y el nodo donde se ejecutó falla, entonces también se vuelve a ejecutar la *tarea map* en otro nodo debido a que su salida se encuentra en el disco local del nodo que falló. Si una *tarea reduce* está en estado finalizado y el nodo donde se ejecutó falla, la *tarea reduce* no se vuelve a ejecutar debido a que su salida se almacenó en el sistema de archivos distribuido. Cuando el proceso *master* falla, el ambiente *MapReduce* se corrompe y todos los *programas mapreduce* se cancelan [1].

2.3. Hadoop

Hadoop es una implementación de software libre de *MapReduce y GFS*. Fue creado por Doug Cutting en el año 2005, un año después de la implementación de *MapReduce de Google*. *Hadoop* surge en respuesta a la problemática que tenía Cutting para almacenar y procesar 1 billón de páginas indexadas recogidas por un crawler del buscador web Apache Nutch creado por él mismo. Si se almacenaba y procesaba esa cantidad de datos con las soluciones *Datawarehouse* que existian, el costo se aproximaba a 1 millón de doláres con un costo de operación de 30,000 doláres mensuales. Con *Hadoop* se disminuyeron los costos de compra y mantenimiento ya que se ejecuta en un clúster conformado por nodos de uso general [2].

La figura 2.3 muestra los componentes de *Hadoop* y algunos subproyectos sobre *Hadoop* [2]. Los componentes de *Hadoop* son:

- *MapReduce*, es una implementación de software libre del *MapReduce* de *Google*.

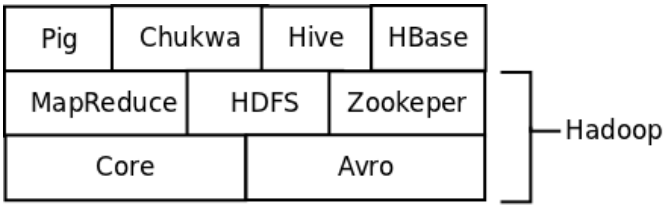


Figura 2.3: Componentes y subproyectos de Hadoop [2].

- *HDFS*, del acrónimo *Hadoop Distributed File System*, es un sistema de archivos distribuido que trabaja en coordinación con *MapReduce*. Es una implementación de software libre de *GFS*.

- Zookeeper, es un servicio de coordinación para desarrollar sistemas distribuidos. Proveé primitivas como candados distribuidos.

- Core, es un conjunto de componentes e interfaces, operadores de entrada y salida para sistemas de archivos distribuidos.

- Avro, permite la persistencia de los datos en el sistema de archivos distribudo a través de estructuras, mecanismos de serialización de datos y llamadas a procedimientos remotos.

Alrededor de *Hadoop* existen otros subproyectos que lo utilizan como base para brindar otros servicios, por ejemplo:

- Pig, es un lenguaje de programación de alto nivel que permite, a través de sentencias parecidas al *SQL*, manejar facilmente grandes volúmenes de datos.

- Chukwa, es un sistema de colección y análisis de datos distribuidos.

- *Hive,* es un *datawarehouse* distribuido. *Hive* administra los datos almacenados en *HDFS*, a través de una interface de usuario parecida al *SQL*. Las consultas SQL son transformadas por el motor de ejecución en *trabajos map_reduce*. *Hive* es presentado en el capítulo 3.

- HBase, es una base de datos distribuida sobre *Hadoop*. Está orientado para facilitar la construcción y manipulación de *tablas* con billones de registros y millones de columnas.

2.3.1. Arquitectura de *Hadoop*

Hadoop puede ejecutarse en uno o más nodos. En ambos casos, su funcionamiento se basa en la ejecución de cinco procesos que se comunican bajo el modelo cliente/servidor. Los procesos son [2] [3]:

- *JobTracker*
- *TaskTracker*
- *NameNode*
- *DataNode*
- *Secondary NameNode*

Los procesos *JobTracker* y *TaskTracker* implementan el modelo *MapReduce*, mientras que los procesos *NameNode*, *DataNode* y *Secondary NameNode* implementan el sistema de archivos distribuido *HDFS*.

La figura 2.4 muestra la arquitectura cliente-servidor del ambiente *MapReduce* y del sistema de archivos paralelo y distribuido *HDFS* en *Hadoop*. Los procesos *JobTracker* y *NameNode* son los procesos servidores, mientras que los procesos *TaskTracker* y *NameNode* son los procesos clientes. El proceso *Secondary NameNode* es un proceso auxiliar del *HDFS*, posteriormente se explicará su funcionamiento.

Hadoop permite a los usuarios especificar los nodos servidores y clientes. Permite también especificar en que nodos servidores ejecutar el proceso *JobTracker*, el proceso *NameNode* y el proceso *Secondary NameNode*. En clústers grandes se recomienda ejecutar cada proceso en un nodo servidor diferente, mientras que en clústers pequeños se recomienda ejecutarlos en un solo nodo servidor donde no se ejecute algún proceso

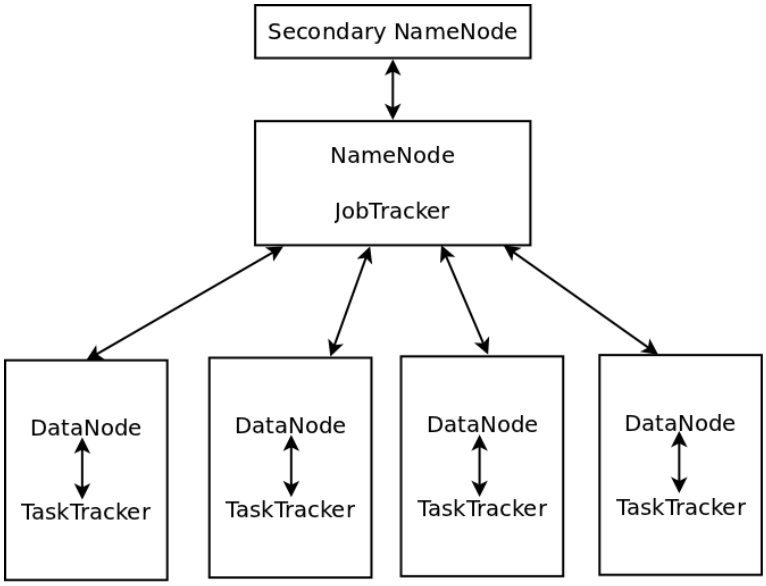


Figura 2.4: Topología de un clúster *Hadoop* [3].

cliente. Por otra parte, en cada uno de los nodos clientes se ejecuta un proceso *TaskTracker* y un proceso *DataNode*.

El proceso *JobTracker* recibe los *programas mapreduce* del usuario, crea y asigna las *tareas map y reduce* a los procesos *TaskTracker*. Posteriormente, mantiene comunicación con dichos procesos para dar seguimiento al avance de ejecución de cada una de las *tareas map y reduce*. Si el proceso *JobTracker* falla se corrompe el ambiente *MapReduce* y no se pueden ejecutar los *programas mapreduce*.

Los procesos *TaskTracker* se encargan de ejecutar las *tareas map y reduce* que han sido asignadas por el *JobTracker* y reportan su avance de ejecución al mismo. Aunque solo se ejecuta un *TaskTracker* por nodo cliente, cada *TaskTracker* genera múltiples Máquinas Virtuales de Java (del inglés *Java Virtual Machine o JVM*) para ejecutar una *tarea map* o una *tarea reduce* de manera paralela [2] [3], ver figura 2.5.

El proceso *NameNode* mantiene el árbol de directorios y archivos del sistema de archivos *HDFS*. Conoce en que nodos se ubican todos los *splits* de cada archivo y demás metadatos relacionados. Está información no es persistente, se construye con ayuda de los *DataNodes* cuando inicia el sistema. Particiona los archivos en *splits* (por defecto con un tamaño de 64MB cada *split*, aunque puede ser configurado por el

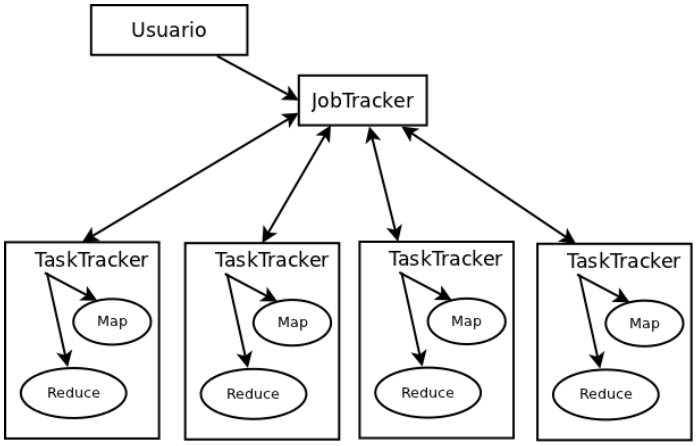


Figura 2.5: Interacción entre el *JobTracker* y *TaskTracker* [3].

usuario) y distribuye los *splits* en los *DataNodes* a quiénes les ordena la replicación correspondiente. Si un nodo que ejecuta un proceso *DataNode* falla, ordena a los otros *DataNodes* realicen la réplica de los *splits* que se ubican en dicho nodo para mantener el factor de replicación (por defecto 3, aunque puede ser configurado por el usuario). Si el *NameNode* falla se corrompe el sistema de archivos *HDFS*.

Los procesos *DataNodes* se encargan de realizar las operaciones de entrada/salida en el sistema *HDFS*. Mantienen comunicación con el *NameNode* para reportar donde se localizan los *splits* y recibir información para crear, mover, leer o eliminar *splits*. Por otra parte, se comunican entre ellos para realizar la réplica de los datos.

El proceso *Secondary NameNode (SNN)* es un proceso auxiliar del sistema de archivos *HDFS*. Recoge de manera períodica los metadatos del *NameNode* con un interválo de tiempo que se define en la configuración del cluster. Permite minimizar el tiempo de recuperación y evitar la pérdida de datos cuando el sistema de archivos *HDFS* queda inhabilitado debido a una falla del *NameNode*. Para esto, se necesita la intervención del administrador de *Hadoop* para reconfigurar el clúster y hacer que se utilice el *SNN* como *NameNode* [2] [3].

La figura 2.6 muestra la interacción entre los procesos *NameNode y DataNode*. Se observa que se tienen dos archivos de datos, uno en el directorio *HDFS* /user/alexis/ y otro en el directorio /user/buenabad/. El archivo data1 fue dividido en 3 *splits* representados por la numeración 1,2,3 y el archivo data2 fue divido en 2 *splits*

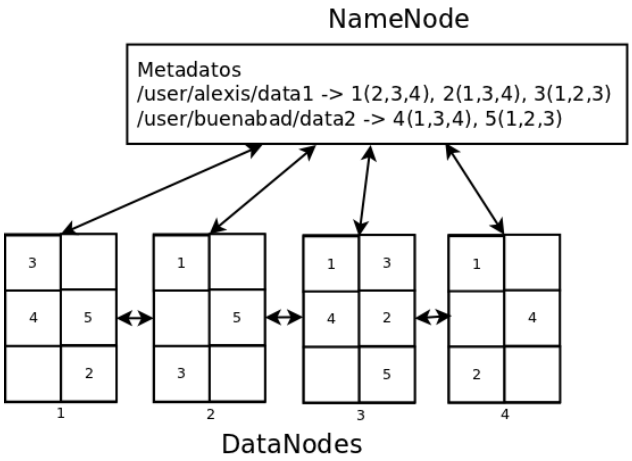


Figura 2.6: Interacción de *NameNode* y DataNodes en HDFS [3]. Los números entre paréntesis corresponden a los *ids* de los *DataNodes*.

representados por la numeración 4,5. El proceso *NameNode* mantiene los metadatos del sistema *HDFS* que contiene la información de los directorios, en cuantos *splits* fue divido un archivo, donde se encuentra cada *split*, etcétera. Por otra parte, los procesos *DataNodes* se comunican entre ellos para realizar la réplica de datos, por ejemplo, del *splits* 3 se tienen 3 réplicas una en el nodo 1, otra en el nodo 2 y una última en el nodo 3. Esto asegura, que si un *DataNode* falla o es inaccesible en la red, se tenga otra copia del mismo *split* en otro nodo y se pueda leer.

2.3.2. Configuración de *Hadoop*

Hadoop permite configurar el ambiente de ejecución de *MapReduce* y el sistema de archivos distribuido *HDFS*. La configuración de *Hadoop* determina el modo de ejecución y el desempeño de los *programas mapreduce*.

La configuración de *Hadoop* se realiza a través de archivos .xml y archivos de texto. La estructura interna de un archivo de configuración .xml se observa en el código 2.2.

```
<configuration>
        <property>
                <name> nombre de la propiedad </name>
                <value> valor de la propiedad </name>
        </property>
        ...
        <property>
                <name> nombre de la propiedad </name>
                <value> valor de la propiedad </name>
        </property>
</configuration>
```

Código 2.2: Estructura interna de un archivo de configuración de Hadoop .xml

Los archivos de configuración .xml de *Hadoop* por defecto son: *core-default.xml, mapred-default.xml y hdfs-default.xml*. Cada archivo mantiene las configuraciones por defecto que se realizan para el entorno en general de *Hadoop*, el ambiente *MapReduce* y el sistema de archivos *HDFS* respectivamente. Si el usuario desea realizar alguna configuración en particular, se recomienda crear o modificar los archivos core-site.xml, mapred-site.xml y hdfs-site.xml que se ubican en la carpeta conf.

```
<configuration>
        <property>
                <name> fs.default.name </name>
                <value> hdfs://nodo servidor:puerto </name>
        </property>
</configuration>
```

Código 2.3: Estructura interna del archivo de configuración core-site.xml

En el archivo *core-site.xml* se pueden configurar, entre otros aspectos, el sistema de archivos distribuido a utilizar, a través de la propiedad *fs.default.name*. Para especificar que se va a ejecutar el sistema de archivos *HDFS*, la propiedad se debe de configurar de la manera como se observa en el código 2.3. En el mismo código los parametros *nodo servidor y puerto* es el nodo servidor y el puerto donde se ejecutará el

proceso *NameNode* del sistema de archivos *HDFS*.

En el archivo *mapred-site.xml* se puede configurar, entre otros aspectos, el nodo y el puerto donde se va a ejecutar el proceso *JobTracker*, a través de la propiedad *mapred.job.tracker*. Además, se puede configurar la cantidad de memoria heap que pueden utilizar las *tareas map y reduce*, a través de la propiedad *mapred.child.java.opts*. La memoria heap es el área de memoria dinámica donde se cargan los *programas mapreduce* y los datos a procesar. Por defecto, se asignan 200 Megabytes a cada *tarea map o reduce*. Ver código 2.4.

```
1  <configuration>
2          <property>
3                  <name> mapred.job.tracker </name>
4                  <value> nodo servidor:puerto </name>
5          </property>
6          <property>
7                  <name> mapred.child.java.opts </name>
8                  <value> -Xmx200m </name>
9          </property>
10 </configuration>
```

Código 2.4: Estructura interna del archivo de configuración mapred-site.xml

En el archivo *hdfs-site.xml* se puede especificar, entre otros aspectos, el tamaño de cada *split* de un archivo, a través de la propiedad *dfs.block.size* (por defecto el valor es de 67108864 bytes que corresponden a 64MB). Por otra parte, también se puede configurar el factor de replicación de los splits (por defecto el valor es de 3), a través, de la propiedad *dfs.replication*. Ver código 2.5.

```
1  <configuration>
2          <property>
3                  <name> dfs.block.size </name>
4                  <value> 67108864 </name>
5          </property>
6          <property>
7                  <name> dfs.replication </name>
8                  <value> 3 </name>
9          </property>
10 </configuration>
```

Código 2.5: Estructura interna del archivo de configuración hdfs-site.xml

Los archivos de texto masters y slaves son los archivos de texto de configuración de *Hadoop*. En estos archivos se especifican correspondientemente los nodos servidores y clientes en la arquitectura cliente/servidor del ambiente *MapReduce* y del sistema de archivos *HDFS*. Si se desea ejecutar a *Hadoop* en un solo nodo, entonces en ambos archivos se especifica el nodo localhost. En caso contrario, si se desea ejecutar a *Hadoop* en un clúster, entonces se especifican los nodos correspondientes en cada archivo.

2.4. Ejemplos

En *Hadoop*, los *programas mapreduce* se pueden escribir en varios lenguajes de programación como Java, Python, C++, Perl, PHP, entre otros. Por defecto, *Hadoop* ejecuta *programas mapreduce* escritos en Java a través de la instrucción en línea de comandos:

```
hadoop jar <jar> [clasePrincipal] args ...
```

Donde $< jar >$ es el archivo comprimido que contiene el *programa mapreduce* escrito en java.

Sin embargo, *Hadoop* permite la ejecución de *programas mapreduce* escritos en cualquier otro lenguaje de programación que no sea Java, a través de la utilidad

Hadoop Streaming que está incluida en *Hadoop*. Para ejecutar un *programa mapreduce* con *Hadoop Streaming* se utiliza la siguiente instrucción en línea de comandos:

```
hadoop jar contrib/hadoop-streaming.jar
          -mapper funcion_map -reducer funcion_reducer
          -input input_dir
          -output output_dir
```

Hadoop Streaming replica los programas especificados como $funcion\_map$ y $funcion\_reduce$ en *tareas map y reduce* respectivamente. Utiliza la entrada y salida estándar para recuperar y enviar los datos a procesar a las *funciones map y reduce*.

A continuación se presenta un *programa mapreduce* escrito en Java y Python que cuenta el número de veces que se repite las palabras en varios archivos. Desde ahora este *programa mapreduce* se refirá como "programa cuenta".

2.4.1. *Programa mapreduce escrito* en *Java*

En Java, un *programa mapreduce* se constituye básicamente de tres clases: Una clase principal que contiene la *función main*, una clase que contiene la *función map* y una clase que contiene la *función reduce*.

En general, la función main configura el ambiente de ejecución de *MapReduce* y los *trabajos mapreduce* a ejecutar. Un *trabajo mapreduce* es la unidad de ejecución del ambiente *MapReduce*. Está constituido por *una función map, una función reduce y los datos a procesar*. Un *programa mapreduce* puede consistir de uno o más *trabajos mapreduce*.

El código 2.6 presenta la clase principal con la *función main* del "programa cuenta" presentado antes en pseudocódigo. Observese que se crea un objeto de la *clase Configuration*, con el que se puede realizar y obtener configuraciones específicas del ambiente *MapReduce* para el "programa cuenta". Con el *objeto configuration* se puede especificar el valor de alguna propiedad como se realiza en los archivos core-site.xml, mapred-site.xml o hdfs-site.xml y el ambiente *MapReduce* se configuraría

```
1  public class CountWords {
2     public static void main(String[] args) throws Exception{
3           // Configuracion del ambiente mapreduce
4           Configuration conf = new Configuration();
5
6           /* Configuración del trabajo mapreduce */
7             Job job = new Job(conf, "wordcount");
8
9             // Configuración de las clases que contiene la función
10            // map y reduce del usuario
11            job.setMapperClass(Map.class);
12            job.setReducerClass(Reduce.class);
13
14            // Configuración del tipo de clave y valor que emite
15            // la función map
16            job.setOutputKeyClass(Text.class);
17            job.setOutputValueClass(IntWritable.class);
18
19            // Configuración de los tipos de archivos de entrada y salida
20            job.setInputFormatClass(TextInputFormat.class);
21            job.setOutputFormatClass(TextOutputFormat.class);
22
23            // Configuración de los directorios de entrada y salida
24            FileInputFormat.addInputPath(job, new Path(args[0]));
25            FileOutputFormat.setOutputPath(job, new Path(args[1]));
26
27            // Envia el trabajo mapreduce para su ejecución en MapReduce
28            // y espera hasta que finalice
29            job.waitForCompletion(true);
30            // Otra opción es job.submit();
31          /* Fin de configuración del trabajo mapreduce */
32
33          /* Se pueden agregar otros objetos job para ejecutar otros
34           * trabajos mapreduce de manera
35           * secuencial (si se utiliza waitForCompletion(true))
36           * o paralela (si se utiliza submit()).
37           * Por ejemplo:
38
39           * Job job2 = new Job(conf, "nombre_trabajo_2");
40           * job2.setMapperClass(Map2.class);
41           * job2.setReducerClass(Reducer2.class);
42           * ... configuración del trabajo
43           * job2.submit();
44
45           * Job job3 = new Job(conf, "nombre_trabajo_3");
46           * ...
47           */
48    }
49 }
```

Código 2.6: Clase principal para ejecutar el trabajo *MapReduce* en *Hadoop* para contar el número de veces que se repite una palabra en varios archivos.

especificamente para este programa. En el código 2.6 se mantienen las configuraciones por defecto. Posteriormente, se crea un objeto de la clase *Job* que permite configurar un *trabajo mapreduce*. Algunas de las configuraciones que se realizan en un *trabajo mapreduce* son: la especificación de las clases que contienen las *funciones map y reduce*; el tipo de datos de la clave y el valor que emite la *función map*; los tipos de los archivos de entrada y salida; los directorios de entrada y salida; entre otras configuraciones.

Una vez que el *trabajo mapreduce* ha sido configurado se envía a ejecutar al ambiente *MapReduce*. Existen dos maneras de ejecutar el *trabajo mapreduce*: esperando hasta que finalice, o enviarlo a ejecución y continuar con las siguientes instruciones. Para esperar hasta que el *trabajo mapreduce* finalice se utiliza el método *waitForCompletion* con el parámetro en *True*. Para ejecutar el *trabajo mapreduce* sin esperar su finalización, se utiliza el método *submit*.

Aunque en este ejemplo no se observa, se pueden crear otros *trabajos mapreduce* creando un nuevo objeto de la *clase Job*. Así mismo, se pueden crear otras clases que contengan otras funciones map y reduce que esten relacionadas con otros *trabajos mapreduce*, con esto se logra ejecutar una serie de *trabajos mapreduce* de manera secuencial (si se utiliza el método *waitForCompletion*) o paralela (si se utiliza el método submit).

La clase que contiene a la función map del "programa de cuenta" se muestra en el código 2.7. Observe que la clase implementa la *Interfaz Mapper*. Una interfaz en Java contiene la declaración de los métodos sin especificar su implementación. La implementación de dichos métodos se realiza en la clase que implementa dicha interfaz. Por tal motivo, al implementar la *Interfaz* Mapper se debe de implementar la función map, esta función recibe tres parámetros que son: una clave, un valor y un contexto. El contexto permite configurar los pares $< clave, valor >$ que se emiten.

La *función map* se ejecuta de forma iterativa por cada línea de cada archivo de texto recibiendo como clave el desplazamiento dentro del archivo de entrada y como valor la línea del archivo. Por cada línea de un archivo de entrada, la *función map*

```
1  public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
2
3    private Text word = new Text();     // Guarda una palabra
4
5    public void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException        {
6      String line = value.toString();
7      // Se obtiene cada palabra de la línea
8          StringTokenizer tokenizer = new StringTokenizer(line);
9          // Cada palabra de cada línea se envía a la función reduce como
               clave y
10         // como valor se envía un 1.
11         while(tokenizer.hasMoreTokens()){
12           // Obtiene una palabra
13           word.set(tokenizer.nextToken());
14           // Emite la clave y el valor de la función map
15           context.write(word, new IntWritable(1));
16         }
17   }
18 }
```

Código 2.7: Clase map en *Hadoop* para contar el número de veces que se repite una palabra en varios archivos.

obtiene las palabras de la línea y por cada palabra emite como clave la palabra y como valor un uno. Posteriormente, el ambiente *MapReduce* recoge todos los pares $< clave, valor >$ que emite la *función map* (o las *funciones map* si se ejecuta en dos o más nodos) y agrupa los valores con una misma clave en listas. La clave y la lista de valores asociada se envía a la función reduce. El código 2.8 presenta la clase que contiene a la función reduce del "programa de cuenta". Observe que la clase implementa la *interfaz Reducer*, razón por la cuál, implementa la función reduce. Esta función recibe tres parámetros: la clave, la lista de valores y el contexto *MapReduce*.

En este caso, la *función reduce* se ejecuta de manera iterativa por cada palabra diferente que emite la *función map*. Por cada palabra se obtiene una lista de unos que se suman para conocer el número de veces que se repite una palabra en varios archivos. Por último, se emite como clave la palabra y como valor la cuenta.

Para ejecutar el "programa de cuenta" escrito en Java en *Hadoop* se deben de empaquetar todas las clases en un *Jar*, por ejemplo *cuenta.jar*, entonces la ejecución del programa se realiza de la siguiente manera:

```
1  public class Reduce extends Reducer<Text, IntWritable, Text, IntWritable
       > {
2    public void reduce(Text key, Iterable<IntWritable> values, Context
       context) throws IOException, InterruptedException {
3        int sum = 0;
4        // Suma los unos de cada palabra
5        for(IntWritable val: values){
6          sum += val.get();
7        }
8        // Emite el par <palabra, cuenta>
9          context.write(key, new IntWritable(sum));
10   }
11 }
```

Código 2.8: Clase reduce en *Hadoop* para contar el número de veces que se repite una palabra en varios archivos.

```
1  #!/usr/bin/env python
2   import sys
3  # La entrada viene de la entrada estándar
4  for line in sys.stdin:
5     # divide la línea en palabras
6      words = line.split()
7      # Por cada palabra encontrada
8      for word in words:
9          # Imprime en la salida estándar una cadena
10         # constituida por la palabra, un carácter tabulador y un uno
11         print '%s\t%s' %(word, 1)
```

Código 2.9: Función map del *programa mapreduce* que cuenta las veces que las palabras se repiten en varios archivos en Python (map.py).

```
hadoop jar cuenta.jar CountWords inputPathInHDFS outputPathInHDFS
```

Nótese que se pasó un directorio y *MapReduce* va a leer todos los archivos del directorio. Se puede enviar un solo archivo.

2.4.2. *Programa mapreduce* escrito en *Python*

El *"programa cuenta"* se puede implementar a través de un *programa mapreduce* escrito en *Python*. El código 2.9 presenta la *función map* y el código 2.10 presenta la *función reduce* en *Python*.

Observe que la *función map* en Python es idéntica a la *función map* en Java.

```
#!/usr/bin/env python
from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None
# La entrada viene de la entrada estándar
for line in sys.stdin:
    # Obtiene la entrada y la cuenta que se envia de map.py
    word, count = line.split('\t', 1)
    # convierte count (actualmente un string) a entero
    try:
        count = int(count)
    except ValueError:
        # Si no fue un número se ignora el error
        # y se obtiene la siguiente palabra
        continue

    # Si es la misma palabra, continuar con la cuenta
    # en caso contrario, imprimir la palabra con su cuenta y actualizar
        la palabra
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # Escribe el resultado a la salida estándar
            print '%s\t%s' %(current_word, current_count)
        current_count = count
        current_word = word
# Si es necesario en la última palabra
if current_word == word:
    print '%s\t%s' %(current_word, current_count)
```

Código 2.10: Función reduce del *programa mapreduce* que cuenta las veces que las palabras se repiten en varios archivos en Python (reduce.py).

La *función map* recibe los datos línea por línea de la entrada estándar, obtiene las palabras de cada línea y por cada palabra imprime en la salida estándar una cadena *"palabra \t 1"* donde la palabra es la clave y el 1 es el valor.

Por otra parte, la *función reduce* en Python para el "programa cuenta" es un poco diferente a la *función reduce* en Java. En vez de recibir una lista de 1's asociada a una misma clave, recibe todas las cadenas *"palabra \t 1"* con la misma clave generadas por las *tareas map*. Es decir con la misma palabra. Como las cadenas están ordenadas por palabra, entonces se realiza la suma de los unos hasta que se encuentra una palabra diferente. Cuando este es el caso se imprime la cadena *"palabra \t cuenta"* que especifica el número de veces que una palabra se encuentra en varios archivos.

Para ejecutar el "programa cuenta" escrito en Python en *Hadoop* se debe de ejecutar la siguiente instrucción:

```
hadoop jar contrib/hadoop-streaming.jar
           -mapper map.py -reducer reduce.py
           -input inputPathInHDFS -output outputPathInHDFS
```

2.5. Resumen

Este capítulo presentó a *MapReduce*, un modelo de programación y ambiente de ejecución creado por *Google* para el procesamiento de grandes volúmenes de datos de manera paralela y distribuida. Los problemas de balanceo de carga y tolerancia a fallas que conlleva el procesamiento paralelo y distribuido los administra *MapReduce* de manera transparente para el usuario. El usuario solo tiene que especificar pares de *funciones map y reduce*.

La *función map* lee cada registro de los archivos de entrada a través de un par $< clave, valor >$, y por cada registro se realiza lo que en la *función map* se especifica y se emite un par intermedio $< clave, valor >$. Posteriormente, *MapReduce* se encarga de recoger todos los pares que son emitidos por la función *map* y todos los valores con una misma clave intermedia se agrupan en una lista y se envían a la función

reduce. *MapReduce* se asegura que todos los valores con una misma clave intermedia se envien a una misma *función reduce*.

La función reduce se ejecuta por cada clave intermedia generada por la *función map*, recibe un par $< clave, lista\ de\ valores\ asociados\ con\ la\ clave >$ y procesa cada elemento de la lista de valores recibida. Al final, generalmente emite un par $< clave, valor >$.

Las *funciones map y reduce* se ejecutan en el ambiente *MapReduce* de manera paralela y distribuida a través de *tareas map y reduce*. El número de tareas map a ejecutar depende del número de *splits* en que está dividido el archivo a procesar en el sistema de archivos distribuido. En cambio, el número de *tareas reduce* a ejecutar es especificado por el usuario.

Hadoop es una implementación de software libre de *MapReduce* y un sistema de archivos distribuido llamado *HDFS* por el acrónimo en inglés *Hadoop Distributed File System*. La implementación de *MapReduce* y *HDFS* en *Hadoop* se realiza a través de una arquitectura cliente/servidor. *Hadoop* tiene la versatilidad de poder ejecutar ejecutar *programas mapreduce* escritos en cualquier lenguaje. Por defecto, ejecuta *programas mapreduce* escritos en Java, aunque puede ejecutar *programas mapreduce* escritos en Python, Perl, C++, PHP, entre otros, a través de la utilidad *Hadoop Streaming* que aprovecha la entrada y salida estándar de los programas para obtener y enviar los pares $< clave, valor >$ entre las *funciones map y reduce*.

Capítulo 3

Hive

En el Capítulo 2 vimos que Mapreduce no sólo oculta los detalles de la programación paralela, sino que además lleva a cabo tolerancia a fallas y balance de carga de manera transparente al usuario. Esta funcionalidad es esencial para procesar grandes cantidades de datos en un tiempo razonable. La tolerancia a fallas, por ejemplo, evita que en caso de que uno o más nodos fallen, no se tenga que ejecutar una aplicación de nuevo desde el principio. A pesar de estos beneficios, el desarrollo de aplicaciones mapreduce complejas (compuestas de uno o más pares de funciones map y reduce) no es simple. Esto se debe en parte a que el modelo de programación de mapreduce no es de uso general todavía, pero también a que no es tan intuitivo como otros modelos de programación como memoria compartida y paso de mensajes.

Por lo anterior, se han desarrollado otras capas de software sobre mapreduce que ofrecen un modelo de programación más simple e intuitivo. En particular, existen esfuerzos para permitir el procesamiento de grandes volúmenes de información por medio de sentencias SQL (*OLAP*) en lugar de escribir *programas mapreduce*.

Un primer enfoque, adaptó al *ambiente MapReduce* para poder implementar de manera más sencilla las ideas de una *sentencia SQL* dentro de un *programa mapreduce*. Ejemplos de este enfoque son los proyectos: MapReduce-Merge [19] y Hadoop++ [20].

Pig es un proyecto que toma las ventajas de las sentencias SQL y de un *programa mapreduce*. El usuario ya no piensa en términos de *funciones map y reduce*, sino

escribe programas cuyo lenguaje de programación son sentencias muy parecidas a las sentencias SQL. De esta manera, los programas se realizan con mayor facilidad y son más fáciles de mantener [21].

Otro enfoque para facilitar el procesamiento de datos en *MapReduce* es a través de un datawarehouse. Este enfoque construye de manera lógica una infraestructura de base de datos, tablas, etcétera, que se almacenan en un sistema de archivos distribuido. El procesamiento de datos se realiza a través de sentencias SQL o sentencias parecidas al SQL que posteriormente se transforman en *trabajos mapreduce*. Algunos proyectos que siguen este enfoque son: Dremel desarrollado por Google [22]; *Hive* desarrollado por Facebook [4]; Dryad desarrollado por Microsoft [23]; Cheetah desarrollado por Turn [24]; HadoopDB desarrollado por la *universidad de Yale*. HadoopDB tiene la peculiaridad de brindar una infraestructura datawarehouse (base de datos, tablas, etcétera) física a través de un Sistema de Gestión de Base de Datos (SGBD) como PostgreSQL. Para procesar los datos de manera distruida, utilizan a *MapReduce* como medio de comunicación entre los *SGBD* [25].

En nuestro caso, se decide seguir el enfoque de un datawarehouse sobre *MapReduce*, debido a que consideramos es la manera más sencilla y universal de procesar datos, a través de una infraestructura de base de datos y sentencias SQL. Además, varias empresas utilizan un datawarehouse y sentencias SQL (OLAP) para procesar grandes volúmenes de datos a través de sistemas de soporte a decisiones, sistemas de minería de datos, entre otros. Por lo tanto, la migración de un datawarehouse comercial que utiliza base de datos comerciales a uno que utiliza *MapReduce* sería más sencilla. En particular, se decide utilizar *Hive* debido a que es una implementación de código abierto que está implementado sobre *Hadoop*, existe una comunidad oficial de desarrollo y existen empresas como Cloudera que brindan soporte técnico [16].

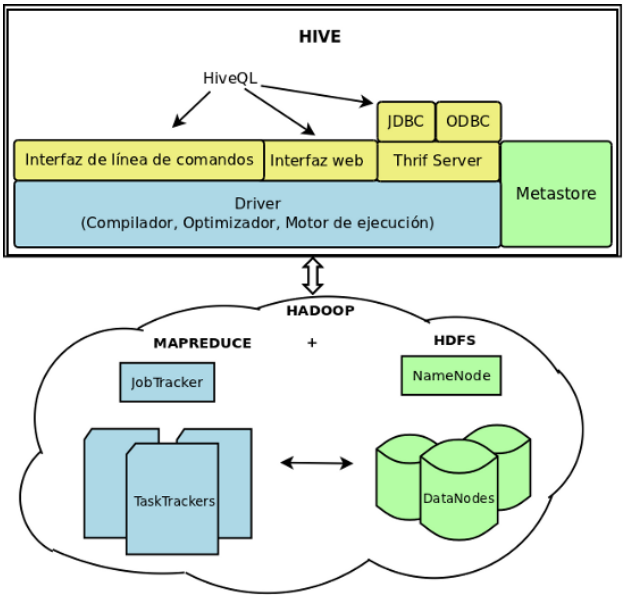


Figura 3.1: Arquitectura de hive [4].

3.1. Arquitectura de Hive

Hive es un datawarehouse de código abierto implementado sobre *Hadoop*. Fue desarrollado por Facebook con el objetivo de eficientar el acceso a los datos y optimizar su modelo de negocios [4]. El procesamiento de los datos se realiza a través de sentencias parecidas al SQL llamadas sentencias *HiveQL*. Estas sentencias pueden involucrar *scripts MapReduce* dentro de sí mismas, de manera que es posible especificar el procesamiento de datos estructurados y no estructurados.

Hive está constituido por 3 grupos de componentes como se observa en la figura 3.1: *Metastore, interfaces de usuario y Driver*.

El componente *Metastore* mantiene la infraestructura del datawarehouse y guarda información relacionada con las estructuras de *Hive*. Por ejemplo, guarda la ubicación de las bases de datos y tablas, los permisos de los usuarios sobre las base de datos y tablas, los tipos de datos de las columnas, etcétera. Esta información se le conoce como metadatos. Los metadatos se almacenan en un sistema de archivos local o en un SGBD como MySQL o PostgreSQL. Si se desea almacenar los metadatos en un *SGBD*, entonces *Hive* utiliza un *framework* llamado DataNucleus que convierte un objeto que contiene los metadatos en registros de una tabla o relación y viceversa.

Las interfaces de usuario de *Hive* son: Línea de comandos, interfaz web y Thrif server. Las interfaces de línea de comandos y web permiten al usuario introducir directamente sentencias HiveQL. La interfaz *Thrif server* permite comunicar a *Hive* con otra aplicaciones desarrolladas por el usuario en cualquier lenguaje de programación como C, Java, PHP, etcétera. Estas aplicaciones se comunican con el *Thrif server* a través de un conector a base de datos conocido como *ODBC* (del inglés *Open DataBase Connectivity*). Si no existe un *ODBC* para un lenguaje de programación, entonces se puede crear uno ya que *Thrif server* brinda las herramientas para poder comunicarse con él. El conector específico para vincular un programa Java con *Thrif server* es *JDBC* (del inglés, *Java Database Connectivity*).

Por último, el *Driver* es un grupo de componentes que compilan las sentencias HiveQL en una secuencia de *trabajos mapreduce* que se ejecutan en *Hadoop*. Los componentes del Driver son: compilador, optimizador y motor de ejecución. El compilador se encarga de verificar si una sentencia HiveQL está bien formada y es coherente. Si lo es, entonces construye una representación gráfica de la sentencia HiveQL (ver sección 3.4). Posteriormente, el optimizador se encarga de optimizar dicha representación gráfica (ver sección 3.5). Por último el motor de ejecución se encarga de construir los *trabajos mapreduce* a partir de la representación gráfica optimizada y ejecuta dichos *trabajos mapreduce* en *Hadoop* (ver sección 3.6).

3.2. Estructuras del datawarehouse

Hive brinda una infraestructura de base de datos de manera lógica. Las estructuras que soporta son: Bases de datos, tablas, particiones, buckets, vistas, funciones e índices. Las primeras cuatro estructuras tienen una representación física en el sistema de archivos distribuidos *HDFS*. La figura 3.2 muestra la representación física de estas estructuras.

Una base de datos es una estructura que está conformada por un conjunto de tablas. En el sistema *HDFS* una base de datos es un directorio. Una tabla es una

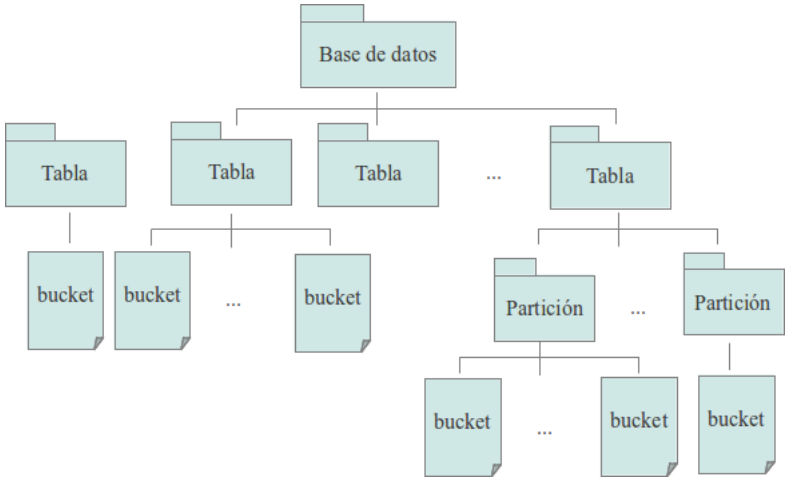


Figura 3.2: Representación física de las estructuras de *Hive* en el sistema de archivos *HDFS*.

estructura que contiene grandes cantidades de información de manera organizada. En el sistema *HDFS* una tabla es un directorio y si la tabla se crea dentro de una base de datos, entonces es un subdirectorio del directorio de la base de datos. Los registros de una tabla pueden estar divididos en particiones y/o buckets.

Una partición es una estructura que divide los registros de una tabla en función de alguna condición. Esta condición puede o no estar en función de alguna columna de la tabla. Por ejemplo, suponga que se tiene una tabla de empleados con las columnas *id, nombre, apellido parterno, apellido materno*. Suponga que se desea tener dividido a los empleados de acuerdo a su horario de salida. Entonces se crean varias particiones, donde cada partición contiene los empleados que corresponden a un horario de salida. Una partición tiene a los empleados que salen a las 6pm, otra partición tiene a los empleados que salen a las 7pm y así sucesivamente.

Un bucket es uno o varios archivos que guardan los registros de una tabla o partición. Cuando son varios archivos, cada archivo contiene los registros que se encuentran dentro de un rango de acuerdo a una columna de la tabla. Por ejemplo, la tabla empleados puede almacenar sus registros en un bucket o en varios buckets. Si se almacena en varios buckets, cada bucket almacena cierta cantidad de registros. Por ejemplo, un bucket puede almacenar los empleados con un id entre 1-100, el siguiente bucket almacena los empleados con un id entre 101-200 y así sucesivamente.

Una vista es una "tabla virtual" que almacena una consulta. Se dice que es una "tabla virtual" porque se puede manipular como una tabla, pero no lo es. En realidad los datos que devuelve no se encuentran almacenados físicamente como tabla, sino más bien es el resultado de aplicar la consulta almacenada [26].

Una función recoge un conjunto de parámetros, realiza un procesamiento determinado y regresa un valor. *Hive* permite el uso de funciones como *avg(columna), sum(columna)*, entre otras, pero también permite al usuario definir sus propias funciones [26].

Un índice es una estructura de datos que permite acceder a los registros de una tabla de una manera más rápida y organizada. Un índice en *Hive* actúa de igual manera que un índice en un diccionario: cuando se requiere buscar una palabra nos vamos al índice e identificamos la página donde comienzan las palabras que inician con la palabra buscada. Si se agrega o elimina una palabra, el índice se actualiza. De igual manera un índice en *Hive* permite identificar registros con mayor facilidad y si se agrega o elimina un registro se actualiza el índice [26].

3.3. HiveQL

El *lenguaje SQL* es un estándar ANSI/ISO que permite la definición, manipulación y control de las bases de datos, a través de sentencias declarativas que se agrupan en dos grandes grupos: Lenguaje de Definición de Datos (del inglés *Data Definition Language o DDL*) y Lenguaje de Manipulación de Datos (del inglés *Data Manipulation Language o DML*). El lenguaje DDL crea, modifica o elimina las estructuras de un SGBD (base de datos, tablas, funciones, etcétera). El lenguaje DML crea, modifica, consulta o elimina los datos.

HiveQL soporta el lenguaje DDL y DML de Hive. Es muy parecido al estándar SQL motivo por el cuál un usuario de otro SGBD pueden utilizar *Hive* con facilidad. En esta sección se presentan las diferencias que existen entre *HiveQL* y SQL.

3.3.1. *Lenguaje de Definición de Datos (DDL)*

El lenguaje DDL de SQL y *HiveQL* es casi el mismo, ambos soportan las sentencias: *create, drop, alter, rename, show, describe*. Estas sentencias se aplican en las mismas estructuras. La tabla 3.1 muestra las estructuras reconocidas por *Hive* con sus respectivas DDL soportadas. En [26] se detalla la sintaxis de cada sentencia DDL de *Hive*.

La única diferencia entre el lenguaje DDL de SQL y *HiveQL* es que *HiveQL* no soporta disparadores (*triggers*). Esto se debe a que los disparadores son procedimientos que se ejecutan cuando se cumplen una condición establecida al realizar una operación INSERT, DELETE y UPDATE en la base de datos. Como estas operaciones no se realizan con frecuencia en un datawarehouse como *Hive*, entonces no son necesarios.

3.3.2. *Lenguaje de Manipulación de Datos (DML)*

El estándar SQL soporta las siguientes sentencias DML: *INSERT, DELETE, UPDATE, SELECT.*

En *Hive*, existen operaciones de inserción, eliminación y modificación, pero trabajan diferente a las operaciones en un SGBD.

La operación de inserción en un SGBD se realiza registro por registro, mientras que en *Hive* la inserción de los datos se realiza por bloques.

La operación de eliminación en SQL elimina los registros asociados a una condición, mientras que en *Hive* no se permite la eliminación parcial de los datos debido a que por definición, un datawarehouse mantiene el historial de una organización. Si los datos se eliminan se deben de eliminar todos los registros de la tabla.

La operación de actualización en SQL actualiza los registros asociados a una condición, mientras que en un *Hive* no se permite la actualización de los registros que han sido almacenados en él, lo único que se permite es agregar nuevos registros

Estructura	Sentencia
Base de datos	create
	drop
	show
	describe
Tabla	create
	drop
	alter
	rename
	show
	describe
Partición	add
	drop
	alter
	show
Vistas	create
	drop
	alter
Funciones	create
	drop
	show
Índices	create
	drop
	show

Tabla 3.1: Estructuras y sentencias *DDL* soportadas en *HiveQL*.

a la tabla.

Por tales motivos, las sentencias *UPDATE y DELETE* no se soportan en *HiveQL*. Las operaciones de inserción, eliminación total y agregación de registros a una tabla, directorio *HDFS* o directorio local se realiza con la sentencia *INSERT*, la cual tiene una sintaxis diferente al SQL. En [26] se detalla la sintaxis de la sentencia *INSERT* en *HiveQL*.

Por otra parte, la sentencia SELECT en *HiveQL* tiene la misma sintaxis que en SQL para realizar consultas a tablas, vistas, subconsultas. Sin embargo, las subconsultas en *HiveQL* sólo se soportan en la cláusula *FROM* y toda subconsulta debe de tener un nombre. El número de subconsultas es arbitrario por lo que una subconsulta a su vez puede contener otra subconsulta [26].

3.4. Compilador de Hive

El *compilador de Hive* verifica que una *sentencia HiveQL* este bien formada y sea coherente, esta verificación lo realiza en dos fases: Análisis léxico y sintáctico, y verificación y análisis semántico.

La fase de análisis léxico y sintáctico se encarga de identificar las palabras reservadas del lenguaje *HiveQL* (select, insert, join, groupby, etcétera) y su correcta sintaxis. En otras palabras, se encarga de verificar que una sentencia *HiveQL* este bien formada. Para realizar la verificación, se apoya de *Antlr*, un framework parecido a *Lex & Yacc* utilizado para construir reconocedores, interprétes, compiladores y traductores a partir de una descripción gramatical. Al final, si una *sentencia HiveQL* está bien formada, *Antlr* devuelve un *Árbol Sintáctico Abstracto (del inglés Abstract Syntax Tree o AST)* que es una representación gráfica de una sentencia *HiveQL*. En caso contrario, se cancela el procesamiento de la sentencia *HiveQL* y se devuelve un error léxico.

La fase de verificación de tipos y análisis semántico se encarga de evaluar la gramática y el sentido de una sentencia *HiveQL* en el *AST*. También verifica la compatibilidad de los tipos en las expresiones, para lo cuál recupera del *metastore* los tipos de las columnas implicadas en dichas expresiones. Si se produce un error, se cancela el procesamiento de una sentencia *HiveQL* y se devuelve un error semántico. En caso contrario, se identifican las subconsultas de una consulta en el *AST* y por cada subconsulta que encuentra se crea un árbol llamado bloque de consulta o *Query Block (QB)*. Los *QB's* se enlanzan entre sí de tal manera que se hace visible el orden en que se deben de ejecutar las subconsultas. Los *QB's* representan graficamente a una sentencia *HiveQL* en función de operadores *HiveQL* (*select, join, where, group*, etcétera). Posteriormente, cada *QB* se convierte en una parte de un Grafo Aciclíco Dirigido [1] (del inglés *Directed Acyclic Graph o DAG*), de tal manera que cada

[1] Un grafo es un conjunto de vértices o nodos que están vinculados por enlaces o aristas. Se dice que un grafo es dirigido cuando las aristas indican una dirección. Un Grafo Aciclíco Dirigido es un grafo dirigido que no contiene ciclos.

operador *HiveQL* (*select, join, where, groupby,* etcétera) se convierte a un operador *Hive*(SelectOperator, JoinOperator, FilterOperator, GroupByOperator, etcétera) y se mantiene el orden de las subconsultas. La tabla 3.2 resúme la relación que existe entre algunos operadores *HiveQL* y los operadores de *Hive*. Cada operador *Hive* tiene asociada una implementación en *MapReduce*. En esta fase se configuran los operadores *Hive* y en la fase de generación del plan físico se ejecutan.

Cláusula HiveQL	**Operador Hive**
Select	SelectOperator
From	TableScanOperator
Where	FilterOperator
Función de agregación y agrupación Group By	GroupByOperator
Join	JoinOperator
Order By	OrderByOperator
Sort By	SortByOperator
	ReduceSinkOperator
	FileSinkOperator

Tabla 3.2: Relación entre *las cláusulas de una sentencias HiveQL* y *operadores Hive*.

Nótese que los operadores *Hive ReduceSinkOperator (RS)* y *FileSinkOperator (FS)* no tienen asociado un operador *HiveQL* debido a que sólo se utilizan en el ambiente *MapReduce*. El operador *RS* permite leer los registros del sistema de archivos distribuido *HDFS* y configura los pares intermedios $< clave, valor >$ que emiten las *tareas map*. El operador *FS* configura los pares $< clave, valor >$ que emiten las *tareas reduce* y los escribe en un archivo en el sistema de archivos distribuido *HDFS*.

3.5. Optimizador de Hive

El optimizador de *Hive* aplica consecutivamente una serie de transformaciones al *DAG* construido por el compilador. El objetivo es que al final de aplicar todas las optimizaciones se tenga, en principio, un *DAG* que al convertirlo en *trabajos mapreduce* tenga un mejor desempeño en *Hadoop*. Las *optimizaciones de Hive* se

```
1  create view subq1 as
2              select a.status, b.gender
3              from status_updates as a join profiles as b
4              on (a.userid = b.userid)
5              where a.ds='2012-08-10';
6
7  // consulta facebook
8  select subq1.gender, count(1)
9              from subq1
10             group by subq1.gender;
```

Código 3.1: Consulta de cambio de status de facebook por género.

basan en reglas. Es decir, se basan en expresiones condicionales *"Si [regla], entonces [transformación del DAG]"*. Cada optimización recibe el *DAG* transformado por la optimización anterior, evalúa si alguna regla de la optimización se cumple y si se cumple transforma al *DAG* en un *DAG "más optimizado"*.

Antes de describir las optimizaciones actuales de *Hive* conviene definir un caso de estudio que nos servirá para ejemplificar las optimizaciones. Este caso de estudio es una versión corta del caso de estudio propuesto en [4].

Caso de estudio: Suponga que *Facebook* guarda las actualizaciones de status que realiza cada usuario en una *tabla* llamada *status_updates* y por cada usuario tiene la información de su escuela y su género (masculino, femenino) en una *tabla* llamada *profiles*. La estructura de cada *tabla* es la siguiente:

```
status_updates(userid int, status string, ds string)
profiles(userid int, school string, gender int)
```

En la tabla *status_updates* la columna *userid* es el id del usuario, la columna *status* es el *status* que ha escrito el usuario y la columna *ds* es el día de la actualización.

En la tabla *profiles* la columna *userid* es el id del usuario, la columna *school* es la escuela del usuario y la columna *gender* es el género del usuario (masculino, femenino).

Suponga que Facebook desea conocer ¿Cuántas actualizaciones de *status* se realizan en un día determinado por cada género?. Las sentencias *HiveQL* que se muestran en el código 3.1 resuelven la pregunta. La vista *subq1* obtiene de la tabla

status_profiles los *status* de los usuarios que modificaron su *status* en el día *10-08-2012* y los relaciona con su género que obtiene de la tabla *profiles* (ver tabla 3.3). Posteriormente, la siguiente consulta agrupa los registros de la vista *subq1* por la columna género y por cada agrupación (género) cuenta el número de actualizaciones de status (ver tabla 3.4). A partir de ahora, esta consulta se referirá como consulta facebook.

a.status	b.gender
"¿Hola como estás?"	Femenino
"Vamos a jugar"	Masculino
"Por fin es viernes!!"	Femenino
"Vamos al cine"	Femenino
"De errores se aprende"	Masculino

Tabla 3.3: Ejemplo de resultado de la vista *subq1*. El alias de la tabla *status_updates* es 'a' y el alias de la tabla *profiles* es 'b'.

subq1.gender	cuenta de status actualizados
Femenino	3
Masculino	2

Tabla 3.4: Ejemplo de resultado de la consulta facebook.

La figura 3.3 muestra el *DAG* optimizado que genera *Hive* para la *consulta facebook*. Observe que el orden de las subconsultas se mantiene. Primero se ejecuta la *vista subq1* y después se ejecuta la *consulta facebook*.

Una vez que hemos descrito nuestro caso de estudio, ahora describamos las optimizaciones de *Hive*.

3.5.1. Optimizaciones actuales en Hive

Actualmente, *Hive* soporta ocho optimizaciones. Tres optimizaciones son comúnes con los Sistemas de Gestión de Base de Datos *SGBD* como *MySQL*. Las cinco optimizaciones restantes se enfocan a optimizar el *DAG* para que cuando se transforme a secuencias de *trabajos mapreduce* cada *trabajo mapreduce* se ejecute de manera eficiente considerando las condiciones del ambiente *MapReduce*.

Las optimizaciones que son comúnes con cualquier SGBD son:

1. ***Filtrar registros lo antes posible***: Esta optimización está relacionada con la

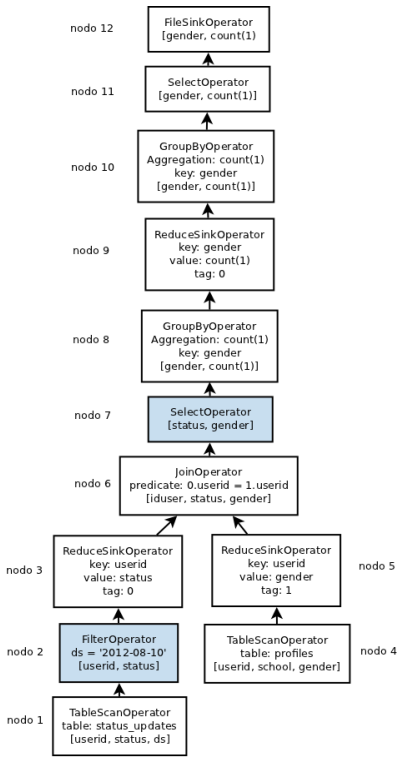


Figura 3.3: DAG de la *consulta facebook*.

operación de selección en un SGBD. La operación de selección son condiciones que se expresan en las cláusulas *where* u *on* en una consulta *SQL* [27]. El objetivo de la optimización es filtrar los registros que cumplan con dichas condiciones lo antes posible. Por ejemplo, la consulta facebook (ver código 3.1) filtra los registros de la tabla *status_update* que han sido actualizados el día *10-08-2012* a través de la condición *a.ds='2012-10-8'* en la cláusula *where* en la vista *subq1*. La tabla *status_updates* puede tener millones de registros de cambio de *status* de otras fechas, pero sólo se necesita los de la fecha indicada. En *Hive* el filtro de los registros se realiza con el operador *FilterOperator*. En el *DAG* de la consulta facebook que se observa en la figura 3.3 en el nodo 2 se observa el operador *FilterOperator* filtra los registros deseados.

2. ***Podado de partición:*** Al igual que la optimización anterior, esta optimización está relacionada con la operación de selección en una base de datos. Sin embargo, en lugar de filtrar registros de una tabla por una condición que involucra alguna

columna de la tabla, esta optimización filtra los registros por una condición que involucra alguna partición (ver sección 3.2). Por ejemplo, supongáse que los registros de la tabla *status_updates* de la consulta facebook se particiona de acuerdo al país en el que se hizo una actualización. El país no es una columna de la tabla, sin embargo, todos los registros de un mismo país se van a almacenar en una partición. *Hive* permite seleccionar los registros de una partición filtrando con condiciones en la cláusula *where*.

3. ***Podado de columnas***: Esta optimización está asociada a la operación de proyección en un SGBD. La operación de proyección permite seleccionar columnas que se especifican en la cláusula *select* en una consulta SQL [27]. El objetivo de esta optimización es garantizar que sólo las columnas que son necesarias para el procesamiento de la consulta se consideren. El filtro de las columnas se debe de realizar lo antes posible. En *Hive* el podado de columnas se realiza con el operador *SelectOperator*. Por ejemplo, en la vista *subq1* de la consulta facebook, la cláusula *select* selecciona las columna *status* de la tabla *status_updates* y *gender* de la tabla *profiles*. Como las columnas pertenecen a tablas diferentes, entonces se espera a que se realice la operación *Join* y posteriormente se selecciona dichas columnas. En el *DAG* de la consulta facebook que se observa en la figura 3.3 la optimización se observa en el operador *SelectOperator* del nodo 7. Observe que el operador selecciona las columnas *status y gender* evitando que la columna *userid* continue siendo procesa por los operadores superiores.

Las optimizaciones que están pensadas en optimizar el *DAG* para que cuando se transforme a secuencias de *trabajos mapreduce* cada *trabajo mapreduce* se ejecute de manera eficiente considerando las condiciones del ambiente *MapReduce* son:

1. ***Reducción del uso de la red por medio de agregaciones parciales en tareas map***: Las funciones de agregación son funciones estadísticas que resumen un conjunto de registros en un sólo valor. Ejemplo de ellas son las

funciones: *sum(), min(), max(), avg(), count()*, entre otras. Estas funciones generalmente se aplican sobre grupos de registros de tal manera que por cada grupo de registros se obtiene un sólo valor de acuerdo a la función de agregación aplicada. Los grupos de registros en una consulta *HiveQL* se forman a partir de las columnas que se especifican en la cláusula *group-by*. Por ejemplo, en la consulta facebook (ver código 3.1) en la cláusula *group-by* se específica la columna *gender*. Por lo tanto, los registros se agrupan por género (masculino, femenino) y por cada género (grupo) aplica la función de agregación *count()*.

Esta optimización está pensada para aplicarse a la hora en que se transforma el *DAG* a *trabajos mapreduce*, por lo que son necesarios algunos aspectos relacionados con la ejecución de *tareas map y reduce*.

Cuando una consulta *HiveQL* involucra funciones de agregación y agrupación que actúan sobre los registros de una tabla, entonces *Hive* genera un sólo *trabajo mapreduce* donde en cada *tarea map* se lee un split del archivo (bucket) que contiene los registros de la tabla a procesar. Como cada *tarea map* no puede determinar si ha leído en el split todos los registros que pertenecen a un grupo especificado por la cláusula *group-by* (por ejemplo cuando se procesan más de un split), entonces no se puede realizar la agrupación total en una *tarea map*. Por lo tanto, cada *tarea map* emite un par intermedio $< clave, valor >$ de tal manera que la clave son las columnas que se especifican en la cláusula *group-by*. De este modo, todos los registros con el mismo valor en las columnas *group-by* se agrupan y se envían por la red a una *tarea reduce*. Por lo tanto, cada *tarea reduce* recibe los registros agrupados por la cláusula *group-by* y sólo falta aplicar la función de agregación deseada. Sin embargo, esta implementación tiene el problema que envía todos los registros de una relación de una o más *tarea map* a una *tarea reduce* a través de la red y si cada grupo tiene varios millones de registros, entonces se puede ocasionar un cuello de botella en la red.

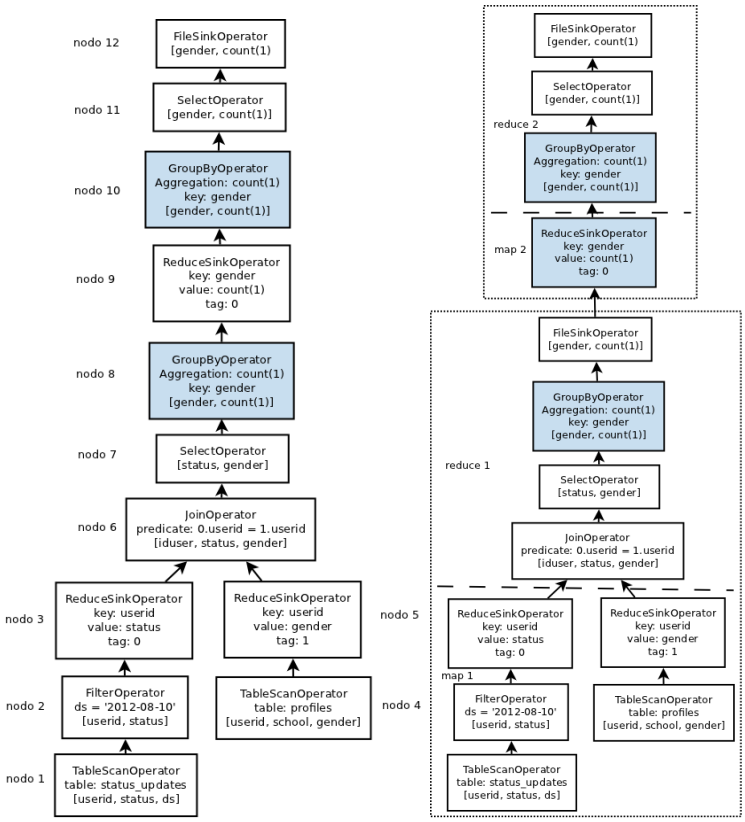
Esta optimización consiste en evitar el envío de tantos registros por medio de

que cada *tarea map* realice una agregación parcial y sólo se envie este valor parcial por la red a la *tarea reduce* correspondiente. La agregación parcial se realiza a través una *tabla hash*[2] que utiliza como clave las columnas que se expresan en la cláusula *group-by* y como valor el resultado parcial de aplicar la función de agregación a los registros del grupo que se identifica con la clave.

2. ***Reducción de la escritura a HDFS por medio de agregaciones parciales en tareas reduce***: Cuando una consulta *HiveQL* involucra funciones de agregación y agrupación que se aplican a registros que provienen de una operación *join* o de otras subconsultas, entonces *Hive* crea dos *mapreduce*. El primer *trabajo mapreduce* realiza la operación *join* o la subconsulta y el segundo *trabajo mapreduce* lee los registros generados por la operación *join* o la subconsulta y realiza la función de agregación y agrupación como se ha descrito anteriormente. En este caso, se presenta el mismo problema de red que se ha planteado. Además, se presenta otro problema de entrada y salida ocasionado por los registros de la operación *join* o subconsulta que se escriben al sistema de archivos distribuido *HDFS* y se leen en el segundo *trabajo mapreduce* correspondiente a la función de agregación y agrupación.

   Para resolver el problema de entrada y salida al *HDFS*, *Hive* aplica la optimización de agregación parcial basada en una tabla hash, sólo que esta vez se aplica al final de las *tareas reduce* del *trabajo mapreduce* de la operación *join* o subconsulta. De la misma manera que en el caso anterior, los registros que se generan en la operación *join* o subconsulta se agrupan de acuerdo a las columnas que se especifican en la cláusula *group-by*, para esto se utiliza una tabla hash que utiliza como clave dichas columnas y como valor el resultado parcial de aplicar la función de agregación y agrupación a cada grupo de los registros del *join* o subconsulta. Los resultados parciales son los que se escriben al sistema *HDFS* reduciendo la cantidad de datos a escribir.

[2]Una tabla hash es una estructura de datos conformada por una clave y un valor donde el valor se identifica por la clave que nunca se repite.



(a) DAG (b) Plan físico

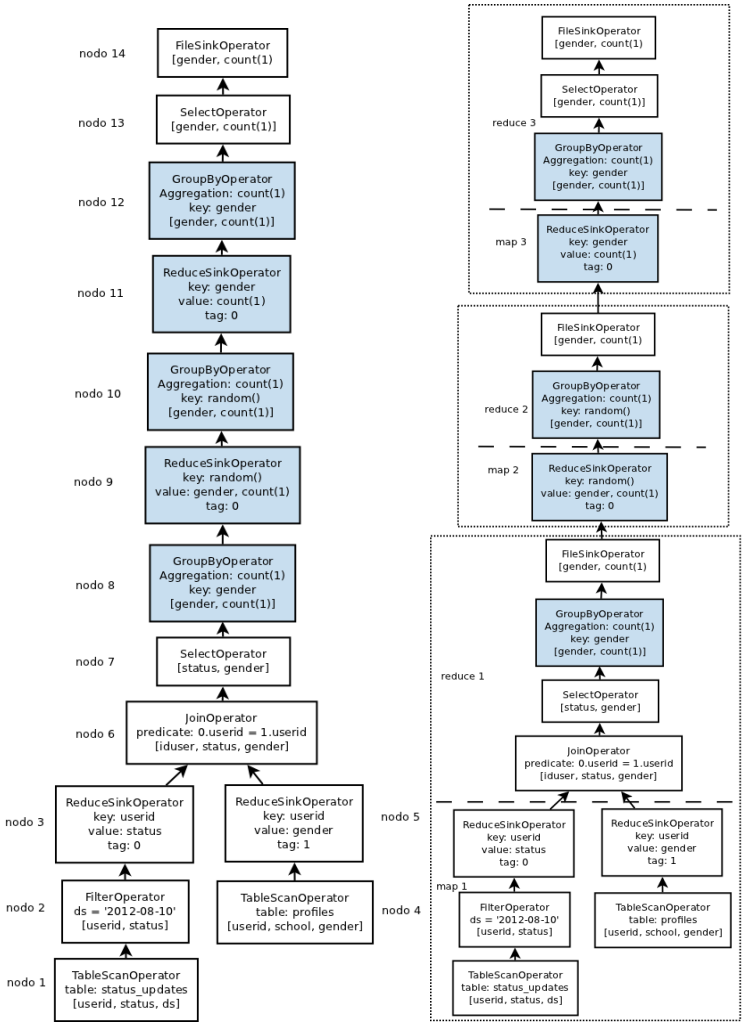
Figura 3.4: DAG y plan físico de la *consulta facebook* con la optimización de reducción de la escritura a HDFS por medio de agregaciones parciales en *tareas reduce*.

Por ejemplo, en la consulta facebook (ver código 3.1) la función de agregación y agrupación *count()* se aplica sobre los registros que se generan en la vista *subq1*. Por lo tanto, es necesario resolver primero la vista *subq1* para después aplicar la función *count()*. Esto se observa en el plan físico que genera *Hive* para la consulta (ver figura 3.4b). Obsérvese que el *trabajo mapreduce* 1 resuelve la vista *subq1* y el *trabajo mapreduce* 2 resuelve la función de agregación y agrupación *count()*. Obsérvese también que en el *trabajo mapreduce* 1 existe un operador *GroupByOperator* el cuál aplica la función *count()* de manera parcial a los registros de la vista *vistaq1*. Los resultados parciales se escriben al sistema de archivos *HDFS* a través del operador *FileSinkOperator*. Posteriormente, en las *tareas map* del *trabajo mapreduce* 2 se leen los resultados parciales a través del operador *ReduceSinkOperator* y se envían a las *tareas reduce* donde a través del operador *GroupByOperator* se vuelve a realizar la función de agregación y agrupación *count()* para obtener los resultados definitivos.

3. ***Aprovechamiento del paralelismo de* MapReduce *a través de la división de datos en la operación* Group-BY*:*** Otro de los problemas de las *funciones de agregación y agrupación* en *MapReduce* se observa cuando los registros se agrupan por una columna que puede tener pocos valores diferentes. Por ejemplo, en la *consulta facebook*, los registros se agrupan por la columna género, pero esta columna sólo acepta dos valores: masculino y femenino. En este caso, como la tarea map emite los pares $< clave, valor >$ utilizando como clave la columna género, entonces todos los registros con un mismo género se envían a una *tarea reduce*. Esto ocasiona que sólo se creen dos *tarea reduce* y cada *tarea reduce* procese una gran cantidad de registros, en lugar de aprovechar el paralelismo que ofrece *MapReduce*.

   Para solucionar este problema, esta optimización transforma al *DAG* para resolver la *función de agregación y agrupación* en *dos trabajos mapreduce*.

   En el primer *trabajo mapreduce* los registros que se envían entre *una tarea*



(a) DAG (b) Plan físico

Figura 3.5: DAG y plan físico de la *consulta facebook* con la optimización de sesgo de datos en la operación GroupBy.

map y una tarea reduce no se envían utilizando como clave a las columnas de agrupación, en su lugar se utiliza como clave cualquier otro valor aletorio. De esta forma, se logra una mejor distribución de los registros en varias *tareas reduce* y en cada *tarea reduce* se realiza una *agregación parcial*.

En el segundo *trabajo mapreduce* los registros entre *una tarea map y una tarea reduce* se envían utilizando como clave a las columnas de agrupación. Aun cuando se van a crear pocas *tareas reduce*, cada *tarea reduce* procesa menos cantidad de registros.

Para que esta optimización se aplique se debe de configurar la variable *hive.groupby.skewindata* en *true* en el archivo *hive-site.xml*.

Por ejemplo, si se aplica esta optimización a la *consulta de facebook*, el *DAG optimizado* que construye *Hive* se observa en la figura 3.5a. Obsérvese que a diferencia del *DAG* de la *consulta facebook* en condiciones normales (ver figura 3.4a) se ha agregado un *operador ReduceSinkOperator y un operador GroupByOperator* (nodos 11 y 12) en el DAG de la figura 3.5a. Esto ocasiona que se generé otro *trabajo mapreduce* como se observa en el plan físico de la consulta en la figura 3.5b. El *operador GroupByOperator* de la *tarea reduce 1* realiza una cuenta parcial y envía el resultado al sistema de archivos con el operador *FileSinkOperator*. Después, el *operador ReduceSinkOperator* de la *tarea map 2* lee los registros anteriores y los envía a la *tarea reduce 2* utilizando como clave un número aleatorio. De esta manera se tiene una mejor distribución en varias *tarea reduce*. Entonces, el *operador GroupByOperator* de cada *tarea reduce 2* realiza una cuenta parcial y envía los registros nuevamente al sistema de archivos con el operador *FileSinkOperator*. Por último, el *operador ReduceSinkOperator* de la *tarea map 3* lee los registros anteriores y envía los registros a la *tarea reduce 3* utilizando como clave las columnas de agrupación. El *operador GroupByOperator* de la *tarea reduce 3* realiza la cuenta final.

4. ***Reducción del uso de memoria por reordenamiento de los datos en el operador join:*** La operación *Join* es una operación binaria que permite combinar registros de dos *relaciones (tablas, vistas, subconsultas)*. En *Hive*, la operación *Join* se realiza a través del operador *JoinOperator*. En condiciones normales, este operador se realiza en una *tarea reduce*. Por ejemplo, en el plan físico de la consulta facebook en las optimizaciones anteriores (figuras 3.4b y 3.5b) se observará que en los *trabajos mapreduce* 1 de ambos planes físicos la operación *join* se realiza en las *tareas reduce*. La optimización *reordenamiento del Join* hace que se cargue en memoria a la *relación* más pequeña para después

vincular los registros con la *relación* más grande. Eso disminuye la posibilidad de que la *operación join* exceda los límites de memoria en una *tarea reduce*.

5. ***Reducción del uso de red realizando el* join *del lado del map:*** Como se ha comentado, normalmente la *operación Join* se realiza en una *tarea reduce*. Sin embargo, la *operación Join* también se puede realizar en una *tarea map* cuando una de las *relaciones* que vincula tiene un tamaño pequeño. Para indicar que la *operación join* se realice en una *tarea map*, la *sentencia HiveQL* debe de tener una estructura parecida a la siguiente:

```
SELECT /*+ MAPJOIN(t2) */ t1.c1, t2.c1 from t1 JOIN t2
ON (t1.c2 = t2.c2);
```

Con la instrucción *MAPJOIN* se indica que la operación *join* se va a realizar en las *tarea map* e indica la relación más pequeña, en este caso la *tabla t2*. Esta tabla se replica completamente en cada uno de los nodos donde se va ejecutar una *tarea map*, de tal manera que en cada *tarea map* se carga la *tabla t2* en una *tabla hash*. Posteriormente, cada *tarea map* lee un split de la *tabla t1* (tabla con mayor tamaño) y realiza el *join* con los registros que están almacenados en la *tabla hash*. Con esta optimización se evita enviar los registros de ambas tablas por la red para realizar el *Join* en la *tarea reduce*, logrando un mejor desempeño.

3.5.2. Componentes de una optimización

Las optimizaciones en *Hive* se basan en reglas. Es decir, cada optimización tiene un conjunto de reglas preestablecidas y cada regla tiene asociada una transformación del *DAG*. Es decir, una optimización puede transformar un *DAG* de diferentes maneras dependiendo de la regla que se cumpla. Para verificar que al menos una regla se cumple, cada optimización recorre al *DAG* en busca de que al menos una regla se cumpla. Cuando una regla se cumple se aplica en el *DAG* la transformación asociada

a la regla. Si dos o más reglas se cumplen, entonces se aplica la transformación de la regla que involucra más nodos.

Una optimización se basa en 5 entidades: *Node, Rule, Processor, Dispatcher y GraphWalker*.

La entidad *Node* está asociada a los nodos del *DAG*. Un nodo es un operador de *Hive* (tabla 3.2). Esta entidad permite reconocer que operador tiene un determinado nodo. Cada operador tiene asignada una abreviación relativa a su nombre. Por ejemplo, el operador *GroupByOperator* se asocia con la abreviación *GBY*, el operador *ReduceSinkOperator* se asocia con la abreviación *RS* y así sucesivamente.

En la entidad *Rule* se especifican las reglas de cada optimización. Estas reglas se expresan a través de expresiones regulares, que indican la secuencia de operadores que se deben de encontrar al recorrer el *DAG* para determinar que se ha encontrado una regla. Una expresión regular describe un conjunto de cadenas que concuerdan con un patrón general. Por ejemplo, la expresión regular *B[an]*s* describe a las cadenas *Bananas, Baaas, Bs, Bns*; esto se debe a que el símbolo '*' indica que las palabras que están dentro del corchete pueden aparecer de 0 a N veces en una palabra. Además de este símbolo existen otros simbolos como '+', '?' que nos ayudan a expresar diferente patrones [28]. En *Hive* actualmente, el manejo de una expresión regular para expresar una regla es sencillo. Se utilizan sólo 2 símbolos: '%' y '*'. El símbolo '%' indica que se debe encontrar una secuencia específica de operadores. El símbolo '*' indica que no importa que secuencia de operadores se encuentre. Por ejemplo, la regla *RS%*GBY%RS%GBY%* indica que se debe de encontrar un operador *ReduceSinkOperator* seguido de cualquier secuencia de operadores y que posteriormente se debe de encontrar un operador *GroupByOperator* seguido de otro operador *ReduceSinkOperator* y otro operador *GroupByOperator*.

La entidad *Processor* guarda la transformación que se debe de aplicar cuando una regla se cumple. Es decir, cada entidad *Rule* tiene asociada una entidad *Processor*.

La entidad *Dispatcher* mantiene la relación entre las reglas y los procesadores (transformaciones). Esta entidad verifica si se cumple alguna regla al recorrer el *DAG*,

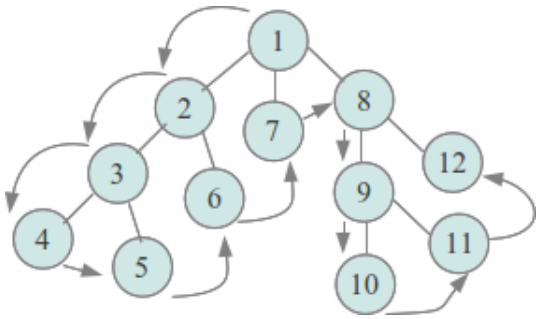


Figura 3.6: Ejemplo de recorrido de un árbol con el algoritmo *DFS*.

y si este es el caso ejecuta el Processor (transformación) asociado. Si dos o más reglas se cumplen, entonces selecciona la regla que implica el mayor número de operadores.

La entidad *GraphWalker* se encarga de recorrer el *DAG*. El *DAG* se puede recorrer de diferentes maneras de acuerdo a las necesidades de cada optimización. El *GraphWalker* inicia el recorrido del *DAG* a partir de los nodos de más abajo, es decir los *TableScanOperator*. Por defecto, *Hive* utiliza el algoritmo de *búsqueda primero en profundidad (del inglés Depth First Search o DFS)*. Este algoritmo recorre un árbol expandiendo los nodos que va localizando hasta que no quedan nodos que visitar en dicho camino. Entonces, regresa al nodo anterior que tenga otro camino por recorrer; y realiza el mismo procedimiento anterior hasta terminar de recorrer el árbol completo. La figura 3.6 muestra como se recorre el árbol con el algoritmo *DFS*. La secuencia de nodos visitados es: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12. *Hive* utiliza este algoritmo por cada nodo *TableScanOperator*. Cuando coincide con un nodo que ya ha sido visitado, entonces detiene su recorrido. Aplicando el algoritmo *DFS* para recorrer el *DAG* de la *consulta facebook* que se observa en la figura 3.4a, los nodos se recorreran de la siguiente manera: *TS %FIL %RS %JOIN %SEL %GBY %RS %GBY %SEL %FS %TS %RS %*. Cabe mencionar que por cada nodo visitado se obtiene la abreviación del operador que contiene y así es como se va formando la cadena de los nodos que visita. Un *GraphWalker* invoca al *Dispatcher* por cada nodo que visita para verificar si alguna regla se cumple.

El procedimiento general para aplicar una optimización a un *DAG* se muestra en

la figura 3.7 y consiste en:

1. Construir las reglas que se desean encontrar en el *DAG*.

2. Definir por cada regla que se construye la transformación (*Proccesor*) que se va a aplicar al *DAG* cuando la regla se encuentre.

3. Inicializar al *Dispatcher* con las reglas y sus transformaciones asociadas.

4. Empezar a recorrer el árbol con ayuda del *GraphWalker* a partir de los nodos *TableScanOperator*. Por cada nodo visitado obtener la abreviación del nodo y mandar a llamar al *Dispatcher* para verificar si alguna regla se cumple. Si no se cumple alguna regla continuar el recorrido del árbol hasta que se termine de recorrer. Si en algún momento se cumple alguna regla, entonces aplicar la transformación(*Processor*) correspondiente y continuar recorriendo el árbol hasta que se termine de recorrer.

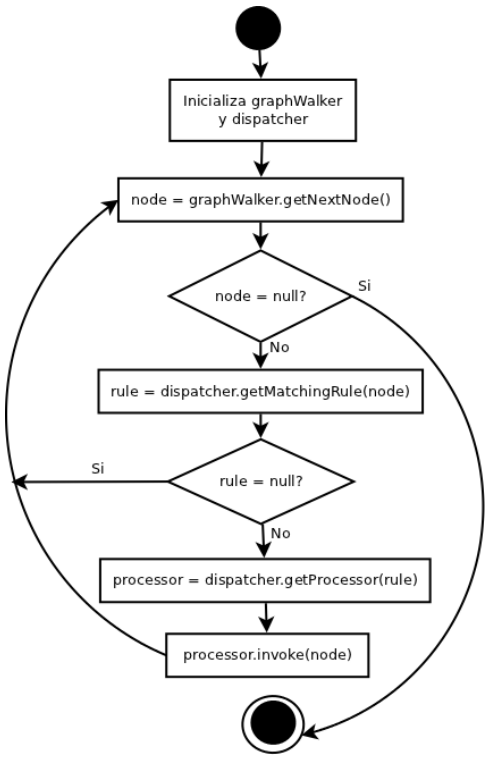


Figura 3.7: Diagrama de actividad de una optimización en *Hive*.

3.6. Motor de ejecución de Hive

El motor de ejecución en el *driver de Hive* se encarga de transformar el *DAG* ya optimizado en una secuencia de *trabajos mapreduce* a ejecutar en *Hadoop*.

Está constituido por dos fases: Generación del plan físico y ejecución.

La fase de generación del plan físico identifica y divide al *DAG optimizado* en una secuencia de *trabajos mapreduce* a ejecutar según el orden del *DAG*.

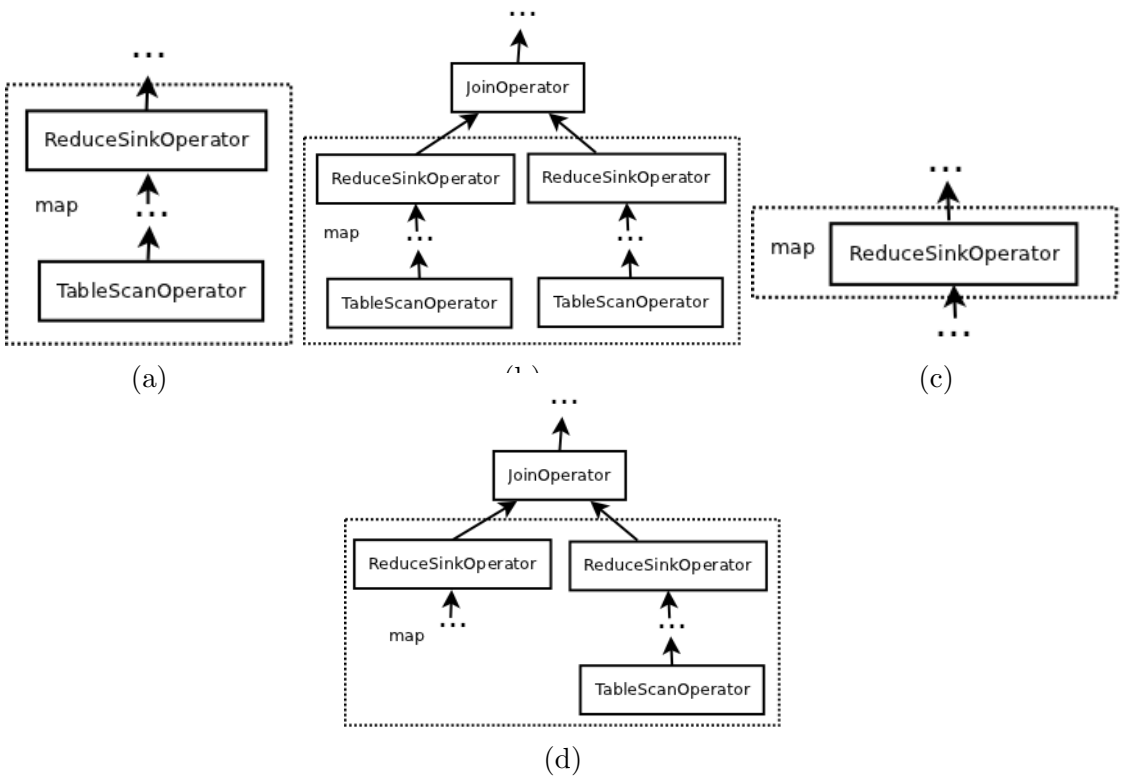
La fase de ejecución ejecuta cada *trabajo mapreduce* en el orden establecido por el *DAG* mapeado a *trabajos mapreduce*. Cada operador en el *DAG* tiene asociada una implementación *MapReduce*, por lo que el motor de ejecución se encarga de ejecutar dichas implementaciones en las tareas *mapreduce* correspondientes.

Para identificar *una tarea map o una tarea reduce* en el *DAG optimizado* se toma cómo referencia a los operadores: *TableScanOperator (TS), ReduceSinkOperator (RS) y FileSinkOperator (FS)*. El *operador TS* representa la lectura de una tabla en una *consulta HiveQL*. El *operador RS* es un operador agregado por *Hive* que se encarga de configurar los pares $< clave, valor >$ que emite una *tarea map*. El *operador FS* escribe los resultados de una *tarea reduce* al sistema de archivos distribuido *HDFS*.

La figura 3.8 muestra las cuatro formas en que se puede constituir una tarea map, las cuales son:

1. Una tarea map se puede constituir por todos los operadores que se encuentran entre un *operador TS* y un *operador RS* como se observa en la figura 3.8a.

2. Una tarea map se puede constituir por dos *operadores TS* y dos *operadores RS* cuando los dos *operadores TS* son vinculados por un *operador Join* como se observa en la figura 3.8b. Este caso se presenta en el *DAG* de la *consulta facebook* que se observa en la figura 3.9a. Observe que el *Join* de la vista *subq1* vincula a la *tabla status_updates y la tabla profiles* provocando la construcción de la *tarea map 1* que se observa en el plan físico de la *consulta facebook* en la figura 3.9b.



(a) (c) (d)

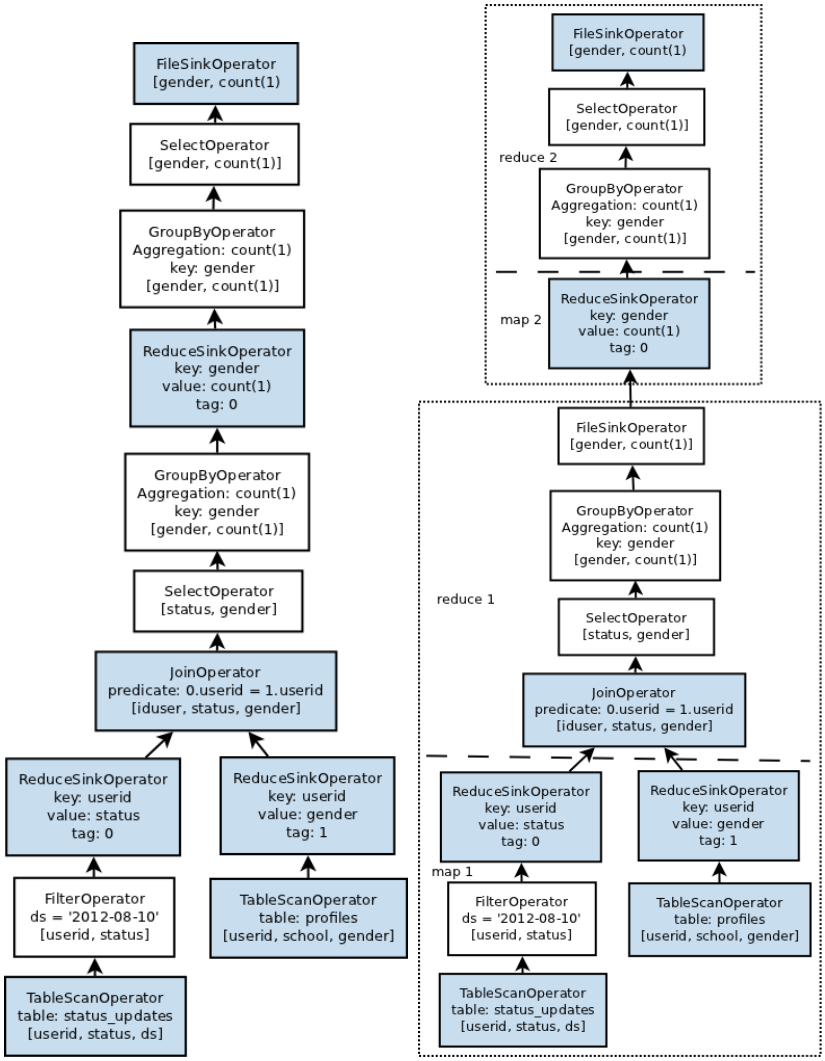
Figura 3.8: Formas de identificar una tarea map.

3. Una *tarea map* se puede constituir únicamente por un *operador RS* como se observa en la figura 3.8c. Esto sucede cuando ya se ha identificado un *trabajo mapreduce* anterior a dicha *tarea map*. Por ejemplo, en el *DAG de la consulta facebook* una vez que se ha realizado la vista *subq1* aparece un *RS* seguido de un operador *GroupByOperator*. En este caso, se crea la *tarea map 2* que se observa en el plan físico de la *consulta facebook* en la figura 3.9b.

4. Una *tarea map* se puede constituir por dos *operadores RS* como se observa en la figura 3.8d. Esto sucede cuando ya se ha identificado un *trabajo mapreduce* anterior a dichos *operadores RS* y ambos operadores son vinculados por un *operador Join*.

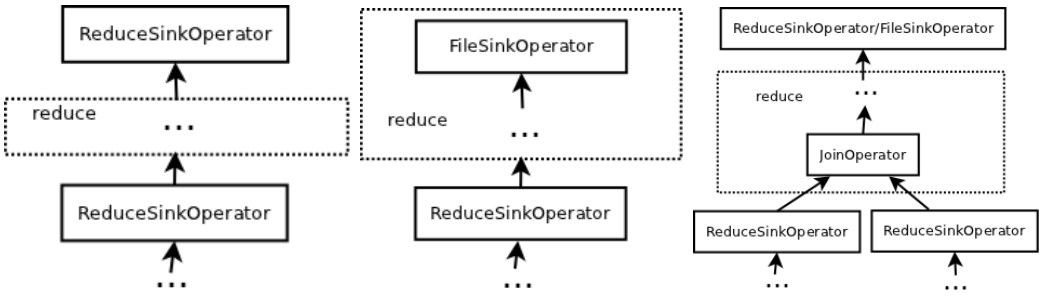
La figura 3.10 muestra las tres formas de constituir una *tarea reduce*, las cuales son:

1. Una *tarea reduce* se puede constituir por todos los operadores después de un



(a) DAG (b) Plan físico

Figura 3.9: DAG y plan físico de la *consulta facebook*.



(a) (b) (c)

Figura 3.10: Formas de identificar una tarea reduce.

operador RS y el operador anterior al siguiente *operador RS* como se observa en la figura 3.10a. Este caso se observa en el *DAG* de la *consulta de facebook* al realizar la vista *subq1*, a partir de que realiza el*Join* de las *tablas status_updates y profiles* hasta que realiza la cuenta parcial, que se refleja en la *tarea reduce 1* en el plan físico de la *consulta facebook* en la figura 3.9b. Observe que la *tarea reduce 1* agrega un operador *FS*, esto se debe a que la salida de la *tarea reduce* la escribe provisionalmente al *sistema de archivos HDFS*.

2. Una *tarea reduce* se puede constituir por todos los operadores después de un *operador RS* hasta encontrar un *operador FS* como se observa en la figura 3.10b. Esto sucede cuando es la última *tarea reduce* de todos los *trabajos mapreduce* identificados, debido a que en un *DAG* el último operador es un *FS*.

3. Una *tarea reduce* se puede constituir por todos los operadores entre un *operador Join* y un *operador RS* o de un operador *join* hasta un *operador FS* como se observa en la figura 3.10c.

Por último, un *trabajo mapreduce* se identifica cuando se encuentra *una tarea map y una tarea reduce* consecutivamente.

3.7. Resumen

Hive es un datawarehouse que implementa de manera lógica una infraestructura de base de datos sobre *Hadoop*. Permite el procesamiento de grandes volúmenes de datos estructurados y no estructurados a través de sentencias *HiveQL* que son muy parecidas a las sentencias SQL.

Hive está constituido por tres componentes: *Metastore, interfaces de usuario y driver*.

El *metastore* mantiene la infraestructura del datawarehouse. Guarda datos como la localización de las bases de datos o tablas en el sistema *HDFS*, los permisos de usuario, etcétera.

Las *interfaces de usuario* de *Hive* son tres: *Línea de comandos, interfaz web, Thrif Server*. Las primeras dos interfaces de usuario permiten al usuario interactuar directamente con *Hive* a través de *sentencias HiveQL*. La interfaz *Thrif Server* permite al usuario interactuar con *Hive* a través de programas escritos en diferentes lenguajes como C o Java.

El *Driver* se encarga de compilar las sentencias *HiveQL* en una secuencia de *trabajos mapreduce* que se ejecutan en *Hadoop*. El *Driver* está constituido por: *compilador, optimizador y motor de ejecución*.

El *compilador* se encarga de verificar que una sentencia *HiveQL* este bien formada y sea coherente. Si lo es, construye un *DAG* de la *sentencia HiveQL* formado por *operadores de Hive*. Cada *operador de Hive* tiene asociada una implementación *MapReduce* que es lo que se ejecuta en el *motor de ejecución*.

El *optimizador* se encarga de optimizar el *DAG* construido en el *compilador*, de tal manera que al convertirse en *trabajos mapreduce* tengan un mejor desempeño.

El *motor de ejecución* construye los *trabajos mapreduce* a partir de la representación gráfica optimizada y ejecuta cada *trabajo mapreduce* en *Hadoop*.

Capítulo 4

Otras posibles optimizaciones para consultas *HiveQL*

Hive es un datawarehouse sobre *Hadoop* que facilita el procesamiento de grandes volúmenes de datos a través de sentencias *HiveQL*. En el capítulo 3, se describen las optimizaciones actuales que aplica *Hive* a una sentencia *HiveQL* durante el proceso de compilación a *trabajos mapreduce*. Sin embargo, aún con las optimizaciones actuales, en algunas ocasiones *Hive* produce *trabajos mapreduce* ineficientes, repetidos e innecesarios que ocasionan un bajo desempeño. En este capítulo se describen los problemas y las condiciones en las que el *DAG* de una consulta *HiveQL* produce un *DAG* poco optimizado. Además se propone una solución para mejorar el desempeño de *Hive* en esos casos. En el capítulo 5 se presenta el diseño e implementación de algunas de las optimizaciones propuestas y en el capítulo 6 se evalúa el desempeño de *Hive* con las optimizaciones realizadas.

4.1. Introducción

Las optimizaciones actuales de *Hive* se enfocan a mejorar el *DAG* en tres aspectos: Algunas optimizaciones aplican un razonamiento que se utiliza ampliamente en un Sistema de Gestión de Base de Datos (*SGBD*), que consiste en que los operadores

superiores de un *DAG* trabajen únicamente con los datos necesarios, filtrando los datos que no son necesarios con operadores *FilterOperator* o *SelectOperator* (ver tabla 3.2). Otras optimizaciones reducen la cantidad de datos que se envían por la red entre las *tareas map y reduce* de un *trabajo mapreduce*. Por último, otra optimización reduce la cantidad de datos que escribe una *tarea reduce* al sistema *HDFS* y que lee una *tarea map* en otro *trabajo mapreduce*. Las optimizaciones actuales de *Hive* sin duda permiten que se cree un *DAG* optimizado. Sin embargo, en algunas ocasiones el *DAG* optimizado que se produce aún tiene algunos inconvenientes, algunos problemas que observamos son:

- El *DAG* optimizado en ocasiones, al transformarlo en *trabajos mapreduce* genera *trabajos mapreduce* innecesarios relacionado con las agregaciones parciales.
- El *DAG* optimizado puede contener secuencias repetidas de operadores *Hive*, ocasionando que al transformar el *DAG* en *trabajos mapreduce*, se repitan las mismas *tareas map o reduce* en distintos *trabajos mapreduce* o incluso que varios *trabajos mapreduce* se repitan.
- El *DAG* optimizado en ocasiones, al transformalo en *trabajos mapreduce* realiza una misma secuencia de operadores dos veces en una *tarea map* de un *trabajo mapreduce* provocando que se envíe dos veces los mismos datos a una *tarea reduce*.

Para poder describir estos problemas a detalle, las situaciones en que se presentan y las propuestas de solución es conveniente definir casos de estudio que permitan ejemplificar los problemas. Los casos de estudio que utilizaremos son algunas consultas OLAP de Chatziantoniou [29] y TPC-H [30]. Estas consultas se utilizan con frecuencia para evaluar el desempeño de un Datawarehouse o base de datos.

4.1.1. Consultas de Chatziantoniou

Las consultas de *Chatziantoniou*, son consultas *OLAP* complejas que se utilizarón en un estudio para evaluar una técnica para identificar consultas group-by y optimizarlas.

Son cuatro consultas de las cuales sólo utilizamos las primeras tres consultas [29]. A continuación se describe el caso de estudio y las tres consultas.

Caso de estudio: Suponga que una empresa periódistica *"For Your Information (FYI)"* tiene una tabla donde mantiene la información de las secciones visitadas por sus clientes. La tabla es la siguiente:

```
FYILOG(id int, date string, section string, duration double)
```

Donde: *id* es el id del cliente, *date* se refiere a la fecha de conexión, *section* es la sección que visitó el cliente (deportes, política, negocios, etcétera), y *duration* es el tiempo de conexión del cliente en la sección. Entonces, la empresa *FYI* desea conocer:

- Consulta 1 (Q1): Por cada cliente, ¿Cuántas veces accedió a la sección *"WORLD"*? y ¿Cuántas veces el tiempo que estuvo en la sección *"WORLD"* fue mayor que el promedio que permaneció en todas las secciones? Esta consulta permite saber por cada cliente la frecuencia con que visita la sección *"WORLD"* y el tiempo que permanece en dicha sección en comparación con todas las secciones.

- Consulta 2 (Q2): Por cada cliente, encontrar la máxima duración promedio por cada sección con su respectivo nombre. Esta consulta permite encontrar por cada cliente la sección de su preferencia.

- Consulta 3 (Q3): Por cada cliente que accedió a la sección *"WORLD"* en una conexión *t*, encontrar el promedio de tiempo gastado en la sección *"WORLD"* en las conexiones anteriores a *t*, y el promedio de tiempo gastado en la sección *"WORLD"* en las conexiones posteriores a *t*. Esta consulta permite observar si a un cliente le interesó más la sección *"WORLD"* antes o posterior a una fecha.

4.1.2. Consultas del estudio TPC-H

TPC-H es un conjunto de consultas de un estudio de mercado para el soporte a decisiones. Está constituido por un conjunto de consultas OLAP (22 consultas) que

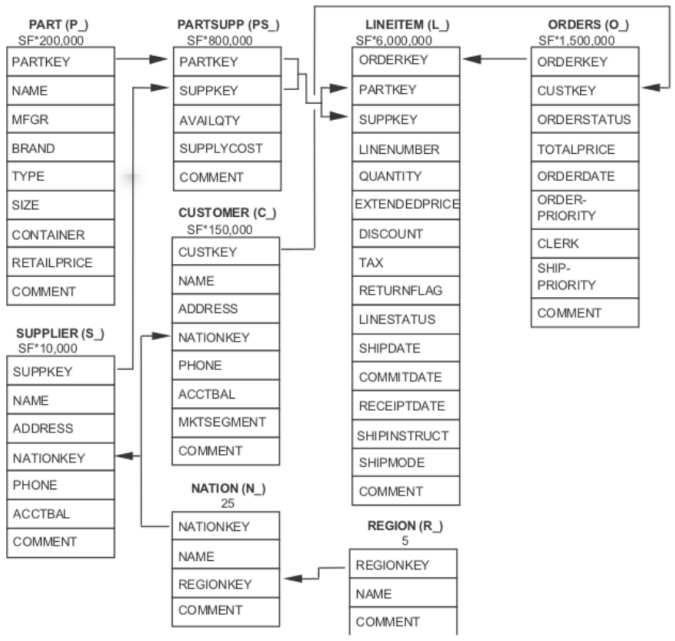


Figura 4.1: Esquema relacional del estudio TPC-H

evalúa el rendimiento de sistemas de soporte de decisiones a través de condiciones controladas. El estudio está avalado por compañías como: AMD, CISCO, Oracle, IBM, Microsoft, Dell, HP, entre otras. El esquema relacional de las tablas del estudio *TPC-H* se observa en la figura 6.1.

La consulta *TPC-H* que se toma en cuenta para ejemplificar las problemáticas es la consulta 11 la cual consiste en encontrar el stock más importante de los proveedores de una determinada nación.

Las consultas *HiveQL* y el *DAG* que genera *Hive* por cada de una de las consultas anteriores se presentaran en el momento que se utilicen.

4.2. Eliminación de *trabajos mapreduce* de operadores de agregación y agrupación (GroupByOperator)

Los operadores de agregación y agrupación *GroupByOperator* aparecen en un *DAG* cuando una consulta *HiveQL* involucra funciones de agregación y agrupación (*avg(), sum(), max()*, entre otras). Por ejemplo, la consulta 1 de Chatziantoniou se puede resolver con las sentencias *HiveQL* que se observan en el código 4.1. La vista *Q1V1* obtiene por cada cliente el tiempo promedio que permaneció en todas las secciones del periódico *Fyi*. La vista *Q1V2* obtiene por cada cliente el número de veces que visitó la sección *"WORLD"*. La vista *Q1V3* obtiene por cada cliente el número de veces en el que el tiempo que permaneció en la sección *"WORLD"* fue mayor que el tiempo promedio que permaneció en todas las secciones. Por último, la vista *Q1* resuelve la consulta 1 de *Chatziantoniou* y obtiene por cada cliente el número de veces que visitó la sección *"WORLD"* y el número de veces que el tiempo que permaneció en la sección *"WORLD"* fue mayor que el tiempo promedio de todas las secciones que ha visitado. Para esto se apoya de las vistas *Q1V2* y *Q1V3*.

El *plan físico* que resulta de la consulta 1 de *Chatziantoniou* se observa en la figura 4.2. Obsérvese que las vistas *Q1V1* y *Q1V2* se resuelven con los *trabajos mapreduce 1 y 2* respectivamente. En ambas vistas, las funciones de agregación *avg()* y *count()* respectivamente se aplican a registros de una sola tabla. Por esta razón, en las *tareas map* de ambos *trabajos mapreduce* se realiza la función de agregación respectiva de manera parcial, y en las *tareas reduce* se realiza la función de agregación de manera total (ver optimización de reducción del uso de la red por medio de agregaciones parciales en *tareas map* en la sección 3.5.1). Por el contrario, la vista *Q1V3* se realiza con dos *trabajos mapreduce*: el 3 y el 4. La vista *Q1V3* utiliza la función de agregación *count()* que procesa los registros que resultan de vincular la tabla *fyilog* con la vista *Q1V1* a través de un *Join*. Por esta razón, el *trabajo mapreduce 3*

```
1  CREATE VIEW Q1V1 as
2            SELECT id, avg(duration) as avg_d
3            FROM FYILOG
4            GROUP BY id;
5
6  CREATE VIEW Q1V2 as
7           SELECT id, COUNT(*) as cnt
8           FROM FYILOG
9           WHERE SECTION="WORLD"
10          GROUP BY id;
11
12 CREATE VIEW Q1V3 AS
13          SELECT c.id, COUNT(*) AS cnt
14          FROM fyilog c JOIN q1v1 q ON (c.id = q.id)
15          WHERE c.section="WORLD" and c.duration > q.avg_d
16          GROUP BY c.id;
17
18
19 CREATE VIEW Q1 AS
20          SELECT q1v2.id, q1v2.cnt as cntq1v2, q1v3.cnt as cntq1v3
21          FROM q1v2 join q1v3 ON (q1v2.id = q1v3.id);
```

Código 4.1: Sentencias *HiveQL* para resolver la consulta 1 de Chatziantoniou en Hive

realiza el *Join* de la tabla *fyilog* con la vista *Q1V1* y en las *tareas reduce* de dicho *trabajo mapreduce* se realiza la función de agregación *count()* de manera parcial en el operador *GroupByOperator*. Posteriormente, en el *trabajo mapreduce 4* se termina de realizar la función de agregación *count()*, leyendo los registros parciales con el operador *ReduceSinkOperator* y realizando la agregación final en las *tareas reduce* con el operador *GroupByOperator* (ver optimización de reducción de la escritura a *HDFS* por medio de agregaciones parciales en *tareas reduce* en la sección 3.5.1).

El problema se observa cuando existen consultas *HiveQL* que involucran funciones de agregación y agrupación que se aplican a registros que provienen de una operación *Join* y/o de otras subconsultas, como la vista *Q1V3* de la consulta 1 de Chatziantoniou. En este caso, las funciones de agregación se realizan de manera parcial en las *tareas reduce* y se realizan de manera total en otro *trabajo mapreduce* (ver optimización de reducción de la escritura a *HDFS* por medio de agregaciones parciales en *tareas reduce* en la sección 3.5.1). Esto se debe a que *Hive* no verifica si cada *tarea reduce* de un *trabajo mapreduce* recibe todos los registros agrupados por

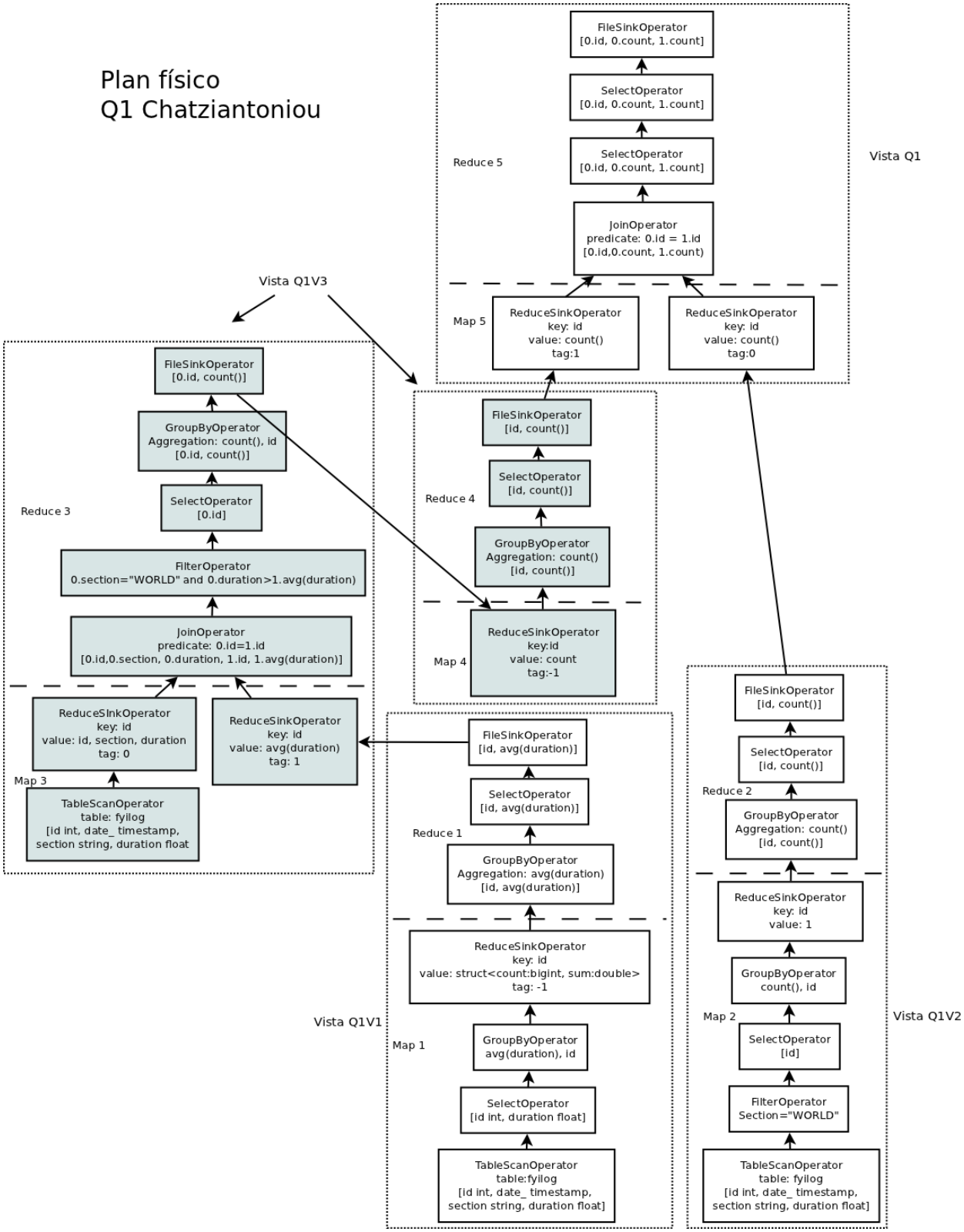


Figura 4.2: Plan físico de la consulta 1 de Chatziantoniou con las optimizaciones actuales de Hive

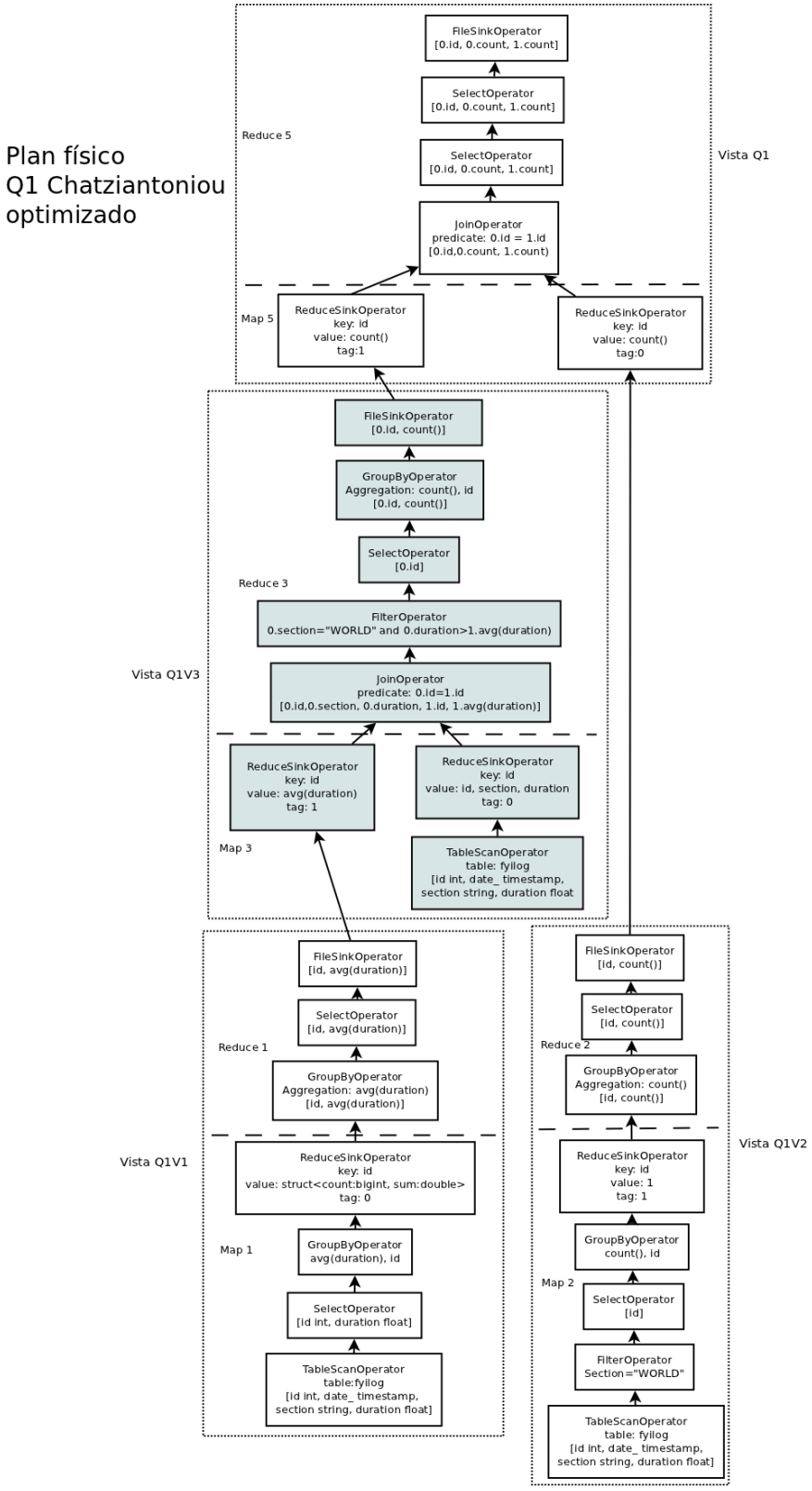


Figura 4.3: Plan físico de la consulta 1 de Chatziantoniou con la optimización propuesta

las columnas que se especifican en la cláusula *group-by*, dando por echo que cada *tarea reduce* no recibe todos los registros de un grupo. Sin embargo, en algunas ocasiones cada *tarea reduce* de un *trabajo mapreduce* si recibe todos los registros de un grupo, lo cual se puede determinar recuperando las columnas de agrupación de la cláusula *Group-By* y comparando dichas columnas con las columnas que se envían como clave en el operador *ReduceSinkOperator*. En este caso, la función de agregación se puede aplicar de manera total y no es necesario crear otro *trabajo mapreduce*.

Por ejemplo, en el *trabajo mapreduce 3* del plan físico de la consulta 1 de Chatziantoniou, los dos operadores *ReduceSinkOperator* configuran los pares intermedios $< clave, valor >$ que emiten las *tareas map*, de tal manera que la clave es el id de un cliente. Por lo tanto, cada *tarea reduce* del *trabajo mapreduce 3* recibe todos los registros de un mismo *id*. Como la consulta *Q1V3* agrupa los registros por *id*, entonces la función de agregación *count* que se realiza con el operador *GroupByOperator* en las *tareas reduce* pueden realizar la agrupación total de los registros y el *trabajos mapreduce 4* no es necesario. De esta manera, la consulta 1 se puede resolver con los trabajos mapreduce que se observan en el plan físico de la figura 4.3, del cual se ha eliminado el *trabajo mapreduce 4*.

Con esta optimización se elimina un *trabajo mapreduce* innecesario por cada operador de agregación y agrupación (*GroupByOperator*) que se le pueda aplicar esta optimización, eliminando lecturas de registros del sistema *HDFS*, procesamiento de registros, envío de registros por red y escritura de registros al sistema *HDFS* innecesarios.

4.3. Eliminación de secuencia de operadores *Hive* redundantes en un DAG

En ocasiones el *DAG* optimizado por las optimizaciones actuales de *Hive* contiene secuencias de operadores redundantes. Esto ocurre cuando una consulta *HiveQL* involucra subconsultas similares o lecturas a mismas tablas que son vinculadas

```
1  CREATE VIEW Q2V1 AS
2          SELECT id, section, avg(duration) AS avg_d
3          FROM FYILOG
4          GROUP BY id, section;
5
6  CREATE VIEW Q2V2 AS
7          SELECT id, max(avg_d) AS max_s
8          FROM Q2V1
9          GROUP BY id;
10
11 CREATE VIEW Q2 AS
12         SELECT q2v1.id, q2v1.section, q2v2.max_s
13         FROM q2v1 JOIN q2v2 ON (q2v1.id = q2v2.id)
14         WHERE q2v2.max_s = q2v1.avg_d;
```

Código 4.2: Sentencias *HiveQL* para resolver la consulta 2 de Chatziantoniou en Hive

por operadores *Join*. Cada operador *Join* en una consulta *HiveQL* vincula dos subconsultas o dos tablas. Por lo tanto, los operadores *JoinOperator* en un *DAG* vinculan dos ramas de operadores donde cada rama representa una subconsulta o lectura de una tabla. Una rama de un operador *JoinOperator* son todos los operadores que están abajo a la izquierda o a la derecha del operador *JoinOperator* en un *DAG*. Dichas ramas pueden ser simples o compuestas. Las ramas son simples cuando no tienen bifurcaciones y son compuestas cuando tienen bifuraciones. Las bifurcaciones se deben a que en dicha rama se encuentra otro operador *Join* que tiene otras dos ramas. Esta optimización se aplica cuando hay *joins* en una consulta *HiveQL* y por lo tanto, al menos hay dos ramas. A continuación se describe el problema de los operadores redundantes entre ramas simples y compuestas.

4.3.1. Eliminación de operadores redundates entre ramas simples

El problema de operadores redundantes entre ramas simples se puede presentar cuando en una consulta *HiveQL* se utiliza un sólo *join* que vincula subconsultas que leen una sola *tabla* o cuando el *join* vincula a la misma tabla.

Para describir el problema de los operadores redundantes entre ramas simples se

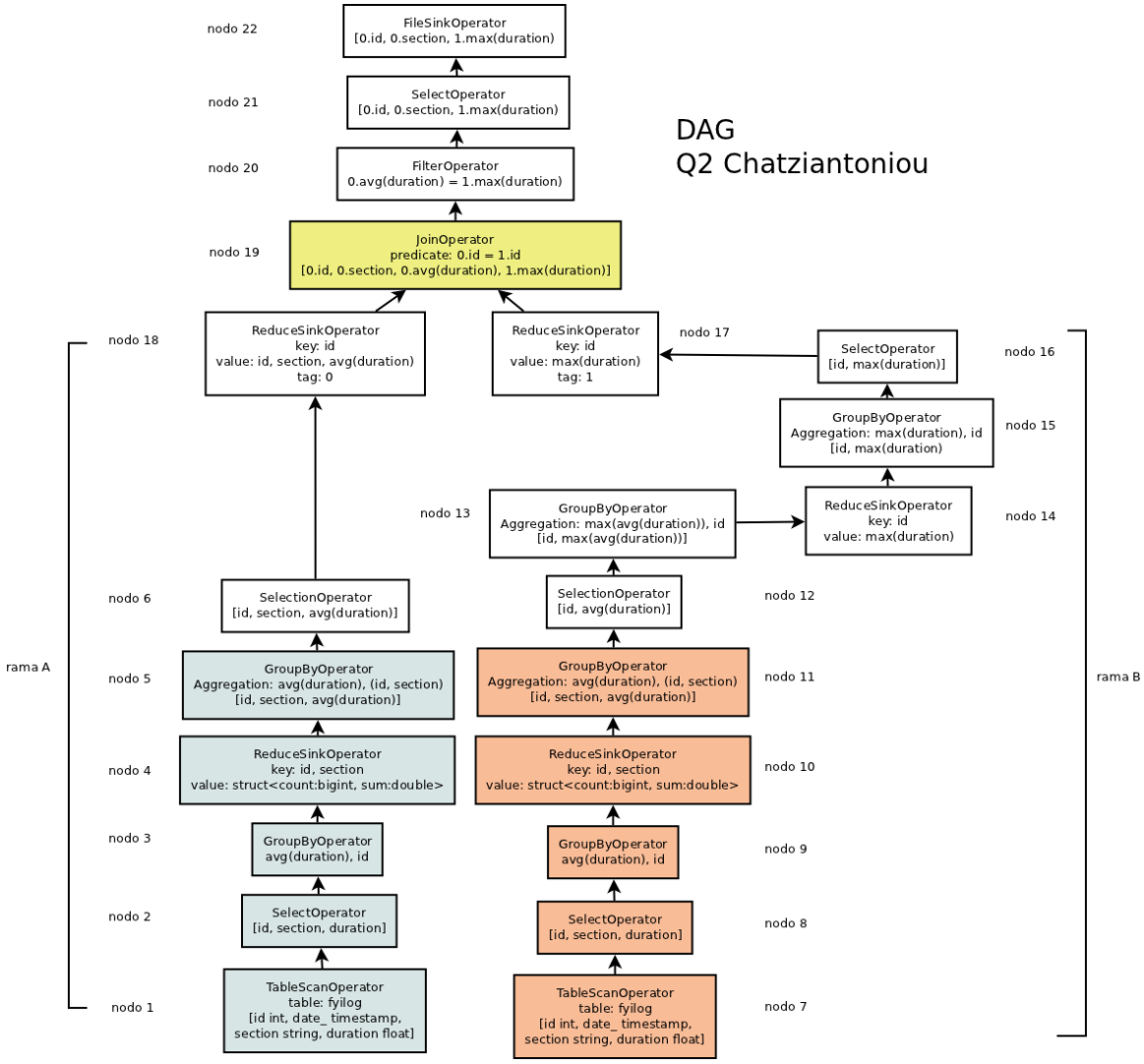


Figura 4.4: DAG de la consulta 2 de Chatziantoniou

tomará como ejemplo la consulta 2 de Chatziantoniou. Para resolver esta consulta se utilizan las consultas *HiveQL* que se observan en el código 4.2. La vista *Q2V1* obtiene el promedio de tiempo que gasta cada usuario en una sección. La vista *Q2V2* obtiene el máximo de los promedios de todas las secciones que visitó cada cliente, esta vista utiliza la vista *Q2V1* para obtener los promedios de cada sección. Por último, la vista *Q2* obtiene el nombre y el promedio del tiempo de la sección donde cada cliente gastó más tiempo; esta vista utiliza como subconsultas las vistas *Q2V1* y *Q2V2*. El *DAG* optimizado de la consulta 2 producido por las optimizaciones de *Hive* se observa en la figura 4.4. Obsérvese que el operador *JoinOperator* del nodo

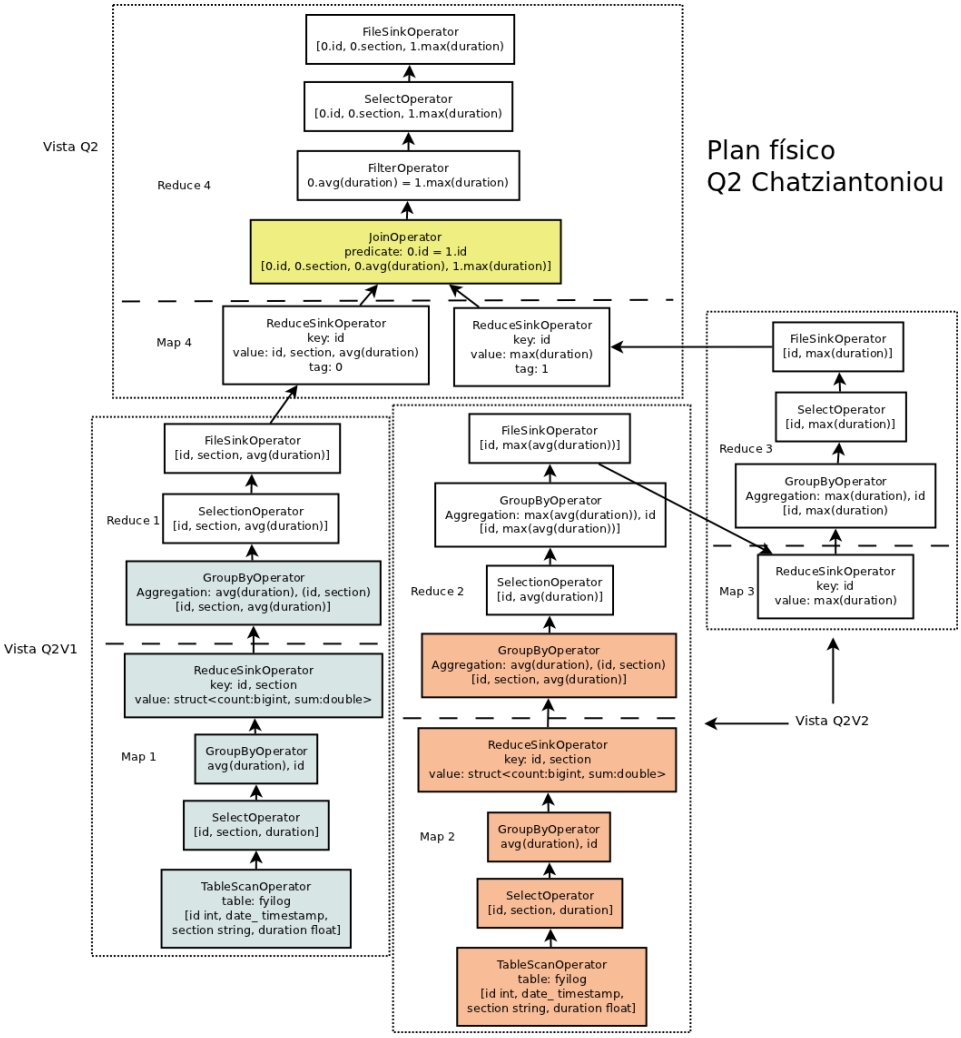


Figura 4.5: Plan físico de la consulta 2 de Chatziantoniou con las optimizaciones actuales de Hive

19 vincula dos ramas simples. Una rama que contiene los nodos del 1 al 6 y el nodo 18 la cual nombraremos rama A, y otra rama que contiene los nodos del 7 al 17 la cual nombraremos rama B. Los nodos del 1 al 6 de la rama A resuelven la vista *Q2V1*, los nodos del 7 al 16 de la rama B resuelven la vista *Q2V2*, y los nodos del 17 al 22 resuelven la vista *Q2*. Obsérvese también que los nodos del 1 al 5 de la rama A realizan las mismas operaciones que los nodos del 7 al 11 de la *rama B*, esto se debe a que la vista *Q2* vincula a través de un *join* las vistas *Q2V1* y *Q2V2*. Sin embargo, la vista *Q2V2* también hace uso de la vista *Q2V1*. Por lo tanto, la vista *Q2V1* se realiza dos veces. Este problema ocasiona la creación de *trabajos mapreduce*

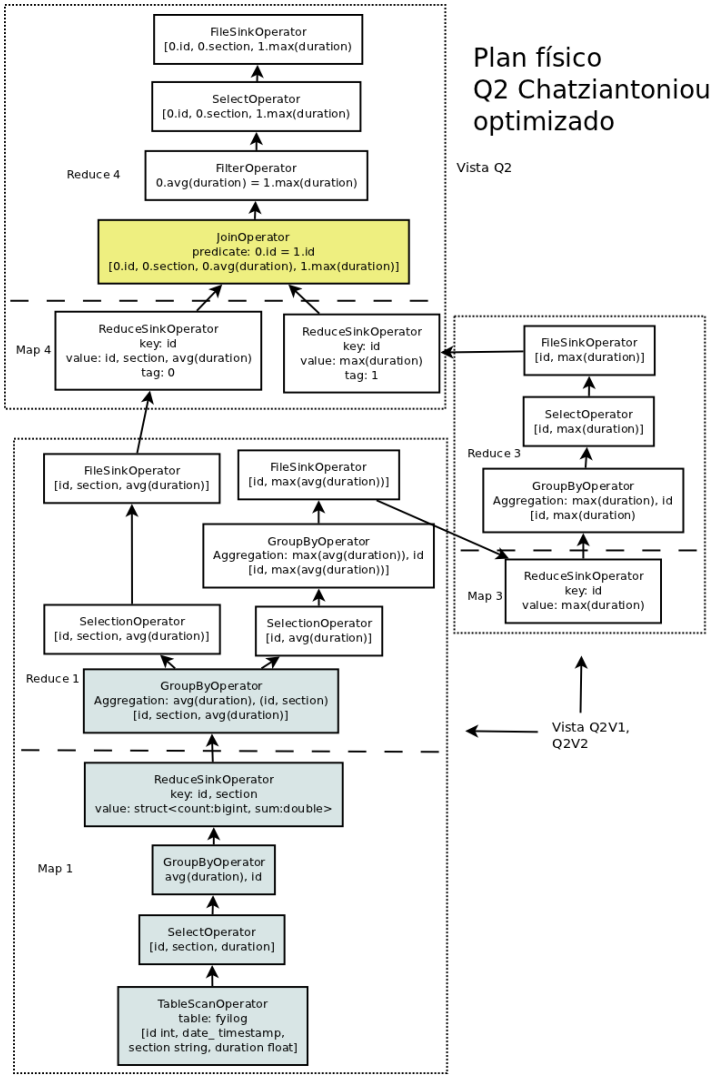


Figura 4.6: Plan físico de la consulta 2 de Chatziantoniou con la optimización propuesta

que realizan operaciones parecidas. La figura 4.5 muestra el plan físico de la consulta 2 de Chatziantoniou que se crea a partir del *DAG* de la figura 4.4. Obsérvese que los *trabajos mapreduce 1 y 2* realizan operaciones similares. Por supuesto, esto no es óptimo debido a que se está realizando una doble lectura de una misma *tabla*. Además, las *tareas map 1 y 2* están enviando los mismos datos por la red a las *tareas reduce 1 y 2* respectivamente, y por último se están realizando las mismas operaciones en ambos *trabajos mapreduce* hasta el operador *GroupByOperator* de las *tareas reduce*.

Aunque *Hive* optimiza un *DAG* para que al transformarse a secuencias de

trabajos mapreduce cada *trabajo mapreduce* tenga un buen desempeño, *Hive* no se da cuenta que algunas operaciones ya las ha realizado y las repite provocando la creación de *trabajos mapreduce* con operaciones similares. Lo que se propone es eliminar los operadores redundantes entre ramas simples vinculadas por un operador *JoinOperator*, de tal manera que las operaciones se realicen una sola vez. Los beneficios que se obtienen al realizar esta propuesta son: Lograr que las lectura de las *tablas* se realicen una sola vez; lograr la optimización del recurso de red al no enviar datos repetidos; y en ocasiones eliminar *trabajos mapreduce* completos. Con esto se lograría un mejor desempeño de *Hive*.

Por ejemplo, si se aplica la propuesta al DAG de la consulta 2 de Chatziantoniou, entonces se generá el plan físico de la figura 4.6. Obsérvese que se ha eliminado el *trabajo mapreduce 2* del plan físico de la figura 4.5. Esto se debe a que se han eliminado los nodos 7 al 11 de la rama B en el *DAG* de la consulta 2, de tal manera que la secuencia de los nodos del 1 al 6 en la rama A sólo se realiza una vez, la lectura de la tabla *fyilog* sólo se realiza una vez, y se optimiza el recurso de la red, al enviar una sola vez los datos entre las *tareas map 1* y la *tareas reduce 1*.

4.3.2. Eliminación de operadores redundantes entre ramas compuestas

El problema de operadores redundantes entre ramas compuestas se puede presentar cuando una consulta *HiveQL* vincula a través de operadores *joins* subconsultas que leen más de una tabla. Entonces, cada subconsulta involucra operadores *joins*.

Por ejemplo, la consulta 3 de Chatziantoniou se puede resolver con las consultas *HiveQL* que se observan en el código 4.3. La vista *Q3V1* obtiene el promedio que gastó cada usuario en la sección *"WORLD"* en las conexiones anteriores a t, donde t es la fecha de cualquier conexión de un usuario a la sección *"WORLD"*. La vista *Q3V2* obtiene el promedio que gastó cada usuario en la sección *"WORLD"* en las conexiones posteriores a t, donde t es la fecha de cualquier conexión de un usuario a la sección

```
1  CREATE VIEW Q3V1 AS
2          SELECT c1.id, c1.date_, avg(c2.duration) as avg_d
3          FROM fyilog c1 JOIN fyilog c2 ON (c1.id = c2.id)
4          WHERE c1.section="WORLD" and c2.section="WORLD" AND c2.date_ <
               c1.date_
5          GROUP BY c1.id, c1.date_;
6
7  CREATE VIEW Q3V2 AS
8          SELECT c1.id, c1.date_, avg(c2.duration) as avg_d
9          FROM fyilog c1 JOIN fyilog c2 ON (c1.id =c2.id)
10         WHERE c1.section="WORLD" and c2.section="WORLD" AND c2.date_ >
               c1.date_
11         GROUP BY c1.id, c1.date_;
12
13 CREATE VIEW Q3 AS
14         SELECT c1.id, c1.date_, c1.avg_d as avg_q3v1, c2.avg_d as
               avg_q3v2
15                 FROM q3v1 c1 JOIN q3v2 c2 ON (c1.id = c2.id and c1.date_
                       = c2.date_);
```

Código 4.3: Sentencias *HiveQL* para resolver la consulta 3 de Chatziantoniou en Hive

"WORLD". Por último, la vista *Q3* obtiene los promedios de tiempo gastados por cada usuario en la sección *"WORLD"* en las conexiones anteriores y posteriores a la conexión t. Para ello, utiliza como subconsultas las vistas *Q3V1* y *Q3V2*. Obsérvese que las vistas *Q3V1* y *Q3V2* vinculan dos tablas y realizan operaciones similares, la única diferencia se encuentra en la cláusula *where* donde se realiza la comparación de las fechas de conexión. En la vista *Q3V1* se encuentran las conexiones menores a una conexión t ($c2.date < c1.date$) y en la vista *Q3V2* se encuentran las conexiones mayores a una conexión t ($c2.date > c1.date$). El uso de ambas subconsultas en la vista *Q3* se refleja en el *DAG* de la consulta que se observa en la figura 4.7. Obsérvese que el operador *JoinOperator* del nodo 28 vincula dos ramas compuestas donde cada rama contiene otro operador *JoinOperator*. Los nodos del 1 al 12 y el nodo 27 constituyen una rama a la cual nombraremos rama A, y los nodos del 13 al 26 constituyen otra rama a la cual nombraremos rama B. Los nodos del 1 al 12 de la rama A resuelven la vista *Q3V1*, los nodos de 13 al 25 de la rama B resuelven la vista *Q3V2*, y los nodos del 26 al 32 resuelven la vista *Q3*. Obsérvese también que los nodos 1 al 7 de la rama A realizan las mismas operaciones que los nodos del 13 al 19 de la rama

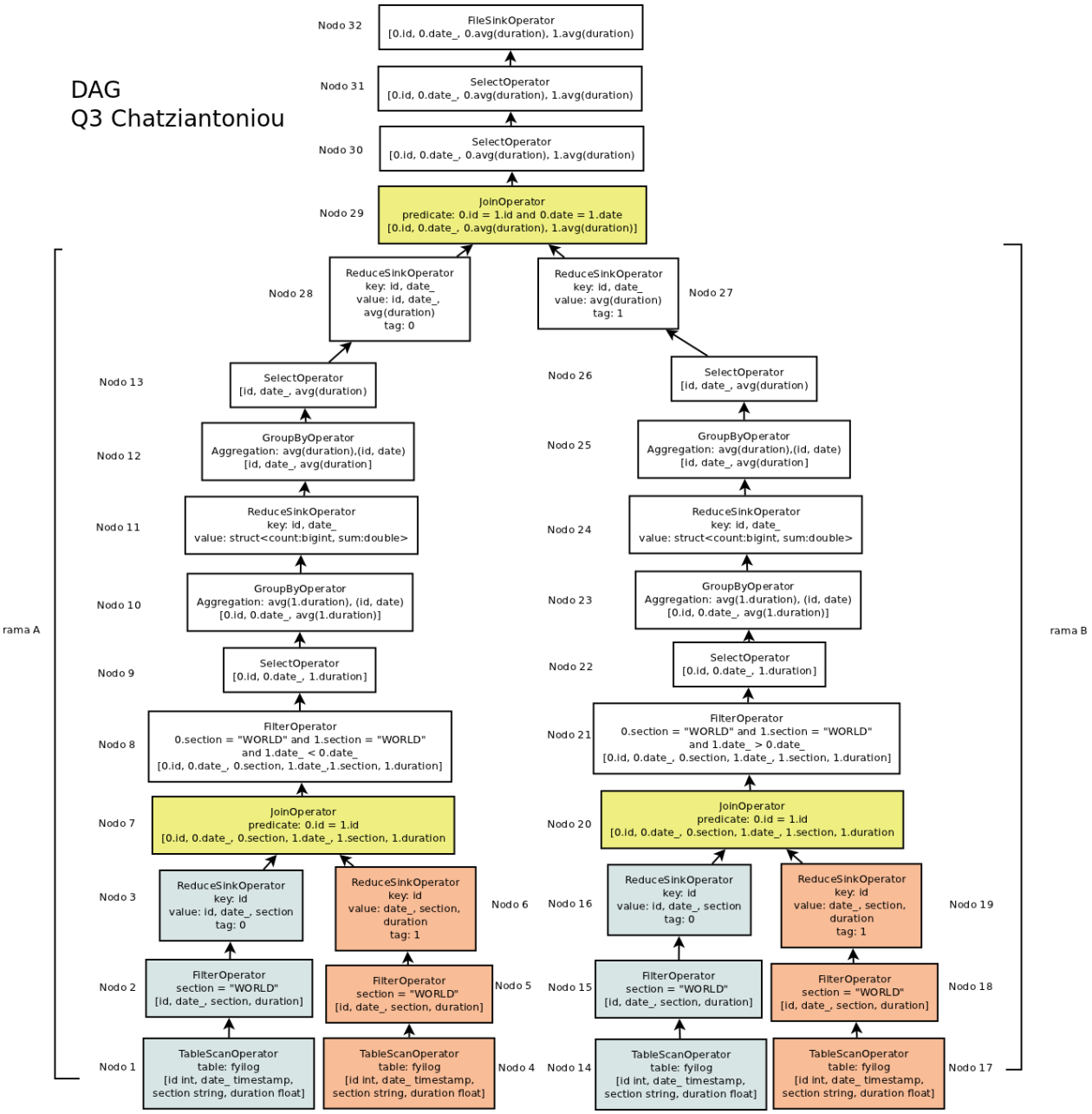


Figura 4.7: DAG de la consulta 3 de Chatziantoniou

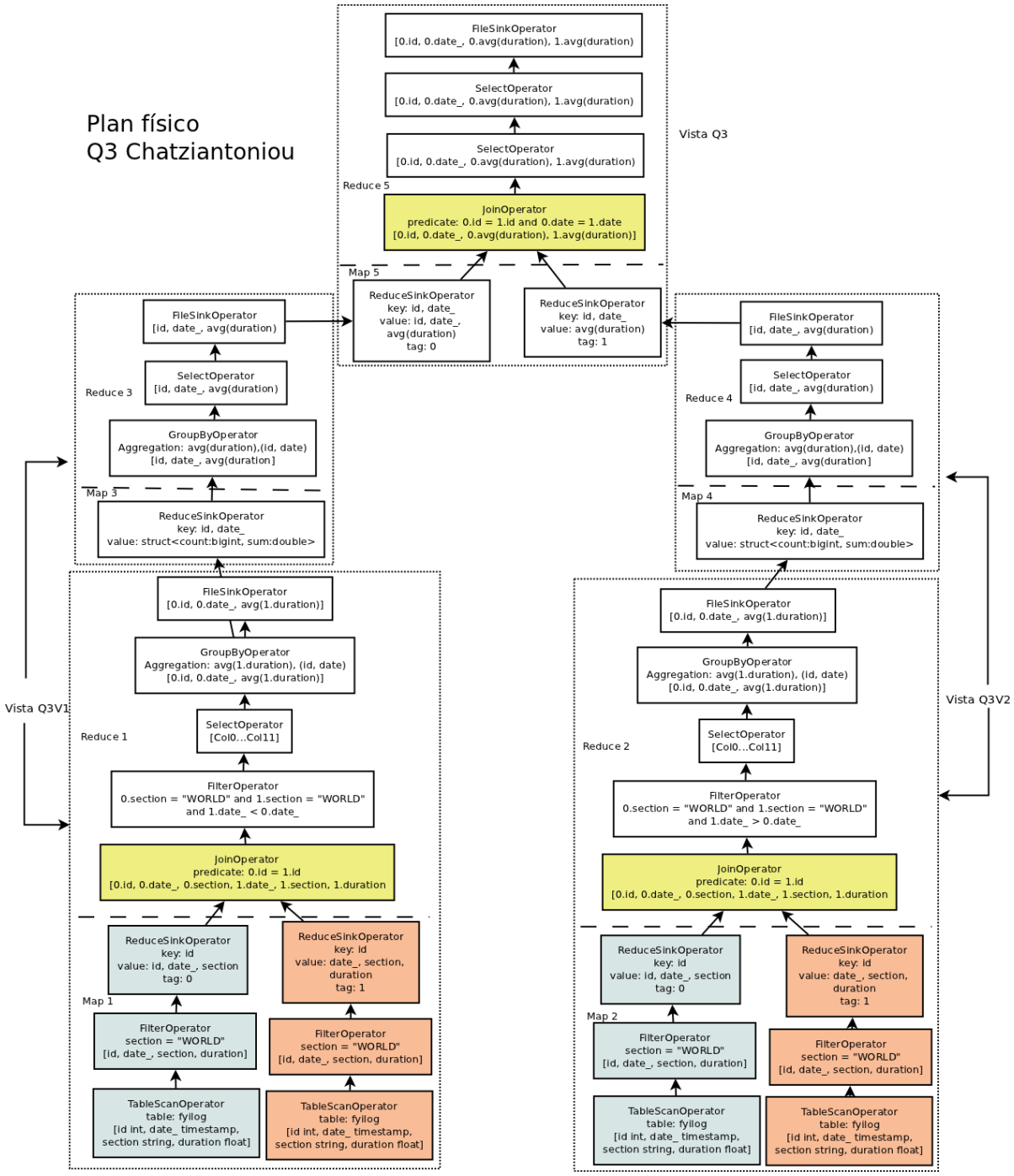


Figura 4.8: Plan físico de la consulta 3 de Chatziantoniou con las optimizaciones actuales de Hive

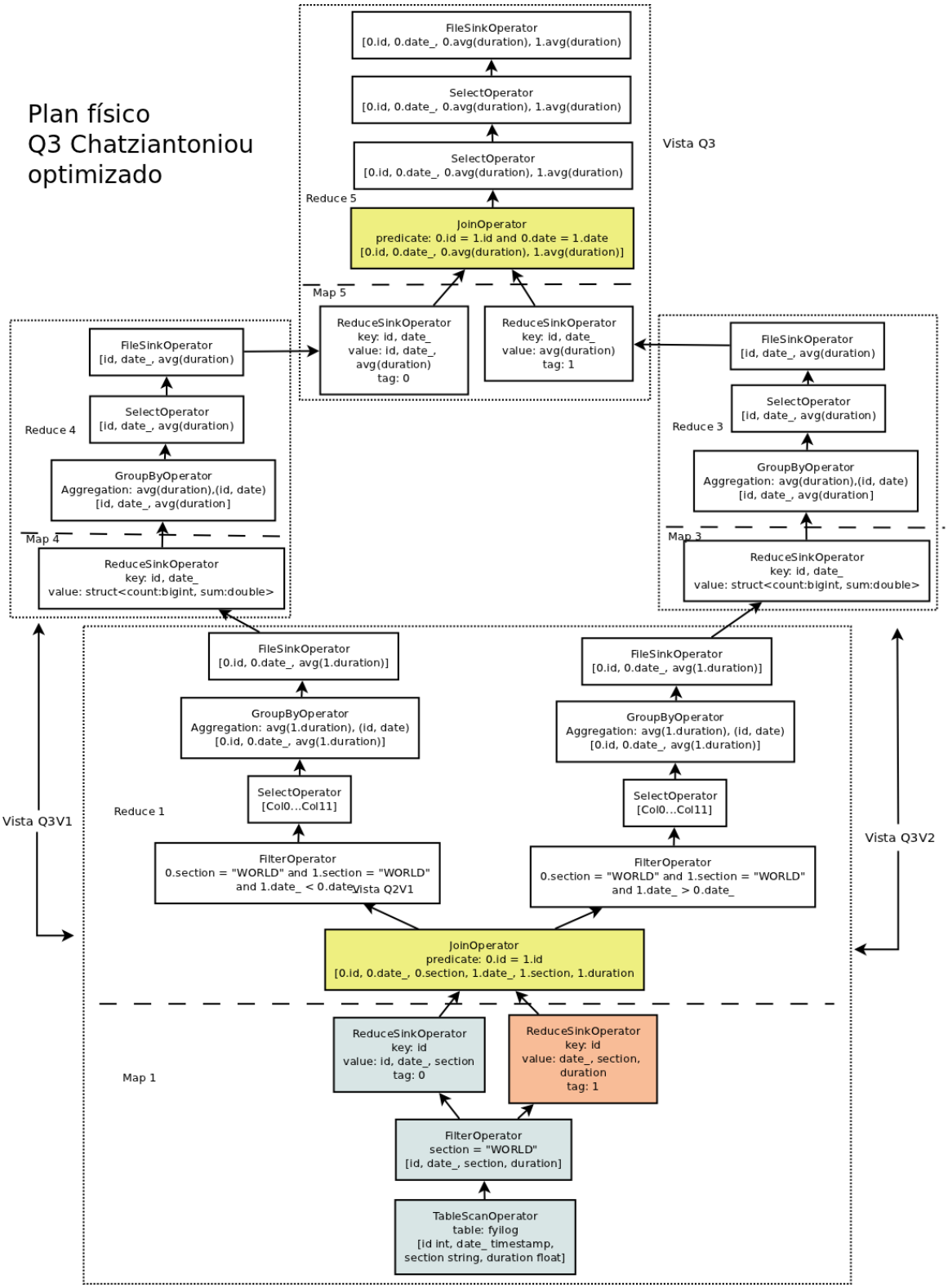


Figura 4.9: Plan físico de la consulta 3 de Chatziantoniou con la optimización propuesta

B, esto se debe a que las vistas *Q3V1* y *Q3V2* realizan operaciones iguales hasta la cláusula *where* que se representa en el *DAG* a través de los operadores *FilterOperator*. Este problema, ocasiona la creación de *trabajos mapreduce* que realizan operaciones parecidas. La figura 4.8 muestra el plan físico de la consulta 3 de Chatziantoniou que se crea a partir del *DAG* de la figura 4.7. Obsérvese que los *trabajos mapreduce 1 y 2* realizan operaciones similares. Además, obsérvese que en las *tareas map 1* se están realizando las mismas operaciones dos veces. Lo mismo sucede en la *tarea map2*. Todos estos inconvenientes provocan que la tabla *fyilog* se lea 4 veces, que se este realizando cuatro veces el mismo procesamiento y que las *tareas map 1 y 2* esten enviando los mismos datos por la red a las *tareas reduce 1 y 2* respectivamente.

Si se aplica la propuesta de eliminar operadores redundantes entre ramas compuestas al *DAG* de la consulta 3 de Chatziantoniou, entonces se generará el plan físico de la figura 4.9. Obsérvese que se ha eliminado el *trabajo mapreduce 2* del plan físico de la figura 4.8. Esto se debe a que se han eliminado los nodos del 13 al 19 del *DAG* de la consulta 3. Además se han eliminado los nodos 4 y 5 debido a que realizaban las mismas operaciones que los nodos 1 y 2. Con esto, se logra que la tabla *fyilog* se lea una sola vez y que las operaciones se realicen una sola vez. Además las *tareas map 1* envían sólo una vez los datos a las *tareas reduce 1*.

Aunque cada subconsulta en la consulta 3 de Chatziantoniou lee un par de tablas, y ambas tablas son las mismas (fyilog), no necesariamente tiene que ser así. Es decir, cada subconsulta puede leer un par de tablas o más que sean diferentes entre sí.

Otro ejemplo se puede observar en la consulta 11 del estudio TPC-H. Esta consulta se puede resolver con las consultas *HiveQL* que se observan en el código 4.4. La vista *Q11V1* encuentra por cada pieza diferente el costo total de todas las piezas de ese tipo que se encuentra en el país *"GERMANY"*. La vista *Q11V2* encuentra el costo de todas las piezas que se encuentran en *stock* del país *"GERMANY"*, para esto ocupa el costo total de cada una de las piezas diferentes disponibles en el país *"GERMANY"* que brinda la vista *Q11V1*. Al final la vista *Q11* por cada pieza muestra el número de pieza y el costo de todas las piezas de ese tipo que se encuentra en el *stock* del país

```
1  create table q11(ps_partkey INT, value DOUBLE);
2
3  CREATE VIEW Q11V1 AS
4      select  ps_partkey, sum(ps_supplycost * ps_availqty) as part_value
5      from  nation n join supplier s
6          on   s.s_nationkey = n.n_nationkey and n.n_name = 'GERMANY'
7          join partsupp ps
8          on  ps.ps_suppkey = s.s_suppkey
9      group by ps_partkey;
10
11 CREATE VIEW Q11V2 AS
12     select  sum(part_value) as total_value
13     from q11_part_tmp;
14
15 insert overwrite table Q11
16 select   ps_partkey, part_value as value
17     from (
18         select ps_partkey, part_value, total_value
19            from Q11V1 join Q11V2
20     ) a
21     where part_value > total_value * 0.0001
22     order by value desc;
```

Código 4.4: Sentencias *HiveQL* para resolver la consulta 11 del estudio TPC-H en Hive

"GERMANY" ordenados de la pieza con un costo total mayor a la pieza con un costo total menor. La vista *Q11V1* involucra el *join* de varias tablas. La vista *Q11V2* utiliza la vista *Q11V1*. Posteriormente, ambas vistas son subconsultas de la consulta *Q11*. Esto se refleja en el *DAG* optimizado por las consultas actuales de *Hive* que se observa en la figura 4.10. Obsérvese que el operador *JoinOperator* del nodo 38 corresponde al *join* de la consulta *Q11* principal y vincula dos ramas compuestas donde cada rama contiene dos operadores *JoinOperator*. Los nodos del 1 al 15 y el nodo 37 constituyen una rama a la cual nombraremos rama A, y los nodos del 16 al 36 constituyen otra rama a la cual nombraremos rama B. Los nodos 1 al 15 de la rama A resuelven la vista *Q11V1*, los dos operadores *JoinOperator* de dicha rama son los operadores *join* en la vista *Q11V1*. Los nodos 16 al 35 de la rama B resuelven la vista *Q11V2*, los dos operadores *JoinOperator* de dicha rama son los operadores *join* en la vista *Q11V2*. Por último los nodos del 36 al 44 resuelven la vista *Q11*. Obsérvese también que los nodos del 1 al 14 de la rama A realizan las mismas operaciones que los nodos del 16

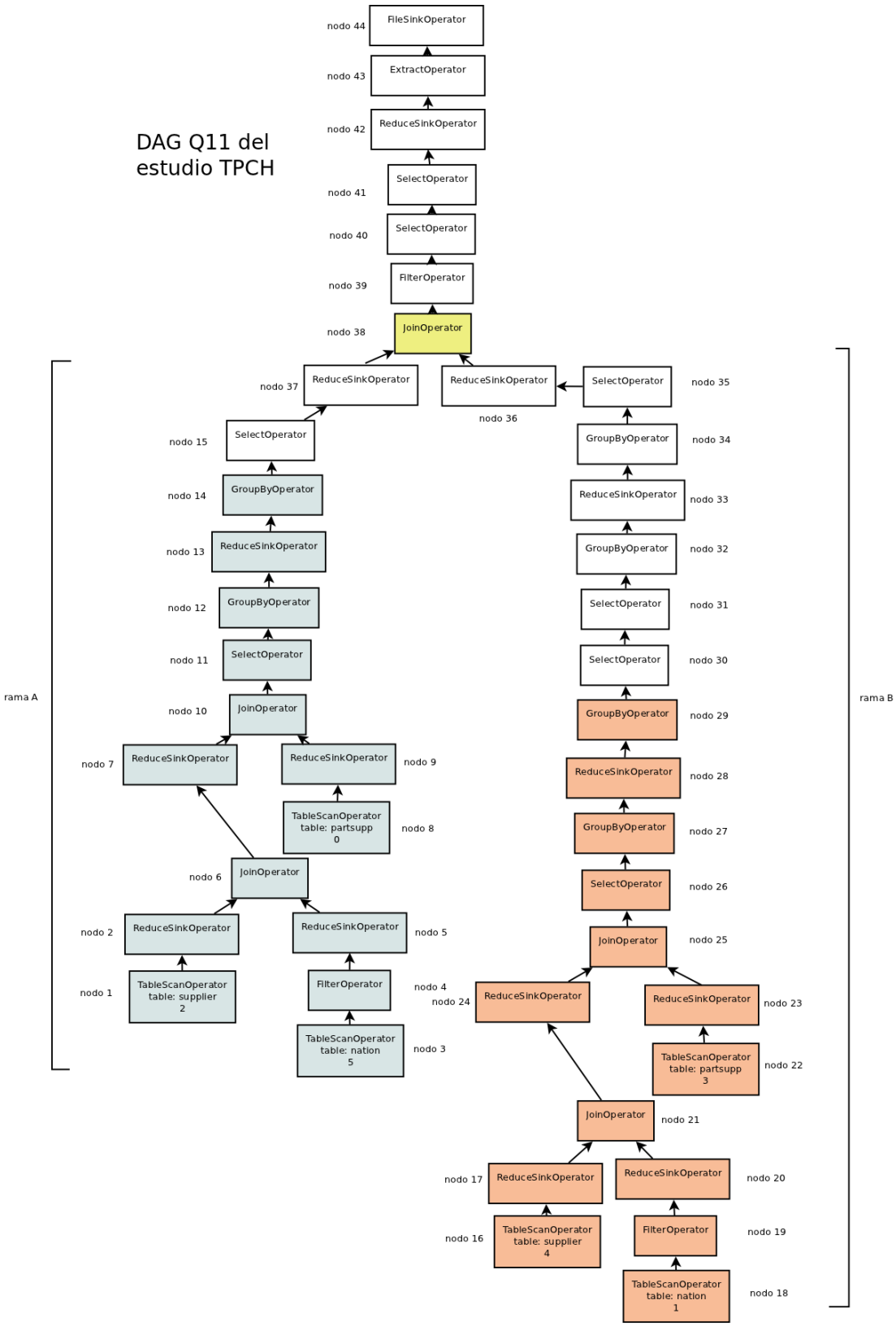


Figura 4.10: DAG de la consulta 11 de TPC-H

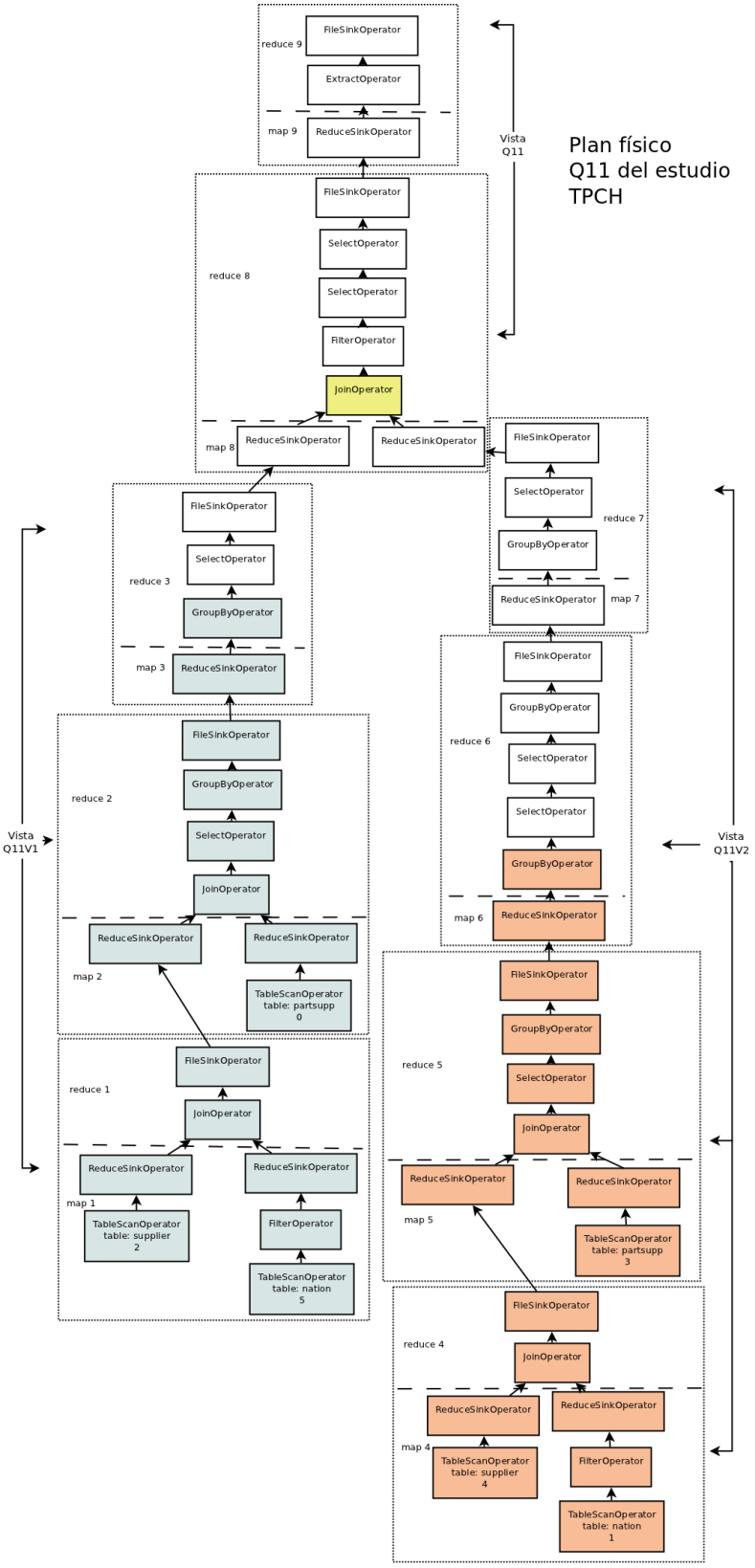


Figura 4.11: Plan físico de la consulta 11 de TPC-H con las optimizaciones actuales de Hive

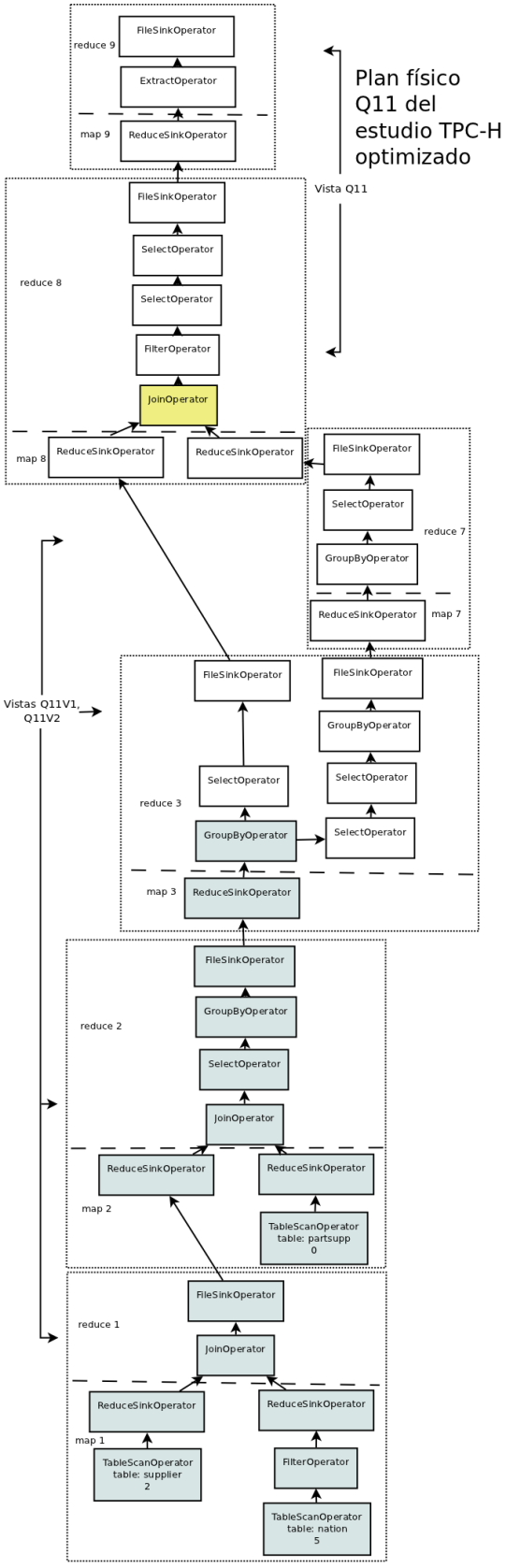


Figura 4.12: Plan físico de la consulta 11 de TPC-H con la optimización propuesta

al 29 de la rama B, esto se debe a que la vista *Q11* utiliza como subconsultas a las vistas *Q11V1* y *Q11V2*. Sin embargo, la vista *Q11V2* también hace uso de la vista *Q11V1* y *Hive* no se da cuenta. Esto ocasiona la creación de varios *trabajos mapreduce* repetidos como se observa en la figura 4.11. Los *trabajos mapreduce 1 y 2* se repiten en los *trabajos mapreduce 6 y 7*, y las operaciones hasta el operador *GroupByOperator* del *trabajo mapreduce 3* se repiten en el *trabajo mapreduce 8*. En *Hive*, por cada cláusula *join* en una consulta *HiveQL* se crea un *trabajo mapreduce* (obsérvese la figura 4.11). Por esta razón, cuando se hace uso repetido de una subconsulta que involucra varios join, se crean varios *trabajos mapreduce* repetidos como es el caso de la vista *Q11V1* en la consulta 11 del estudio *TPC-H*.

Si se aplica la propuesta de eliminar secuencias de operadores redundantes al *DAG* de la consulta 11 del estudio TPC-H, entonces se generará el plan físico de la figura 4.12. Obsérvese que se han eliminado completamente los *trabajos mapreduce 6 y 7*, y el *trabajo mapreduce 8* se ha eliminado hasta el operador *GroupByOperator*. Con esto se logra un mejor desempeño de *Hive*, debido a que se ha eliminado la lectura redundante de tres tablas (*supplier, nation y partsupp*), la eliminación de dos operadores *join*, operación que es una de las que demanda mayor procesamiento en una sentencia *HiveQL*, y se ha conseguido que las *tareas map 1 y 2* envían una sola vez los datos a las *tareas reduce 1 y 2* respectivamente.

En resumen, aunque *Hive* optimiza un *DAG* para que cada *trabajo mapreduce* que se forma del *DAG* se ejecute de manera óptima, *Hive* no se da cuenta que en ocasiones realiza operaciones redundantes. Las operaciones redudantes en un *DAG* provocan que al transformar el *DAG* en *trabajos mapreduce*, se creen *trabajos mapreduce* similares o incluso iguales. Con un poco más de detalle, se ha observado que los operadores redundantes ocasiona problemas de:

- Entrada y salida: Se leen varias tablas de manera redundante. Si cada tabla contiene millones y millones de registros del orden de *Gigabytes, Terabytes, o Petabytes*, leer cientos de *Terabytes* de datos más de una vez por supuesto que puede ocasionar un bajo rendimiento de *Hive*, y si son varias tablas, el problema

es aún más grande.

- Procesamiento redundante: Procesar cientos de *Terabytes* conlleva un tiempo considerado, y si las operaciones son redundantes, entonces el rendimiento de *Hive* se ve disminuido.

- Datos redundantes en la red: Los operadores redundantes pueden crear *tareas map* repetidas en diferentes *trabajos mapreduce* y cada *tarea map* envía los registros procesados a una *tarea reduce* a través de la red. Al haber *tareas map* repetidas, se envían los mismos datos por la red lo cuál no es óptimo sobre todo cuando son millones y millones de registros.

La secuencia de operadores redundates se encuentran en un *DAG* cuando una consulta *HiveQL* involucra operaciones *Join* para vincular N tablas iguales o N subconsultas que realizan operaciones similares o N subconsultas que leen M tablas iguales.

4.4. Optimización del operador JoinOperator

Cuando se realizó la optimización de eliminar operadores redundantes de la consulta 3 de Chatziantoniou, el plan físico resultante fue el mostrado en la figura 4.9 (página 76). En esta figura, obsérvese que en el *trabajo mapreduce* 1, las operaciones anteriores a cada operador *ReduceSinkOperator*(RS) son las mismas. Sin embargo, se utilizan dos operadores *RS* para enviar registros similares al operador *JoinOperator*. Esto se debe a que la operación *Join* es binaria y combina registros de dos relaciones (tablas, subconsultas, vistas). Sin embargo, enviar registros similares dos veces por la red no es óptimo sobre todo cuando son millones y millones de registros. Obsérvese en el mismo *trabajo mapreduce 1* que ambos operadores *RS* configuran como clave del par intermedio $< clave, valor >$ la columna id. Lo único que cambia es que en el operador RS de la izquierda configura como valor las columnas *id, date_, section* y el operador *RS* de la derecha envía las columnas *date_, section, duration*.

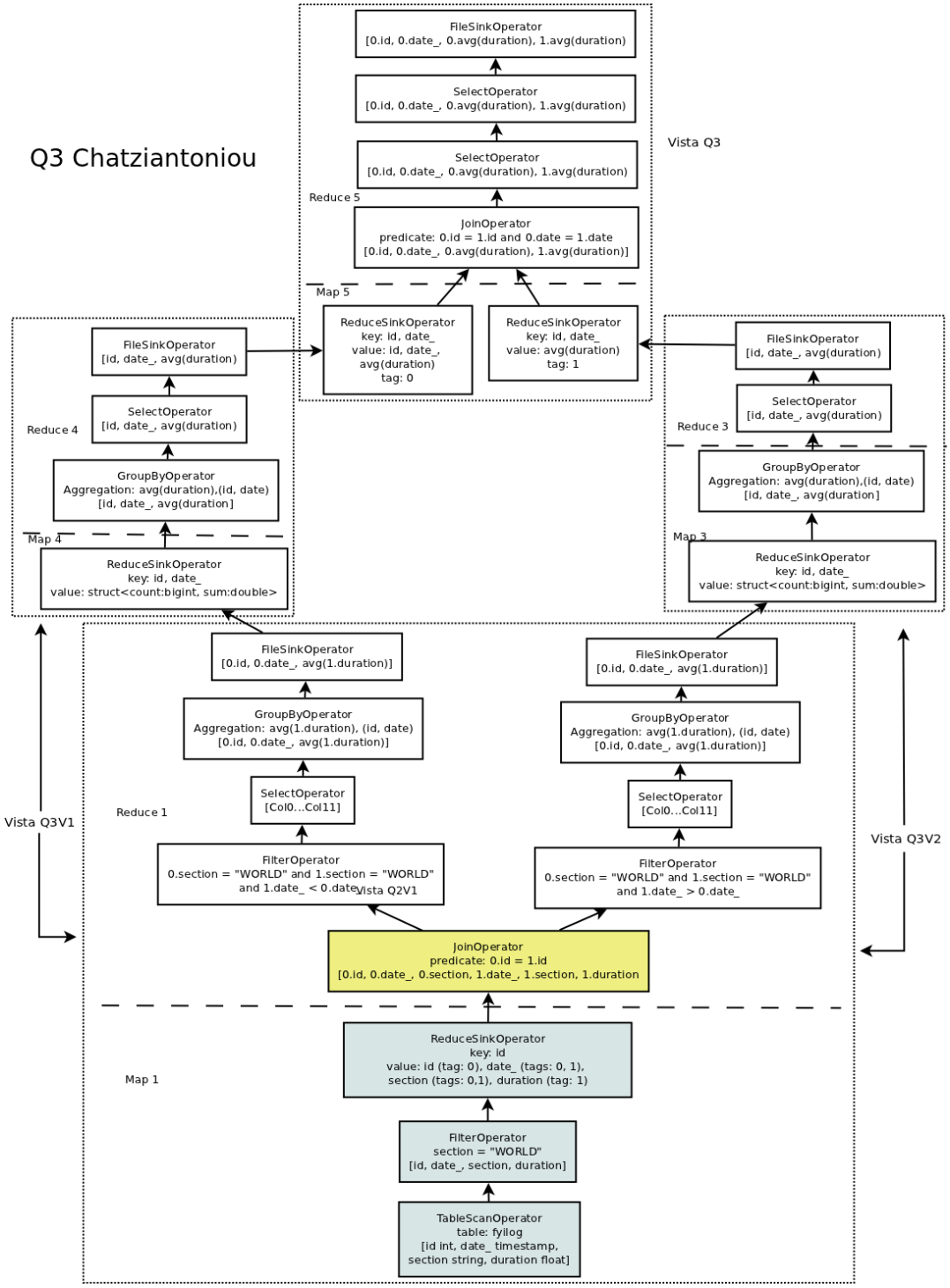


Figura 4.13: Plan físico de la consulta 3 de Chatziantoniou con la optimización del operador *Join* propuesta

Lo que se propone es:

- Identificar los casos en que los operadores anteriores a cada operador *RS* sean los mismos.
- Revisar las columnas que envía cada operador *RS* como valor e identificar las columnas iguales.
- Realizar la unión de las columnas a enviar en ambos *RS* y etiquetar cada columna a la relación a la que pertenece, de tal manera que las columnas repetidas se van a enviar una sola vez.
- Modificar la funcionalidad del operador *RS* de tal manera que acepte columnas etiquetadas para especificar a que relaciones pertenece cada columna.
- Eliminar un operador *RS*.
- Modificar la funcionalidad del operador *JoinOperator* de tal manera, que también reciba una sola entrada, y cuando reciba una sola entrada, entonces sepa que va a realizar el *Join* de los mismo registros, es decir, un *SELF-JOIN*.

De esta manera, los registros similares de una *tarea map* se enviarán una sola vez cuando existan dos operadores *RS*, las operaciones anteriores a ambos *RS* sean las mismas y exista un operador *JoinOperator* en las *tareas reduce*. Esto sucede cuando se realiza un *SELF-JOIN*. Un SELF-JOIN es una operación *JOIN* que vincula a dos relaciones (tablas, subconsultas, vistas) iguales.

La figura 4.13 muestra el plan físico que se forma después de aplicar esta optimización al plan físico de la consulta 3 de Chatziantoniou que se muestra en la figura 4.9. Obsérvese que las operaciones anteriores al operador *RS* se realizan una sola vez y que las columnas se etiquetan, de tal manera, que los registros se envían una sola vez por la red entre las *tareas map y reduce*.

4.5. Eliminación de *trabajos mapreduce* innecesarios en una misma rama de un *DAG*.

Nuestra optimización de eliminar operadores redundantes entre ramas se enfoca en eliminar secuencias de operadores que realizan el mismo procesamiento en diferentes ramas de un *DAG* vinculadas por un operador *JoinOperator*, ocasionando que se eliminen *trabajos mapreduce* redundantes. Sin embargo, en ocasiones en una misma rama del *DAG* existen secuencias de operadores que aunque no son redundantes, crean *trabajos mapreduce* innecesarios, ya que esa secuencia de operadores se pueden realizar con otra secuencia de operadores de la misma rama en un sólo *trabajo mapreduce*. Por ejemplo, en el *DAG* optimizado de la consulta 1 de Chatziantoniou que se forma después de aplicar nuestra optimización de eliminar *trabajos mapreduce* asociados a operadores de agregación y agrupación que se observa en la figura 4.14. En el *trabajo mapreduce 3* se obtiene por cada cliente (id) el número de veces en el que el tiempo que estuvo un cliente en la sección *"WORLD"* fue mayor que el promedio que permaneció en todas las secciones. Se auxilia del *trabajo mapreduce 1* donde se obtuvo el promedio que gastó cada cliente en todas las secciones. Ambos *trabajos mapreduce* se crean con una rama de un *DAG* y se pueden reducir a un sólo *trabajo mapreduce* como se observa en la figura 4.15.

Obsérvese en la figura 4.15 que se ha eliminado el *trabajo mapreduce 3* y sus operaciones se han mezclado con las operaciones del *trabajo mapreduce 1*, de tal modo que en las *tareas map* sólo se lee la tabla *fyilog* una vez y el operador *RS* configura los pares intermedios $< clave, valor >$ de tal modo, que cada *tarea reduce* reciba todos los registros de un cliente, entonces en cada *tarea reduce* se calcula el promedio que gastó cada cliente en todas las secciones y en otra rama se filtran los registros de las veces que un cliente visitó la sección *"WORLD"*, después se realiza el *Join* para vincular el tiempo que gastó cada cliente en la sección *"WORLD"* con el tiempo promedio que gastó en todas las secciones. Posteriormente, filtra los registros en los que el tiempo que gastó en la sección *"WORLD"* fue mayor que el tiempo que

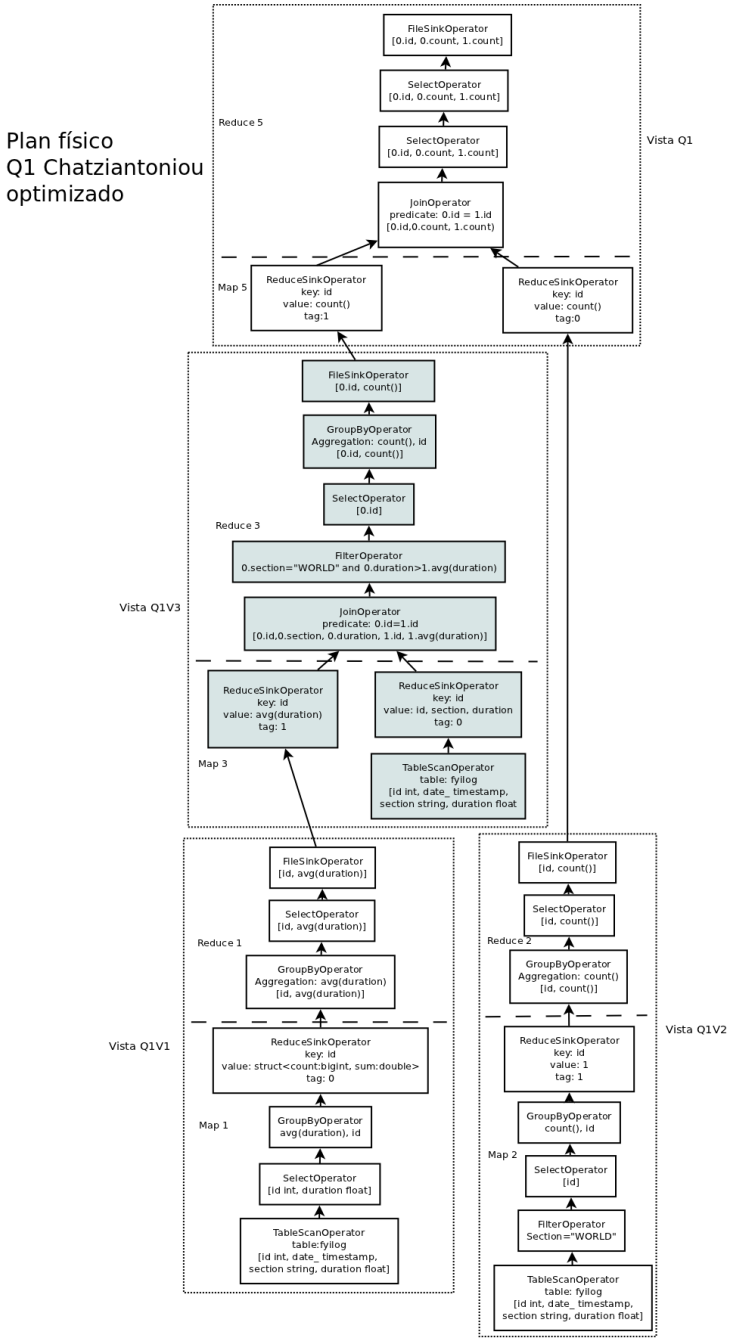


Figura 4.14: Plan físico de la consulta q1c que resulta del *DAG* que se optimizó después aplicar nuestra optimización de eliminar *trabajos mapreduce* asociados a operadores de agregación.

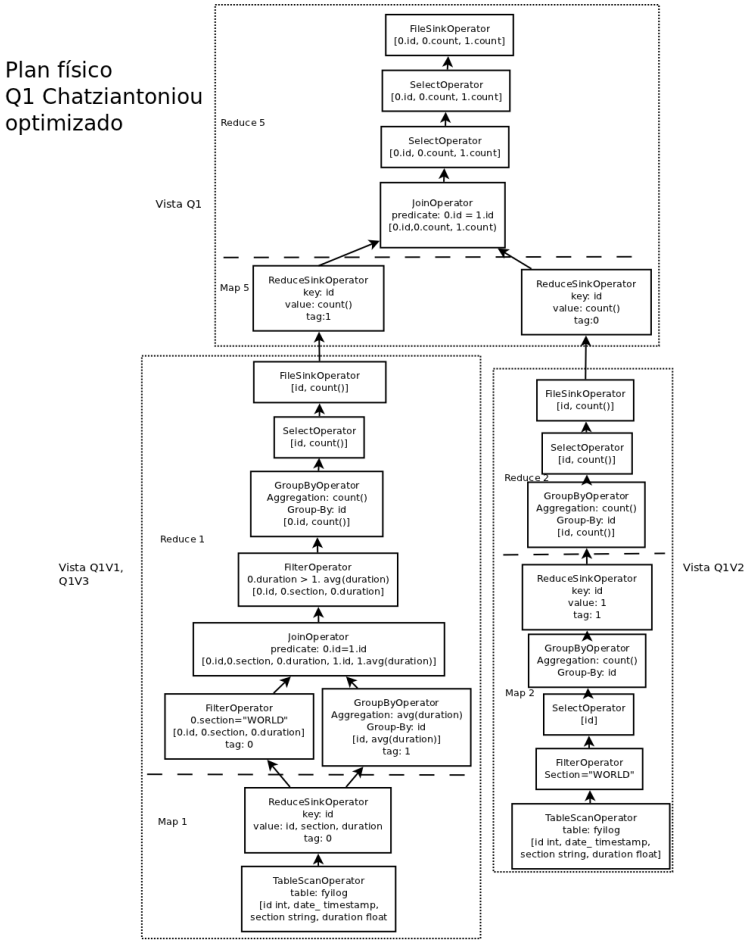


Figura 4.15: Plan físico de la consulta q1c que resulta del *DAG* después de aplicar la optimización propuesta para eliminar *trabajos mapreduce* innecesarios en una misma rama de un *DAG*.

gastó en todas las secciones para al final realizar la cuenta del número de registros que cumplen con esta condición y escribir los resultados al sistema *HDFS* una sola vez.

Con esta optimización se lograría reducir una doble lectura de la misma tabla, eliminar registros repetidos a la red, y además eliminar un escritura al sistema de archivos *HDFS*.

4.6. Resumen

En este capítulo se presentarón las optimizaciones que se pudieron identificar al analizar los *DAG's* que se forman de las consultas *HiveQL* con subconsultas y funciones de agregación y agrupación. Se observaron tres optimizaciones: La primera optimización se enfoca a eliminar secuencia de operadores redundates. Se observó que las optimizaciones actuales de *Hive* optimizan los *DAGs* para que al transformarlos en *trabajos mapreduce*, cada *trabajo mapreduce* se ejecute de manera óptima. Sin embargo, *Hive* no se da cuenta cuando realiza operaciones redundantes causando la generación de *trabajos mapreduce* similares o incluso iguales. Con esta optimización se eliminan *trabajos mapreduce* a veces de manera parcial y en ocasiones totalmente. Esto ocasiona que las tablas se lean una sola vez, las operaciones redundantes se realicen una sola vez y en ocasiones que los registros repetidos se envíen una sola vez por la red.

La segunda optimización se enfoca a eliminar *trabajos mapreduce* relacionados con operadores *GroupByOperator*. Cuando una consulta *HiveQL* involucra funciones de agregación y agrupación que se aplica a registros que provienen de una operación *Join* y/o de otras subconsultas, se ocupan dos *trabajos mapreduce*. Cuando no es necesario el segundo *trabajo mapreduce*, entonces se elimina y la función de agregación se realiza de manera total en las *tareas reduce* del primer *trabajo mapreduce*. Con esta optimización se elimina un *trabajo mapreduce* por cada función de agregación y agrupación que tenga un segundo *trabajo mapreduce* innecesario. De esta manera, se elimina una lectura de registros del sistema *HDFS*, un envío de registros por la red y una escritura de registros al sistema *HDFS* por cada *trabajo mapreduce* eliminado.

La tercera optimización se enfoca a reducir la cantidad de registros que se envían por la red cuando en las *tareas map* de un *trabajo mapreduce* se realizan operaciones similares y en las *tareas reduce* existe el operador *JoinOperator*. El operador *JoinOperator* requiere dos entradas que se envían de las tareas map. Sin embargo, las dos entradas del operador *JoinOperator* reciben registros similares de

las *tareas map*. La optimización consiste en enviar los registros una sola vez por la red y el operador *JoinOperator* reciba una sola entrada y realice el *Join* con los mismos registros. Esta situación se da en consultas que involucran *SELF-JOIN*.

Por último, la cuarta optimización se enfoca a eliminar operadores en una misma rama de un *DAG* que aunque no están repetidos se pueden realizar en otro *trabajo mapreduce* de la misma rama, de tal manera que se eliminen *trabajos mapreduce* innecesarios.

Capítulo 5

Nuestras Optimizaciones realizadas a consultas *HiveQL*

Este capítulo describe el diseño e implementación de las optimizaciones que realizamos.

5.1. Introducción

En el capítulo 4 vimos que el "*DAG* optimizado" que generan las optimizaciones actuales de *Hive* no es óptimo cuando una consulta *HiveQL* involucra subconsultas similares o iguales y/o involucra funciones de agregación y agrupación. En ambos tipos de consultas se crean *trabajos mapreduce* innecesarios. Las optimizaciones analizadas y propuestas en el capítulo 4 tienen el objetivo de eliminar los *trabajos mapreduce* innecesarios.

El optimizador de *Hive* aplica un conjunto de optimizaciones (transformaciones) a un *DAG* de una consulta *HiveQL*. Cada optimización está constituida por cinco entidades: *Node, Rule, GraphWalker, Dispatcher, y Processor* (ver secciones 3.5 y 3.5.2). La entidad *Node* representa un nodo en un *DAG*. La entidad *Rule* permite especificar una regla. Una regla es una secuencia de operadores que se desean encontrar durante el recorrido de un *DAG* para aplicar la optimización. La entidad *GraphWalker*

permite recorrer un *DAG* y por cada nodo que visita manda a llamar al *Dispatcher* quién mantiene las reglas de la optimización y cuando se cumple una regla, manda a llamar al *Processor* para transformar un *DAG* en un *DAG* deseado.

Nuestras optimizaciones se agregarón al optimizador de *Hive* y se aplican después de las optimizaciones actuales de *Hive*. Las optimizaciones que se implementaron son: la optimización de eliminar *trabajos mapreduce* de operadores de agregación y agrupación (*GroupByOperator*), y la optimización de eliminar operadores rendundantes entre ramas simples o complejas. La implementación de ambas optimizaciones siguen la lógica de una optimización en *Hive*. No implementamos la optimización *"join"* de la sección 4.4, ni la optimización de eliminar *trabajos mapreduce* innecesarios en una misma rama de un *DAG* de la sección 4.5 por falta de tiempo, debido a que se requiere modificar el comportamiento de varios operadores como el operador *ReduceSinkOperator*, *JoinOperator*, *FilterOperator*, entre otros, y para realizar la modificación se debe de realizar un estudio que establezca el comportamiento de dichos operadores en diferentes situaciones y poder así adaptar nuestras optimizaciones de manera exitosa.

Nuestra optimización para eliminar *trabajos mapreduce* de operadores de agregación y agrupación utiliza las cinco entidades. Se utiliza la entidad *Rule* debido a que se busca una secuencia estática de operadores para identificar y eliminar los *trabajos mapreduce* innecesarios que son generados por la optimización de reducción de la escritura a *HDFS* por medio de agregaciones parciales en *tareas reduce*.

Nuestra optimización para eliminar operadores redundantes entre ramas simples o complejas utiliza cuatro entidades: *Node, GraphWalker, Dispatcher y Processor*. No se utiliza la entidad *Rule* debido a que las reglas especifican una secuencia estática de operadores a encontrar en un *DAG* y bajo esta optimización la secuencia de operadores redundantes que se buscan eliminar en las ramas no se conocen de antemano.

Las secciones 5.2 y 5.3 describen las implementaciones de nuestras optimizaciones en *Hive* explicando lo que se realiza en cada entidad de cada optimización. La sección

5.4 presenta un trabajo relacionado con nuestras optimizaciones.

5.2. Eliminación de *trabajos mapreduce* asociados a operadores de agregación y agrupación (*GroupByOperator*)

Esta optimización elimina el segundo *trabajo mapreduce* generado por la optimización actual de *Hive* llamada *"Reducción de la escritura a HDFS por medio de agregaciones parciales en tareas reduce"* (ver sección 3.5.1) cuando el segundo *trabajo mapreduce* es innecesario.

La optimizacion de *Hive* crea dos *trabajos mapreduce* para una consulta cuando una función de agregación y agrupación (*avg(), max(), count(),* etcétera) se aplica sobre registros que provienen de un *JOIN* o de otra subconsulta. El primer *trabajo mapreduce* realiza la operación *join* o la subconsulta y el segundo *trabajo mapreduce* lee los registros generados por la operación *join* o la subconsulta y realiza la función de agregación y agrupación implicada. Sin embargo, para reducir la cantidad de registros que escribe el primer *trabajo mapreduce* al sistema *HDFS*, esta optimización realiza la función de agregación implicada de manera parcial al final de las *tareas reduce* del primer *trabajo mapreduce*. Se dice que la función de agregación implicada se realizó de manera parcial porqué *Hive* no verifica si cada *tarea reduce* del primer *trabajo mapreduce* recibe todos los registros de un grupo. Los grupos de registros se forman con los registros que tienen los mismos valores en las columnas que se especifican en la cláusula *Group-By*. Si cada *tarea reduce* del primer *trabajo mapreduce* recibe todos los registros de un grupo, entonces los resultados que se obtienen de aplicar la función de agregación implicada en dichas *tareas reduce* no son parciales, sino son los resultados finales debido a que dicha función de agregación se habrá aplicado sobre todos los registros de un grupo. Y por lo tanto, el segundo *trabajo mapreduce* es innecesario.

Nuestra optimización trabaja con el *DAG*, y elimina del mismo los operadores que

conforman el segundo *trabajo mapreduce* después de verificar que es innecesario.

Esta optimización identifica los dos *trabajos mapreduce* asociados a funciones de agregación y agrupación con ayuda de las entidades *Rule*, *GraphWalker*, y *Dispatcher*; y en la entidad *Processor* verifica si cada *tarea reduce* del primer *trabajo mapreduce* asociado a la función de agregación recibe todos los registros de un grupo, si es el caso, elimina los operadores que conforman el segundo *trabajo mapreduce* asociado a la función de agregación. A continuación se describe la optimización a través de la entidades, la optimización se inicia en la entidad *GraphWalker*.

5.2.1. Rule

La regla, es decir, la secuencia de operadores que se busca durante el recorrido del *DAG* en nuestra optimización es: $RS\,\%.^*\,GBY\,\%RS\,\%GBY\,\%$. Lo que indica es que se desea encontrar un operador *ReduceSinkOperator (RS)* seguido de cualquier secuencia de operadores, seguido de un operador *GroupByOperator (GBY)*, otro operador *RS*, y otro operador *GBY*. Cuando se encuentra esta secuencia de operadores entonces se han identificado los dos *trabajos mapreduce* asociados a funciones de agregación y agrupación. Esto se debe a que como se explicó en la sección 3.6 un *trabajo mapreduce* se compone de *tareas map y reduce*, y el operador *RS* es el mediador entre dichas *tareas*. El primer operador *RS* en la expresión regular indica el final de las *tareas map* del primer *trabajo mapreduce* y los operadores que le siguen contituyen las *tareas reduce* del primer *trabajo mapreduce* hasta el primer operador *GBY* que se encuentra antes del segundo *RS*, el segundo *RS* contituye las *tarea map* del segundo *trabajo mapreduce* y el operador *GBY* pertenece a las *tareas reduce* del segundo *trabajo mapreduce*.

Por ejemplo, en el *DAG* de la consulta 1 de *Chatziantoniou* que se muestra en la figura 5.1 la secuencia de operadores se encuentra del nodo 16 al nodo 22. Obsérvese que en el nodo 16 se encuentra un operador *RS*, si se continúa el recorrido del *DAG* hacia arriba encontramos los operadores *JoinOperator*, *FilterOperator*, *SelectOperator* (cualquier secuencia de operadores), después encontramos un operador *GBY*, seguido

de un operador *RS*, y otro operador *GBY*. Cuando se transforma el *DAG* de la figura 5.1 al plan físico de la figura 5.2, obsérvese que los nodos que se encuentran por la regla forman parte de los *trabajos mapreduce* 3 y 4 los cuáles realizan una operación de agregación y agrupación *count()* sobre los registros que provienen del operador *JoinOperator* del nodo 17.

Una vez que se cumple esta regla, entonces se manda a llamar al *Processor* para ver si se puede eliminar el segundo *trabajo mapreduce*.

5.2.2. GraphWalker

La entidad *GraphWalker* en esta optimización recorre el *DAG* de una consulta utilizando el algoritmo de Búsqueda de Primero Profundidad (del inglés *Depth First Search*, DFS). Por cada nodo del *DAG* que se visita se manda a llamar al *Dispatcher* donde se verifica si se cumple la regla especificada. Para ejemplificar el recorrido de un *DAG* con el algoritmo *DFS* utilizaremos el "*DAG* optimizado" por *Hive* para la consulta 1 de Chatziantoniou que se observa en la figura 5.1. Un *DAG* es recorrido a partir de cada operador *TableScanOperator (TS)*. En la figura 5.1, el recorrido del *DAG* a partir del operador *TS* del nodo 1 es la secuencia de nodos del nodo 1 al 6, luego el nodo 14, luego del nodo 17 al 24, y por último del nodo 26 al 29. Posteriormente, se inicia un nuevo recorrido con el operador *TS* del nodo 7. Con este nodo la secuencia de recorrido sería del nodo 7 al nodo 13, luego el nodo 25. Al llegar al nodo 26, el algoritmo se da cuenta que ya ha recorrido los nodos del 26 al 29 y por lo tanto se termina el recorrido. Después se inicia un nuevo recorrido del *DAG* con el operador *TS* del nodo 15, con este operador se recorre del nodo 15 al 17. Al llegar al nodo 17 el algoritmo se da cuenta que ya ha recorrido todos los nodos posteriores y por lo tanto se termina el recorrido.

Durante el recorrido del *DAG* se va formando una cadena con los nodos que se van visitando y por cada nodo que se visita se llama al *Dispatcher* pasándole dicha cadena. Por ejemplo, cuando se inicia el recorrido con el operador *TS* del nodo 1, entonces se alcanza el operador *SelectOperator (SEL)* y se forma la cadena:

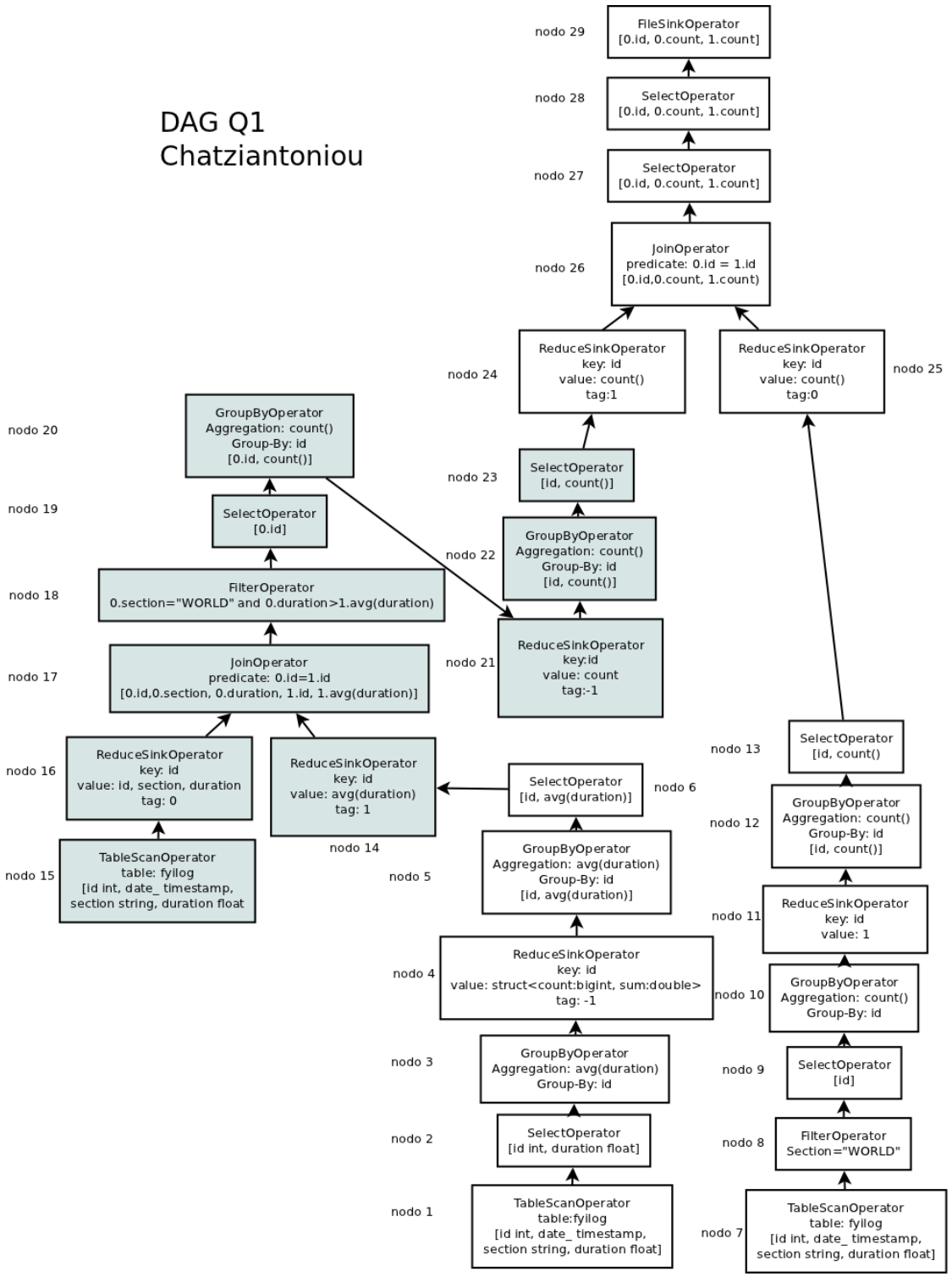


Figura 5.1: *DAG* de la consulta 1 de Chatziantoniou

"TS %SEL %", entonces, se manda a llamar el *Dispatcher* con esta cadena y se verifica si se cumple la regla especificada. Posteriormente, se sigue el recorrido y se alcanza al operador *GBY* del nodo 3, entonces se forma la cadena *"TS %SEL %GBY %"*, y se manda a llamar nuevamente al dispatcher con esta última cadena, el proceso continúa así sucesivamente hasta que se termina de recorrer el *DAG*.

5.2.3. Dispatcher

El *Dispatcher* recibe del *GraphWalker* la cadena que representa los nodos visitados en un *DAG*. En esta entidad se evalúa si la regla especificada se cumple. Por ejemplo, al recorrer el *DAG* de la figura5.1 en el *GraphWalker* con el operador *TS* del nodo 1 y alcanzar el operador *GBY* del nodo 22, la secuencia de operadores recorridos sería: *"TS %SEL %GBY %RS %GBY %SEL % RS %JO %FIL %SEL %GBY %RS %GBY %"*. Obsérvese, que la regla que se busca se cumple en este recorrido en la subcadena: *RS %JO %FIL %SEL %GBY %RS %GBY %*. Por lo tanto, en este momento se manda a llamar al *Processor* con el último nodo que se ha recorrido, en este caso, sería el nodo 22.

5.2.4. Processor

La entidad *Processor* verifica si cada *tarea reduce* del primer *trabajo mapreduce* de una función de agregación y agrupación recibe todos los registros de un grupo especificado en la cláusula *group-by* en una consulta. Si los recibe, entonces la función de agregación que se aplica en cada *tarea reduce* del primer *trabajo mapreduce* genera resultados finales, y por lo tanto, los operadores del segundo *trabajo mapreduce* no son necesarios. La entidad *Processor* realiza dicha verificación con los siguientes pasos:

1. La entidad *Processor* recibe del *Dispatcher* el último nodo en el que se comprueba la regla que se especifica. El último nodo siempre será el operador *GroupByOperator (GBY)* del segundo *trabajo mapreduce* para resolver la función de agregación.

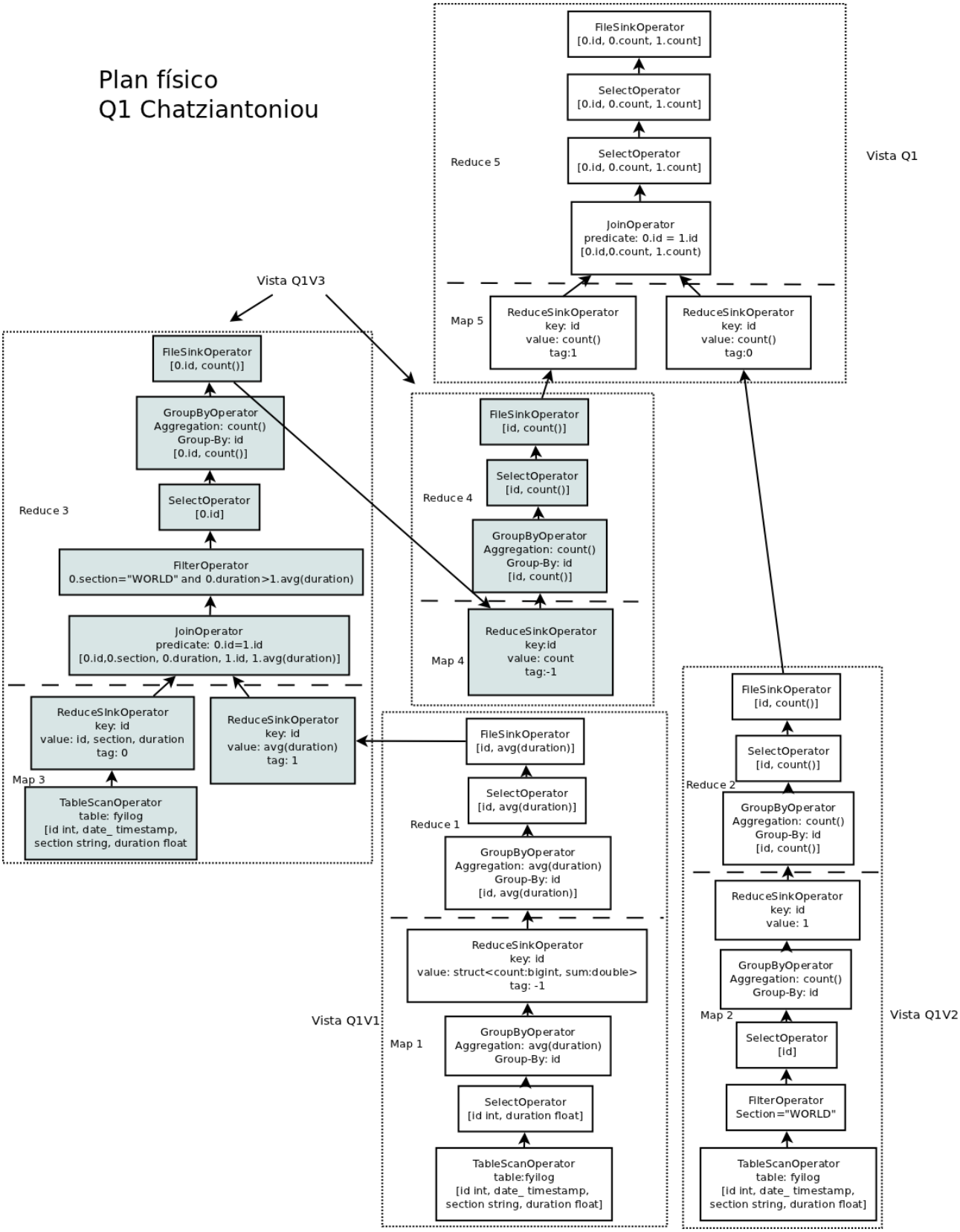


Figura 5.2: *Plan físico* de la consulta 1 de Chatziantoniou.

Por ejemplo, del *DAG* de la consulta 1 de Chaziantoniou (ver figura 5.1) se recibe del *Dispatcher* el operador *GroupByOperator* del nodo 22 que es último operador que se visitó cuando la regla se cumplió. Si observamos el plan físico que se forma a partir de dicho *DAG* en la figura 5.2, el nodo 22 se encuentra en el *trabajo mapreduce 4* el cual es el segundo *trabajo mapreduce* asociado a la operación *GroupByOperator count()*.

2. A partir del nodo *GBY* del segundo *trabajo mapreduce* se retrocede en el recorrido del *DAG*, es decir, se dirige hacia los nodos de abajo hasta encontrar dos operadores *RS* durante el recorrido en reversa. El operador *RS* que se encuentra configura los pares intermedios $< clave, valor >$ de las *tareas map* del primer *trabajo mapreduce* asociado a la función de agregación.

   Por ejemplo, a partir del operador *GroupByOperator* del nodo 22 se retrocede en el recorrido del *DAG* y se alcanza el nodo 21 que es un operador *RS*, y por lo tanto, continuamos retrocediendo en el *DAG* hasta encontrar los operadores *RS* de los nodos 14 o 16. En cualquiera de los dos casos hemos alcanzados dos *RS* durante el recorrido en reversa. Si observamos el plan físico de la consulta 1 de Chatziantoniou en la figura 5.2 los nodos 14 o 16 pertenecen a la *tarea map* del *trabajo mapreduce 3* el cuál es el primer *trabajo mapreduce* asociado a la operación *GroupByOperator count()*.

3. En el operador *RS* que se alcanza se verifica que la clave que utiliza para emitir los pares intermedios $< clave, valor >$ sean las mismas columnas que se utilizan en la cláusula *Group BY* de la consulta. Si es el caso, entonces se eliminan los operadores que se encuentran entre el primer operador *GBY* y el segundo operador *GBY* que se encuentra en la cadena que le pasa la entidad *Dispatcher*. Estos operadores constituyen el segundo *trabajo mapreduce* de una función de agregación. Para eliminar dichos operadores, el primer operador *GBY* debe de tener como nodo hijo (nodo de arriba) al nodo hijo que tiene el segundo operador *GBY*, y el hijo del segundo operador *GBY* debe de tener como nodo

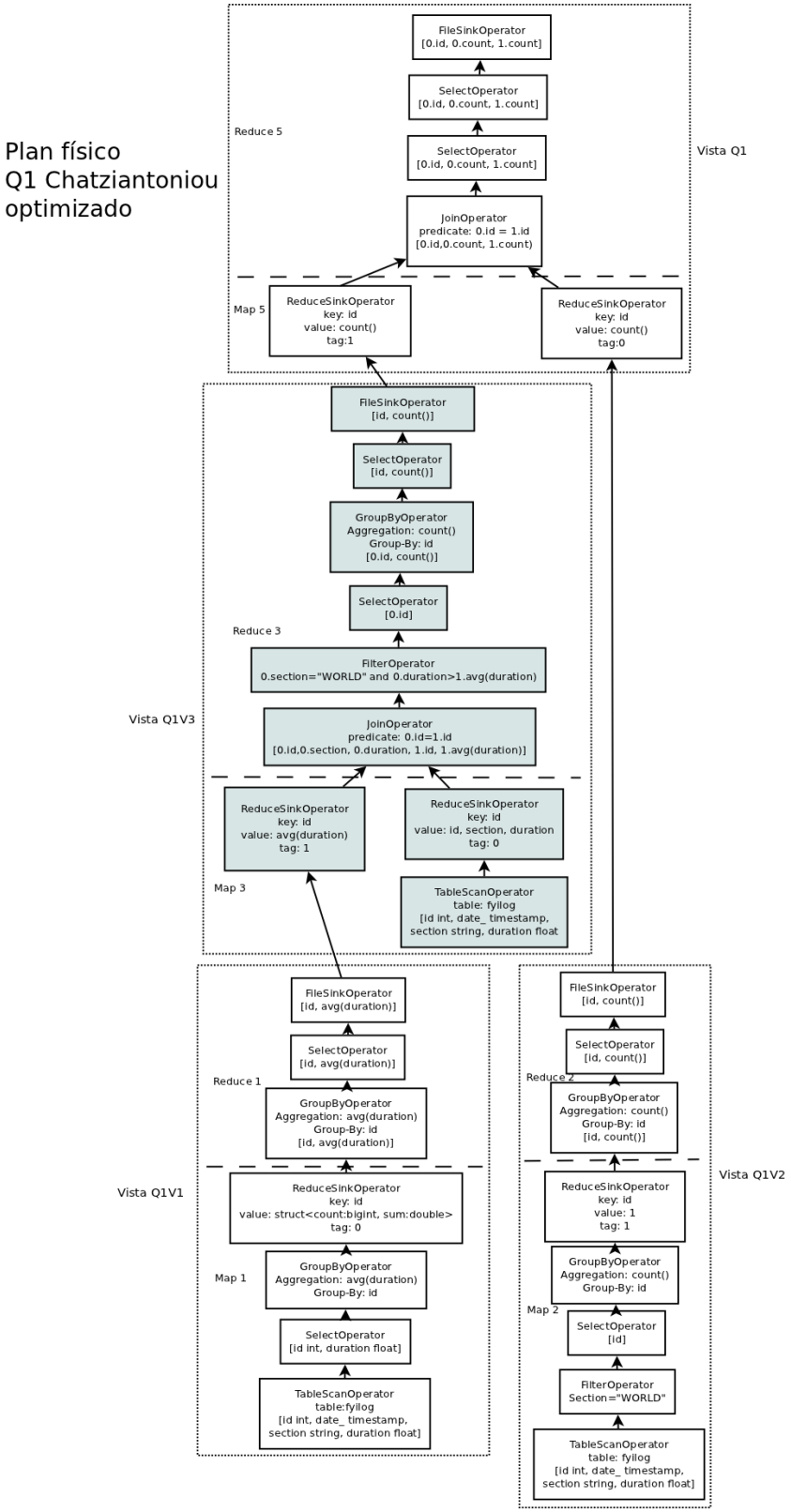


Figura 5.3: *Plan físico* de la consulta 1 de Chatziantoniou después de aplicar nuestra optimización al *DAG* de la consulta.

padre (nodo de abajo) al primer operador *GBY*.

Por ejemplo, el operador *RS* del nodo 14 o 16 que se encontró en el paso anterior y que se encuentra en el *trabajo mapreduce 3* de la figura 5.2 configura los pares intermedios $< clave, valor >$ que emiten con la clave id, la cual es la misma columna que se utiliza para agrupar los datos en el operador *GroupByOperator* de la *tarea reduce* en el *trabajo mapreduce 3*. Por lo tanto, cada *tarea reduce* del *trabajo mapreduce 3* recibe todos los registros con el mismo id, y la función de agregación *count()* que se realiza en el operador *GBY* en la *tarea reduce 3* se aplica sobre todos los registros de un grupo. Por lo tanto, el *trabajo mapreduce 4* no es necesario y se debe de eliminar. Para eliminar los operadores entre el operador *GBY* del nodo 20 y el operador *GBY* del nodo 22, el nodo 20 debe de tener como hijo (nodo de arriba) al nodo 23 y el nodo 23 debe de tener como padre (nodo de abajo) al nodo 20. La figura 5.3 muestra el plan físico que se forma con el *DAG* de la consulta 1 después de aplicar esta optimización. Obsérvese que el *trabajo mapreduce 4* se ha eliminado.

5.3. Eliminación de operadores redundantes entre ramas simples o compuestas

Esta optimización elimina operadores redundantes entre ramas simples o compuestas que están vinculadas por un operador *JoinOperator (JO)* en un *DAG*. Las ramas simples son aquellas que no tienen ramificaciones y que están vinculadas por un sólo operador *JO*. Las ramas compuestas están constituidas por varias ramificaciones donde cada ramificación tiene un *JO*, y al final todos las ramificaciones convergen en un sólo *JO*.

La optimización inicia en la entidad *GraphWalker*, esta entidad encuentra los *TablesScanOperators (TSs)* que leen las mismas tablas en diferentes ramas simples o compuestas. Estos *TSs* identifican ramas que probablemente tengan operadores

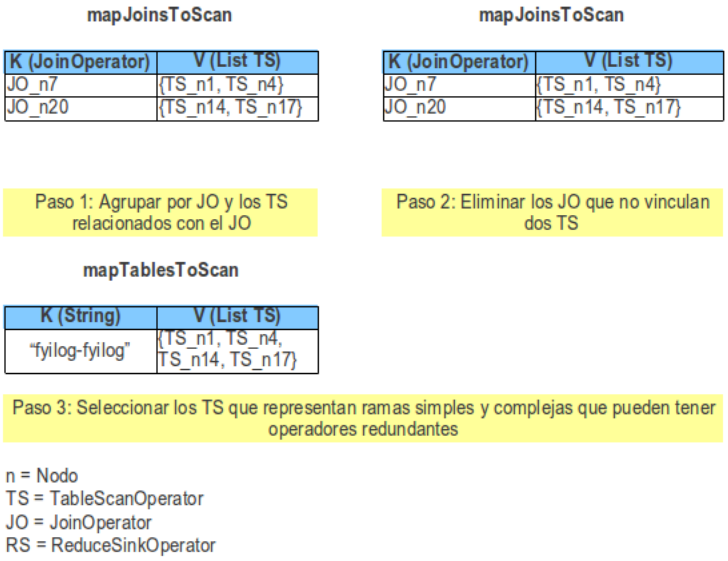


Figura 5.4: Nuestras estructuras de datos utilizadas en la entidad *GraphWalker* de la optimización para eliminar secuencia de operadores *Hive* redundante en un DAG. Los valores corresponden a la ejecución de la entidad *GraphWalker* en el *"DAG optimizado"* que entregan las optimizaciones actuales de *Hive* para la consulta 3 de Chatziantoniou.

redundantes y se les pasa a la entidad *Dispatcher* donde se identifican los operadores redundantes en las ramas implicadas y en la entidad *Processor* se eliminan dichos operadores redundantes. A continuación se describe con más detalle lo que se realiza en cada entidad.

5.3.1. GraphWalker

En esta optimización, la entidad *GraphWalker* encuentra los *TSs* que leen las mismas tablas en diferentes ramas simples o complejas. Para ilustrar como funciona nuestro *GraphWalker* vamos a ilustrar su funcionamiento con el *DAG* de la consulta 3 de Chatziantoniou que se muestra en la figura 5.5. Los pasos que se realizan para identificar los *TSs* que leen las mismas tablas en diferentes ramas son:

1. Primero por cada operador *TS* de un *DAG* se recorre el mismo hasta encontrar un operador *JO* y se guarda en un mapa[1] llamado *mapJoinsToScan*, guardando

[1]un mapa es una estructura de datos donde cada registro se constituye por una clave y un valor. La clave no se repite e identifica de manera única a un valor

en dicho mapa como clave el JO y como valor el *TS*. Por ejemplo, cuando se inicia el recorrido del *DAG* de la figura 5.5 con el nodo 1 se alcanza al *JO* del nodo 7. Entonces se añade el *JO_7* como clave y el *TS_1* como valor al mapa *mapJoinsToScan* (del paso 1) de la figura 5.4, esta figura muestra las estructuras de datos que añadimos para auxiliar el proceso de nuestra optimización. Los números 7 y 1 indican el número de nodo del operador.

Posteriormente, se inicia un nuevo recorrido del *DAG* con el operador *TS* del nodo 4, con este operador se alcanza nuevamente al operador *JO* del nodo 7. Por lo tanto, se agregar el operador *TS_4* a la lista que se asocia con el *JO* del nodo 7 en *mapJoinsToScan*. De esta manera, dicho operador *JO* ya asocia dos *TS* como valor y representa a una rama compuesta. Lo mismo se realiza con los operadores *TS* de los nodos 14 y 17 que alcanzan al operador *JO* del nodo 20. Estos dos nodos *TS* representan a otra rama compuesta. Durante esta fase el mapa *mapJoinsToScan* se llena con los valores que se observan en el paso 1 en la figura 5.4.

2. Después, si sólo se encontró un *JO* significa que el *JO* vincula ramas simples que están representadas por los dos *TSs* que vincula. Entonces, se comprueba que los dos *TSs* lean la misma tabla, y si es el caso, entonces significa que ambas ramas simples pueden tener operadores redundantes, y por lo tanto, los dos *TS* se agregan como valor al map *mapTablesToScan* y como clave el nombre de la tabla que leen ambos *TS*.

   Si se encontró más de un *JO* significa que cada *JO* pertenece a una rama compuesta. Entonces, por cada JO del mapa *mapJoinsToScan* se recupera los dos *TS* y se conoce las dos tablas que leen dichos TS (TS que representan un rama compuesta), entonces se busca si existen otros dos *TS* en otro *JO* del mapa *mapJoinsToScan* que lean el mismo par de tablas (TS que representan otra rama compuesta). Si es el caso, se agregan como valor los 4 o más *TS* al mapa *mapTablesToScan* y como clave un string que indica las dos tablas que

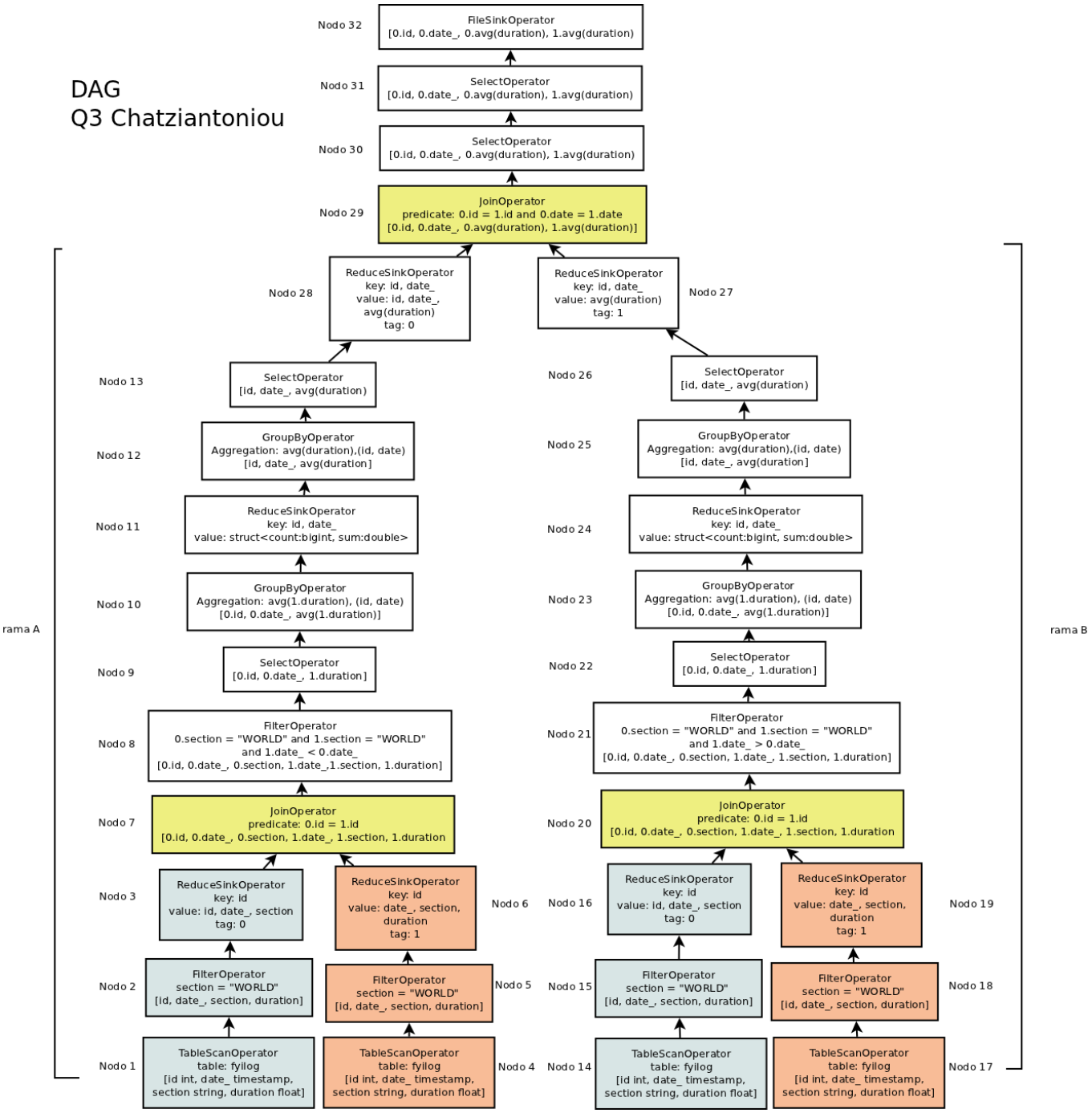


Figura 5.5: *DAG* de la consulta 3 de Chatziantoniou

lee cada par de *TS*, donde cada par de *TS* representa una rama compuesta.

Por ejemplo, en la consulta 3 de Chatziantoniou, el mapa *mapJoinsToScan* tiene dos *JO*. Se obtienen los dos *TS* del primer *JO* (nodos 1 y 4), en este caso ambos *TS* leen la tabla fyilog. Después se obtienen los otros dos *TS* del segundo *JO* (nodos 14 y 17), como ambos pares de *TS* leen el mismo par de tablas "fyilog-fyilog", entonces los 4 TS se guardar como valor en el mapa *mapTablesToScan* y se utiliza como clave el string "fyilog-fyilog" (ver mapa *mapTablesToScan* en paso 3 de la figura 5.4). Cabe mencionar que cada par de *TSs* representa una rama compuesta y no forzasamente tienen que leer la misma tabla como en este caso, puede ser cualquier par de tablas X y Y, pero ambas ramas compuestas deben de leer el mismo par de tablas.

3. Una vez que se han identificado los *TSs* que representan ramas simples o complejas que pueden tener operadores redundantes, entonces manda a llamar al *Dispatcher* y le envía el mapa *mapTablesToScan*.

Si el mapa *mapTablesToScan* contiene un registro con un sólo par de *TSs*, entonces se han identificado ramas simples, y cada *TS* representa una rama simple que lee la misma tabla. Por el contrario, si el mapa *mapTablesToScan* contiene registros con 4 o más operadores *TS* como valor, entonces cada registro almacena ramas compuestas que leen el mismo par de tablas, es decir, cada par de *TSs* en cada registro representan un rama compuesta y leen el mismo par de tablas.

Al final, la entidad *GraphWalker* obtiene los operadores *TS* que leen las mismas tablas en diferentes ramas simples o compuestas. Estas ramas pueden tener operadores redundantes que se encuentran en la entidad *Dispatcher*.

5.3.2. Dispatcher

En nuestra optimización, el objetivo de la entidad *Dispatcher* es encontrar la secuencia de operadores redundantes en las ramas de un *DAG*. Para lograrlo recibe del *GraphWalker* el mapa *mapTablesToScan* que contiene los *TS* de las ramas simples

o complejas que pueden tener operadores redundantes. Se inician dos recorridos un recorrido con un *TS* de una rama que lee una tabla, y otro recorrido con otro *TS* de otra rama que lee la misma tabla. Durante el recorrido se comparan si los nodos de ambas ramas son iguales. En el momento que los nodos de ambas ramas ya no son iguales, entonces se detienen ambos recorridos y se llama a la entidad *Processor* con los últimos nodos hasta donde ambas ramas son iguales.

Dos nodos son iguales si:

- Ambos nodos tienen el mismo operador (TS, FilterOperator, JO, entre otros).
- Las columnas de salida son las mismas en ambos nodos. Ver apéndice A para más detalle.
- Si son nodos con operadores *FilterOperator*, entonces la condicional de ambos operadores es la misma.
- Si son nodos con operadores *RS*, entonces ambos operadores deben de configurar las mismas columnas como clave, y la mismas columnas como valor en los pares $< clave, valor >$ que emiten. Ver apéndice A para más detalle.

A continuación se describe con más detalle lo que se realiza en la entidad *Dispatcher* de nuestra optimización:

1. Primero se obtiene la lista *TS* de cada elemento del mapa *mapTablesToScan* y se realiza el paso 2 o 3.
2. Si la lista contiene dos *TS*, entonces se inicia el recorrido de ambas ramas a partir de los dos *TS*. Cuando se recorre un nodo en una rama, también se recorre un nodo en la otra rama. Por cada nodo que se alcanza en una rama se compara con el nodo que se alcanza en otra rama. Si los nodos son iguales se continúa ambos recorridos hasta encontrar un par de nodos que no sean iguales. En este momento, se manda a llamar al *Processor* con los últimos nodos en los que son iguales ambos recorridos.

3. Si la lista de *TS* contiene cuatro o más *TS*. Simplifiquemos y supongámos que sólo contiene cuatro *TSs* que nombraremos: *TS_1, TS_2, TS_3, TS_4*. Los nodos *TS_1 y TS_2* representan una rama compuesta y leen el mismo par de tablas que los nodos *TS_3 y TS_4* que representan otra rama compuesta.

   Entonces, se inician dos recorridos: un recorrido con el *TS_1* y otro recorrido con el *TS_3*. Cuando se recorre un nodo con el operador *TS_1*, también se recorre un nodo con el operador *TS_3*. Por cada nodo que se alcanza en una rama se compara con el nodo que se alcanza en la otra rama. Si los nodos son iguales se continúa ambos recorridos hasta encontrar un par de nodos que no sean iguales. En este momento, se manda a llamar al *Processor* con los últimos nodos en los que son iguales ambos recorridos.

   Posteriormente, se inician otros dos recorridos: uno con el nodo *TS_2* y otro con el nodo *TS_4* hasta encontrar nodos que no son iguales o alcanzar nodos que ya han sido visitado por los dos recorridos anteriores. Si se alcanzan nodos que no son iguales, entonces se manda a llamar al *Processor* con los últimos nodos en los que son iguales ambos recorridos, por el contrario si se alcanza nodos que ya han sido visitado por los recorridos anteriores, entonces no se hace nada.

   Por último si los *TS_1* y *TS_2* leen la misma tabla, entonces ir al paso 1.

   Por ejemplo, para la consulta 3 de *Chatziantoniou* se obtiene del *GraphWalker* el mapa *mapTablesToScan* que contiene un sólo elemento como se observa en el paso 3 de la figura 5.4. Este elemento contiene una lista de 4 *TSs*: *TS_1, TS_4, TS_14, TS_17*. Los nodos *TS_1 y TS_4* representan una rama compuesta que leen la tabla *fyilog* en ambos nodos. Los nodos *TS_14 y TS_17* representan otra rama compuesta que leen el mismo par de tablas, en este caso la misma tabla fyilog dos veces.

   Entonces se inician dos recorridos, un recorrido con *TS_1* (TS del nodo 1), y otro recorrido con *TS_14* (TS del nodo 14). Con el *TS* del nodo 1 se alcanza el operador *FilterOperator* del nodo 2 y con el *TS* del nodo 14 se alcanza el

operador *FilterOperator* del nodo 15. Ambos nodos son iguales porque cumplen con las condiciones que se describieron anteriormente. Si se continúan ambos recorridos los nodos que se alcancen seguirán siendo iguales hasta los operadores *JO* del nodo 7 y 20. Al alcanzar los operadores *FilterOperators* del nodo 8 y 21 respectivamente ya no son iguales debido a que la condicional no es la misma en ambos nodos. Por lo tanto, se detiene el recorrido y **se manda a llamar al *Processor* con los nodos 8 y 21**.

Posteriormente, se inician dos nuevos recorridos: un recorrido con el *TS_4* (TS del nodo 4) y otro recorrido con el *TS_17* (TS del nodo 17). En ambos recorridos se van comparando los nodos que se van visitando, hasta alcanzar a los operadores *JO* de los nodos 8 y 21 donde nos damos cuenta que ya fueron visitados por los dos recorridos anteriores y por lo tanto, se detiene el recorrido y no se realiza nada más.

Por último, como los nodos *TS_1* y *TS_4* leen la misma tabla entonces se ejecuta el paso 1, y se vuelve a iniciar dos recorridos: uno con *TS_1* y otro con *TS_4*. En ambos recorridos se van comparando los nodos que se visitan y se descubre que los operadores *RS* de los nodos 3 y 6 no son iguales debido a que no configuran las mismas columnas como valor del par intermedio $< clave, valor >$. Entonces se detienen ambos recorridos y se **manda a llamar al *Processor* con los nodos 2 y 5**.

El *Dispatcher* descubre las secuencias de operadores redundantes en las ramas de un *DAG* y manda a llamar a la entidad *Processor* con los últimos nodos hasta donde son iguales los operadores en las ramas para eliminar los operadores redundantes.

5.3.3. Processor

En nuestra optimización, la entidad *Processor* elimina los operadores redundantes que se encontraron en el *Dispatcher*. Esta entidad recibe del *Dispatcher* los dos últimos nodos en los que dos ramas de un *DAG* tienen operadores redundantes.

Para eliminar los operadores redundantes entre dos ramas de un *DAG* sólo es necesario eliminar en el *DAG* las referencias al último nodo de la secuencia de operadores redundantes y los operadores *TSs* de la rama redundante.

Por comodidad nombremos los dos nodos que se reciben del *Dispatcher* como: *N_1* y *N_2*. A continuación se describe con más detalle lo que se realiza en la entidad *Processor* :

1. Para eliminar las referencias al último nodo de la secuencia de operadores redudantes se debe de hacer lo siguiente: Agregar como nodo hijo (nodo superior) del nodo *N_1* el nodo hijo del nodo *N_2*, y al nodo hijo del nodo *N_2* cambiar la referencia del nodo padre (nodo inferior) al nodo *N_1*.

   En nuestro ejemplo de la consulta 3 de Chatziantoniou, la entidad *Processor* se manda a llamar dos veces.

   En la primera llamada se recibieron los nodos 8 y 21 del *DAG* (ver figura 5.5). Entonces, al nodo 8 se le debe de agregar como nodo hijo (nodo superior) el nodo 22 que es el nodo hijo del nodo 21. Posteriormente, el nodo 22 debe de cambiar de referencia del nodo padre (nodo inferior) que ahora será el nodo 8.

   En la segunda llamada se recibieron los nodos 2 y 5. Entonces, al nodo 2 se le debe de agregar como nodo hijo el nodo 6, y el nodo 6 debe de cambiar de referencia del nodo padre al nodo 2.

2. Por último se deben de eliminar los operadores *TS* de la rama redundante del *DAG*.

   En nuestro ejemplo, en la primera llamada del *Processor* se deben de eliminar los operadores *TS* de los nodos 14 y 17. De este modo, los operadores redundantes del nodo 14 al 21 se han eliminado por completo, y el *DAG* que se forma se observa en la figura 5.6.

   En la segunda llamada del *Proccesor* se debe de eliminar el operador *TS* del nodo 4. De este modo, los operadores redundantes del nodo 4 al 5 se han eliminado por completo, y el *DAG* que se forma se observa en la figura 5.7.

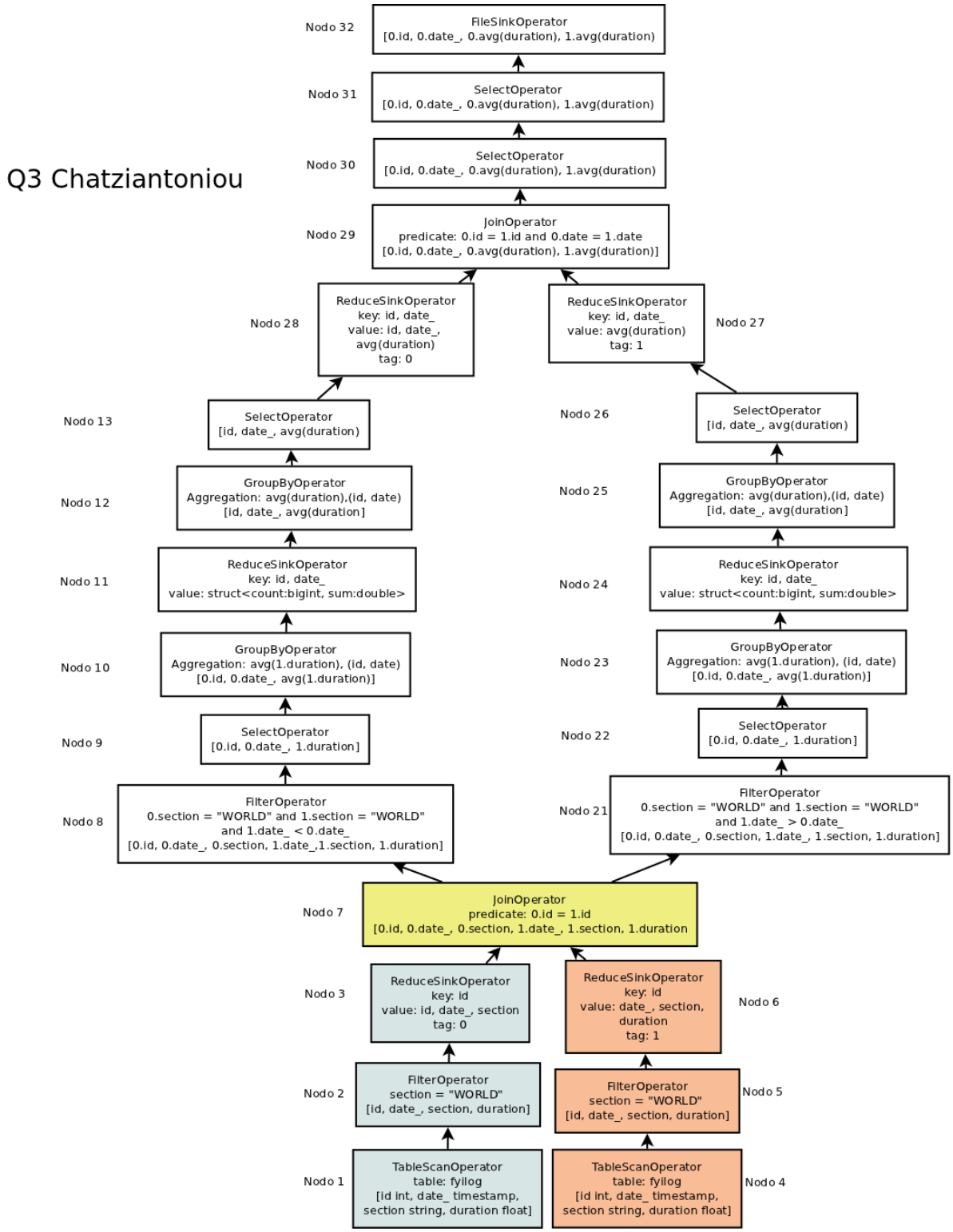


Figura 5.6: *DAG* de la consulta 3 de Chatziantoniou después de eliminar la primera secuencia de operadores redundantes.

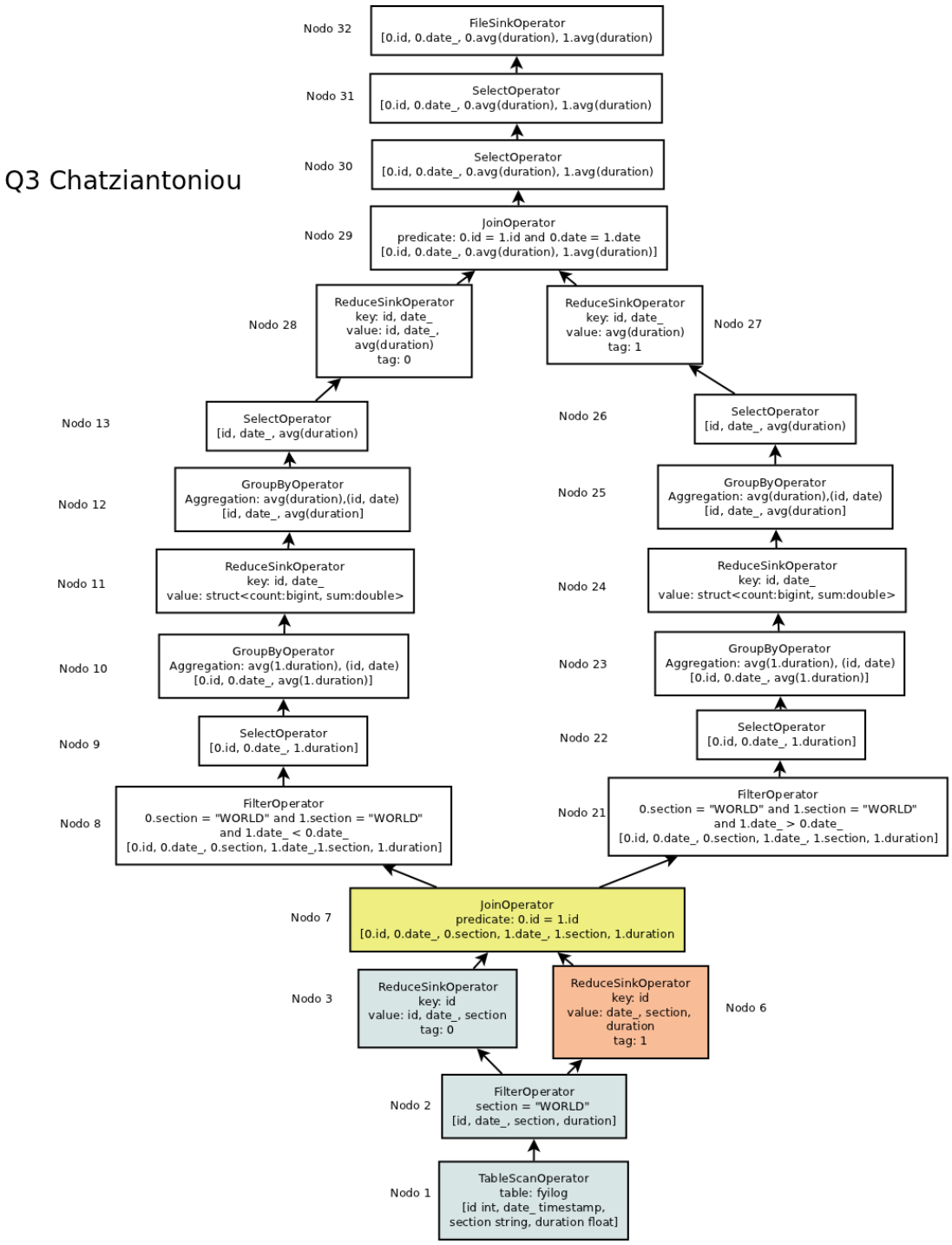


Figura 5.7: *DAG* de la consulta 3 de Chatziantoniou después de eliminar la segunda secuencia de operadores redundantes.

En resumen, elimina secuencias de operadores redundantes de un *DAG* lo que resulta en la eliminación de *trabajos mapreduce*. Por ejemplo para la consulta 3 de Chatziantoniou el *DAG optimizado* que se forma por la optimizaciones actuales de *Hive* se observa en la figura 5.8 y el *DAG optimizado* que se forma después de aplicar esta optimización se observa en la figura 5.9. Obsérvese que con esta optimización se ha eliminado el *trabajo mapreduce* 2, se ha reducido la lectura de la tabla fyilog de cuatro lecturas a una sola lectura, además en vez de realizar dos veces las mismas operaciones en las *tareas map 1 y 2*, ahora sólo se realizan una vez. También la operación *Join* de los trabajos mapreduce 1 y 2 sólo se realiza una vez, con esto se reduce la cantidad de datos que se envían por la red. También obsérvese que en un *trabajo mapreduce* se puede realizar más de una ramificación en una *tarea reduce*, cómo se observa en el *trabajo mapreduce* de la figura 5.2.

La consulta 3 de Chatziantoniou es un caso de eliminación de operadores redundantes entre ramas compuestas. Las ramas puede ser aún más simples con un sólo *Join*, o más complejas con múltiples *join*. Estos casos se analizan con detalle en el apéndice A.

5.4. Trabajo relacionado

YSMART es un trabajo relacionado que realiza optimizaciones parecidas a las que realizamos [31]. A diferencia de nuestras optimizaciones, las optimizaciones de *YSMART* se aplicaron a un framework que ellos desarrollaron para convertir sentencias *SQL* a *trabajos mapreduce*, y actualmente están integrando dichas optimizaciones a *Hive*. Las optimizaciones de *YSMART* están diseñadas para descubrir 3 tipos distintos de correlaciones entre las subconsultas de una consulta.

Una aproximación para explicar el trabajo de *YSMART* en un *DAG* y plan físico de *Hive* se observa en las figuras 5.10 y 5.11. Recuerde que el *DAG* es el árbol sin los cuadros punteados que representan los *trabajos mapreduce*. Se presenta el plan físico para explicar de mejor manera el efecto de la optimización. Sin embargo, el trabajo

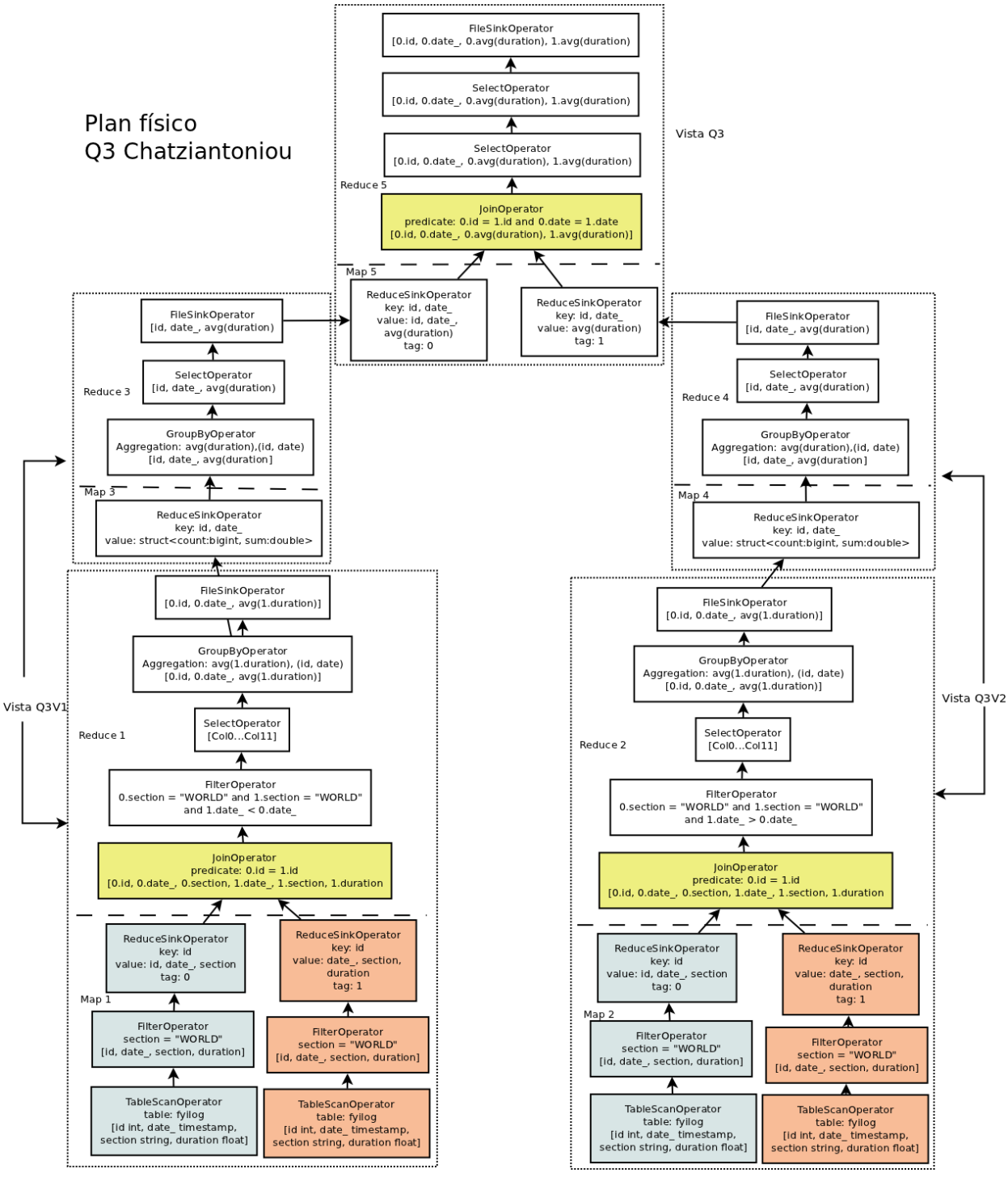


Figura 5.8: *Plan físico* de la consulta 3 de Chatziantoniou con el *DAG* optimizado que se forma con las optimizaciones actuales de *Hive*.

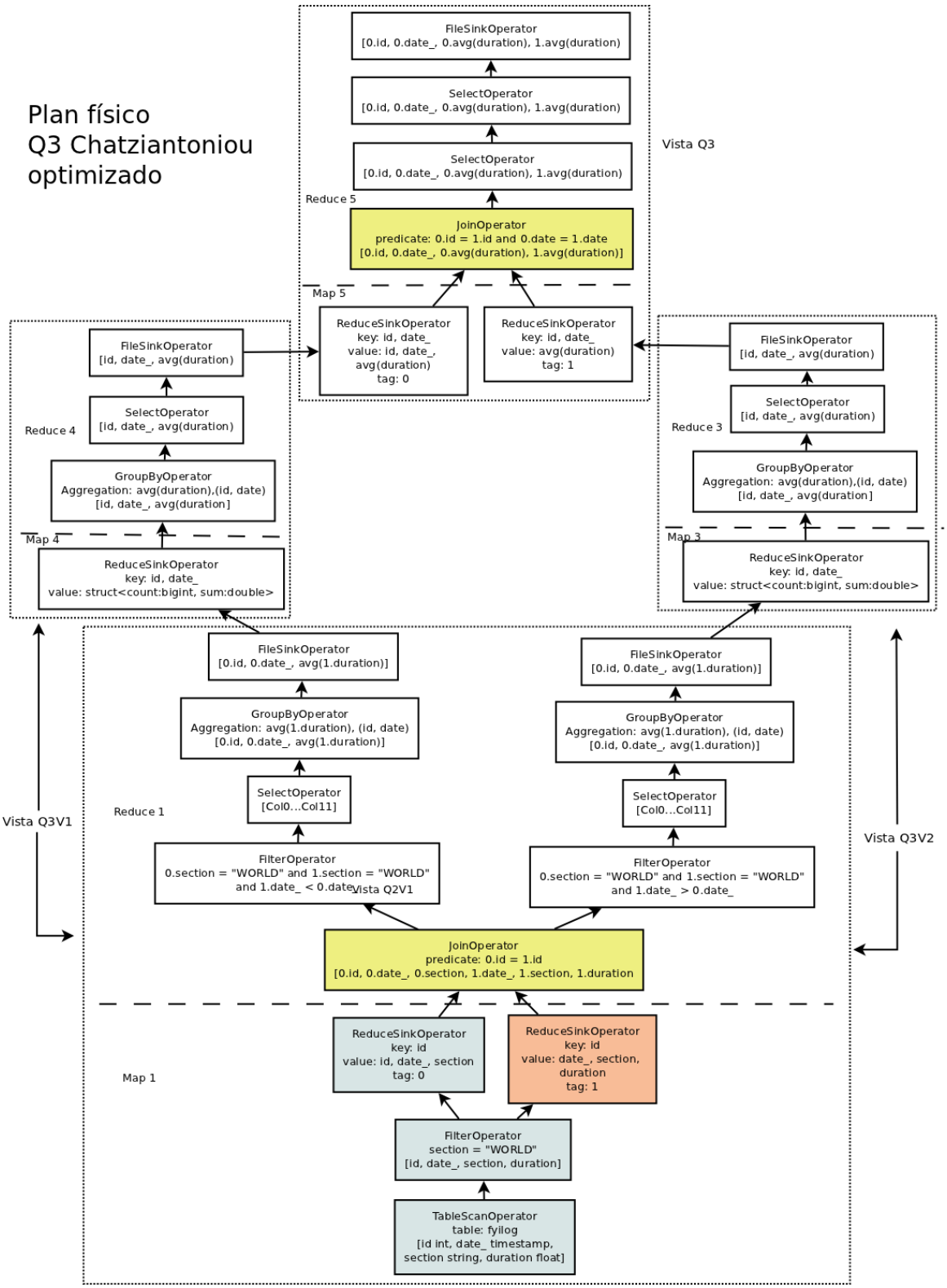


Figura 5.9: *Plan físico* de la consulta 3 de Chatziantoniou con el *DAG* optimizado que se forma después de aplicar nuestra optimización de eliminar operadores redundantes entre ramas.

de *YSMART* trabaja con un árbol de operadores que en *Hive* corresponde a un *DAG*.

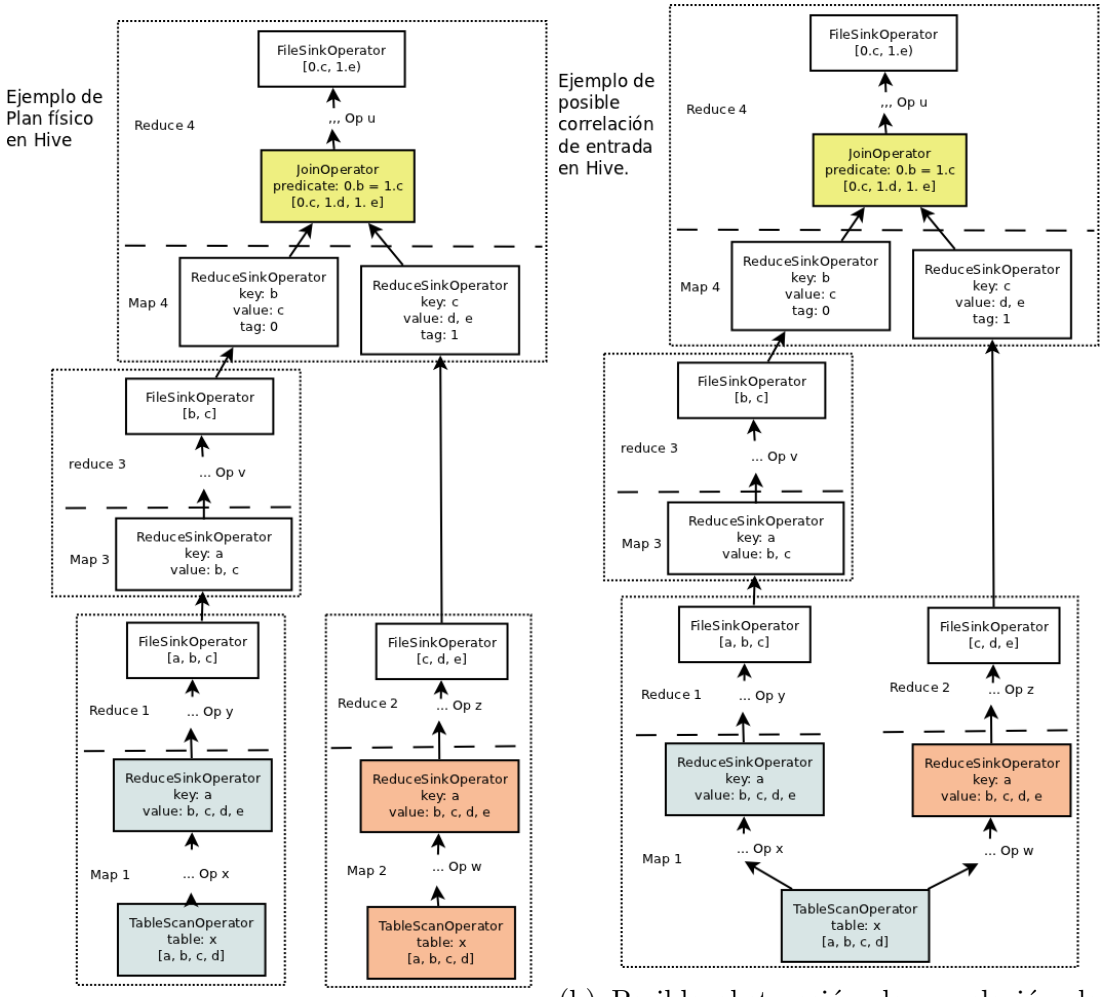
Las correlaciones que se buscan en el árbol de la consulta son:

1. Correlación de entrada (ce): Dos conjuntos de nodos tienen correlación de entrada (ce) si su conjunto de tablas de entrada no son disjuntas. En *Hive*, sería que los operadores *TS* lean las mismas tablas. Si dos conjuntos de nodos tienen correlación de entrada, entonces los dos *trabajos mapreduce* correspondientes pueden compartir la lectura de la misma tabla en las *tareas map*.

   Por ejemplo, en la figura 5.10a se observa el plan físico de una consulta *HiveQL*. Obsérvese que el conjunto de nodos de la izquierda inicia con un nodo *TS* que lee una tabla x y forma el *trabajo mapreduce 1*, y el conjunto de nodos de la derecha inicia con otro nodo *TS* que también lee la tabla x y forma el *trabajo mapreduce 2*. Cómo ambos conjuntos inician con un operador *TS* que lee la misma tabla, entonces ambos conjuntos tienen correlación de entrada y los *trabajos mapreduce 1 y 2* pueden compartir la lectura de la misma tabla en las *tareas map* como se observa en la figura 5.10b.

2. Correlación de transición (ct): Dos conjuntos de nodos tienen correlación de transición (ct) si ellos tienen correlación de entrada, y los nodos que envian los pares intermedios $< clave, valor >$ (*ReduceSinkOperator* en *Hive*) utilizan la misma clave. Si dos nodos tienen correlación de transición (en Hive, que dos operadores *RS* configuren los pares intermedios $< clave, valor >$ de las tareas map con la misma clave), entonces los dos *trabajos mapreduce* se pueden unir en un sólo *trabajo mapreduce* acomodando el orden de los nodos.

   Por ejemplo, en la figura 5.10b se observa que los operadores *RS* del *trabajo mapreduce 1* configuran los pares $< clave, valor >$ con la misma clave: *a*. Por lo tanto, ambos conjuntos de nodos (de la izquierda y la derecha) tienen correlación de transición y se unen en una sola secuencia de nodos acomodando el orden de los nodos *Op x* con *Op w* en las tareas map, y *Op y* con *Op z* en las *tareas reduce* cómo se observa en la figura 5.11a.



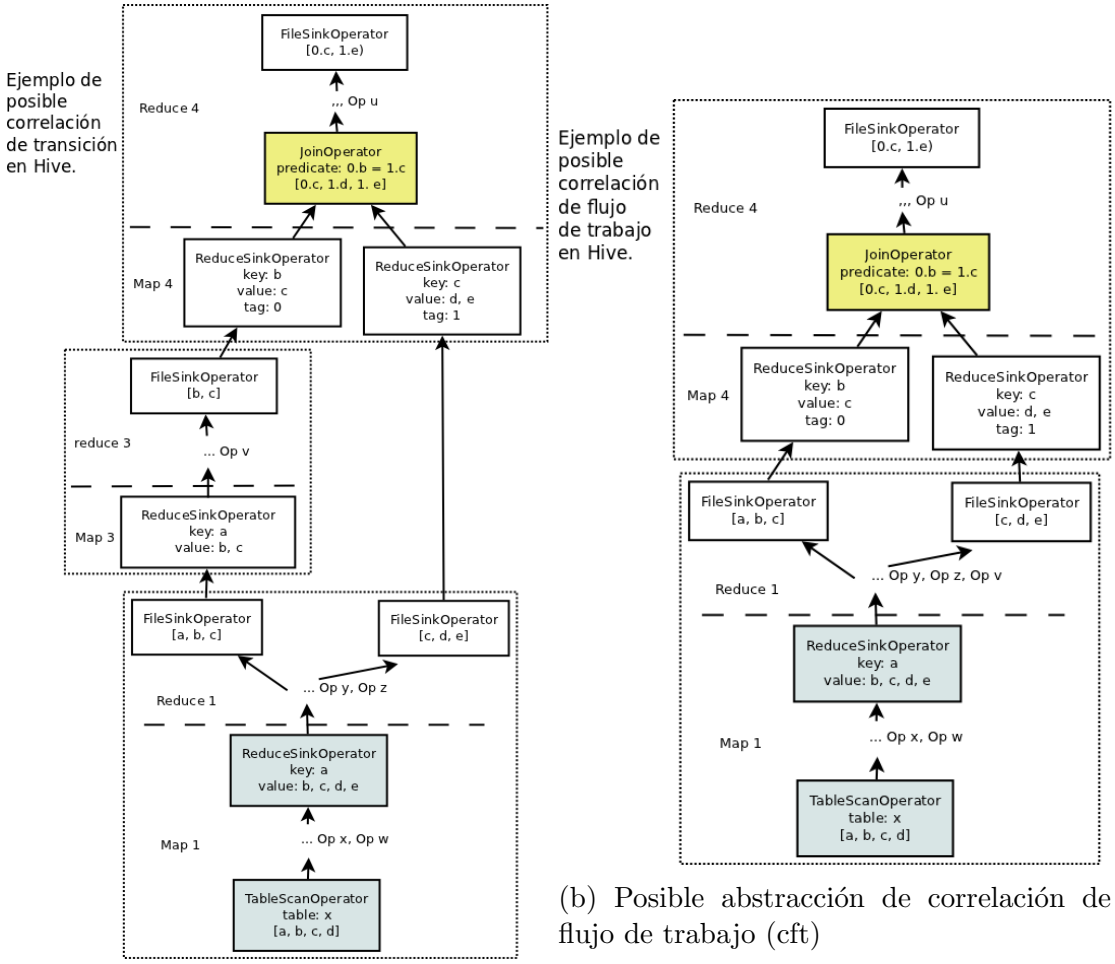
(a) Ejemplo de plan físico

(b) Posible abstracción de correlación de entrada (ce)

Figura 5.10: Posible abstración de la correlación de entrada de *YSMART* en un plan físico de *Hive*

3. Correlación de flujo de trabajo (cft): Un nodo tiene correlación de flujo de trabajo con uno de sus nodos hijos (nodos de arriba) si ambos utilizan la misma clave en el par intermedio $< clave, valor >$. Si un nodo tiene correlación de flujo de trabajo con uno de sus nodos hijos, entonces las operaciones que se encuentran en las *tareas reduce* del *trabajo mapreduce* donde se encuentra el nodo hijo (superior) se pueden ejecutar en las *tareas reduce* del *trabajo mapreduce* donde se encuentra el nodo inferior.

   Por ejemplo, en la figura 5.11a se observa que el nodo *RS* del *trabajo mapreduce 3* tiene como hijo el nodo *RS* del *trabajo mapreduce 1*. Ambos nodos tienen



(a) Posible abstracción de correlación de transición (ct)

(b) Posible abstracción de correlación de flujo de trabajo (cft)

Figura 5.11: Posible abstración de las correlaciones de transición y flujo de trabajo de *YSMART* en un plan físico de *Hive*

correlación de flujo de trabajo debido a que ambos configuran los pares intermedios $< clave, valor >$ con la clave *a*. Por lo tanto, las operaciones *v* del *trabajo mapreduce 3* se pueden ejecutar en las *tareas reduce* del *trabajo mapreduce 1* como se observa en la figura 5.11b.

En resumen, la correlación de entrada hace que cuando se lean las mismas tablas, las *tareas map se unan* y se haga una. La correlación de transición elimina *trabajos mapreduce* de manera horizontal, y la correlación de flujo de trabajo elimina *trabajos mapreduce* de manera vertical. Creemos que la correlación de entrada es un subconjunto de la correlación de *transición* por lo que se puede considerar como una

misma correlación.

Nuestra optimización de eliminar operadores redundantes entre ramas simples o complejas de alguna manera aplica las correlaciones de entrada y transición de *YSMART*, mientras que la optimización de eliminar *trabajos mapreduce* asociados a operadores de agregación aplica la correlación de flujo de trabajo de *YSMART*. El trabajo de *YSMART* ofrece una panorámica más general para eliminar *trabajos mapreduce* similares debido a que ellos realizan una *transformación* de un *DAG* eliminando los operadores redundantes y reordenando la ejecución de los operadores, mientras que nosotros sólo eliminamos los operadores redundantes. Además la correlación de flujo de trabajo de *YSMART* se aplica para cualquier condición, mientras que nosotros sólo detectamos el caso en los *trabajos mapreduce* asociados a operadores de agregación.

5.5. Resumen

En este capítulo se describio como se realizan nuestras optimizaciones al *DAG* optimizado resultante de las optimizaciones actuales de *Hive*. Las optimizaciones que se explicaron fueron: la optimización de eliminar secuencias de operadores redundantes, y la optimización de eliminar *trabajos mapreduce* innecesarios asociados a funciones de agregación y agrupación. Estas optimizaciones buscan eliminar *trabajos mapreduce* que son innecesarios con el objetivo de: eliminar operaciones de entradas/salida al eliminar la lecturas redundantes de *tablas*; eliminar los registros repetidos que se envian por una red al eliminar *tareas map* que emiten los mismos pares intermedios $< clave, valor >$; y eliminar procesamiento de datos repetidos en distintos nodos de un cluster al eliminar operadores en el *DAG* que se encuentran repetidos. *YSMART* es un trabajo relacionado con nuestras optimizaciones que realiza optimizaciones parecidas a nuestras optimizaciones pero en otro *framework* y que actualmente, las están tratando de integrar a *Hive*.

Capítulo 6

Evaluación experimental y resultados

Para evaluar el desempeño de nuestras optimizaciones en *Hive* utilizamos varias consultas *OLAP*, algunas propuestas por Chatziantoniou [29] y otras del estudio de mercado llamado TPC-H [30]. El resultado fue que *Hive* versión 0.8 con nuestras optimizaciones tuvó mejor desempeño que el *Hive* versión 0.8 sin las mismas. En algunas consultas se redujo el tiempo de ejecución hasta en un 30 %. Este capítulo describe la plataforma experimental: Hardware, software y aplicaciones; la metodología experimental y los resultados de los experimentos realizados.

6.1. Plataforma experimental

6.1.1. Hardware

El hardware utilizado fue el clúster del Departamento de Computación del Cinvestav-IPN, Zacatenco de la Ciudad de México. Este clúster está constituido por 32 nodos, de los cuáles solo se utilizaron: 8 nodos, 16 nodos y 20 nodos. Cada nodo del clúster tiene las siguiente características: un procesador Intel Core i7 a 2.67 Ghz con 4 núcleos con tecnología *HyperThreading* con 8MB de caché; 4GB de memoria RAM; 500GB de

disco duro. Los nodos del clúster se conectan a través de una conexión *Switch Gigabit Ethernet*.

6.1.2. Software

Las herramientas de software que se utilizaron son:

- Sistema operativo *Linux* de 64 bits en cada uno de los nodos del clúster. La distribución de linux que se utilizó fue *Fedora 15*.
- Máquina virtual de Java versión 1.6. Java se utiliza debido a que *Hadoop* y *Hive* se ejecutan en una máquina virtual de *Java*.
- *Hadoop* versión 0.21 que era la versión estable disponible cuando se inició la investigación. Para configurar el ambiente de *Hadoop* en el clúster se configuraron los archivos de configuración: *core-site.xml, mapred-site.xml, hdfs-site.xml*. Aunque los *programas mapreduce* en *Hadoop* se pueden escribir en varios lenguajes, nosotros utilizamos *Java* para escribir nuestros *programas mapreduce* de prueba.
- *Hive* versión 0.8 que era la versión estable disponible cuando se inició la investigación. Para configurar *Hive* se utilizó el archivo *hive-site.xml*.
- Apache ant, es una herramienta de software que permite construir software de manera automática. Es muy similar al *make* en un ambiente *linux* pero está implementado para utilizarse con el lenguaje *Java*. Esta herramienta se utilizó para volver a construir *Hive* una vez que nuestras optimizaciones se agregarón. El comando que se utiliza para construir *Hive* en modo consola es: *ant clean package*.
- Eclipse versión 3.7 (Indigo). Eclipse es ambiente de desarrollo integrado (del inglés *Integrated Development Enviroment, IDE*) que facilita el desarrollo de aplicaciones escritas en varios lenguajes como: *Java*, *C*, *Python*, entre otros. Se

facilita el desarrollo debido a que eclipse integra un compilador, un *debugger*, entre otras herramientas.

Eclipse se utilizó para implementar *programas mapreduce* de prueba escritos en *Java*, estos *programas mapreduce* se ejecutaban desde eclipse en el ambiente *Hadoop* de manera local.

Eclipse también facilitó la implementación de nuestras optimizaciones en *Hive*. Para esto, se importó todo el proyecto de *Hive* (Clases, interfaces, archivos de prueba, etcétera) a eclipse. En eclipse se implementó nuestras optimizaciones y con ayuda del *debugger* que trae incluido *Hive* para eclipse se logró depurar nuestras optimizaciones con algunas consultas prueba de Chatziantoniou y TPC-H. De esta manera eclipse nos facilitó el desarrollo ya que identificamos nuestros erróres con mayor facilidad. Además, desde eclipse realizamos pruebas del comportamiento de *Hive* con nuestras optimizaciones en un entorno local. También agregamos a eclipse la herramienta *ant* para construir el proyecto de *Hive* desde eclipse. De este modo, para evaluar *Hive* con nuestras optimizaciones en un entorno distribuido solo copiamos la carpeta de *Hive* a nuestro clúster.

6.1.3. Aplicaciones

Para evaluar el desempeño de nuestras optimizaciones en *Hive* utilizamos algunas consultas *OLAP* propuestas por Chatziantoniou [29] y algunas consultas *OLAP* propuestas en el estudio de mercado llamado TPC-H [30]. A continuación describimos ambos conjuntos de consultas.

Las consultas de Chatziantoniou son consultas *OLAP* complejas que se utilizaron en un estudio para evaluar una técnica para identificar consultas group-by y optimizarlas. Se compone de cuatro consultas de las cuales solo utilizamos tres. El caso de estudio es el siguiente:

Caso de estudio: Suponga que una empresa periódistica en línea en la web *"For Your Information (FYI)"* tiene una tabla donde mantiene la información de las

secciones visitadas por sus clientes. La tabla es la siguiente:

```
FYILOG(id int, date string, section string, duration double)
```

Donde: *id* es el id del cliente, *date* se refiere a la fecha de conexión, *section* es la sección que revisa el cliente (deportes, política, negocios, etcétera), y *duration* es el tiempo en segundos que gasta el cliente en la sección durante una conexión. Entonces, la empresa *FYI* desea conocer:

- Consulta 1 (q1c): Por cada cliente, ¿Cuántas veces accedió a la sección *"WORLD"*? y ¿Cuántas veces el tiempo que estuvo en la sección *"WORLD"* fue mayor que el promedio que permaneció en todas las secciones?. Las consultas *HiveQL* para esta consulta se observan en el código 6.1.

- Consulta 2 (q2c): Por cada cliente, encontrar la máxima duración promedio por cada sección con su respectivo nombre. Las consultas *HiveQL* para esta consulta se observan en el código 6.2.

- Consulta 3 (q3c): Por cada cliente que accedió a la sección *"WORLD"* en una conexión *t*, encontrar el promedio de tiempo gastado en la sección *"WORLD"* en las conexiones anteriores a *t*, y el promedio de tiempo gastado en la sección *"WORLD"* en las conexiones posteriores a *t*. Las consultas *HiveQL* para esta consulta se observan en el código 6.3.

```
1  CREATE VIEW Q1V1 as
2            SELECT id, avg(duration) as avg_d
3            FROM FYILOG
4            GROUP BY id;
5
6  CREATE VIEW Q1V2 as
7            SELECT id, COUNT(*) as cnt
8            FROM FYILOG
9            WHERE SECTION="WORLD"
10           GROUP BY id;
11
12 CREATE VIEW Q1V3 AS
13           SELECT c.id, COUNT(*) AS cnt
14           FROM fyilog c JOIN q1v1 q ON (c.id = q.id)
15           WHERE c.section="WORLD" and c.duration > q.avg_d
16           GROUP BY c.id;
17
```

```
18 CREATE VIEW Q1 AS
19         SELECT q1v2.id, q1v2.cnt as cntq1v2, q1v3.cnt as cntq1v3
20         FROM q1v2 join q1v3 ON (q1v2.id = q1v3.id);
```

Código 6.1: Sentencias *HiveQL* para resolver la consulta 1 de Chatziantoniou en Hive que realiza lo siguiente: Por cada cliente ¿Cuántas veces accedió a la sección *"WORLD"*? y ¿Cuántas veces el tiempo que estuvo en la sección *"WORLD"* fue mayor que el promedio que permaneció en todas las secciones?.

```
1  CREATE VIEW Q2V1 AS
2          SELECT id, section, avg(duration) AS avg_d
3          FROM FYILOG
4          GROUP BY id, section;
5
6  CREATE VIEW Q2V2 AS
7          SELECT id, max(avg_d) AS max_s
8          FROM Q2V1
9          GROUP BY id;
10
11 CREATE VIEW Q2 AS
12         SELECT q2v1.id, q2v1.section, q2v2.max_s
13         FROM q2v1 JOIN q2v2 ON (q2v1.id = q2v2.id)
14         WHERE q2v2.max_s = q2v1.avg_d;
```

Código 6.2: Sentencias *HiveQL* para resolver la consulta 2 de Chatziantoniou en Hive que realiza lo siguiente: Por cada cliente encontrar la máxima duración promedio por cada sección con su respectivo nombre.

```
1  CREATE VIEW Q3V1 AS
2          SELECT c1.id, c1.date_, avg(c2.duration) as avg_d
3          FROM fyilog c1 JOIN fyilog c2 ON (c1.id = c2.id)
4          WHERE c1.section="WORLD" and c2.section="WORLD" AND c2.date_ <
               c1.date_
5          GROUP BY c1.id, c1.date_;
6
7  CREATE VIEW Q3V2 AS
8          SELECT c1.id, c1.date_, avg(c2.duration) as avg_d
9          FROM fyilog c1 JOIN fyilog c2 ON (c1.id =c2.id)
10         WHERE c1.section="WORLD" and c2.section="WORLD" AND c2.date_ >
               c1.date_
11         GROUP BY c1.id, c1.date_;
12
13 CREATE VIEW Q3 AS
14         SELECT c1.id, c1.date_, c1.avg_d as avg_q3v1, c2.avg_d as
               avg_q3v2
15                 FROM q3v1 c1 JOIN q3v2 c2 ON (c1.id = c2.id and c1.date_
                       = c2.date_);
```

Código 6.3: Sentencias *HiveQL* para resolver la consulta 3 de Chatziantoniou en Hive que realiza lo siguiente: Por cada cliente que accedió a la sección *"WORLD"* en una conexión t encontrar el promedio de tiempo gastado en la sección *"WORLD"* en las conexiones anteriores a t y el promedio de tiempo gastado en la sección *"WORLD"* en las conexiones posteriores a t.

Por otra parte, *TPC-H* es un estudio de mercado de soporte a decisiones.

Está constituido por un conjunto de consultas OLAP (22 consultas) que evalúa el rendimiento de sistemas de soporte de decisiones a través de condiciones controladas. El estudio está avalado por compañías como: AMD, CISCO, Oracle, IBM, Microsoft, Dell, HP, entre otras. El esquema relacional de las tablas del estudio *TPC-H* se observa en la figura 6.1.

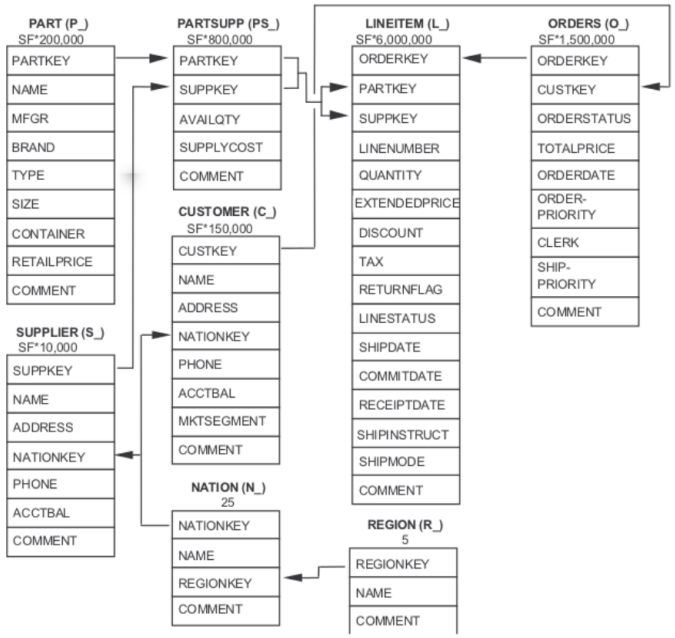


Figura 6.1: Esquema relacional del estudio TPC-H

Las consultas *TPC-H* que se utilizaron son:

- Consulta 2 (q2t): Por cada proveedor de una región se obtiene el costo mínimo de cada parte (artículo) de un determinado tipo y tamaño. Con esta información por cada proveedor se lista el saldo total, el nombre, dirección, número de telefóno y nación del proveedor. También se enlista el número de pieza y el fabricante. Las consultas *HiveQL* para esta consulta se observan en el código 6.4.

- Consulta 3 (q3t): Obtiene los pedidos junto con su ingreso que aun no se han entregado en una fecha determinada. Los pedidos se ordenan de manera decreciente de acuerdo al ingreso. Las consultas *HiveQL* para esta consulta se observan en el código 6.5.

- Consulta 11 (q11t): Por cada nación, obtiene por cada proveedor el número y el valor de las partes (artículos) en orden descendente de acuerdo a su valor. Las consultas *HiveQL* para esta consulta se observan en el código 6.6.

- Consulta 13 (q13t): Clasifica a los clientes de acuerdo al número de pedidos que ha realizado. Consulta ¿cuántos clientes no tienen órdenes?, ¿Cuántos tienen 1, 2, 3, etcétera?. Se realiza una comprobación para asegurarse de que las órdenes de contado no se tomen en cuenta. Las consultas *HiveQL* para esta consulta se observan en el código 6.7.

```
1  create table q2_minimum_cost_supplier (s_acctbal double, s_name string,
       n_name string, p_partkey int, p_mfgr string, s_address string,
       s_phone string, s_comment string);
2
3  create view q2_minimum_cost_supplier_tmp1 as  select  s.s_acctbal, s.
       s_name, n.n_name,
4    p.p_partkey, ps.ps_supplycost, p.p_mfgr, s.s_address, s.s_phone, s.
         s_comment
5      from   nation n join region r
6        on    n.n_regionkey = r.r_regionkey and r.r_name = 'EUROPE'
7        join supplier s
8        on  s.s_nationkey = n.n_nationkey
9        join partsupp ps
10       on  s.s_suppkey = ps.ps_suppkey
11       join part p
12       on  p.p_partkey = ps.ps_partkey and p.p_size = 15 and p.p_type
             like '%BRASS';
13
14 create view q2_minimum_cost_supplier_tmp2 as select p_partkey, min(
       ps_supplycost) as
15     ps_min_supplycost
16     from  q2_minimum_cost_supplier_tmp1
17     group by p_partkey;
18
19 insert overwrite table q2_minimum_cost_supplier
20     Select  t1.s_acctbal, t1.s_name, t1.n_name, t1.p_partkey, t1.p_mfgr
           , t1.s_address,
21     t1.s_phone, t1.s_comment
22     from   q2_minimum_cost_supplier_tmp1 t1
23       join q2_minimum_cost_supplier_tmp2 t2
24       on t1.p_partkey = t2.p_partkey and t1.ps_supplycost=t2.
             ps_min_supplycost
25     order by s_acctbal desc, n_name, s_name, p_partkey
26     limit 100;
```

Código 6.4: Sentencias *HiveQL* para resolver la consulta 2 del estudio TPC-H en Hive que realiza lo siguiente: Por cada proveedor de una región se obtiene el costo mínimo de cada parte (artículo) de un determinado tipo y tamaño. Con esta información por cada proveedor se lista el saldo total, el nombre, dirección, número de telefóno y nación del proveedor. También se enlista el número de pieza y el fabricante.

```
1  create table q3_shipping_priority (l_orderkey int, revenue double,
       o_orderdate string, o_shippriority int);
2
3  Insert overwrite table q3_shipping_priority
4  select
5    l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue,
         o_orderdate, o_shippriority
6  from
7    customer c join orders o
8      on c.c_mktsegment = 'BUILDING' and c.c_custkey = o.o_custkey
9    join lineitem l
10     on l.l_orderkey = o.o_orderkey
11 where
12   o_orderdate < '1995-03-15' and l_shipdate > '1995-03-15'
13 group by l_orderkey, o_orderdate, o_shippriority
14 order by revenue desc, o_orderdate
15 limit 10;
```

Código 6.5: Sentencias *HiveQL* para resolver la consulta 3 del estudio TPC-H en Hive que realiza lo siguiente: Obtiene los pedidos junto con su ingreso que aun no se han entregado en una fecha determinada. Los pedidos se ordenan de manera decreciente de acuerdo al ingreso.

```
1  create table q11_important_stock(ps_partkey INT, value DOUBLE);
2
3  CREATE VIEW q11_part_tmp AS select  ps_partkey, sum(ps_supplycost *
       ps_availqty) as part_value
4      from  nation n join supplier s
5        on   s.s_nationkey = n.n_nationkey and n.n_name = 'GERMANY'
6        join partsupp ps
7        on  ps.ps_suppkey = s.s_suppkey
8    group by ps_partkey;
9
10 CREATE VIEW q11_sum_tmp AS select  sum(part_value) as total_value
11   from q11_part_tmp;
12
13 insert overwrite table q11_important_stock
14 select   ps_partkey, part_value as value
15   from (
16       select ps_partkey, part_value, total_value
17          from q11_part_tmp join q11_sum_tmp
18   ) a
19   where part_value > total_value * 0.0001
20   order by value desc;
```

Código 6.6: Sentencias *HiveQL* para resolver la consulta 11 del estudio TPC-H en Hive que realiza lo siguiente: Por cada nación obtiene por cada proveedor el número y el valor de las partes (artículos) en orden descendente de acuerdo a su valor.

```
1  create table q13_customer_distribution (c_count int, custdist int);
2
3  insert overwrite table q13_customer_distribution
4  select
5    c_count, count(1) as custdist
6  from
7    (select
```

```
8       c_custkey, count(o_orderkey) as c_count
9     from
10      customer c left outer join orders o
11      on
12        c.c_custkey = o.o_custkey and not o.o_comment like '%special%
              requests%'
13    group by c_custkey
14    ) c_orders
15  group by c_count
16  order by custdist desc, c_count desc;
```

Código 6.7: Sentencias *HiveQL* para resolver la consulta 13 del estudio TPC-H en Hive que realiza lo siguiente: Clasifica a los clientes de acuerdo al número de pedidos que ha realizado. Consulta ¿cuántos clientes no tienen órdenes? ¿Cuántos tienen 1, 2, 3, .., n órdenes?. Se realiza una comprobación para asegurarse de que las órdenes de contado no se tomen en cuenta.

6.2. Organización de experimentos y resultados

Para evaluar el rendimiento de nuestras optimizaciones organizamos tres grupos de experimentos. En cada grupo de experimentos se ejecutaron las consultas *OLAP* de Chatziantoniou y TPC-H de la sección 6.1 en *Hive* versión 0.8 y en *Hive* versión 0.8 con nuestras optimizaciones. En los tres grupos de experimentos se maneja de manera constante: 1) el tamaño de cada split de la entrada de datos, cuyo tamaño es de 64 MB; 2) la réplica de cada split en 3 nodos; 3) y el tamaño del *heapsize*[1] de 1 GB.

Las condiciones que se variaron en los experimentos son: 1) el tamaño de los datos de entrada para el primer grupo de experimentos, 2) el número de nodos del cluster para el segundo grupo de experimentos, 3) y el número de *tareas reduce* por *trabajo mapreduce* para el tercer grupo de experimentos. Otros aspectos configurables de *Hadoop y Hive* (por ejemplo: cantidad máxima de *tareas map o reduce* por nodo, los nodos que se utilizan para el sistema *HDFS*, entre otras) utilizan su configuración por omisión.

A partir de ahora, el *Hive* con nuestras optimizaciones se nombrará como *HiveC* y *Hive* versión 0.8 simplemente se nombrará *Hive*. A continuación se describe el

[1]Heapsize, es un espacio de memoria que se reserva para ejecutar el código de un programa y los datos asociados al programa.

objetivo, las condiciones, la metodología, las hipótesis y los resultados de cada grupo de experimentos.

6.2.1. Experimento 1: Hive vs HiveC

Objetivo

El objetivo de este grupo experimentos es comparar el desempeño de *HiveC* con *Hive*. El desempeño se evalúa entorno al tiempo de ejecución de las consultas de Chatziantoniou y TPC-H en ambas versiones de *Hive*.

Condiciones Además de las configuraciones constantes descritas en la sección 6.2. En la tabla 6.1 se observan las configuraciones específicas para este grupo de experimentos.

Tamaño del archivo de entrada	4GB, 8GB, 16GB
Número de nodos	8 nodos
Número de *tareas reduce* por *trabajo mapreduce*	Por omisión[1].

Tabla 6.1: Configuraciones específicas del primer grupo de experimentos.

El tamaño de los datos de entrada varía en 4GB, 8GB y 16GB. Se utiliza de manera constante 8 nodos del clúster. De los 8 nodos del clúster, un nodo se utiliza como servidor de *MapReduce* y *HDFS*, es decir, ejecuta los procesos *JobTracker*, *NameNode* y *Secondary NameNode*. Los otros 7 nodos del clúster ejecutan los procesos clientes de *MapReduce* y *HDFS*, es decir, los procesos *TaskTrackers* y *Datanodes*. Esta configuración se lleva acabo en los archivos *masters* y *slaves*. En el archivo *masters* se enlista el nodo en el cual se ejecutan los procesos servidores de *Hadoop* y *HDFS*, en el archivo *slaves* se enlistan los 7 nodos donde se ejecutan los procesos clientes de *Hadoop* y *HDFS*. El *framework* de *Hive* solo se coloca en el nodo servidor. El número de *tareas reduce* que se utilizan por cada *trabajo mapreduce* es el que generá *Hadoop* por omision, excepto para la consulta 3 por razones presentadas más adelante. Las demás configuraciones son las que trae por omisión *Hadoop y Hive*.

[1]Excepto para la consulta 3 de Chatziantoniou por razones presentadas más adelante

Tabla	4 GB	8 GB	16 GB	% de los datos
lineitem	2.9 GB	5.8 GB	11 GB	70.25 %
orders	664 MB	1.4 GB	2.5 GB	15.15 %
partsupp	458 MB	918 MB	1.7 GB	10 %
customer	94 MB	187 MB	351 MB	2.24 %
part	93 MB	187 MB	350 MB	2.23 %
supplier	5.5 MB	11 MB	21 MB	0.13 %
nation	4 KB	4 KB	4 KB	0.00000055 %
region	4 KB	4 KB	4 KB	0.00000055 %

Tabla 6.2: Distribución de los datos en las 8 tablas del estudio *TPC-H*.

En las consultas de Chatziantoniou los 4GB, 8GB o 16GB se encuentran en la tabla *fyilog*. Los registros en esta tabla se distribuyen de manera uniforme, es decir, por cada usuario se crea el mismo número de registros. Por otro lado, en las consultas del estudio *TPC-H* los 4GB, 8GB o 16GB se distribuyen en las 8 tablas que conforman el estudio. La manera en que se distribuyen los datos se observa en la tabla 6.2. Los datos se crean con un programa que de manera oficial brinda el estudio *TPC-H*. Obsérvese que la tabla *lineitem* es la que mayor cantidad de datos tiene con un 70.25 % de los datos, seguido de la tabla *orders* con un 15.15 % de los datos y así sucesivamente. Notése que el tamaño de las tablas *nation y region* siempre son constantes con 4 KB de datos, cantidad que es muy pequeña con respecto a los 4 GB, 8 GB o 16 GB de datos generados.

Metodología

La metodología utilizada en este grupo de experimentos fue la siguiente:

1. Primero se cargan 4GB de datos al sistema *HDFS* para las consultas de Chatziantoniou y 4GB de datos para las consultas TPC-H.

2. Posteriormente, las consultas de Chatziantoniou y TPC-H se ejecutan con los datos respectivos en *HiveC* y en *Hive*.

3. Se comparan los resultados obtenidos.

4. Se eliminan los 4GB de datos que se cargarón en el sistema *HDFS* y se cargan

8GB de datos para cada grupo de consultas. Posteriormente se vuelve a repetir a partir del paso 2. Después, se eliminan los 8GB de datos y se cargan 16GB de datos por cada grupo de consultas y se vuelve a repetir a partir del paso 2. Los datos se eliminan debido a que cada tabla de *Hive* tiene asociado un directorio y todos los archivos que se encuentran en ese directorio son los datos de la tabla.

Hipótesis

Se espera obtener un menor tiempo de respuesta de las consultas *HiveQL* de Chatziantoniou y TPC-H en *HiveC* que en *Hive*. Conforme aumenta el tamaño de los datos entrada las consultas *HiveQL* tardan más en ejecutarse, proporcionalmente al tamaño del archivo de entrada a procesar.

Resultados

Las consultas cuyo nombre termina con "c" son de Chatziantoniou, y con "t" son del estudio de mercado TPC-H. Los resultados de cada consulta se presentan con dos gráficas de barras, la primera corresponde al tiempo de ejecución de la consulta en *Hive* y la segunda al tiempo de ejecución de la consulta en *HiveC*. A las consultas q2c, q3c y q11t se aplicó la optimización de eliminación de secuencias de operadores *Hive* redundantes en un *DAG* que en las gráficas se nombró como: *HiveC-opt_subqueries*. En las demás consultas se aplicó la optimización de eliminación de *trabajos mapreduce*; esta optimización en las gráficas se nombró como: *HiveC-opt_aggregation*. Aunque ambas optimizaciones se pueden aplicar a una misma consulta, en ninguna de las consultas mostradas se aplicarón ambas. Aún así, la figura 6.2 muestra que el tiempo de ejecución de las consultas en *HiveC* es menor que el de *Hive* para todas las consultas.

La optimización *opt_subqueries* lográ mejor desempeño que la optimización *opt_aggregation*. En las consultas *OLAP* en las que se aplicó la optimización *opt_subqueries* se redujo el tiempo de ejecución con respecto a *Hive* entre 21 % y 33 %, y en las consultas *OLAP* donde se aplicó la optimización *opt_aggregation* se redujo el tiempo de ejecución con respecto a *Hive* entre 2 % y 22 %.

La figura 6.3 muestra los tiempos de ejecución que se obtuvierón al ejecutar las

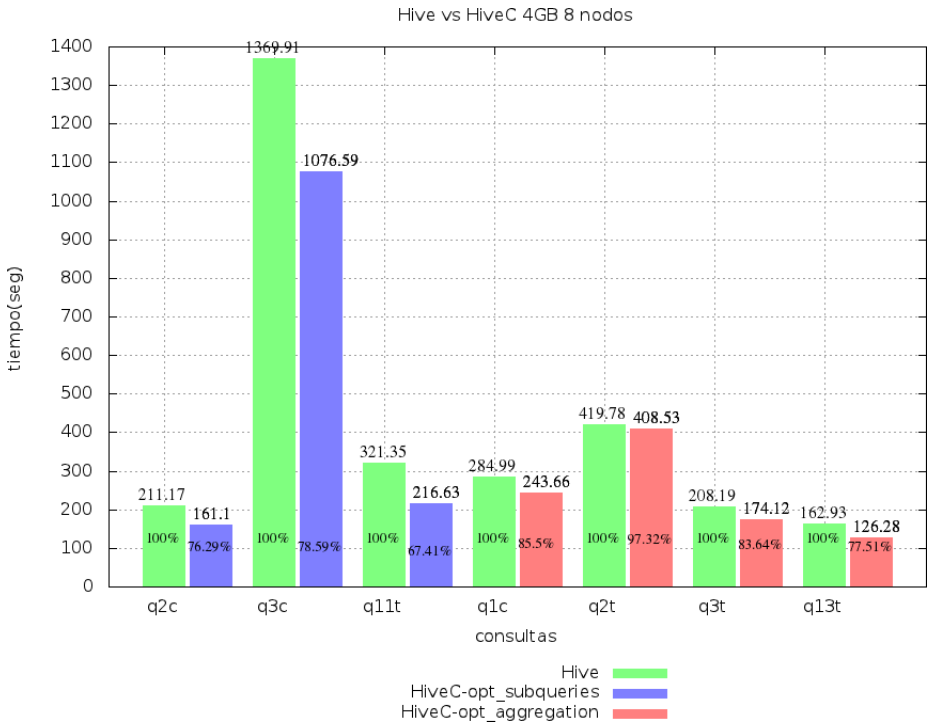


Figura 6.2: Tiempo de ejecución del 1er. Grupo de experimentos con 4GB.

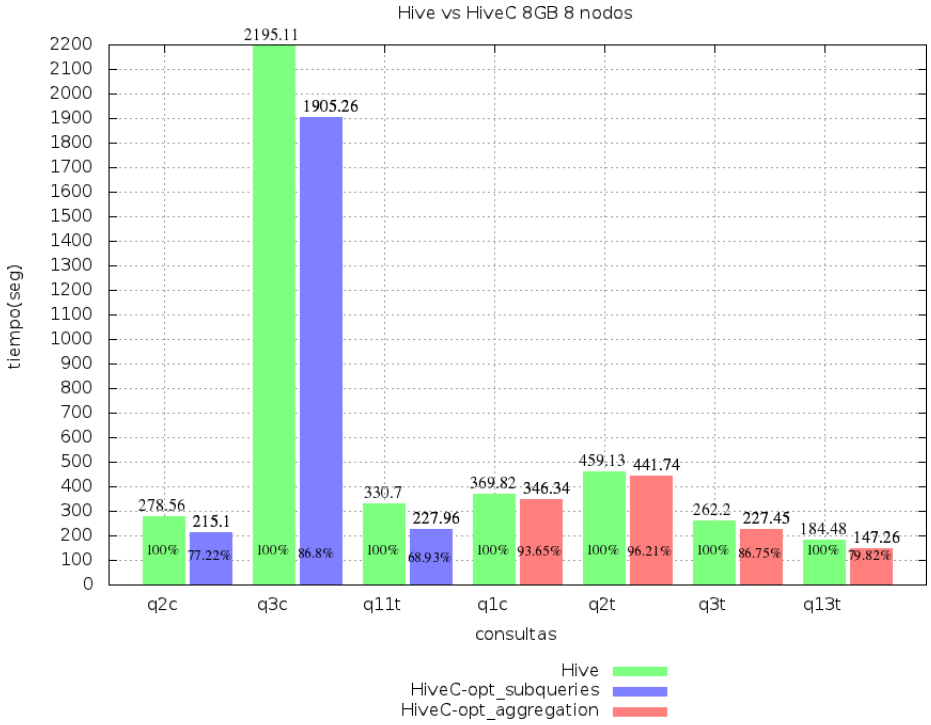


Figura 6.3: Tiempo de ejecución del 1er. Grupo de experimentos con 8GB.

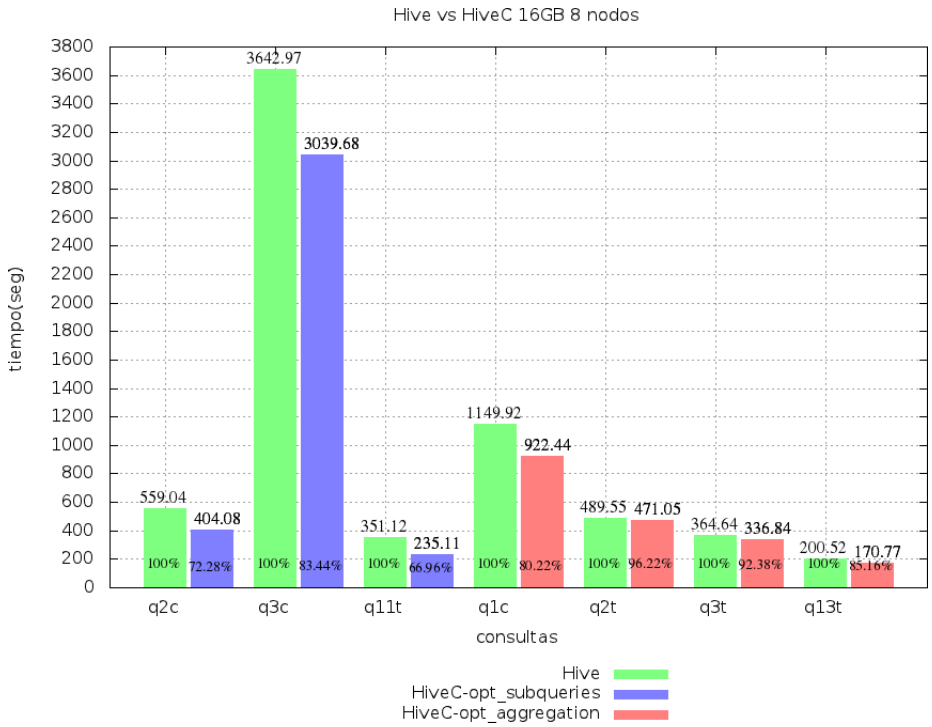


Figura 6.4: Tiempo de ejecución del 1er. Grupo de experimentos con 16GB.

consultas *OLAP* seleccionadas con 8GB de datos. En las consultas *OLAP* donde se aplicó la optimización *opt_subqueries* se redujo el tiempo de ejecución con respecto a *Hive* entre un 13 % y 31 %, y en las consultas *OLAP* donde se aplicó la optimización *opt_aggregation* se redujo el tiempo de ejecución con respecto a *Hive* entre 3 % y 20 %.

La figura 6.4 muestra los Tiempo de ejecución que se obtuvierón al ejecutar las consultas *OLAP* seleccionadas con 16GB de datos. En las consultas *OLAP* donde se aplicó la optimización *opt_subqueries* se redució el tiempo de ejecución con respecto a *Hive* entre un 16 % y 33 %, y en las consultas *OLAP* donde se aplicó la optimización *opt_aggregation* se redujo el tiempo de ejecución con respecto a *Hive* entre 3 % y 20 %.

Obsérvese que la consulta donde más se disminuye el tiempo de ejecución es la consulta q11t, esto se debe a que en esta consulta se eliminan más *trabajos mapreduce* que en las demás consultas. Las consulta *q11t* produce 9 *trabajos mapreduce* en *Hive* y en *HiveC* produce 6 trabajos mapreduce. La consulta q3c es la consulta que mayor tiempo se lleva en ambas versiones de *Hive*, esto se debe a que la consulta opera sobre un mismo conjunto de registros que pertenecen a una misma tabla *fyilog* y cada

registro se compara con todos los registros de la misma tabla, en otras palabras, cada registro se compara con los otros 4GB, 8GB, 16GB de registros. Esto provoca un gran consumo de memoria RAM. Ésta es la única consulta en la que no se considerá el número de *tareas reduce* que generá *Hive* por omisión, debido a que en ambas versiones de *Hive* falla su ejecución. Después de realizar varias pruebas se encontró que, para que la consulta se ejecute de forma satisfactoria en ambas versiones de *Hive*, se debe de configurar a *Hive* para que ejecute con 128 *tareas reduce* por cada *trabajo mapreduce* que generá *Hive* para la consulta.

Por otra parte, también se observa que conforme se aumenta la cantidad de datos de entrada y se mantiene el mismo número de nodos (8 nodos), el tiempo de ejecución de todas las consultas aumenta, esto se debe a que requiere más tiempo de procesamiento. Por lo cuál nos preguntamos, ¿Qué sucederá si mantenemos constante el tamaño de los datos de entrada y se incrementa el número de nodos? Debería de disminuir los tiempos de ejecución. Por es razón se planteó el siguiente grupo de experimentos.

6.2.2. Experimento 2: Variando el número de nodos

Objetivo

Evaluar la escalabilidad de *MapReduce*.

Condiciones

Además de las configuraciones constantes descritas en la sección 6.2, en la tabla 6.3 se observan las configuraciones específicas para este grupo de experimentos.

Tamaño del archivo	16GB
Número de nodos	8, 16 y 20 nodos
Número de *tareas reduce* por *trabajo mapreduce*	Por omisión.

Tabla 6.3: Configuraciones específicas del segundo grupo de experimentos

Se utilizan 16GB de datos de entrada en cada grupo de consultas *OLAP*. Recuerde que en las consultas de *Chatziantoniou* los 16GB están en una tabla, mientras que

en las consultas del estudio *TPC-H* los 16GB se distribuyen en las 8 tablas como se observa en la tabla 6.2. En este grupo de experimentos lo que se varía es la cantidad de nodos en 8, 16 y 20 nodos. En cada configuración un nodo se utiliza para ejecutar los procesos que son servidores en *MapReduce* y *HDFS* y el resto de los nodos se utilizan para ejecutar los procesos clientes en *MapReduce* y *HDFS*. El nodo que actúa como servidor se enlista en el archivo *masters* y los nodos que actuan como clientes se enlistan en el archivo *slaves*. La cantidad de *tareas reduce* que se utilizan por cada *trabajo mapreduce* son los que generá *Hadoop* por omisión, excepto para la consulta q3c en la que se utilizan 128 *tareas reduce* por las razones dadas en el grupo de experimentos anterior. Las demás configuraciones son las que trae por omisión *Hadoop y Hive*.

Metodología

La metodología a utilizar en este grupo de experimentos es la siguiente:

1. Primero se cargan 16GB de datos al sistema *HDFS* para las consultas de Chatziantoniou y 16GB de datos para las consultas TPC-H.

2. Después se ejecutan las consultas de Chatziantoniou y TPC-H con los datos respectivos en *HiveC* y en *Hive*.

3. Se comparan los resultados obtenidos.

4. Cada vez que se varía el número de nodos se eliminan los 16GB de datos del sistema HDFS debido a que se aumentará el número de nodos y se tienen que volver a distribuir los 16GB de datos en los nodos que se aumentan. Posteriormente, se detienen los procesos de *MapReduce* y *HDFS*, y se agregan los nodos clientes al archivo *slaves*. Se inician nuevamente los procesos de *MapReduce* y *HDFS* y se vuelven a cargar los 16GB de datos para cada grupo de consultas. Una vez realizada la configuración, se vuelve a repetir a partir del paso 2, hasta que se configure con 16 nodos.

Hipótesis

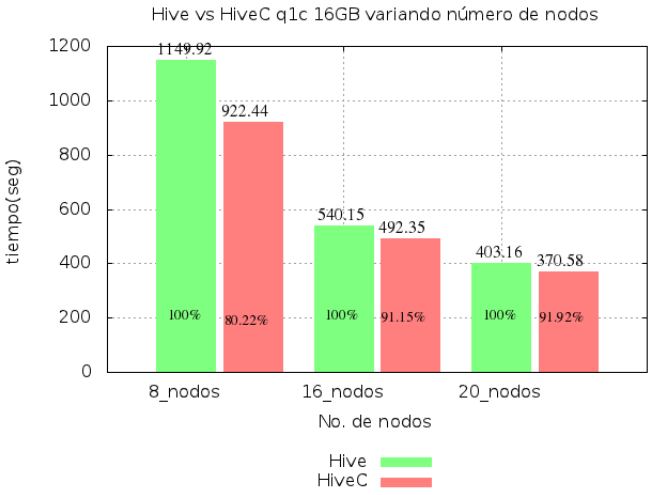


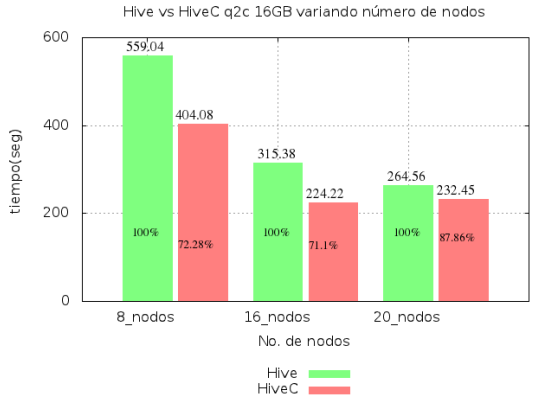
Figura 6.5: Tiempos de ejecución de la consulta q1c procesando 16GB de datos en 8, 16 y 20 nodos.

Conforme se aumente el número de nodos, el rendimiento de las consultas *HiveQL* será mejor. En teoría el rendimiento debe de ser proporcional al número de nodos agregados. Por lo que al agregar más nodos manteniendo el tamaño de datos de entrada se debe de mejorar el tiempo de ejecución proporcionalmente al nivel de paralelismo.

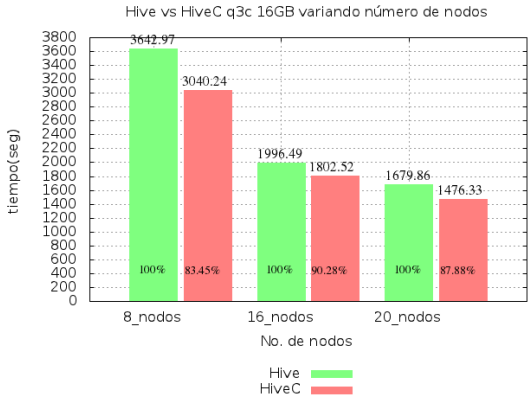
Resultados

En las consultas de *Chatziantoniou* donde los 16GB están en una tabla, los tiempos de ejecución de las consultas en *Hive* y *HiveC* disminuyen conforme se van agregando nodos, esto se debe debido a la escalabilidad que ofrece *MapReduce* para grandes volúmenes de datos. Por ejemplo, en la figura 6.5 se muestran los tiempos de ejecución de la consulta q1c para 8, 16 y 20 nodos. Obsérvese que al aumentar la cantidad de nodos y procesar los mismos 16GB de datos en una tabla, el tiempo de ejecución de la consulta en *Hive* y *HiveC* se disminuye de manera proporcional al nivel de paralelismo. Lo mismo sucede en las consultas *q2c* y *q3c* cuyos tiempos de ejecución se observan en las figuras 6.6a y 6.6b respectivamente.

En las consultas del estudio *TPC-H* donde los 16GB se distribuyen en 8 tablas como se observa en la tabla 6.2, no siempre se disminuyó el tiempo de ejecución de las consultas conforme se agregaron nodos al clúster. Por ejemplo, la figura 6.7a muestra

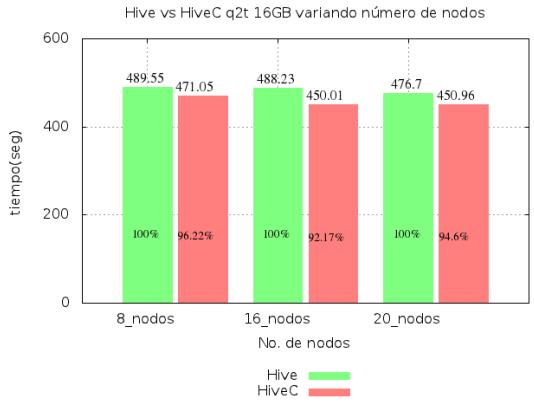


(a) Tiempos de ejecución de la consulta q2c procesando 16GB de datos en 8, 16 y 20 nodos.

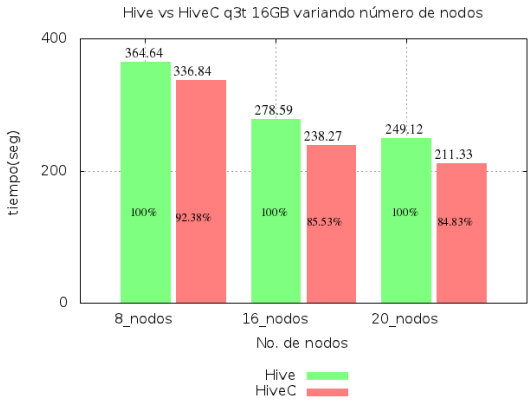


(b) Tiempos de ejecución de la consulta q3c procesando 16GB de datos en 8, 16 y 20 nodos.

Figura 6.6: Tiempos de ejecución de las consultas q2c y q3c procesando 16GB en 8, 16 y 20 nodos. Obsérvese como se disminuye el tiempo de ejecución de las consultas de acuerdo al nivel de paralelismo.

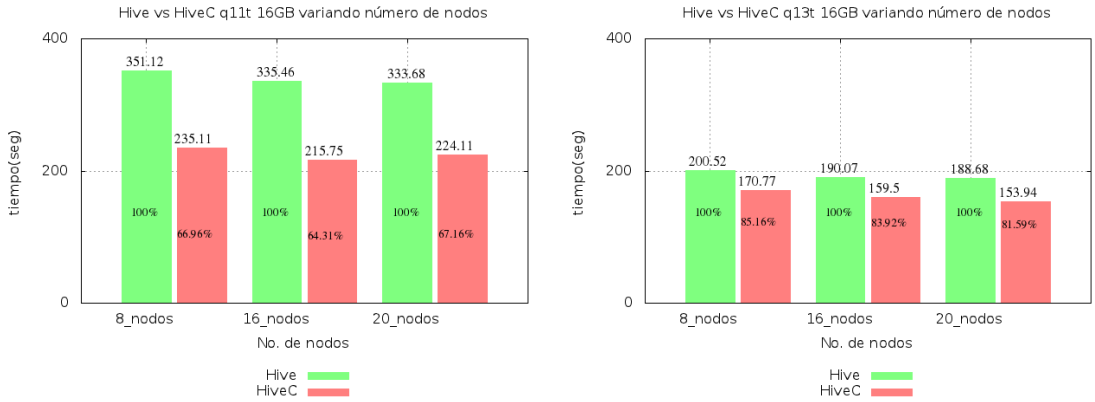


(a) Tiempos de ejecución de la consulta q2t procesando 16GB de datos en 8, 16 y 20 nodos.



(b) Tiempos de ejecución de la consulta q3t procesando 16GB de datos en 8, 16 y 20 nodos.

Figura 6.7: Tiempos de ejecución de las consultas q2t y q3t procesando 16GB de datos en 8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución de la tabla q2t no se disminuye conforme se aumentan los nodos, esto se debe a que la consulta procesa tablas con pocos datos y al parecer *MapReduce* no es escalable con pocos datos. Por lo otro lado, los tiempos de ejecución de la consulta q3t si se reducen conforme se aumentan nodos debido a que involucra tablas con una mayor cantidad de datos.



(a) Tiempos de ejecución de la consulta q11t procesando 16GB de datos en 8, 16 y 20 nodos.

(b) Tiempos de ejecución de la consulta q13t procesando 16GB de datos en 8, 16 y 20 nodos.

Figura 6.8: Tiempos de ejecución de las consultas q11t y q13t procesando 16GB de datos en 8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución de ambas consultas no se disminuye conforme aumentan los nodos, esto se debe a que ambas consultas procesan tablas con pocos datos y al parecer *MapReduce* no es escalable con pocos datos.

los tiempos de ejecución de la consulta *q2t* para procesar 16GB de datos distribuidos en 8 tablas en 8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución no disminuyen significativamente conforme se agregan nodos al clúster, esto se debe a que la consulta procesa las tablas *nation, region, supplier, partsupp y part* (ver código 6.4 en página 125) las cuáles en total tienen 2.06 GB de datos. Como son pocos datos, entonces se generan pocas *tareas map y reduce* y no se aprovecha el nivel de paralelismo que va aumentando conforme se agregan más nodos al clúster. En cambio, en la figura 6.7b se muestran los tiempos de ejecución de la consulta consulta *q3t* para procesar 16GB de datos distribuidos en 8 tablas en 8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución de esta consulta si disminuyen significativamente conforme se agregan nodos al clúster, esto se debe a que la consulta procesa las tablas *customer, orders y lineitem* (ver código 6.5 en pagina 126) las cuáles en total tienen 13.84 GB. Como es una cantidad de datos considerable, entonces se genera una cantidad considerable de *tareas map y reduce* y se aprovecha el nivel de paralelismo que se va aumentando conforme se agregan más nodos al clúster. En la figura 6.8a se muestra los tiempos de ejecución de la consulta *q11t* para procesar 16GB de datos distribuidos en 8 tablas en

8, 16 y 20 nodos. Obsérvese que los tiempos de ejecución de esta consulta tampoco disminuyen significativamente conforme se agregan nodos al clúster, esto se debe al igual que en la consulta *q2t* a que las tablas que procesa son *nation, supplier y partsupp* (ver código 6.6 en página 126) y estas solo suman 1.72GB de datos, por lo tanto, no se aprovecha el nivel de paralelismo. Lo mismo sucede con los tiempos de ejecución de la consulta *q13t* que se muestran en la figura 6.8b donde la cantidad de datos que procesa en todas las tablas que involucra es de 2.84GB.

6.2.3. Experimento 3: Variando el número de *tareas reduce*

Objetivo

Evaluar como el número de *tareas reduce* afectan el desempeño de la ejecución de las consultas *HiveQL* en *Hive* y en *HiveC*.

Condiciones

En este grupo de experimentos además de las configuraciones constantes descritas en la sección 6.2, las configuraciones específicas utilizadas se observan en la tabla 6.4.

Tamaño del archivo	16GB
Número de nodos	8 nodos
Número de *tareas reduce* por *trabajo mapreduce*	Se utiliza la mitad y el doble de las *tareas reduce* que generá el primer *trabajo mapreduce* de cada consulta *HiveQL*

Tabla 6.4: Configuraciones específicas del tercer grupo de experimentos

Se utilizan de manera constante 16GB de datos de entrada en cada grupo de consultas *OLAP*. También se utilizan de manera constante 8 nodos del clúster. Al igual que en los experimentos anteriores se utiliza un nodo como nodo sevidor y siete nodos como nodos clientes. Por cada consulta, se compara la configuración con el número de tareas reduce por omisión, la mitad de ese número de *tareas reduce* y con el doble de ese número de *tareas reduce*. *Hive* por cada consulta genera varios trabajos mapreduce, y a cada *trabajo mapreduce* le asigna un número de *tareas reduce*

```
<configuration>
    <property>
        <name>mapred.reduce.tasks</name>
        <value> 2 </value>
    </property>
</configuration>
```

Código 6.8: Configuración del archivo *hive-site.xml* de *Hive* para configurar el número de *tareas reduce* a ejecutar en cada *trabajo mapreduce*.

diferente. Nosotros tomamos el número de *tareas reduce* del primer *trabajo mapreduce* de cada consulta, y con la mitad y el doble de ese número configuramos el número de *tareas reduce* a ejecutar en el archivo *hive-site.xml*. Este archivo permite entre otras cosas configurar el número de *tareas reduce* que deben de ejecutar los *trabajos mapreduce*. Sin embargo, restringe a que todos los *trabajos mapreduce* se ejecuten con el mismo número de *tareas reduce* especificado. Por lo tanto, al configurar a la mitad y el doble del número que se obtuvo, todos los *trabajos mapreduce* de cada consulta se ejecutarán con el mismo número de *tareas reduce*. Por ejemplo, para la consulta 1 de Chatziantoniou, *Hive* genera 4 *trabajos mapreduce*, y para el primero de estos, se crean cinco *tareas reduce*. Al duplicarse se obtienen diez *tareas reduce*, este es el número de *tareas reduce* que ejecutan todos los *trabajos mapreduce* de la consulta. Posteriormente, se reduce a la mitad, lo cual daría 2.5 *tareas reduce*. Como es un número real, entonces se redondea al número entero inferior si la parte decimal está entre .1 y .4, y si está entre .5 y .9, entonces se redondea al número entero posterior. En este caso, se utilizarán 3 *tareas reduce* en cada *trabajo mapreduce* de la consulta.

Metodología

La metodología a utilizar en este grupo de experimentos es la siguiente:

1. Primero se cargan 16GB de datos al sistema *HDFS* para las consultas de Chatziantoniou y 16GB de datos para las consultas TPC-H.

2. Posteriormente, para especificar el número de *tareas reduce* a utilizar en cada *trabajo mapreduce* de una consulta se configura el archivo *hive-site.xml* de *Hive*

como se observa en el código 6.8.

3. Después se ejecutan las consultas de Chatziantoniou y TPC-H con los datos respectivos en *Hive* y en *HiveC*.

4. Se comparan los resultados obtenidos.

5. Cada vez que se va a ejecutar una nueva consulta de Chatziantoniou o TPC-H con diferente número de *tareas reduce* se debe de configurar el archivo *hive-site.xml* como se observa en el código 6.8 y se vuelve a ejecutar a partir del paso 5.

Hipótesis

Cuando se disminuye la cantidad de *tareas reduce* a ejecutar en un *trabajo mapreduce* se puede tener un mejor desempeño si la cantidad de datos que se distribuyen en cada *tarea reduce* no es tán grande, debido a que se ocupa menos tiempo para administrar las *tareas reduce*. En caso contrario, puede llevar más tiempo de ejecución. Por otra parte, cuando se aumenta al doble la cantidad de *tareas reduce* a ejecutar en un *trabajo mapreduce* se puede tener un mejor desempeño, cuando cada *tarea reduce* recibe una cantidad de datos considerable, debido a que el tiempo de procesamiento de tales datos domina al tiempo que ocupa *MapReduce* para administrar las *tareas reduce*. En caso contrario, puede llevar más tiempo de ejecución.

Resultados

Este grupo de experimentos no se contempló en el objetivo de esta tesis. Sin embargo, se realizó debido a que creemos que es importante encontrar la relación entre la cantidad de datos a procesar y el número de *tareas reduce* a ejecutar en un *trabajo mapreduce*. Desafortunamente, los resultados que obtuvimos no fueron suficientes para poder interpretar dicha relación, ya que no se consideraron cosas como: la manera en que estaban distribuidos los datos, el número de *tareas reduce* que se ejecutaban al mismo tiempo en cada nodo del clúster, cuánta cantidad de datos

procesaba cada *tarea reduce*, y además faltó realizar más experimentos con diferentes números de *tareas reduce*.

Sin embargo, se presentan los siguientes resultados para observar que en algunas consultas su tiempo de ejecución disminuye cuando el número de *tareas reduce* se disminuye a la mitad o cuando se aumenta al doble. En las gráficas 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, 6.15 se muestran los resultados para las consultas q1c, q2c, q3c, q2t, q3t, q11t y q13t respectivamente. En todos los resultados, las dos gráficas que están asociadas a la leyenda *default* indican el tiempo de ejecución de la consulta con las *tareas reduce* por omisión que produce *Hive* y *HiveC* para cada *trabajo mapreduce* de cada consulta. El número que está entre paréntesis después de la leyenda *default* indica el número de *tareas reduce* que produció *Hive* para el primer *trabajo mapreduce* de una consulta. El número que está entre paréntesis después de las leyendas *mitad y doble* indican el numero de *tareas reduce* con el que se configuró todos las *trabajos mapreduce* de una consulta.

El tiempo de ejecución de las consultas q1c y q2c diminuye cuando se diminuye el número de *tareas reduce* a la mitad como se observa en las figuras 6.9 y 6.10, mientras que el tiempo de ejecución de la consulta q13t disminuye cuando se aumenta el número de *tareas reduce* al doble como se observa en la figura 6.15. El tiempo de ejecución de las demás consultas aumenta cuando se disminuye el número de *tareas reduce* a la mitad y cuando se aumenta el número de *tareas reduce* al doble.

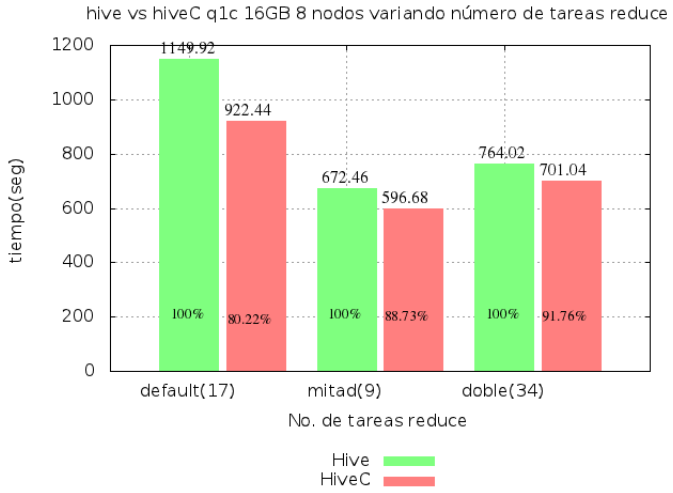


Figura 6.9: Tiempo de ejecución de la consulta q1c con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

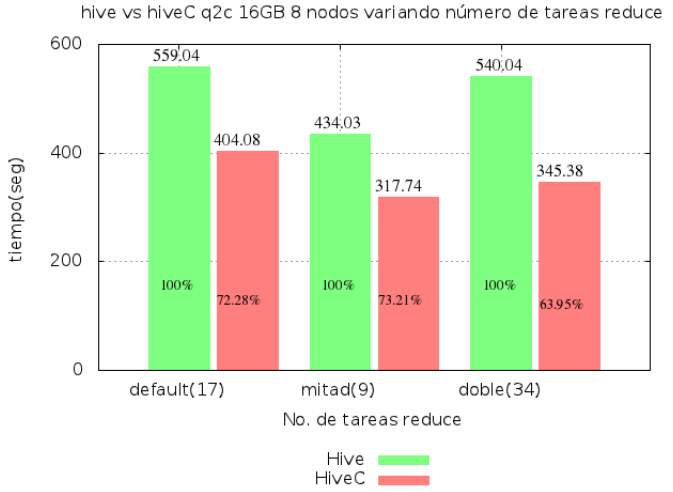


Figura 6.10: Tiempo de ejecución de la consulta q2c con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

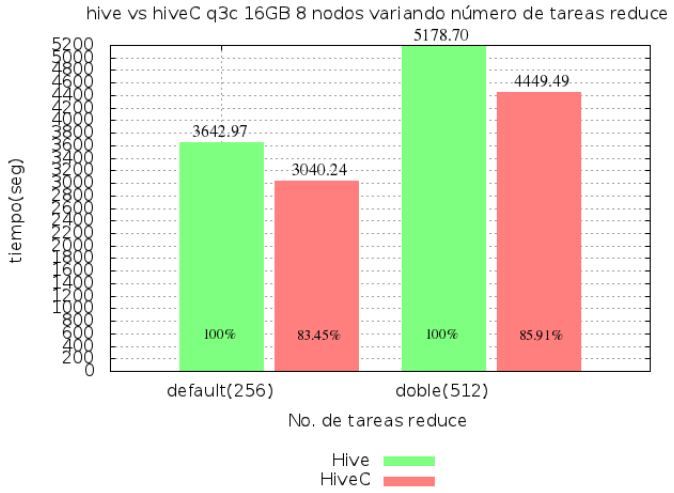


Figura 6.11: Tiempo de ejecución de la consulta q3c con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

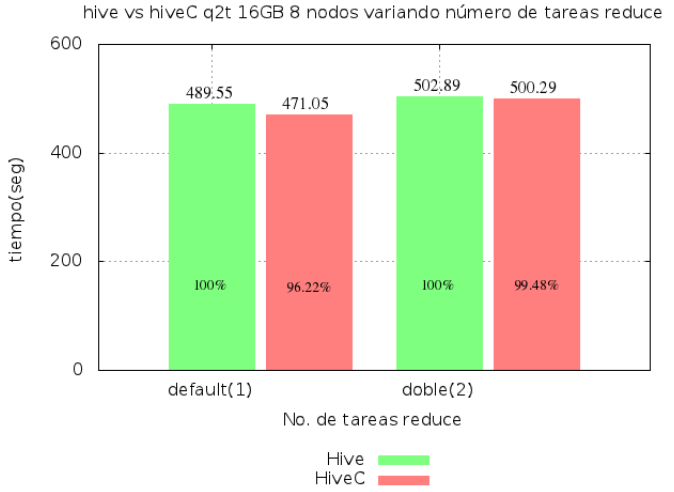


Figura 6.12: Tiempo de ejecución de la consulta q2t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

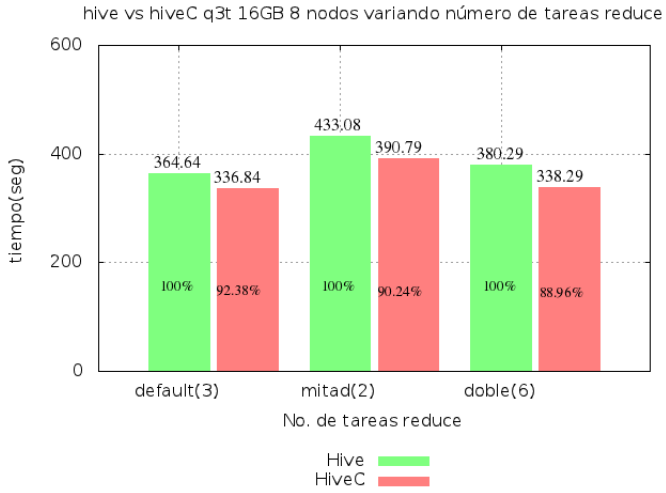


Figura 6.13: Tiempo de ejecución de la consulta q3t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

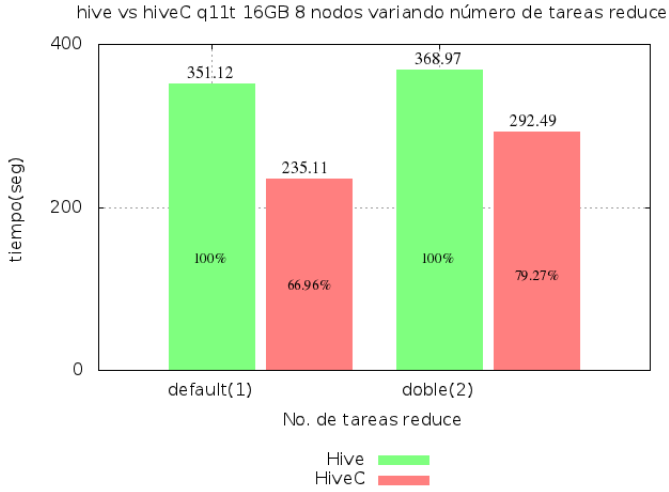


Figura 6.14: Tiempo de ejecución de la consulta q11t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

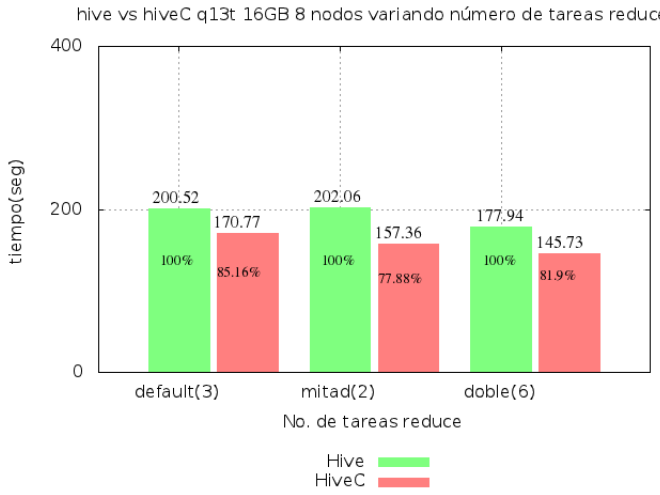


Figura 6.15: Tiempo de ejecución de la consulta q13t con 16GB de datos de entrada de manera constante, 8 nodos, variando la cantidad de *tareas reduce*.

6.3. Resumen

El tiempo de ejecución de las consultas *OLAP* seleccionadas de Chatziantoniou y del estudio TPC-H en *HiveC* fue menor que en *Hive* en los tres grupos de experimentos propuestos. Las consultas que mejor rendimiento tienen son aquellas a las que se les aplica la optimización de eliminación de secuencia de operadores *Hive* redundantes en un *DAG*. Esto se debe a que esta optimización, al eliminar los operadores redundantes, puede eliminar varios *trabajos mapreduce*, a diferencia de la optimización de eliminar *trabajos mapreduce* asociados a funciones de agregación y agrupación que solo puede eliminar un *trabajo mapreduce* por cada función de agregación y agrupación que se encuentre en una consulta o subconsulta.

Por otra parte, se comprobó la escalabilidad y el paralelismo de *Hive* y *Hadoop*, ya que al aumentar el tamaño de datos de entrada y mantener el número de nodos del clúster el tiempo de ejecución de cada consulta aumenta, pero cuando se mantiene el tamaño de datos de entrada y se incrementa el número de nodos el tiempo de ejecución de las consultas disminuye. Sin embargo, este comportamiento no es general, y dependiendo de la consulta, la escalabilidad, puede no ser la esperada en base al número de nodos utilizados.

Capítulo 7

Conclusiones y trabajo futuro

Hive es un *Datawarehouse* sobre *Hadoop*, la versión libre y abierta de *MapReduce*. *Hive* ofrece una infraestructura de bases de datos y un compilador *HiveQL* que permite especificar sentencias muy similares a sentencias SQL. El compilador de *Hive* se encarga de procesar y compilar las sentencias *HiveQL* a una secuencia de *trabajos mapreduce* que se ejecutan en *Hadoop*. Este proceso lo realiza a través de cuatro fases: compilación (análisis léxico, sintáctico y semántico), optimización, generación del plan físico y ejecución de *trabajos mapreduce* en *Hadoop*.

Esta tesis ha investigado la organización de *Hive* y ha propuesto nuevas optimizaciones para el optimizador de *HiveQL* que permitan un mejor desempeño de algunos tipos de consultas *HiveQL*. Para lograrlo se analizaron las ocho optimizaciones que tiene la versión 0.8 de Hive (versión estable cuando se inició la investigación) y se descubrió que algunas de las optimizaciones son obvias/lógicas y se utilizan en otros Sistemas de Gestión de Base de Datos (SGBD) como: filtrar los registros de tablas o particiones, y/o filtrar las columnas de los registros a utilizar lo antes posible para evitar que la consulta procese datos innecesarios en los operadores superiores. Por otra parte, las optimizaciones siguientes están pensadas para que cuando un *DAG* se transforme a una secuencia de *trabajos mapreduce*, cada *trabajo mapreduce* se ejecute de manera eficiente considerando las condiciones del ambiente mapreduce: 1) reducción del uso de la red por medio de agregaciones parciales en *tareas map*,

2) reducción de la escritura a *HDFS* por medio de agregaciones parciales en *tareas reduce*, 3) aprovechamiento del paralelismo de *MapReduce* a través de la división de datos en la operación *Group-BY*, 4) reducción del uso de memoria por reordenamiento de los datos en el operador *join*, y 5) reducción del uso de red realizando el *join* del lado del *map*.

A partir de este análisis se descubrió que cuando un consulta *HiveQL* involucra subconsultas y las subconsultas son similares, *Hive* no se da cuenta de ello. Y por lo tanto, cada subconsulta similar genera ramas de operadores redundantes en un *DAG*. Estas ramas de operadores redundantes generán a su vez *trabajos mapreduce* similares y/o repetidos, cuando el *DAG* se transforma a un plan físico. Los *trabajos mapreduce* similares y/o repetidos provocan un bajo rendimiento de las consultas *HiveQL* debido a que ocasionan que se lean las mismas tablas en diferentes *trabajos mapreduce*, que se realice el mismo procesamiento en diferentes *trabajos mapreduce* y/o se envien los mismo registros por la red a diferentes *trabajos mapreduce*. Por esta razón se propusó e implementó una optimización en el optimizador de *Hive* que identifica ramas de operadores redundantes en un *DAG*, dando como resultado que las operaciones redundantes en una consulta *HiveQL* se realicen una sola vez. De esta manera, se eliminan los *trabajos mapreduce* similares o repetidos y se obtiene un mejor desempeño de las consultas *HiveQL* de este tipo.

También nos dimos cuenta que cuando se procesa una consulta o subconsulta *HiveQL* que involucra funciones de agregación y agrupación (*sum(), avg(), count()*, entre otras) sobre registros que se obtienen después de realizar una operación *join* o una subconsulta, entonces se generan dos *trabajos mapreduce*: en el primer *trabajo mapreduce* se resuelve la operación *join* o la subconsulta y en el segundo *trabajo mapreduce* se resuelve la función de agregación y agrupación. Para reducir la cantidad de registros que escribe el primer *trabajo mapreduce* al sistema *HDFS*, la optimización de *Hive* correspondiente realiza la función de agregación de manera parcial al final de las *tareas reduce* del primer *trabajo mapreduce*. Se dice que la función de agregación implicada se realizó de manera parcial porqué *Hive* no verifica si cada *tarea reduce*

del primer *trabajo mapreduce* recibe todos los registros de un grupo. Los grupos de registros se forman con los registros que tienen los mismos valores en las columnas que se especifican en la cláusula *Group-By*. Esta optimización verifica si cada *tarea reduce* del primer *trabajo mapreduce* recibe todos los registros de un grupo, en cuyo caso los resultados que se obtienen del primer *trabajo mapreduce* no son parciales, sino son los resultados finales debido a que se aplica sobre todos los registros de un grupo. Y por lo tanto, elimina el segundo *trabajo mapreduce* que resulta innecesario.

Las dos optimizaciones descritas se pueden aplicar a una misma consulta *HiveQL* debido a que la consulta *HiveQL* puede involucrar subconsultas similares y cada subconsulta similar puede involucrar funciones de agregación y agrupación. Las optimizaciones se agregarón al optimizador de *Hive* versión 0.8 y la nueva versión la nombramos *HiveC*.

Para evaluar nuestras optimizaciones se seleccionó un subconjunto de las consultas de Chatziantoniou y del estudio TPC-H en las cuales se aplicaron las optimizaciones que realizamos. Estas consultas se ejecutaron en *Hive* versión 0.8 y en *HiveC* en tres condiciones diferentes: variando el tamaño de los datos de entrada, variando el número de nodos y variando el número de *tareas reduce*. Bajo las tres condiciones, el tiempo de ejecución de todas las consultas en *HiveC* fue menor que el tiempo de ejecución en *Hive* versión 0.8, con una reducción del tiempo de ejecución de entre un 3 % y un 30 %, aunque en la mayoría de las consultas seleccionadas se reduce el tiempo entre un 10 % y un 30 %. Cuando se varío el tamaño de los datos de entrada, se observó que conforme se aumenta el tamaño de los datos de entrada y se mantiene el número de nodos del clúster, los tiempos de ejecución de las consulta en *Hive* y *HiveC* se incrementan. Cuando se varío el número de nodos, se observó que conforme se agregan nodos al clúster y el tamaño de los datos de entrada permanece constante, el tiempo de ejecución de las consultas disminuye con lo cual se comprueba la escalabilidad de *MapReduce*.

Por otra parte, la consulta 3 de Chatziantoniou no se pudó ejecutar con la configuración por defecto de Hive. Para esta consulta, *Hive* generó 5 *trabajos*

mapreduce, donde el primer *trabajo mapreduce* configura 18 *tareas map* y 5 *tareas reduce* para procesar 4 GB de datos. Esta consulta falló en el primer *trabajo mapreduce*. Para encontrar la razón, se revisó el Log de *Hive* y se analizó la cantidad de memoria que gastaba esta consulta en todos los nodos cuando se ejecutaba el primer *trabajo mapreduce*. Nos dimos cuenta que la consulta no se ejecutaba debido a que se producía un desbordamiento de memoria a la hora de ejecutar las *tareas reduce* del primer *trabajo mapreduce*. Esto se debió a que la cantidad de datos que recibía y procesaba cada *tarea reduce* era grande. Entonces se buscó una configuración factible para poder ejecutar esta consulta y se encontró que se podía ejecutar con 64 *tareas reduce* utilizando un heapsize de 1GB en cada *tarea reduce*. Sin embargo, el tiempo de ejecución no era bueno. La razón de este mal desempeño es que los demás *trabajos mapreduce* procesaban menos datos y *Hive* solo permite configurar el mismo número de *tareas reduce* para todos los *trabajos mapreduce* que se generan para una consulta, de tal manera que cómo los demás *trabajos mapreduce* procesan menos datos, entonces eran muchas *tareas reduce* para procesar dichos datos y el tiempo de ejecución se elevó considerablemente debido al costo de manejar más *tareas reduce*.

En resúmen, las contribuciones de este trabajo de tesis son:

- Un análisis del funcionamiento global de *Hive* versión 0.8; particularmente del compilador de *Hive* incluyendo al optimizador de *Hive* y las optimizaciones actuales de consultas *HiveQL*.

- Un análisis de posibles nuevas optimizaciones en base a las *DAGs* producidas por consultas *HiveQL*.

- El diseño de dos optimizaciones para el optimizador de *Hive* versión 0.8: Una optimización para eliminar *trabajos mapreduce* similares o repetidos asociados a subconsultas similares; y una optimización para eliminar el segundo *trabajo mapreduce* asociado a consultas o subconsultas que involucran funciones de agregación y agrupación que se aplican sobre registros que se generan después de aplicar una operación *join* y/o subconsultas.

- Extensión de *Hive* versión 0.8 con nuestras optimizaciones. Esta versión la nombramos *HiveC*.

- Una evaluación de nuestras optimizaciones con consultas *OLAP* que se seleccionaron de estudios ampliamente usados para evaluar sistemas de base de datos y *Datawarehouse*. Esta evaluación se realizó ejecutando dichas consultas en *Hive* versión 0.8 y en *HiveC* en diferentes condiciones: variando el tamaño de los datos de entrada, variando el número de nodos y variando el número de *tareas reduce*. Nuestras optimizaciones mejoran el desempeño de *Hive* entre un 3 % y 30 %.

7.1. Trabajo a futuro

Al realizar nuestros experimentos, el primer *trabajo mapreduce* de la consulta 3 de Chatziantoniou no se pudó ejecutar con el número de *tareas reduce* que generó *Hive* por defecto. Entonces se buscó una configuración adecuada y nos dimos cuenta que el número de *tareas reduce* que se especifica en el archivo de configuración de *Hive* es el número *tareas reduce* que van a ejecutar todos los *trabajos mapreduce* de una consulta. Esto es ineficiente debido a que cada *trabajo mapreduce* procesa una cantidad diferente de datos. Para evitarlo, se propone realizar un estudio que relacione distintas cantidades y tal vez tipos de datos a procesar con distintos números de *tareas reduce*.

En base al estudio anterior, se propone agregar a *Hive* la capacidad de poder determinar el número de *tareas reduce* a ejecutar por cada *trabajo mapreduce* de una consulta, en base a la cantidad y tal vez tipos de datos a procesar.

También se propone realizar la optimización del operador *JoinOperator* descrita en el capítulo 4. De tal manera, que se reduzca la cantidad de datos que se envían por la red.

Por otra parte, creemos que aún se pueden eliminar más *trabajos mapreduce* de una consulta *HiveQL* siguiendo nuestro enfoque (en contraste con el de *YSMART*), cuando en una misma rama de un *DAG* existen secuencias de operadores que aunque

no son redundantes, crean *trabajos mapreduce innecesarios* como se explica en la sección 4.5. En *YSMART* sería parecido a lo que realizan con la correlación de flujo de trabajo. En nuestro enfoque, es necesario modificar operadores existentes o añadir nuevos operadores. Esto eliminará las lecturas repetidas de una tabla, se envien menos registros a la red y se eliminen escrituras al sistema *HDFS* innecesarias.

*P*or último, Hive tiene una comunidad de desarrollo de *Software libre* que evalúa las aportaciones de sus miembros, y agrega dichas aportaciones a las versiones oficiales de *Hive*. Se propone someter nuestras optimizaciones en dicha comunidad para que se agreguen a las versiones oficiales de *Hive*.

Apéndice A

Eliminación de operadores redundantes entre ramas simples o compuestas en un *DAG*

Este apéndice extiende la explicación de nuestra optimización de eliminación de operadores redundantes entre ramas simples o compuestas en un *DAG*. Presenta las estructuras de datos de *Hive* que utilizamos en nuestra optimización, las estructuras de datos propias de nuestra optimización, cómo se manipulan las estructuras de datos en las entidades de la optimización (*GraphWalker, Dispatcher, Processor*) y algunos otros ejemplos de *DAGs* que permiten observar otros casos en los cuáles nuestra optimización se aplica.

Esta optimización elimina los operadores redundantes que se encuentran entre ramas simples o compuestas que están vinculadas por un operador *JoinOperator (JO)*. Las ramas simples son aquellas que no tienen ramificaciones. Las ramas compuestas son aquellas que tienen ramificaciones, y cada ramificación está vinculada por un operador *JO*.

Toda optimización incluyendo esta se inicia en la entidad *GraphWalker*. Para ejemplificar como funciona cada una de las entidades de nuestra optimización se tomará como ejemplos los planes físicos de la consulta 2 de Chatziantoniou y

consulta 11 del estudio de mercado *TPC-H* que se observa en las figura A.1 y A.3 respectivamente. Aunque se utiliza el plan físico, esta optimización trabaja con un *DAG* de una consulta. Se utiliza el plan físico de una consulta debido a que la optimización está pensada para que tenga efecto cuando un *DAG* se transforma a un plan físico. Recuerde que el *DAG* en las figuras es el árbol sin los cuadros punteados que representan los *trabajos mapreduce*.

A.1. Estructuras de datos de Hive

Cada optimización en *Hive* utiliza sus propias estructuras de datos en cada una de sus entidades (*GraphWalker, Dispatcher, Processor*). Algunas estructuras de datos se pueden compartir entre las entidades pero no entre optimizaciones.

El *DAG* es la única estructura de datos que es común en todas las optimizaciones. Un *DAG* está constituido por nodos, donde cada nodo es un operador de *Hive*. Un *DAG* se recorre a partir de los operadores *TableScanOperator (TS)*. La estructura de datos que contiene todos los operadores *TS* de un *DAG* es la lista *topNodes*. Todas las optimizaciones de *Hive* recibe esta lista en el *GraphWalker*. Por otro lado, cada operador de *Hive* maneja sus propias estructuras de datos. Las estructuras de datos de un operador que utilizamos en nuestra optimización son:

- *RowSchema:* Es una clase que contiene las columnas que le pasa un operador a los siguientes operadores (hacia arriba en un *DAG*).

- *childOperators:* Es una lista que contiene los operadores hijos de un operador. Los operadores hijos de un operador son aquellos operadores que vincula hacia arriba en un *DAG*.

- *parentOperators:* Es una lista que contiene los operadores padres de un operador. Los operadores padres de un operador son aquellos operadores que vincula hacia abajo en un *DAG*.

- *predicate:* En un operador *FilterOperator*, *predicate* es una clase que contiene la condicional de un cláusula *where*.

- *keyCols:* En un operador *ReduceSinkOperator*, *keyCols* es una lista que contiene las columnas que se utilizan cómo clave en los pares intermedios $< clave, valor >$ que emiten los operadores *ReduceSinkOperator* en una *tarea map*.

- *valueCols:* En un operador *ReduceSinkOperator*, *valueCols* es una lista que contiene las columnas que se utilizan cómo valor en los pares intermedios $< clave, valor >$ que emiten los operadores *ReduceSinkOperator* en una *tarea map*.

A.2. Estructuras de datos propias de nuestra optimización

Además de las estructuras de datos de *Hive* presentadas en la sección A.1, en nuestra optimización utilizamos las siguientes:

- *mapJoinsToScan:* Es un mapa[1] que guarda los operadores *JO* que se van encontrando en el recorrido de un *DAG*, almacena como clave un operador *JO* y como valor la lista de operadores *TS* que vincula.

- *mapTablesToScan:* Es un mapa que almacena los operadores *TS* que representan a las ramas simples o compuestas que leen las mismas tablas. Utiliza como clave un *string "tablax-tablay"* que indica el par de tablas que lee cada par de operadores *TS* en la lista que almacena como valor. Por ejemplo, si la clave es *"fyilog-status"* indica que contiene una lista de *TS* donde cada par de *TS* leen las tablas *fyilog y status* respectivamente.

[1]Un mapa es una estructura de datos compuesta por un conjunto de claves irrepetibles y un conjunto de valores asociados a una clave.

- *listEqualNodes:* Es una lista que almacena los últimos nodos hasta donde dos ramas simples o compuestas tienen operadores iguales.

A.3. GraphWalker

En nuestra optimización, la entidad *GraphWalker* identifica los operadores *TS* que representan a diferentes ramas de un *DAG*, que leen las mismas tablas y por lo tanto pueden tener operadores redundantes. *GraphWalker* recibe del *optimizador de Hive* los *TS* de un *DAG* en la lista *topNodes*. El código A.1 muestra el pseudoalgoritmo que se utiliza en la entidad *GraphWalker*. Obsérvese que:

1. Por cada operador *TS* de la lista *topNodes* se recorre un *DAG* hasta encontrar un operador *JO*, entonces se almacena en el mapa *mapJoinsToScan* utilizando como clave el *JO* y como valor el *TS*. Esto corresponde a los pasos 2 al 11 en el código A.1.

   Por ejemplo, para el *DAG* de la consulta 2 de Chatziantoniou que se observa en la figura A.1, la entidad *GrapWalker* recibe del *optimizador de Hive* la lista *topNodes* con los operadores *TS* del nodo 1 y 8. Cuando se inicia el recorrido del *DAG* con el operador *TS* del nodo 1 se alcanza el operador *JO* del nodo 22, entonces se guarda como clave el *JO_n22*[2] y como valor el *TS_n1* en el mapa *mapJoinsToScan*. Posteriormente, se inicia otro recorrido del *DAG* con el operador *TS* del nodo 8 con el cual se vuelve a alcanzar al operador *JO* del nodo 22, entonces se agrega el operador *TS_n8* a la lista de *TS* que se vincula con el *JO_n22* en el mapa *mapJoinsToScan*. En el paso 1 de la figura A.2 se muestran los registros que se guardan en el mapa *mapJoinsToScan* en este paso para el *DAG* de la consulta 2 de Chatziantoniou.

   Otro ejemplo, con en el *DAG* de la consulta 11 del estudio TPC-H que se observa en la figura A.3, la entidad *GraphWalker* recibe del *optimizador de Hive* la lista

[2] *Op_n#* es para hacer alusión al nodo #, en realidad solo se guarda el operador *Op* correspondiente

```
1  graphWalker(topNodes){
2    foreach TS in topNodes{
3      JO = walk_DAG_until_join(TS);
4      if (JO != null){
5        if mapJoinsToScan.hasKey(JO){
6          mapJoinsToScan.get(JO).add(TS);
7        } else {
8          mapJoinsToScan.put(JO, TS);
9        }
10     }
11   }
12
13   if (mapJoinsToScan. size > 0){
14     remove_all_joins_that_doesnot_have_two_TS(mapJoinsToScan);
15
16     if (mapJoinsToScan.size() == 1){
17       listTwoTS = mapJoinsToScan.firstEntry.getValue();
18       TS1 = listTS.get(0);
19       TS2 = listTS.get(1);
20       if (TS.getTable() == TS.getTable()){
21         mapTablesToScan.put(``TS1.getTable()'', listTS);
22       }
23     }else{
24       foreach JO in mapJoinsToScan{
25         listTwoTS = mapJoinsToScan.get(JO);
26         TS1 = listTS.get(0);
27         TS2 = listTS.get(1);
28         listTSReadSameTables = seek_if_another_JO_in_mapJoinsToScan_has
29         _two_TS_that_read_the_same_pair_of_tables(TS1, TS2);
30
31         if (listTSReadSameTables.size > 0){
32           listTSReadSameTables.add(TS1);
33           listTSReadSameTables.add(TS2);
34           mapTablesToScan.put(``TS1.getTable()-TS2.getTable()'',
                 listTSReadSameTables);
35         }
36       }
37     }
38
39     if (mapTablesToScan.size() > 0){
40       dispatcher(mapTablesToScan);
41     }
42   }
43 }
```

Código A.1: Pseudocódigo del algoritmo de la entidad *GraphWalker* en la optimización de eliminar operadores redundantes entre ramas simples o compuestas en un *DAG*.

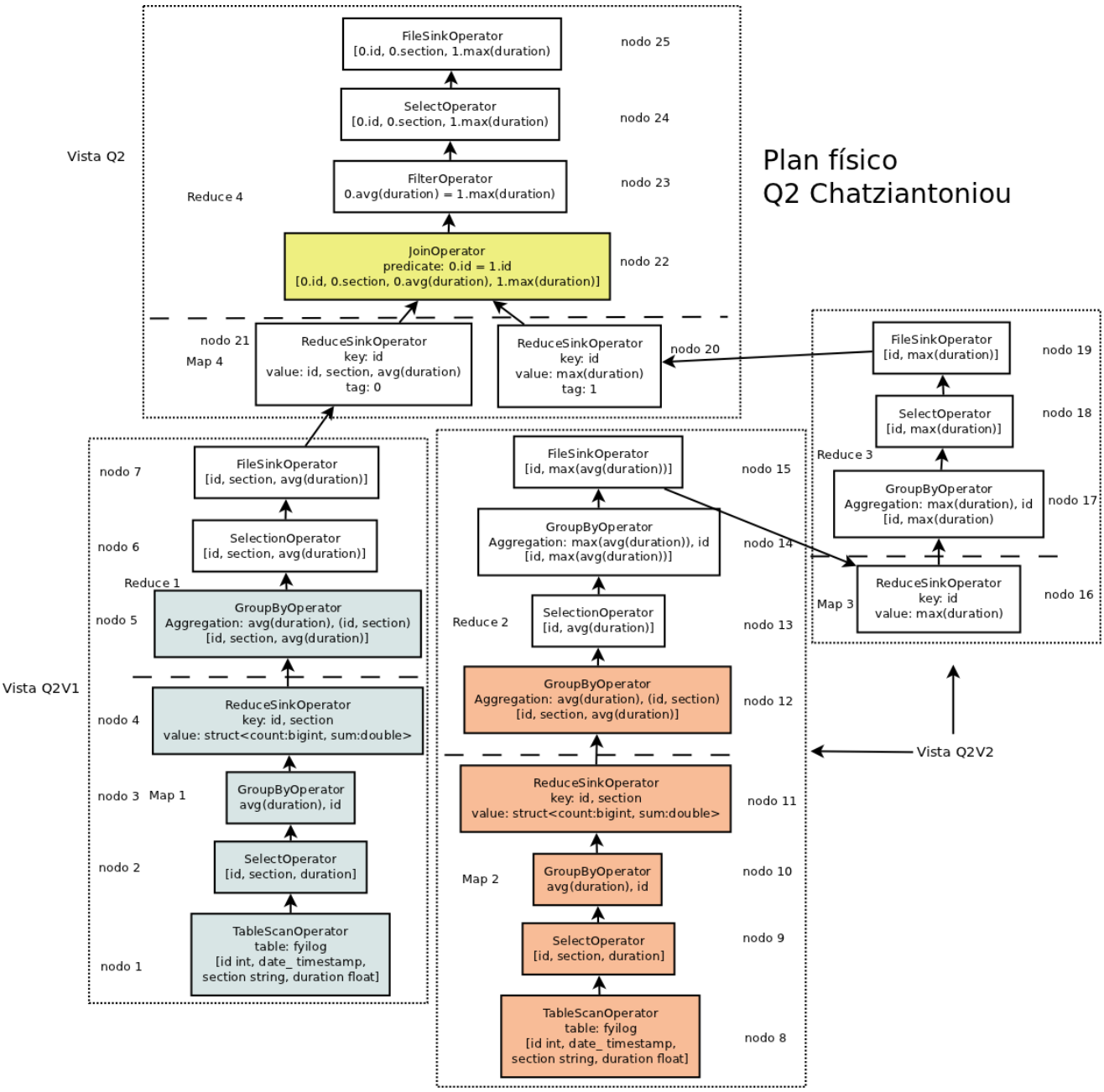


Figura A.1: *DAG* y plan físico de la consulta 2 de Chatziantoniou sin nuestra optimización.

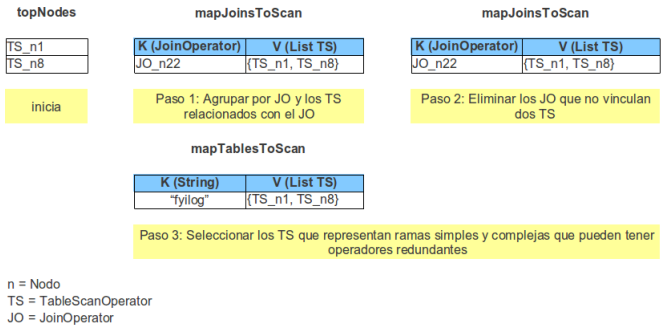
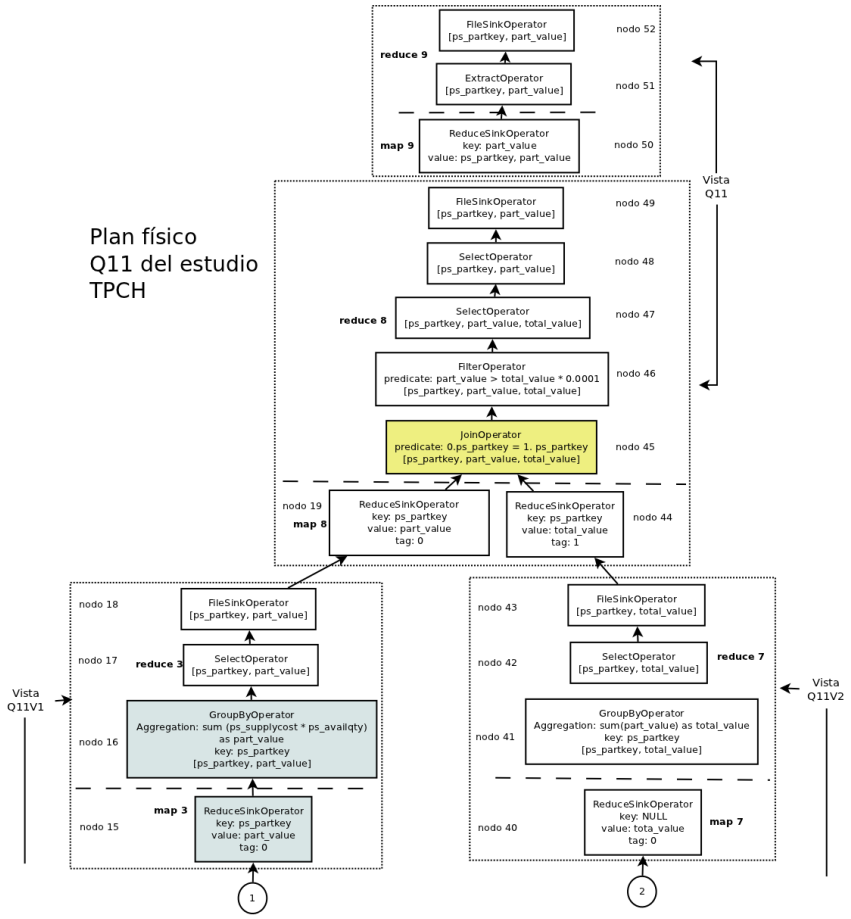


Figura A.2: Datos en las estructuras de datos que se utilizan en la entidad *GraphWalker* para la consulta 2 de Chatziantoniou.



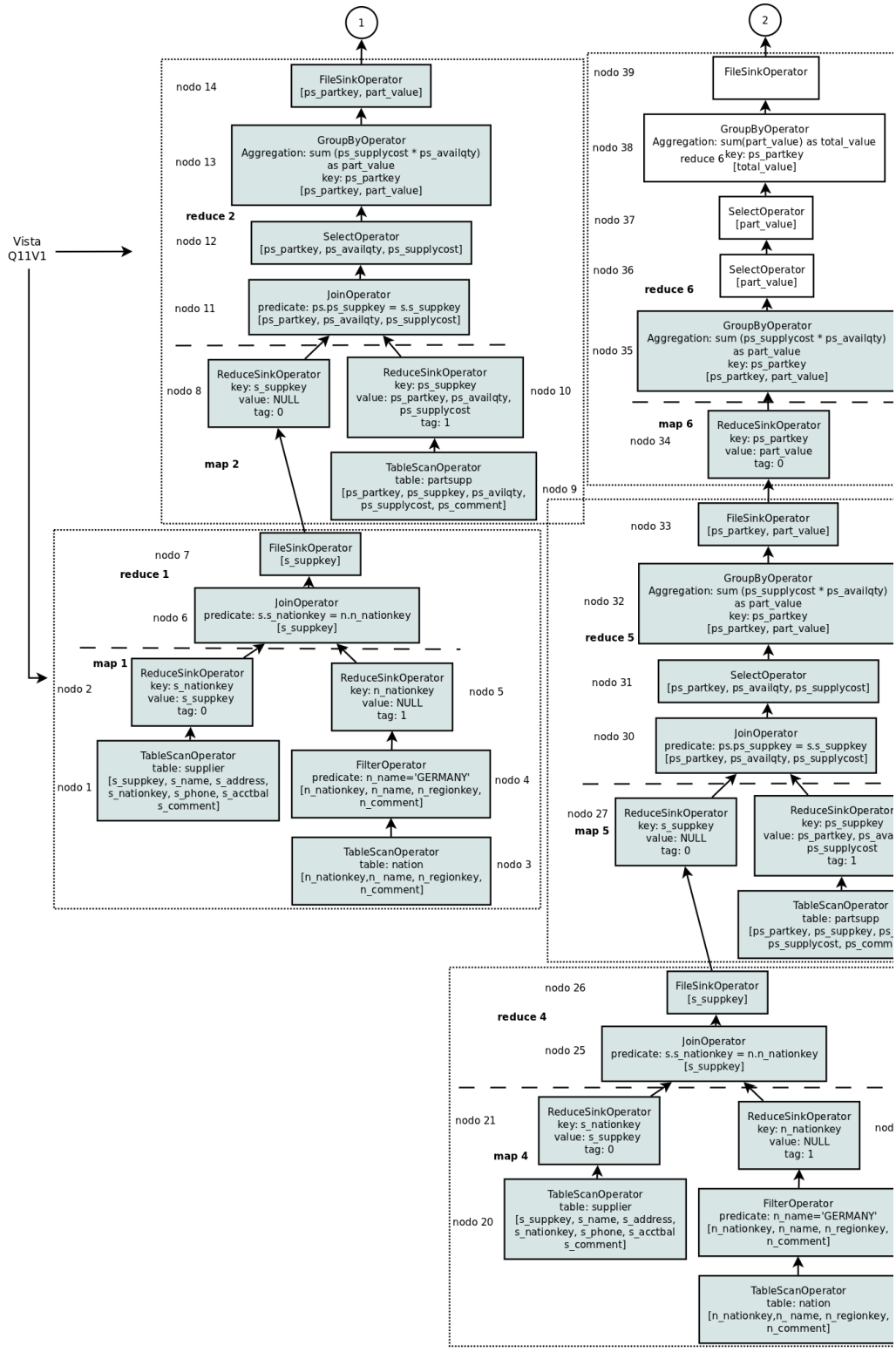


Figura A.3: *DAG* y plan físico de la consulta 11 de tpch sin nuestra optimización.

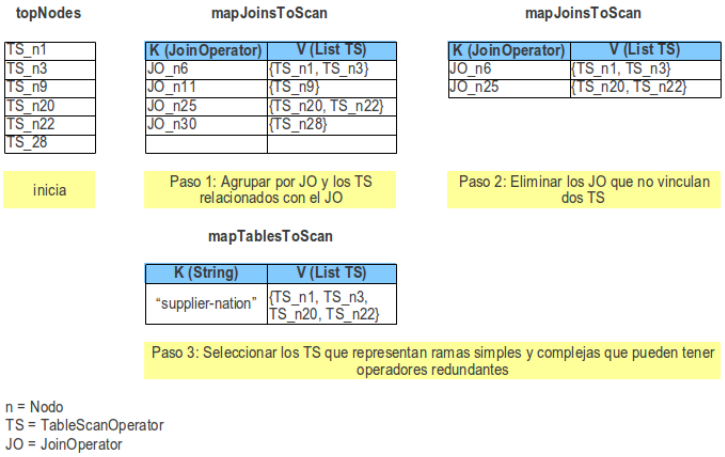


Figura A.4: Datos en las estructuras de datos que se utilizan en la entidad *GraphWalker* para la consulta 11 del estudio TPC-H.

topNodes con los operadores *TS* del nodo 1, 3, 9, 20, 22 y 28. Cuando se inicia el recorrido del *DAG* con el operador *TS* del nodo 1 se alcanza el operador *JO* del nodo 6, entonces se guarda como clave el *JO_n6* y como valor el *TS_n1* el mapa *mapJoinsToScan*. Posteriormente, se inicia otro recorrido del *DAG* con el operador *TS* del nodo 3 con cual se vuelve a alcanzar al operador *JO* del nodo 6, entonces se agrega el operador *TS_n3* a la lista de *TS* que se vincula con el *JO_n6* en el mapa *mapJoinsToScan*. Después, se inicia otro recorrido del *DAG* con el nodo 9 y esta vez se alcanza el operador *JO* del nodo 11, entonces se agrega otro registro al mapa *mapJoinsToScan* donde la clave es el *JO_n11* y el valor es el *TS_n9*. Lo mismo se realiza con los operadores *TS* de los nodos 20, 22 y 28. En el paso 1 de la figura A.4 se muestran los registros que se guardan en el map *mapJoinsToScan* en este paso para el *DAG* de la consulta 11 del estudio *TPC-H*.

2. Si se encontró al menos un *JO*, entonces se eliminan todos los operadores *JO* de *mapJoinsToScan* que no vinculan dos *TS*, esto se realiza debido a que los *JO* que no vinculan dos *TS* en un *DAG* se encuentran en *trabajos mapreduce* que dependen de otros *trabajos mapreduce* y en ese caso, eliminar operadores redundantes puede ocasionar problemas de dependencias. Esto corresponde al

paso 14 en el código A.1.

Por ejemplo, para la consulta 2 de Chatziantoniou se encontró un *JO* y ese *JO* vincula dos *TS* por lo que se mantiene el único registro en el mapa *mapJoinsToScan* como se muestra en el paso 2 de la figura A.2.

Para el ejemplo de la consulta 11 de Chatziantoniou se encontraron 4 *JOs* diferentes. Sin embargo, los operadores *JOs* de los nodos 11 y 30 no vinculan 2 *TS* y por lo tanto se eliminan. Obsérvese que ambos *JOs* se encuentran en los *trabajos mapreduce* 2 y 5 que dependen de los *trabajos mapreduce* 1 y 4 respectivamente. Si se eliminan operadores redundantes en los *trabajos mapreduce* 2 y 5 puede ocasionar problemas de dependencias, debido a que cómo se colapsan ambos *trabajos mapreduce* en uno solo, entonces ese único *trabajo mapreduce* va a depender de los *trabajos mapreduce 1 y 4* y cómo se tiene la incertidumbre de que los *trabajos mapreduce 1 y 4* pueden realizar diferentes cosas en diferentes momentos, puede darse el caso que el único *trabajo mapreduce* que se colapsó ya se tenga que ejecutar y aún no se haya ejecutado uno de los *trabajos mapreduce* que depende y entonces no tiene los datos correspondiente y falla. El paso 2 de la figura A.4 muestra los registros que guarda el mapa *mapJoinsToScan* para este paso.

3. Si solo se encontró un *JO*, entonces significa que se encontraron ramas simples, y es necesario verificar si lo dos TS que representan las dos ramas simples leen la misma tabla, si es el caso, entonces se almacenan ambos *TS* al mapa *mapTablesToScan* utilizando como clave un string que indica la tabla que se lee en ambos *TS* y como valor los dos *TS*. Esto corresponde a los pasos 16 al 22 en el código A.1.

   Por ejemplo, para la consulta 2 de Chatziantoniou solo se encontró un *JO*, entonces se comprueba que los dos *TS* que vincula lean la misma tabla, en este caso ambos *TS* leen la misma tabla *fyilog*. Por lo tanto, ambos *TS* se almacenan en el mapa *mapTablesToScan* como se observa en el paso 3 de la figura A.2.

4. Si se encontró más de un *JO*, entonces significa que se encontraron ramas compuestas. Cada *JO* y los dos *TS* que vincula representan una rama compuesta. Por cada *JO* que se encontró se busca otro *JO* en *mapJoinsToScan* que lea el mismo par de tablas. Si es el caso, entonces todos los pares TS que lean el mismo par de tablas se almacenan en un registro del mapa *mapTablesToScan* utilizando como clave un string que indica el par de tablas que lee cada par de *TS* y como valor todos los *TS* que leen el mismo par de tablas. Esto corresponde a los pasos 25 al 36 en el código A.1.

   Por ejemplo, para la consulta 11 del estudio TPC-H el mapa *mapJoinsToScan* almacena los *JOs* de los nodos 6 y 25, entonces se toman los dos *TS* del nodo 6 y se encuentra que leen las tablas *supplier y nation* y se busca que en el *JO_n25* vincule dos *TS* que lean las tablas *supplier y nation*, como es el caso, entonces los 4 *TS* que vinculan ambos *JOs* se guardan en el mapa *mapTablesToScan* como se observa en el paso 3 de la figura A.4.

5. Si al menos dos *TS* que representan a dos ramas diferentes y leen la misma tabla, entonces se manda a llamar al *Dispatcher* y se le envía el map *mapTablesToScan*. Esto corresponde a los pasos 39 al 41 en el código A.1.

   Para los ejemplos de la consulta 2 de Chatziantoniou y 11 del estudio *TCP-H* existe al menos dos *TS* que representan a ramas diferentes y leen la misma tabla, por lo tanto se manda a llamar al *Dispatcher* y se le pasa el mapa *mapTablesToScan*.

6. Si no se encontró un JO o no se encuentran *TS* que lean las mismas tablas en diferentes ramas, entonces no se aplica la optimización.

A.4. Dispatcher

```
1 dispatcher(mapTablesToScan){
2   foreach registro in mapTablesToScan{
3     listTS = mapTablesToScan.get(registro);
4     TS1 = listTS.get(0);
```

```
5       TS2 = listTS.get(1);
6       if (listTS.size() == 2){
7         listEqualNodes = walk_until_not_equal_node_or_found_JO(TS1, TS2);
8         processor(listEqualNodes);
9       }else{
10        for (i = 2; i < listTS.size(); i+=2){
11          TS3 = listTS.get(i);
12          TS4 = listTS.get(i+1);
13          listEqualNodes1 = walk_until_not_equal_node_or_found_first_JO(
                TS1, TS3);
14          listEqualNodes2 = walk_until_not_equal_node_or_found_first_JO(
                TS2, TS4);
15          if (listEqualNodes1.get(0) != JO || listEqualNodes2 != JO){
16            processor(listEqualNodes1);
17            processor(listEqualNodes2);
18          }else{
19            JO1 = listEqualNodes1.get(0);
20            JO2 = listEqualNodes2.get(1);
21            listEqualNodes = walk_until_not_equal_node_or_found_n_JO(JO1,
                  JO2);
22            processor(listEqualNodes);
23          }
24        }
25        if (TS1.getName() == TS2.getName()){ realizar pasos 7 y 8.}
26      }
27    }
28  }
29  function walk_until_not_equal_node_or_found_n_JO(JO1, JO2){
30    listEqualNodes = walk_until_not_equal_node_or_found_JO(JO1, JO2);
31    if (listEqualNodes.get(0) == JO){
32      if (listEqualNodes.get(0) == listEqualNodes.get(1)){
33        return listEqualNodes;
34      }else{
35        band = branch_has_all_equals_nodes(JO1, JO2);
36        if (band) {
37          JO1 = listEqualNodes.get(0);
38          JO2 = listEqualNodes.get(1);
39          return walk_until_not_equal_node_of_fount_n_JO(JO1, JO2);
40        }else{
41          listEqualNodes.get(0) = JO1;
42          listEqualNodes.get(1) = JO2;
43          return listEqualNodes;
44        }
45      }
46    }else{
47      listEqualNodes.get(0) = JO1;
48      listEqualNodes.get(1) = JO2;
49      return listEqualNodes;
50    }
51  }
```

Código A.2: Pseudocódigo del algoritmo de la entidad *Dispatcher* en la optimización de eliminar operadores redundantes entre ramas simples o compuestas en un *DAG*.

En nuestra optimización, la entidad *Dispatcher* encuentra los operadores redundantes entre ramas simples o complejas que leen las mismas tablas, y manda a llamar a la

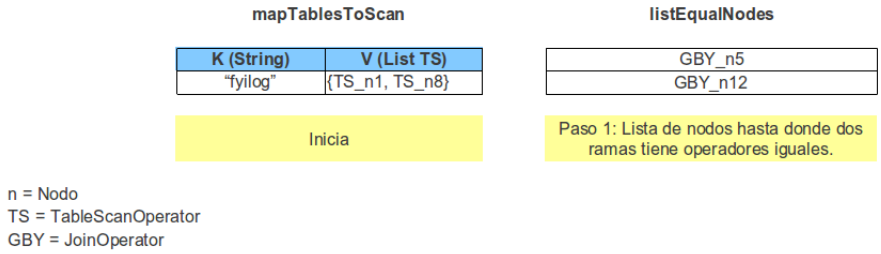


Figura A.5: Datos en las estructuras de datos que se utilizan en la entidad *Dispatcher* para la consulta 2 de Chatziantoniou.

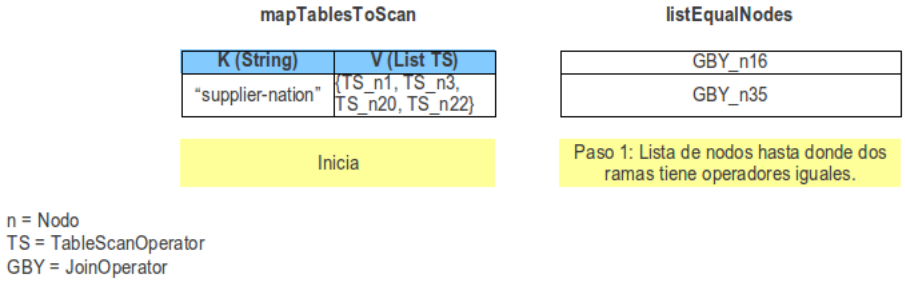


Figura A.6: Datos en las estructuras de datos que se utilizan en la entidad *Dispatcher* para la consulta 11 del estudio de mercado TPC-H.

entidad *Processor* con los últimos nodos del *DAG* hasta donde encuentra operadores redundantes. La entidad *Dispatcher* recibe de la entidad *GraphWalker* el mapa *mapTablesToScan* que por cada registro almacena una lista de *TS* como valor. Si la lista de *TS* solo tiene 2 *TS*, entonces cada *TS* representa una rama simple en un *DAG* que lee la misma tabla. Si la lista de *TS* tiene más de 2 *TS*, entonces cada par de *TS* representan una rama compuesta en un *DAG* que leen el mismo par de *tablas*.

Dos nodos son iguales en un *DAG* si:

- Ambos nodos tienen el mismo operador (TS, FilterOperator, JO, entre otros).
- Las columnas de salida son las mismas en ambos nodos. Ver apéndice A para más detalle.
- Si son nodos con operadores *FilterOperator*, entonces la condicional de ambos operadores es la misma.
- Si son nodos con operadores *RS*, entonces ambos operadores deben de configurar las mismas columnas como clave, y la mismas columnas como valor en los pares

$< clave, valor >$ que emiten. Ver apéndice A para más detalle.

El código A.2 muestra el pseudoalgoritmo que se utiliza en la entidad *Dispatcher*. Obsérvese que:

1. Por cada registro de *mapTablesToScan* se obtienen los primeros 2 *TS*. A los que nombramos *TS1 y TS2*.

   Si la lista solo tiene 2 *TS*, entonces se manda a llamar a la función *walk_until_not_equal_node_or_found_JO(TS1, TS2)*. Esta función realiza dos recorridos del *DAG* al mismo tiempo. Un recorrido con el *TS1* y otro con *TS2* (ambos *TS* representan ramas diferentes). Durante ambos recorridos se van comparando los dos nodos que se alcanzan en ambas ramas y si son iguales, entonces se continúan ambos recorridos hasta encontrar nodos que no son iguales o encontrar un *JO*. En ese momento, se regresa una lista que contiene los dos nodos hasta donde ambos recorridos son iguales y se guardan en la lista *listEqualNodes*. Entonces se manda a llamar a la entidad *Processor* con dicha lista. Esto corresponde a los pasos del 3 al 9 en el código A.2.

   Por ejemplo, en la consulta 2 de Chatziantoniou el mapa *mapTablesToScan* almacena un registro con 2 *TS*. Entonces se inician dos recorridos del *DAG* al mismo tiempo, un recorrido con el operador *TS* del nodo 1 y otro recorrido con el operador *TS* del nodo 8. Con el operador *TS* del nodo 1 se alcanza el operador *SelectOperator (SEL)* del nodo 2 y con el operador *TS* del nodo 8 se alcanza el operador *SEL* del nodo 9, ambos nodos tienen al mismo operador y además emiten las mismas columnas que se encierran entre corchetes en cada nodo, por lo tanto, ambos operadores son iguales. Entonces, se continuan ambos recorridos hasta el operador *SEL* del nodo 6 y el operador *SEL* del nodo 13 respectivamente donde ambos nodos tienen al mismo operador, sin embargo, no emiten las mismas columnas (encerradas entre corchetes). Por lo tanto, los últimos operadores iguales en ambas ramas son los que están en los **nodos 5 y 12**, estos nodos se guardan en la lista ***listEqualNodes*** y se manda a llamar al

Processor al cuál se le pasa dicha lista. El paso 1 de la figura A.5 muestra los datos que almacenan en la lista *listEqualNodes* cuando se aplica este paso a la consulta 2 de Chatziantoniou.

2. Si en cada registro de *mapTablesToScan* se obtiene como valor una lista que tiene más de 2 *TS*, entonces se obtiene el primer par de *TS* al que nombramos *TS1* y *TS2*, después se obtiene otro par de *TS* a los que nombramos *TS3* y *TS4*. En este caso, los operadores *TS1 y TS2* representan una rama compuesta, y los operadores *TS3 y TS4* representan otra rama compuesta. Las tablas *TS1* y *TS3* leen la misma tabla, mientras que las tablas *TS2* y *TS4* leen la misma tabla. Entonces se manda a llamar la función *walk_until_not_equal_node_or_found_first_JO()* dos veces, una vez con los operadores *TS1 y TS3*, y otra vez con los operadores *TS2 y TS4*. Esta función realiza dos recorridos del *DAG* al mismo tiempo. Un recorrido con el primer *TS* que recibe como parámetro, y otro recorrido con el segundo *TS* que recibe como parámetro. Durante ambos recorridos se van comparando los dos nodos que se alcanzan en ambas ramas y si son iguales, enDispatchertonces se continúan ambos recorridos hasta encontrar nodos que no son iguales o encontrar un *JO*. Al final, por cada vez que se llama se regresa una lista con los dos nodos hasta donde ambos recorridos son iguales, esta lista se guarda en *listEqualNodes1* para los recorridos con *TS1 y TS3*, y en *listEqualNodes2* para los recorridos con *TS2 y TS4*. Esto corresponde a los pasos del 3 al 5, y del 11 al 14 en el código A.2.

   Por ejemplo, en la consulta 11 del estudio TPC-H el único registro de *mapTablesToScan* tiene 4 *TS* (ver paso de inicio en la figura A.6), entonces se obtiene el primer par de *TS* que son los nodos 1 y 3, después se obtiene el otro par de *TS* que son los nodos 20 y 22. Entonces se inicia un par de recorridos del *DAG* al mismo tiempo con los operadores *TS_n1* y *TS_n20*. Con el operador *TS_n1* se alcanza al operador *ReduceSinkOperator (RS)* del nodo 2, con el operador *TS_n20* se alcanza al operador *RS* del nodo 21, ambos operadores

son iguales porqué configuran las mismas columnas como clave y valor de los pares intermedio $< clave, valor >$ de las *tareas map* 1 y 4 respectivamente, entonces se continúa ambos recorridos y se encuentra los operadores *JOs* de los nodos 6 y 25 y son iguales, en este momento se detienen ambos recorridos y los operadores *JOs* se guardan en la lista *listEqualNodes1*. Posteriormente, se inicia otro par de recorridos con los operadores *TS_n3* y *TS_n22*, con el operador *TS_n3* se alcanza al operador *FilterOperator (FIL)* del nodo 4 y con el operador *TS_n22* se alcanza al operador *FIL* del nodo 23, ambos operadores son los mismo porqué aparte de emitir las mismas columnas al siguiente nodo que se muestra entre corchetes, tienen la misma condicional. Entonces, se continúan ambos recorrido hasta encontrar nuevamente los operadores *JOs* del nodo 6 y 25, en ese momento se detienen ambos recorridos y se guarda ambos *JOs* en la lista *listEqualNodes2*.

3. Si en los recorridos con *TS1 y TS3* o con *TS2 y TS4* no se alcanza un operador *JO*, entonces se manda a llamar a la entidad *Processor* dos veces, una vez con las lista *listEqualNodes1* que contiene los nodos hasta donde son similares las ramas complejas en el recorrido con *TS1 y TS3*, y otra vez con la lista *listEqualNodes2* que contiene los nodos hasta donde son similares las ramas complejas en el recorrido *TS2 y TS4*. Esto corresponde a los pasos 15 y 18 en el código A.2.

4. En caso contrario, si en los recorridos con *TS1 y TS3* y con *TS2 y TS4* se alcanza un operador *JO*, entonces se continúa el recorrido con la función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)*. Esta función se observa en los pasos del 29 al 51 en el código A.2.

   Por ejemplo, en la consulta 11 del estudio TPC-H se alcanzaron los operadores *JOs* de los nodos 6 y 25. Por lo tanto se continúa el recorrido del *DAG*.

5. La función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)* primero utiliza una función llamada *walk_until_not_equal_node_or_found_JO(JO1, JO2)* que realiza dos recorridos: uno con el JO1 y otro con el JO2 (recuerde que el JO1

corresponde a una rama compuesta del *DAG* y el *JO2* corresponde a otra rama compuesta del *DAG*). Durante ambos recorridos se van comparando los nodos para ver si son iguales, en el momento que no lo son o se alcanzan otros par de nodos *JO*, entonces la función *walk_until_not_equal_node_or_found_JO(JO1, JO2)* regresa una lista de dos elementos que contiene los dos nodos hasta son iguales ambos recorridos, esta lista se guarda en *listEqualNodes* en el paso 30 del código A.2.

Si en ambos recorridos no se alcanzó un nuevo par de *JO*, entonces se confirma si el siguiente *JO* en ambos recorridos es el mismo, si lo es entonces la función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)* regresa una lista que contiene los últimos nodos en los que ambas ramas son iguales, en caso contrario la función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)* regresa una lista que contiene los operadores *JO* a partir de los cuáles se inició el recorrido, esto se debe a que los operadores que le siguen a dichos operadores *JOs* pertenecen a otros *trabajos mapreduce* que pueden depender del resultado de otras ramas que pueden tener otros *trabajos mapreduce*, y por lo tanto, si todos los operadores de los *trabajo mapreduce* implicados no son iguales y se eliminan operadores en dichos *trabajos mapreduce* puede haber problemas de dependencia (esto corresponde a los pasos 46 del 50 en el código A.2).

Por ejemplo, en la consulta 11 del estudio TPC-H se inicia dos recorridos del *DAG* al mismo tiempo con los operadores *JO_n6* y *JO_n25*. Con el operador *JO_n6* se alcanza al operador *RS* del nodo 8, no es el operador *FileSinkOperator (FS)* debido a que ese operador se agrega cuando el *DAG* se transforma a plan físico, con el operador *JO_n25* se alcanza al operador *RS* del nodo 27. Ambos nodos son iguales, entonces se continuan ambos recorridos y se alcanzan los operadores *JOs* de los nodos 11 y 30, ambos nodos son iguales.

Por otra parte, si en ambas ramas se alcanzó otro par de operadores *JO*, entonces se evalúa si en ambos recorridos se alcanzó el mismo *JO*, si es

el caso, entonces se regresa la lista *listEqualNodes* del paso 30 donde los dos registros apuntan al mismo *JO*. En caso contrario, si se alcanza un par de *JO* que no son el mismo nodo, pero son iguales, entonces se utiliza la función *branch_has_all_equals_nodes(JO1, JO2)* para recorrer la otra rama de los operadores *JO* que se alcanzaron. Este recorrido se realiza hacia abajo del *DAG* y durante el recorrido se van comparando los nodos de ambos recorridos y si son iguales hasta el último operador de dichas ramas, entonces se regresa true, en caso contrario se regresa false. Si la otra rama de ambos *JO* que se alcanzaron son iguales, entonces se vuelve a llamar la función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)* de manera recursiva para seguir recorriendo un *DAG* (pasos del 36 al 39 en el código A.2). En caso contrario, si la otra rama de ambos *JO* que se alcanzaron no son iguales, entonces la función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)* regresa en una lista los operadores *JO* anteriores a partir de los cuáles se alcanzaron los nuevos *JOs*, esto debido a la explicación que se dió en el parráfo anterior (pasos del 40 al 44 en el código A.2).

Por ejemplo, en la consulta 11 del estudio TPC-H se alcanzaron los *JOs* de los nodos 11 y 30, entonces se verifica si la rama derecha de ambos operadores son iguales recorriendo al *DAG* hacia atrás en dichas ramas. Para el operador *JO_n11* se alcanza al operador *RS_n10*, y para el operador *JO_n30* se alcanza el operador *RS_n29*, ambos operadores son iguales, entonces se continúan ambos recorridos y se alcanzan los operadores *TSs* de los nodos 9 y 28 respectivamente donde ambos operadores son iguales, como ya no hay más nodos que visitar, entonces ambas ramas son iguales y se continúa el recorrido del *DAG* hacia arriba con los operadores *JO_n11* y *JO_n30* y se vuelve a repetir a partir del paso 5, y en este caso los recorridos de ambas ramas se detienen hasta los operadores *SelectOperator (SEL)* de los nodos 27 y 36 que no son iguales, como el siguiente *JO* es el mismo en ambos recorridos, entonces la función *walk_until_not_equal_node_or_found_n_JO(JO1, JO2)* regresa la lista

```
1  Processor(listEqualNodes){
2    N1 = listNodes.get(0);
3    N2 = listNodes.get(1);
4
5    children_n2 = N2.getChildren();
6
7    N1.addChildrenOperator(children_n2);
8    children_n2.changeParentOperator(N1);
9
10   erase_all_TS_of_redundant_branch();
11 }
```

Código A.3: Pseudocódigo del algoritmo de la entidad *Processor* en la optimización de eliminar operadores redundantes entre ramas simples o compuestas en un *DAG*

listEqualNodes con los nodos ***GroupByOperators (GBY)* 26 y 35** en el paso 5. Posteriormente se manda a llamar al *Processor* con dichos nodos.

6. Por último se evalua si los primeros dos *TSs*(*TS1 y TS2*) que representan una rama compuesta leen la misma tabla, si lo hacen, entonces se realiza lo mismo que se describió en el paso 1 (Pasos 25 en el código A.2).

   Por ejemplo, en la consulta 11 del estudio TPC-H el primer par de nodos *TS* del registro del mapa *mapTablesToScan* (ver figura A.6) son los nodos *TS_n1* y *TS_n3*, y ambos nodos no leen la misma tabla, por lo tanto no se realiza este paso.

A.5. Processor

En nuestra optimización, la entidad *Processor* elimina los operadores redundantes entre ramas que se encontraron en la entidad *Dispatcher*. Esta entidad recibe del *Dispatcher* la lista *listEqualNodes* que contiene los dos últimos nodos en los que dos ramas de un *DAG* tienen operadores redundantes.

Para eliminar los operadores redundantes entre dos ramas de un *DAG* solo es necesario eliminar en el *DAG* las referencias al último nodo de la secuencia de operadores redundantes y los operadores *TSs* de la rama redundante.

El código A.3 muestra el pseudoalgoritmo que se utiliza en la entidad *Processor*.

Obsérvese que:

1. Se obtiene de la lista *listEqualNodes* los dos nodos hasta donde son iguales ambas ramas. Estos nodos se les nombra N1 y N2. Entonces para eliminar la referencia al último nodo de la secuencia de operadores redundantes, se obtiene el nodo hijo del nodo N2 y se le asigna como un nuevo nodo hijo (nodo superior) al nodo N1, y al nodo hijo del nodo N2 se le asigna como nodo padre (nodo de abajo) el nodo hijo 1. Pasos del 2 al 8 en el código A.3.

   Por ejemplo, en la consulta 2 de Chatiziantoniou se recibe la lista *listEqualNodes* del paso 1 de la figura A.5. Entonces el *N1* es el operador *GroupByOperator (GBY)* del nodo 5 y el *N2* es el operador *GBY* del nodo 12. Por lo tanto, el nodo 5 tendrá como hijos a los nodos 6 y 13, y el nodo 13 tendrá como padre al nodo 5 (ver figura A.7).

   Por ejemplo, en la consulta 11 del estudio TPC-H se recibe la lista *listEqualNodes* del paso 1 de la figura A.6. Entonces el *N1* es el operador *GBY_n16* y el *N2* es el operador *GBY_n35*. Por lo tanto, el nodo 16 tendrá como hijos a los nodos 17 y 36, y el nodo 36 tendrá como padre al nodo 16 (ver figura A.8).

2. Después se elimina de la lista *topNodes* los operadores *TS* que pertenecen a la rama redundante que se elimina. Paso 10 en el código A.3.

   Por ejemplo, en la consulta 2 de Chatziantoniou, el operador *TS* que se elimina es el nodo 8, con esto se termina de eliminar los operadores redundantes y se forma el *DAG* de la figura A.7. Obsérvese que cuando el *DAG* se transforma a plan físico se ha eliminado por completo las *tareas map 2* y las *tareas reduce 2* se han agregado a las *tareas reduce 1* con respecto al plan físico de la figura A.1. Con esto se elimina una lectura de la tabla *fyilog* innecesaria, varios procesamientos innecesarios de las *tareas map 2* en cada nodo de un clúster, y se elimina un conjunto de datos que se envíaban por la red de manera innecesaria.

Por ejemplo, en la consulta 11 del estudio TPC-H los operadores *TS* que se elimina son los de los nodos 20, 22 y 28, y el *DAG* que se forma se observa en la figura A.8. Cuando ese *DAG* se transforma a un plan físico obsérvese que se han eliminado por completo los *trabajos mapreduce 4 y 5* con respecto al plan físico de la figura A.3, y además se ha eliminado parte del *trabajo mapreduce 6*. Con esto se eliminan tres lecturas de tablas innecesarios, procesamiento y envío de datos innecesarios en los *trabajos mapreduce 4, 5 y 6*.

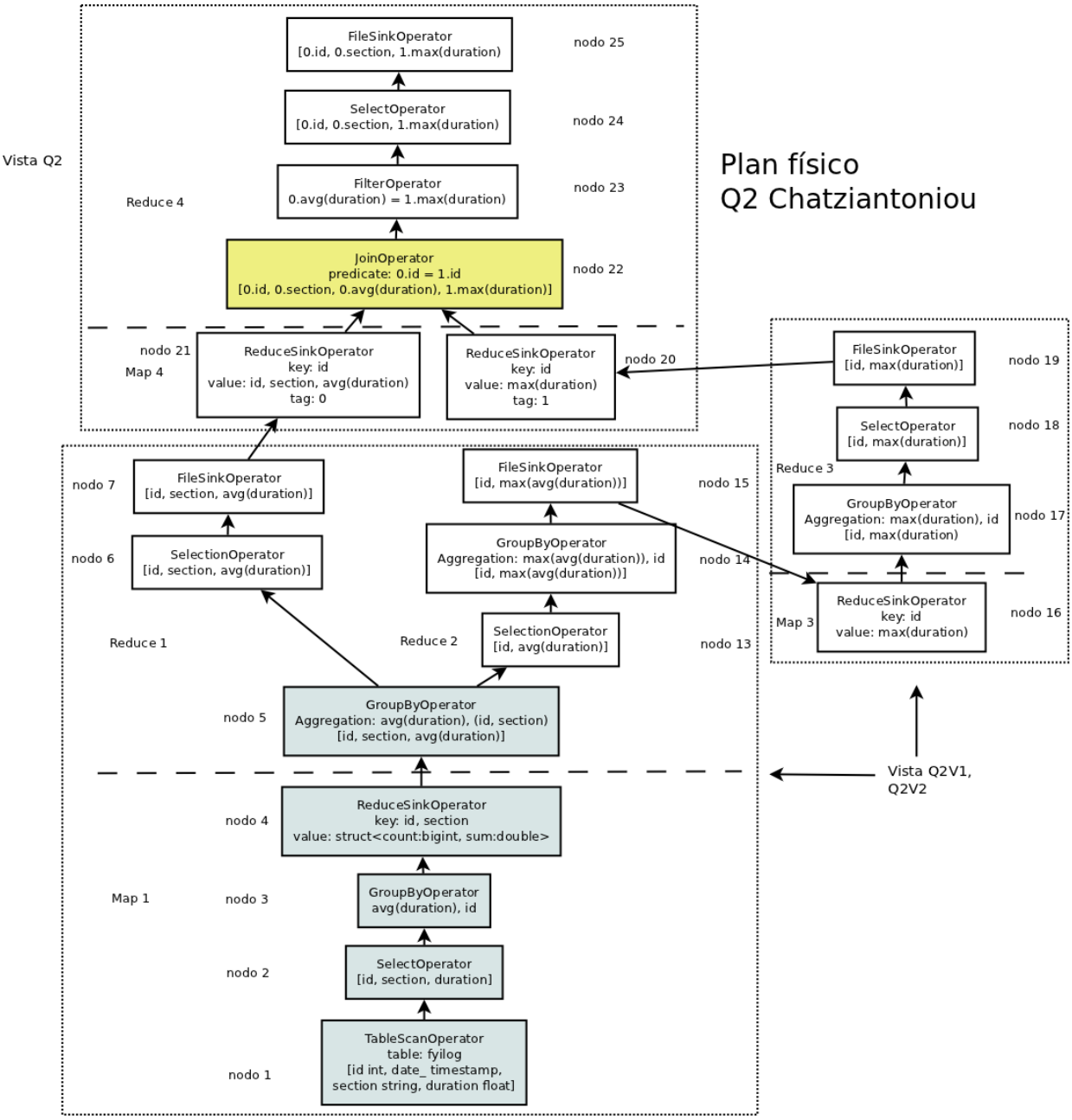
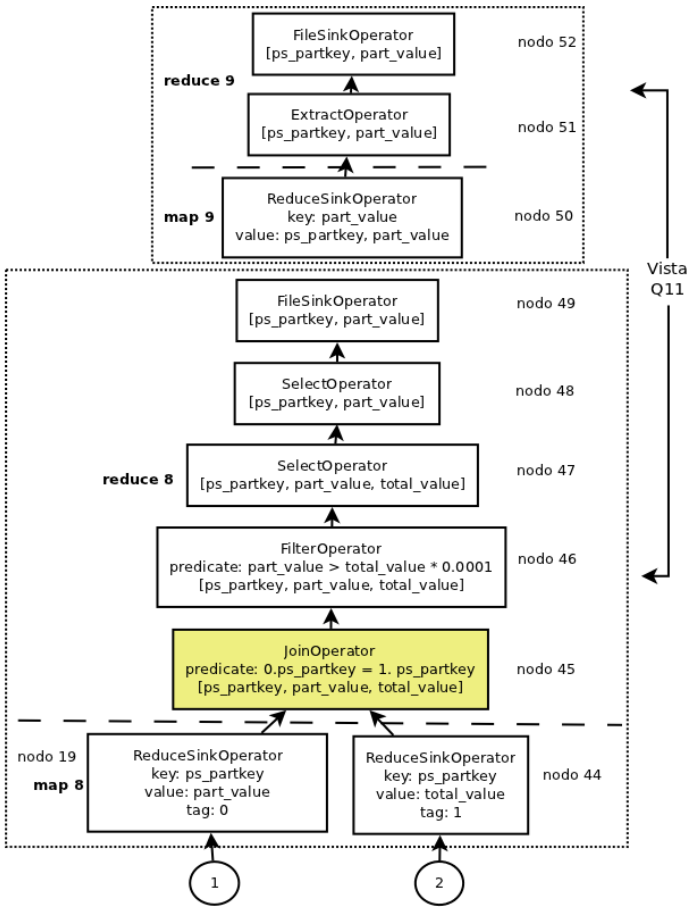


Figura A.7: *DAG* y plan físico de la consulta 2 de Chatziantoniou con nuestra optimización.

Plan físico
Q11 del estudio
TPCH



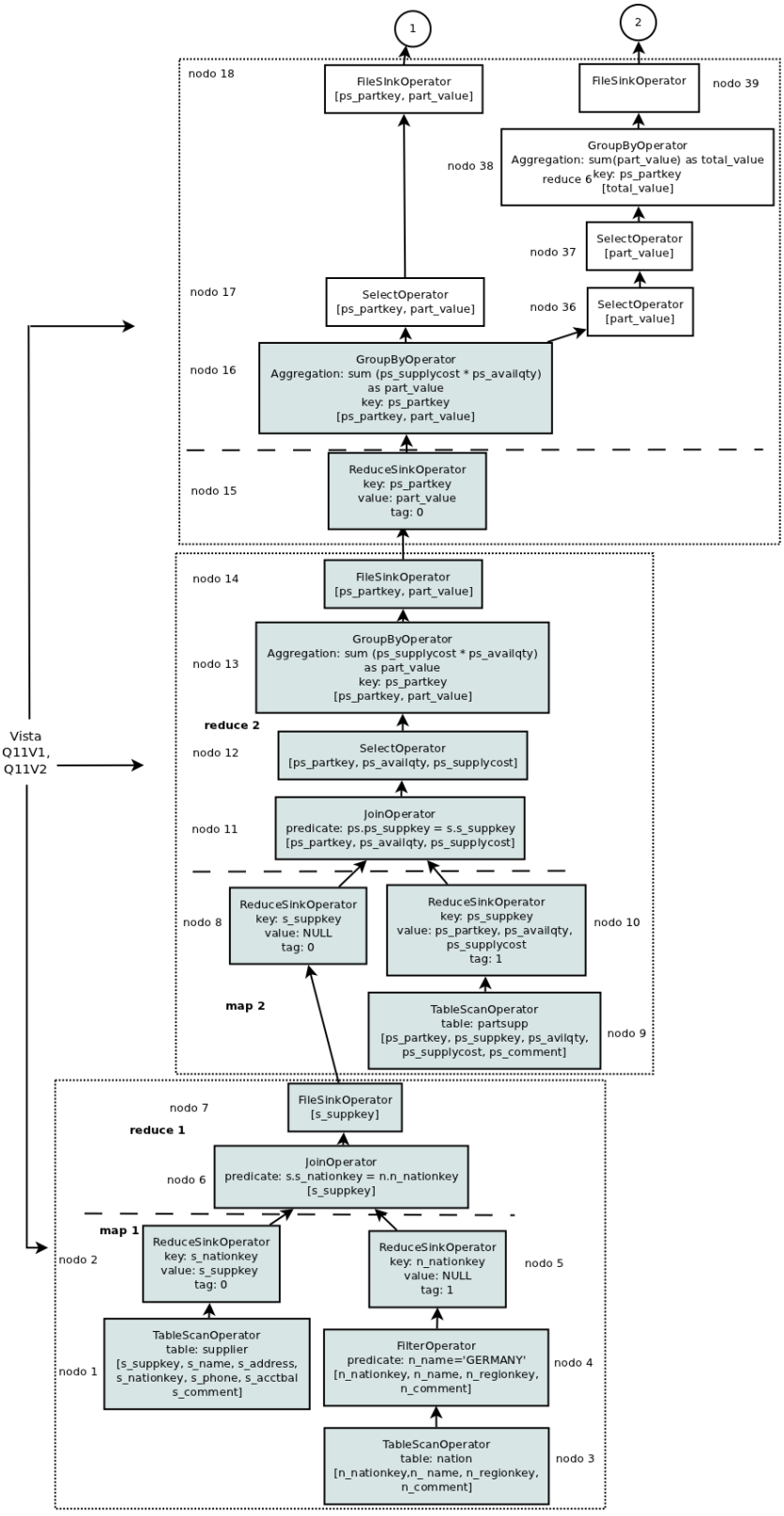


Figura A.8: *DAG* y plan físico de la consulta 11 del estudio de mercado TPC-H con nuestra optimización.

Apéndice B

Arquitectura del optimizador de sentencias *HiveQL*

Hive es un *datawarehouse* que opera sobre *Hadoop*. *Hive* al igual que *Hadoop* está implementado en *Java*. Este apéndice explica cómo se encuentra organizado el optimizador de sentencias *HiveQL* dentro del proyecto *Hive*, cuáles son las interfaces básicas del optimizador, cuáles son las clases que implementan las optimizaciones actuales de *Hive*, donde se encuentran las clases y cómo implementar una nueva optimización.

Antes de continuar, conviene definir los siguientes conceptos:

- Una *interfaz* es una colección de métodos y variables que solo definen lo que se debe de hacer y no cómo se debe de hacer.
- Una *clase* es un conjunto de métodos y variables que indican el comportamiento de un objeto.
- Un *objeto* es una instancia de una clase, que en tiempo de ejecución realiza las tareas de un programa.
- Un *paquete* es un contenedor de clases o interfaces que permite agrupar las distintas partes de un sistema que tienen una funcionalidad en común.

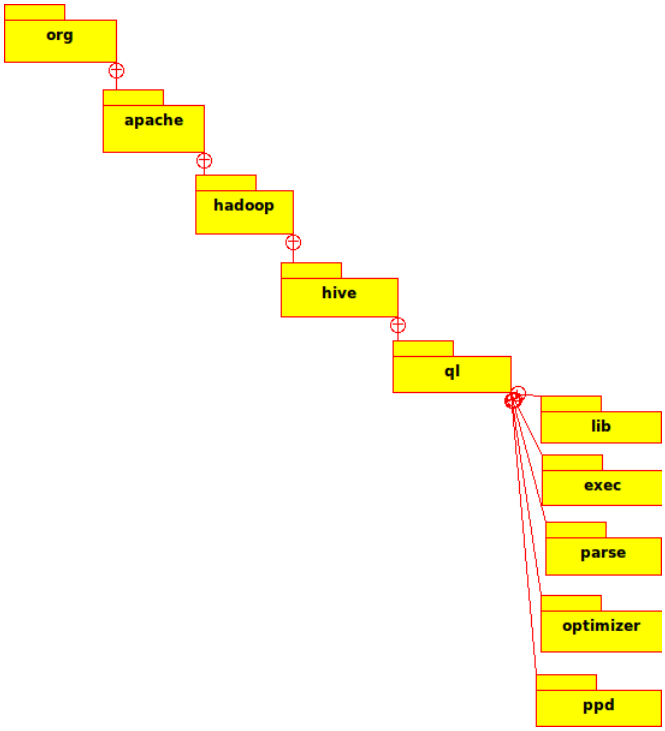


Figura B.1: Diagrama de paquetes del optimizador de consultas *HiveQL*.

La figura B.1 muestra un diagrama de paquetes que representa la manera en que se organizan las optimizaciones de sentencias *HiveQL* en Hive. Cada carpeta representa un paquete. El paquete *org* contiene al paquete *apache* que contiene al paquete *hadoop* y así sucesivamente. Estos paquetes se encuentran dentro del directorio *ql/src/java* en el proyecto *Hive*. El paquete *lib* contiene las interfaces y clases bases del optimizador de consultas *HiveQL* (*GraphWalker, Dispatcher, Rule, Node, Processor*). El paquete *exec* contiene las implementaciones de los operadores de *Hive* (*GroupByOperator*, *FilterOperator*, *ReduceSinkOperator*, *JoinOperator*, etcétera). El paquete *parse* contiene clases que están involucradas en las fases de compilación de una consulta *HiveQL* a *trabajos mapreduce*. Los paquetes *optimize y ppd* contiene las optimizaciones actuales de *Hive*.

La figura B.2 muestra las interfaces y clases que representan las entidades base de cualquier optimización de consultas *HiveQL*: *Node, Rule, GrahpWalker, Dispatcher, NodeProcessor (Processor).* Estas entidades se encuentran dentro del paquete *org.apache.hadoop.hive.ql.lib*.

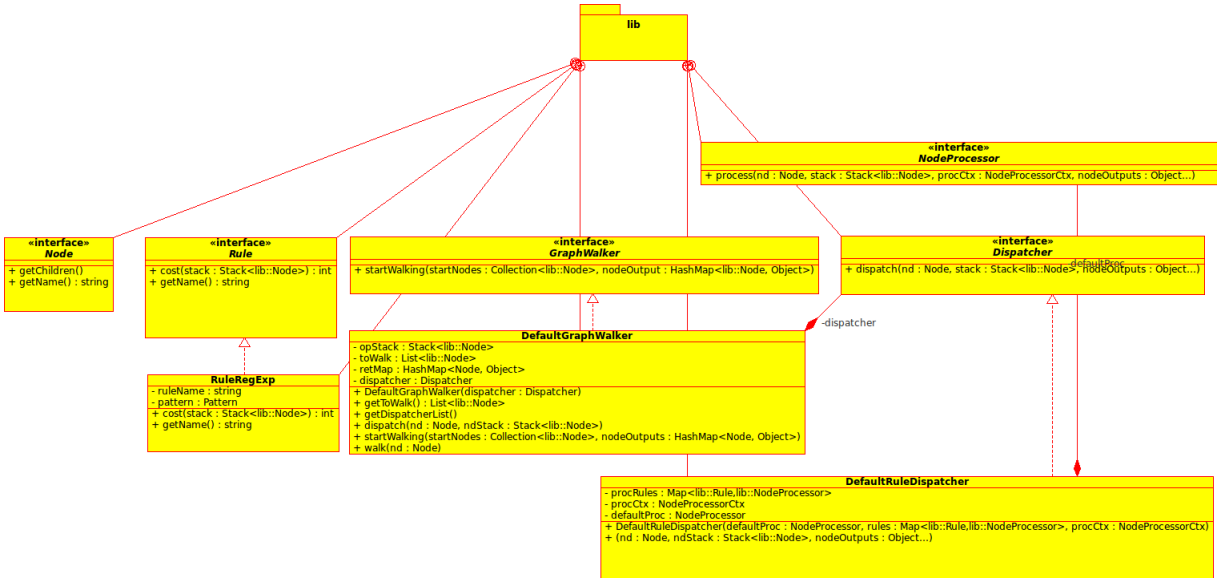


Figura B.2: Diagrama de clases que muestra las interfaces y clases base de cualquier optimización de consultas *HiveQL* en *Hive*.

La interfaz *Node* es una entidad que representa un nodo en un *DAG*, esta interfaz se implementa por cada uno de los operadores de *Hive* (*GroupByOperator, JoinOperator*, etcétera). Contiene varios métodos dos de ellos se muestran en la figura B.2. El método *getChildren()* obtiene los nodos hijos de un nodo (operador), el método *getName()* devuelve el nombre del operador que contiene el nodo.

La interfaz *Rule* es una entidad que representa las reglas a encontrar en una optimización. La clase *RulRegExp* (ver figura B.2) implementa a la interfaz *Rule*, esta implementación permite especificar un conjunto de reglas a través de expresiones regulares (*pattern*). En el método *cost(...)* se comprueba si una o más reglas se cumplen, si dos o más reglas se cumplen, entonces implementa un algoritmo basado en costos que determina la regla a aplicar.

La interfaz *GraphWalker* es la entidad con la que se recorre un *DAG*, proporciona un método llamado *startWalking(...)* que recibe dos parámetros, el parámetro *startNodes* son los nodos con los que se inician los recorridos de un *DAG* (nodos *TS*) y el parámetro *nodeOutput* permite regresar un conjunto nodos que se pueden buscar en un *DAG*. El método *starWalking(...)* se encarga de recorrer un *DAG*. La clase *DefaultGraphWalker* es una implementación de la interfaz *GraphWalker*, permite

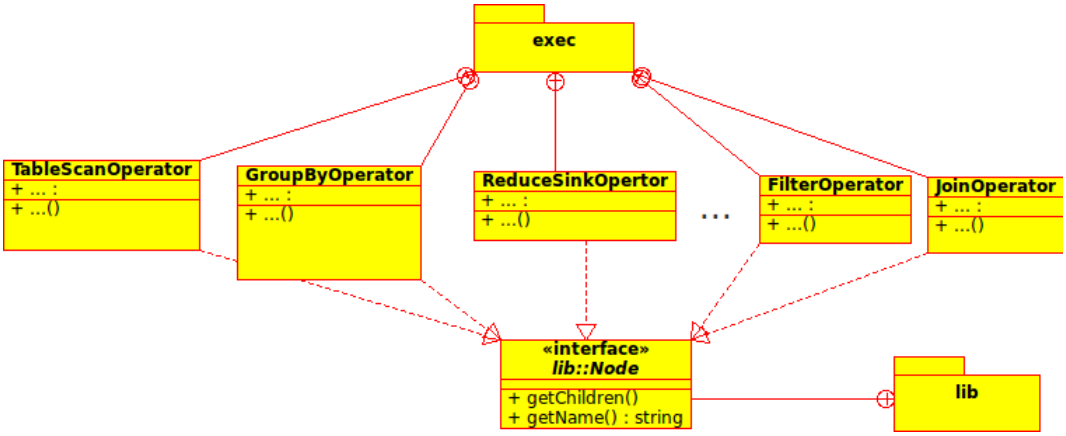


Figura B.3: Diagrama de clases que muestra las clases que representan a los operadores *Hive*.

recorrer un *DAG* utilizando un algoritmo de Búsqueda en profundidad (en inglés *Depth First Search*, DFS). El rombo hacia la interfaz *Dispatcher* indica que contiene una variable que apunta a una clase que implementa la interfaz *Dispatcher*, esto se debe a que cada vez que visita un nodo en un *DAG* manda a llamar al *Dispatcher* asignado para verificar si alguna regla se cumple.

La interfaz *Dispatcher* es la entidad que relaciona un conjunto de reglas con lo que se va a realizar para optimizar un *DAG* (Processor). Contiene un método llamado *dispatch(...)* que se encarga de verificar si alguna regla que se especificó en *RuleRegExpr* se cumple, y si se cumple manda a llamar a la entidad *Processor*. La clase *DefaultRuleDispatcher* es una implementación de la interfaz *Dispatcher*, permite verificar si alguna regla se cumple, si es el caso entonces manda a llamar a un Processor para optimizar un *DAG*. El rombo hacia la interfaz *NodeProcessor* indica que tiene variables que se asignan a clases que implementan la interfaz *NodeProcessor* que representa una entidad *Processor*.

La interfaz *NodesProcessor* representa a una entidad *Processor* y permite en el método *process(...)* especificar lo que se desea hacer para optimizar un *DAG*.

La figura B.3 muestra las clases que implementan los operadores *Hive (GroupByOperator, JoinOperator, ReduceSinkOperator, etcétera)*. Estas clases se encuentran en el paquete *org.apache.hadoop.hive.ql.exec*.

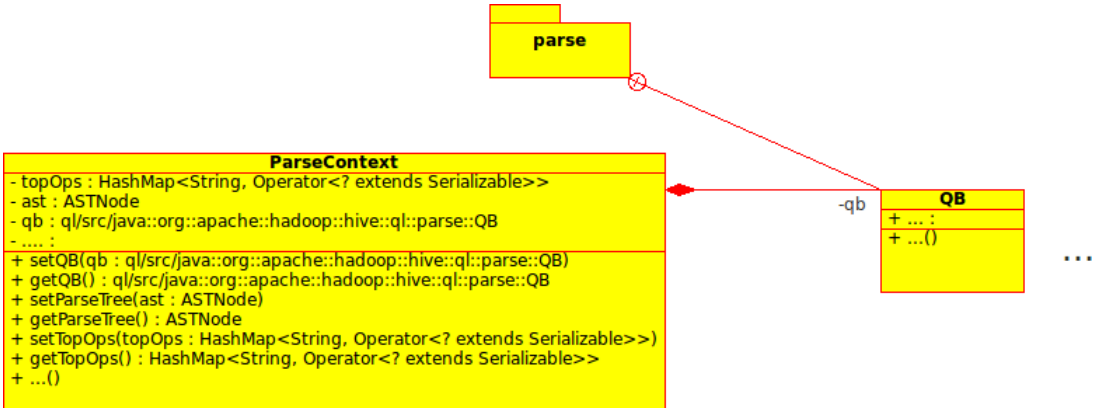


Figura B.4: Diagrama de clases que muestra algunas de las clases auxiliares que se consideran durante la compilación de una sentencia *HiveQL* a *trabajos mapreduce*.

La figura B.4 muestra algunas de las clases que se utilizan como auxiliares durante el proceso de compilación de una sentencia *HiveQL* a trabajos mapreduce. Estas clases se encuentran en el paquete *org.apache.hadoop.hive.ql.parse*. La clase *ParseContext* contiene varias estructuras de datos que se utilizan durante el proceso de compilación de una sentencia *HiveQL*. Por ejemplo, contiene la estructura de datos *topOps* que contiene todos los operadores *TableScanOperator (TS)* de un *DAG*. También contiene la estructura *Abstract Syntax Tree (AST)* que es una representación gráfica que se forma después de realizar un análisis léxico y sintáctico de la consulta *HiveQL*. También contiene la estructura de datos *Query Block (QB)* que es una representación gráfica de una consulta *HiveQL* que se forma a partir del *AST*, en esta estructura se identifican las subconsultas de una consulta *HiveQL*, por esta razón, se localiza un rombo que vincula a la clase *QB*.

La figura B.5 muestra las clases de algunas optimizaciones de *Hive* y la interfaz *Transform*. Estas clases e interfaz se encuentran en los paquetes *org.apache.hadoop.hive.ql.optimizer* y *org.apache.hadoop.hive.ql.ppd*.

La interfaz *Transform* especifica un método *transform(...)* que recibe una variable de tipo *ParseContext* y por lo tanto, cada optimización tiene acceso a las variables *topNodes*, *AST*, *QB*, entre otras. El método *transform* es el método donde se inicializa una optimización de *Hive*, a partir de este método se manda a llamar al *GraphWalker* de cada optimización. Todas las optimizaciones de *Hive* deben de implementar esta

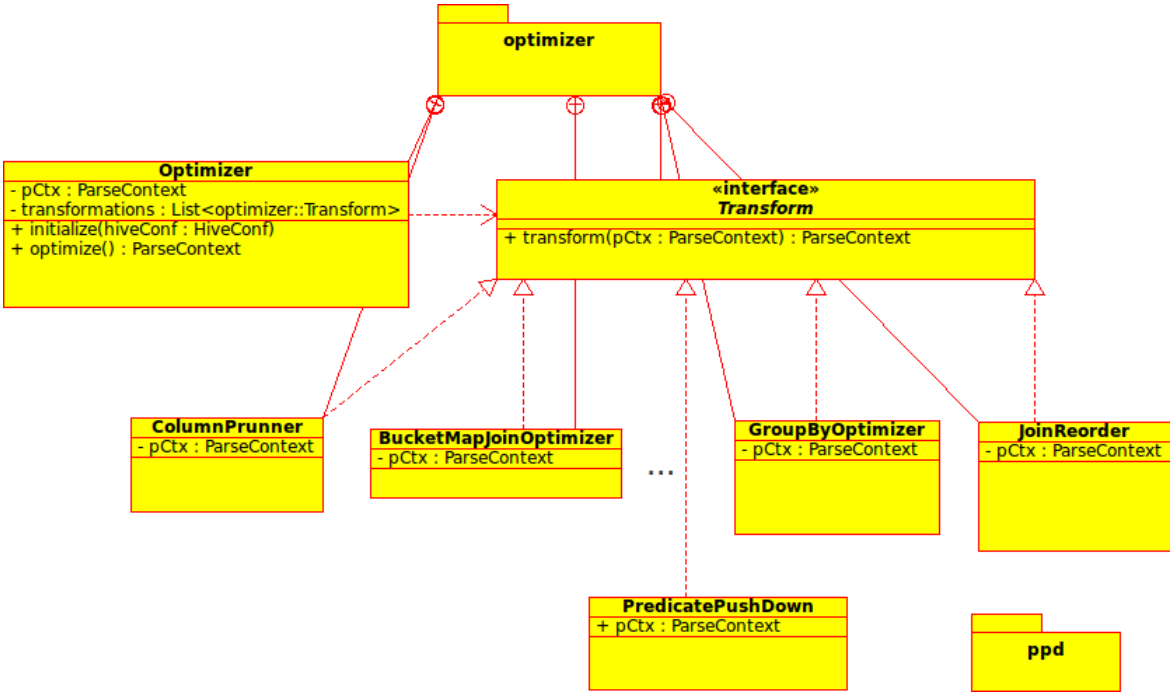


Figura B.5: Diagrama de clases de algunas optimizaciones de *Hive*.

interfaz.

La clase *Optimizer* es aquella donde se registran todas las optimizaciones de *Hive*. Si se desea agregar una nueva optimización, entonces se debe de registrar en esta clase, porqué en esta clase es donde se manda a llamar a todas las optimizaciones de *Hive*.

Las clases *ColumnPrunner, BucketMapJoinOptimizer, PredicatePushDown, GroupByOptimizer, JoinReorder* son clases que representan una optimización actual de *Hive*. Todas estas optimizaciones implementan a la interfaz *Transform* y por lo tanto, todas las optimizaciones tienen un método llamado *transform* donde se inicializa la optimización y se manda a llamar al *GraphWalker*. Todas estas clases se encuentran en el paquete *org.apache.hadoop.hive.ql.optimizer*, excepto la clase *PredicatePushDown* que se encuentra en el paquete *org.apache.hadoop.hive.ql.ppd*.

Para agregar una nueva optimización en *Hive*, se deben de considerar las interfaces *Rule, Node, GraphWalker, Dispatcher, NodeProcessor* del paquete *org.apache.hadoop.hive.ql.lib*, además se debe considerar la clase *Optimizer* y la interfaz *Transform* del paquete *org.apache.hadoop.hive.ql.optimizer*. Los pasos para

agregar una optimización son:

- Crear una clase que implemente la interfaz *Transform* y por lo tanto, tenga el método *tranform ()*. En este método se inicializa las variables de la optimización, una de las variables es un objeto de la clase que implementa a la interfaz *GrapWalker*, de esta manera en esta función se manda a llamar a la función *startWalking()* para inicializar el recorrido de un *DAG* y ver si una optimización se puede aplicar al *DAG*.

- Agregar la optimización en la clase *Optimizer*.

- Si se desea recorrer un *DAG* de diferente manera a la búsqueda en profundidad (DFS) que se realiza en la clase *DefaultGraphWalker*, se puede crear otra clase que implemente la interfaz *GraphWalker* (se surgiere que dicha clase se guarde en el paquete *org.apache.hadoop.hive.ql.lib*) y entonces se debe de implementar el método *startWalking()* para recorrer un *DAG* de la manera que uno desee. En este método se manda a llamar al *Dispatcher*.

- Si se desea encontrar las reglas de una manera diferente a cómo se encuentran en la clase *DefaultRuleDispatcher*, entonces se puede crear una clase que implemente a la interfaz *Dispatcher* (se sugiere que dicha clase se guarde en el paquete *org.apache.hadoop.hive.ql.lib*) y entonces se debe de implementar el método *dispatch()* que es donde se especifica la forma en cómo se deben de encontrar las reglas de una optimización. En este método se manda a llamar al *Processor*.

- Para especificar lo que se debe de hacer en una optimización, se debe de crear una clase que implemente la interfaz *NodeProcessor* (se sugiere que dicha clase se guarde en el paquete *org.apache.hadoop.hive.ql.optimizer*), y entonces se debe de implementar el método *process()* que es donde se especifica lo que se debe de realizar para optimizar un *DAG*.

Apéndice C

Número de *trabajos mapreduce* que generó *Hive* por defecto para cada consulta de Chatziantoniou y TPC-H que se utilizó para evaluar nuestras optimizaciones.

Este apéndice presenta el número de *trabajos mapreduce* que generó *Hive* por defecto para cada consulta de Chatziantoniou y TPC-H que se utilizó para evaluar nuestra optimizaciones de consultas *HiveQL*. Y por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* que se generaron.

En las tablas siguientes las abreviaciones significan lo siguiente: *TM = trabajo mapreduce*, *tm = tareas map*, *tr = tareas reduce*, *q#c = consulta # de Chatziantoniou*, *q#t = consulta # del estudio TPC-H*.

La tabla C.1 muestra el número de *trabajos mapreduce* que generó *Hive* por defecto para cada consulta de Chatziantoniou cuando se manipularon 4 GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* que se generaron. Por ejemplo, para la consulta 1 de Chatziantoniou (q1c) en la tabla C.1 se generaron

5 *trabajos mapreduce*, donde el primer *trabajo mapreduce* generó 18 *tareas map* y 5 *tareas reduce*, el segundo *trabajo mapreduce* generó 18 *tareas map* y 5 *tareas reduce*, y así sucesivamente.

Cuando ya no se especifica números de *tareas map y reduce* a un *trabajo mapreduce*, significa que solo generó el número de *trabajos mapreduce* anterior. Por ejemplo para la consulta 2 de Chatziantoniou (q2c) se generó 4 *trabajos mapreduce*, por lo tanto, en el *trabajo mapreduce 5* ya no se colocó ningún número.

Para la consulta 3 de Chatziantoniou (q3c) obsérvese que todos los *trabajos mapreduce* generan el mismo número de *tareas reduce*, esto es porqué la consulta falló con el número de *tareas reduce* que generó *Hive* por defecto, entonces se buscó una configuración de *tareas reduce* que permitiera ejecutar la consulta, el mínimo número de *tareas reduce* que se encontró fue 64 *tareas reduce*. Sin embargo, al configurar el número de *tareas reduce* de manera manual, todos los *trabajos mapreduce* se ejecutan con el mismo número *tareas reduce*.

Las tablas C.2 y C.3 muestran el número de *trabajos mapreduce* que generó *Hive* por defecto para cada consulta de Chatziantoniou cuando se manipularon 8 GB y 16GB respectivamente. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* que se generarón.

Las tablas C.4, C.5 y C.6 muestran el número de *trabajos mapreduce* que generó *Hive* por defecto para cada consulta del estudio *TPC-H* cuando se manipularon 4 GB, 8 GB y 16GB respectivamente. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* que se generarón.

	TM 1		*TM 2*		*TM 3*		*TM 4*		*TM 5*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q1c	18	5	18	5	21	5	4	1	5	1
q2c	18	5	18	5	3	1	4	1		
q3c	18	64	18	64	8	64	8	64	18	64

Tabla C.1: Número de *trabajos mapreduce* por cada consulta de Chatziantoniou para 4GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas.

	TM 1		*TM 2*		*TM 3*		*TM 4*		*TM 5*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q1c	34	9	35	9	38	9	4	1	6	1
q2c	34	9	35	9	5	1	5	1		
q3c	34	128	34	128	14	128	13	128	24	128

Tabla C.2: Número de *trabajos mapreduce* por cada consulta de Chatziantoniou para 8GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas.

	TM 1		*TM 2*		*TM 3*		*TM 4*		*TM 5*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q1c	65	17	65	17	70	17	6	1	6	1
q2c	65	17	65	17	5	1	6	1		
q3c	64	256	64	256	23	256	25	256	25	256

Tabla C.3: Número de *trabajos mapreduce* por cada consulta de Chatziantoniou para 16GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas.

	TM 1		*TM 2*		*TM 3*		*TM 4*		*TM 5*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	2	1	1	2	1	2	1	5	1	
q3t	9	1	4	14	1	1	1	1	1	
q11t	2	1	1	2	1	1	1	1	1	
q13t	9	1	1	1	1	1	1			

	TM 6		*TM 7*		*TM 8*		*TM 9*		*TM 10*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	4	1	5	1	3	1	1	1	2	1
q3t										
q11t	5	1	1	1	2	1	1	1		
q13t										

	TM 11		*TM 12*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	1	1	1	1
q3t				
q11t				
q13t				

Tabla C.4: Número de *trabajos mapreduce* por cada consulta del estudio TPC-H para 4GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas.

	TM 1		*TM 2*		*TM 3*		*TM 4*		*TM 5*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	2	1	2	1	2	1	2	1	8	1
q3t	11	2	27	7	4	1	1	1	1	1
q11t	2	1	2	1	8	1	8	1	1	1
q13t	11	2	2	1	1	1	1	1		

	TM 6		*TM 7*		*TM 8*		*TM 9*		*TM 10*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	4	1	1	1	8	1	5	1	2	1
q3t										
q11t	1	1	1	1	2	1	1	1		
q13t										

	TM 11		*TM 12*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	1	1	1	1
q3t				
q11t				
q13t				

Tabla C.5: Número de *trabajos mapreduce* por cada consulta del estudio TPC-H para 8GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas.

	TM 1		*TM 2*		*TM 3*		*TM 4*		*TM 5*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	2	1	1	2	1	11	2	5	1	
q3t	16	3	12	5	1	1	1	1	1	
q11t	2	1	1	11	2	2	1	11	2	
q13t	16	3	1	1	1	1	1			

	TM 6		*TM 7*		*TM 8*		*TM 9*		*TM 10*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	1	1	1	2	1	11	2	6	1	2
q3t										
q11t	2	1	1	1	1	2	1	1	1	
q13t										

	TM 11		*TM 12*	
Consulta	No. de *tm*	No. de *tr*	No. de *tm*	No. de *tr*
q2t	1	1	1	
q3t				
q11t				
q13t				

Tabla C.6: Número de *trabajos mapreduce* por cada consulta del estudio TPC-H para 16GB de datos. Por cada *trabajo mapreduce* se muestra el número de *tareas map y reduce* generadas.

Bibliografía

[1] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. *Operating System Design and Implementation (OSDI)*, 51(1):107–113, Enero 2008.

[2] Tom White. *Hadoop The Definitive Guide*. First edition.

[3] Chuck Lam. *Hadoop in Action*. Manning Publications, 180 Broad St. Suite 1323 Stanford, first edition.

[4] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. *International Conference on Data Engineering (ICDE)*, pages 996–1005, Marzo 2010.

[5] David Reinsel John Gantz. Technical report.

[6] IDC. Extracting value from chaos. `http://www.emc.com/digital_universe`, Noviembre 2011.

[7] IBM. Tends in business analytics. `http://www-01.ibm.com/software/data/info/breakfree/trends-in-business-analytics.html`, Noviembre 2011.

[8] Intellego. Bill inmon, el padre del datawarehouse en mÃ©xico. `http://www.intellego.com.mx/interaccion_articulo.php?idarticulo=47`, Junio 2012.

[9] *Netezza Extends TwinFin Appliance Functionality with Availability of SAS/ACCESS Engine Integration.*

[10] Teradata. Hadoop dfs to teradata. `http://developer.teradata.com/extensibility/articles/hadoop-dfs-to-teradata`, 2009.

[11] Asterdata. Sql-mapreduce applications. `http://www.asterdata.com/resources/mapreduce-applications.php`, 2010.

[12] Greenplum. Greenplum mapreduce. `http://www.greenplum.com/technology/mapreduce`, junio 2010.

[13] Oracle. In-database map-reduce. `http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-indatabase-mapreduce-128831.pdf`, Junio 2012.

[14] Oracle. Mysql cluster. `http://www.mysql.com/products/cluster/`, Agosto 2012.

[15] Doug Cutting. Hadoop. `http://hadoop.apache.org/`, Mayo 2012.

[16] Cloudera. Customers cloudera. `http://www.cloudera.com/customers/`, 2011.

[17] Franklin Ricardo Parrales Bravo and Marco Genaro Calle Jaramillo. *Evaluación de MapReduce, Pig y Hive sobre la plataforma de hadoop*. PhD thesis, Escuela Superior Politécnica del Litoral, 2010.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *Special Interest Group on Operating System (SIGOPS) Oper. Syst. Rev.*, 37(5):29–43, 2003.

[19] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D.Stott Parker. Map-reduce-merge: Simplified relational data processing. *Special Interest Group on Management of Data (SIGMOD)*, 36:1029–1040.

[20] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a

cheetah (without it even noticing). *Very Large Database (VLDB)*, 3(1):518–529, Septiembre 2010.

[21] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. *Special Interest Group on Management of Data (SIGMOD) 08*, 6(1):922–933, 2008.

[22] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Special Interest Group on Operating System (SIGOPS) Oper. Syst. Rev.*, 41(3):59–72, 2007.

[24] Songting Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *Very Large Database (VLDB)*, 3(2):1459–1468, 2010.

[25] Daniel J. Abadi Alexander Rasin Avi Silberschatz Azza Abouzeid, Kamil Bajda-Pawlikowski. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Very Large Database (VLDB)*, 2(1):922–933, 2009.

[26] Hive. Hiveql. `https://cwiki.apache.org/confluence/display/Hive/LanguageManual`, Junio 2012.

[27] S.Sudarshan Abraham Silberschatz, Henry F. Korth. *Database System Concepts*. Fifth edition.

[28] Oracle. ApÃ©ndice f. expresiones regulares en mysql. `http://dev.mysql.com/doc/refman/5.0/es/regexp.html`, Septiembre 2012.

[29] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise processing of relational queries. *Very Large Database (VLDB)*, pages 476–485, Agosto 1997.

[30] Transaction Processing Performance Council. Technical report.

[31] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.