23/11/2016

Programación de Objetos Distribuidos

Trabajo Práctico especial

Integrantes:

- Alejandro Bezdjian, 52108
- Jorge Gómez, 52055

Introducción

El objetivo de este trabajo es realizar una aplicación distribuida utilizando **Java** y **Hazelcast** que sea capaz de procesar datos del último censo de Argentina.
Dado un subconjunto del dataset del ***Censo Nacional de Población, Hogares y Viviendas 2010: Censo del Bicentenario*** en un archivo con formato CSV, la aplicación debe poder leer los datos de dicho archivo y ejecutar distintas queries que procesen la información según se pase por parámetros.

Queries requeridas

1. **Cantidad de habitantes total** del país agrupadas por edad en tres categorías: 0 a 14 años, 15 a 64 años, más de 65 años.
2. El **promedio de habitantes por vivienda** para cada tipo de vivienda.
3. Los n **departamentos con mayor índice de analfabetismo**, donde el índice se calcula por el número total de habitantes analfabetos del departamento sobre el total de población del departamento, donde n provee el usuario.
4. Los departamentos de la provincia prov con una **cantidad de habitantes menor a tope**, donde prov y tope lo provee el usuario.
5. Los **pares de departamentos que tienen la misma cantidad de cientos de habitantes**.

Compilación

Para poder compilar el programa se debe seguir los siguientes comandos:

1. Ubicarse en la raíz del código fuente.
2. mvn clean install

Ejecución

Nodos

Para instanciar los nodos de Hazelcast ejecutar los siguientes comandos:

1. Ubicarse en la raíz del código fuente (luego de haber compilado)
2. cd server/target
3. tar -xf mbaracus-server-1.0-SNAPSHOT-bin.tar.gz
4. tar -xf mbaracus-server-1.0-SNAPSHOT-bin.tar.gz
5. chmod +x run-node.sh
6. ./run-node.sh

Cliente

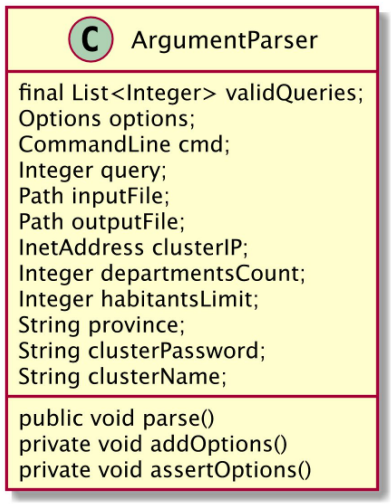
Para ejecutar el cliente se deben realizar los siguientes comandos:

1. Ubicarse en la raíz del código fuente (luego de haber compilado)
2. cd client/target
3. tar -xf mbaracus-client-1.0-SNAPSHOT-bin.tar.gz
4. cd mbaracus-client-1.0.tar.gz
5. Modificar el archivo run-client.sh para utilizar las direcciones a conectarse correctas, la query a ejecutar y los archivos de entrada y salida.
6. chmod +x run-client.sh
7. ./run-client.sh

Diseño e implementación

Parser de argumentos

Se utilizó la librería **Commons Cli** de **Apache** para parsear los argumentos del programa. Si bien esto no era un requerimiento ya que se podían leer los argumentos de la VM (usando System.properties), nos pareció una mejor opción ya que dicha librería se encarga, entre otras cosas, de verificar los argumentos obligatorios devolviendo un mensaje de error si no se reciben. Esto nos ahorró el trabajo de tener que implementar dicha funcionalidad y hace que el código quede más prolijo.



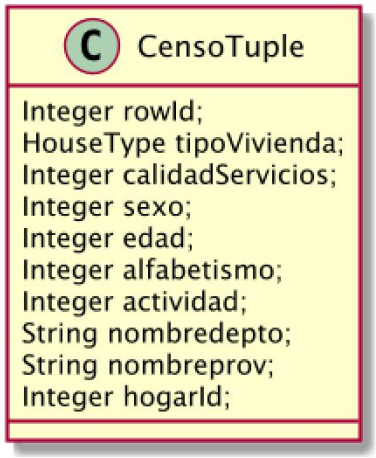
Parser de CSV

Se utilizó la librería **Super CSV** recomendada por la cátedra. Esta librería si bien es fácil de utilizar, nos trajo un problema que se describira en la sección siguiente. La implementación del uso de esta librería se encuentra en la clase CensoReader.

Clase de datos del censo

La librería Super CSV requiere una clase contenedora de los datos de cada fila del archivo CSV y que tenga campos que respeten el nombre de todos los atributos del header del archivo, obligatoriamente tienen que estar todos. Para esto se creó la clase **CensoTuple** que contiene los 9 atributos del archivo CSV.
La clase **IMap<K, V>** de Hazelcast fue utilizada con **K=Integer** y **V=CensoTuple**, donde la key es el numero de fila de la tupla en el CSV.
En un principio probamos el funcionamiento del programa con dicha clase, pero nos dimos cuenta que en las queries no se utilizaban los 9 datos, es más, hay datos que no se usan en ninguna query. Debido a que Super

CSV espera que esten todos los atributos, lo que se decidió fue crear 5 clases nuevas (una para cada query) que extiendan de CensoTuple y que solo utilizan los atributos que necesita cada query. Esto aumentó la performance en la distribución de la información en los nodos.

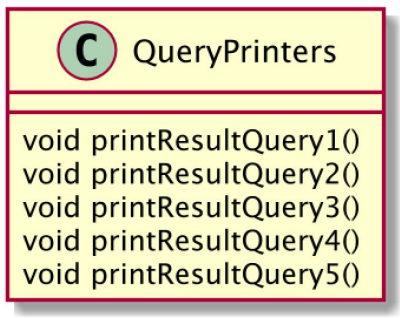


Client y clases utilitarias

Client es la clase que contiene el main del programa. Desde un principio se deseo que solo dirija el trabajo, delegando las acciones en clases utilitarias y así quedando más limpio el código. Entonces las acciones del Client son:

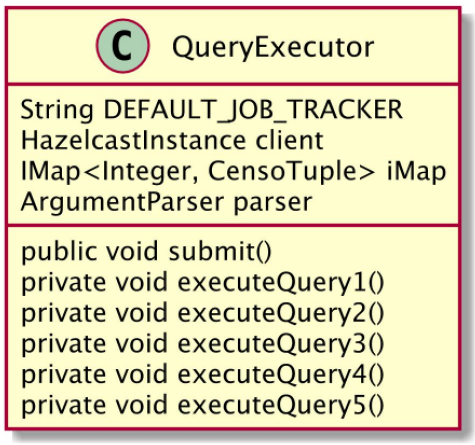
1. Llamar al parser de argumentos
2. Conectarse con los nodos de Hazelcast y obtener el mapa
3. Llamar al parser de CSV
4. Llamar a una clase llamada QueryExecutor, que se encarga de decidir qué query se ejecuta y de qué manera.

Ademas existe la clase QueryPrinters que tiene métodos encargados de la escritura del resultado de las queries en el archivo indicado y con el formato indicado.



QueryExecutor

Esta clase contiene un único método publico llamado **submit** que recibe como parámetro un entero que identifica la query a ejecutar y decide a cuál de sus métodos privados llamar (cada uno ejecuta una query determinada). Cada una de las queries fueron implementadas como trabajos MapReduce utilizando Hazelcast. Como se mencionó anteriormente el tipo de datos del mapa es **IMap<Integer, CensoTuple>**.

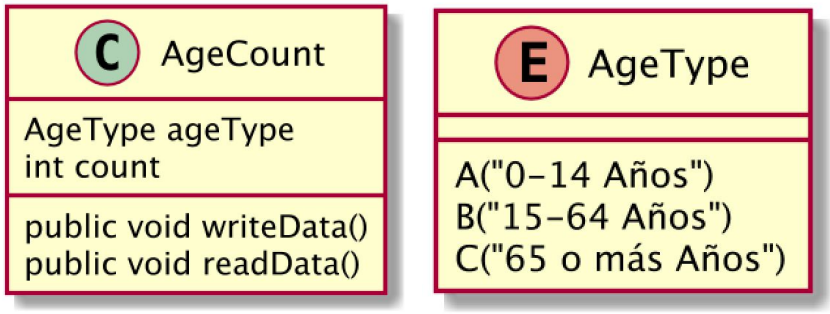


Query 1

Esta query pide la cantidad de habitantes agrupados por 3 rangos de edades distintas.

- **Mapper <Integer, CensoTuple> -> <AgeType, AgeCount>:**
  - Simplemente devuelve el AgeType correspondiente y una nueva instancia de AgeCount que pasa 1 (habitante) como argumento al constructor.
- **Reducer <AgeType, AgeCount> -> AgeCount:**
  - Lleva la cuenta de cuántos habitantes hay para cada AgeType y finalmente devuelve un AgeCount con la cantidad total.

AgeType es un enum con las distintas categorías y AgeCount es una clase que contiene la cantidad de habitantes y a que AgeType pertenecen.
En un principio se utilizó Integer en lugar de AgeType, pero vimos que el código era mucho menos expresivo y al cambiarlo no se notó una disminución en la performance, por lo que se decidió favorecer la mantenibilidad y expresión del código.

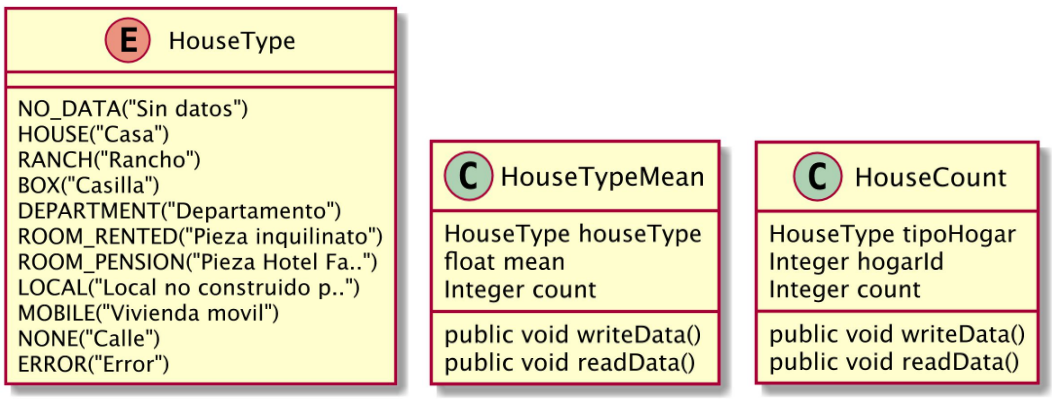


Query 2

Esta query devuelve el promedio de habitantes de cada vivienda para cada tipo de vivienda. Se necesitó utilizar dos trabajos MapReduce en lugar de uno.

- **Mapper 1 <Integer, CensoTuple> -> <Integer, HouseCount>:**
  - Devuelve en el Integer el ID único del hogar y un objeto HouseCount que tiene un unico integrante.
- **Reducer 1 <Integer, HouseCount> -> HouseCount:**
  - Junta a los miembros del mismo hogar y devuelve un HouseCount con la cantidad de personas en esa casa y el promedio (que es el mismo que la cantidad).
- **Mapper 2 <Integer, HouseCount> -> <HouseType, HouseTypeMean>:**
  - Devuelve el tipo de casa (que obtiene de los HouseCount de entrada) y devuelve un objeto HouseTypeMean que representa el promedio en dicha casa.
- **Reducer 2 <HouseType, HouseTypeMean> -> HouseTypeMean:**
  - Junta todas los HouseTypeMean en uno solo que representa el promedio para un HouseType.

HouseType es un enum con los distintos tipos de casas (ya se mencionó en el punto anterior los beneficios de dicha decisión), HouseCount es una clase que contiene el tipo de casa, la cantidad de habitantes y el identificador único de casa y HouseTypeMean es una clase que tiene el tipo de casa, la cantidad de integrantes y el promedio. Es necesario que HouseTypeMean tenga la cantidad de personas para poder generar el nuevo promedio con otro HouseTypeMean.

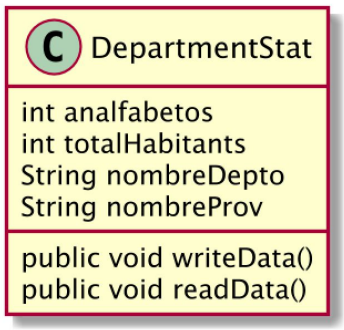


Se implementó dicha solución porque resultaba ser la más simple de implementar. Una vez que se logró que funcione, se intentó utilizar un Collator para no tener que realizar el segundo MapReduce, pero vimos que esto tardaba un 30% mas de forma local con el dataset de 1.000 tuplas. Creemos que esto se debe a que el segundo procesamiento es hecho por el cliente.

Query 3

Esta query devuelve los departamentos con mayor índice de analfabetismo.

- **Mapper <Integer, CensoTuple> -> <Integer, DepartmentStat>:**
  - Devuelve como key un integer que representa al hashCode del DepartmentStat, el cual se forma con la combinación del nombre de la Provincia y el nombre del departamento (para diferenciar los departamentos con el mismo nombre en distintas provincias). El value devuelto es un DepartmentStat con un 1 como total de habitantes y un 1 o 0 en analfabetos dependiendo los datos de la tupla en cuestion.
- **Reducer <Integer, DepartmentStat> -> DepartmentStat:**
  - Para un departamento con un determinado hash, simplemente suma la cantidad de habitantes y la de analfabetos. Al finalizar devuelve una nueva instancia de DepartmentStat con los datos totales.



En un principio se había hecho que el DepartmentStat se guarde el promedio y que el reducer lo setee al final. Si bien esto hace que sea más eficiente al momento de hacer el sort de los departamentos por este valor (requerimiento), hacía que tarde mucho más en la transmisión de los resultados parciales.

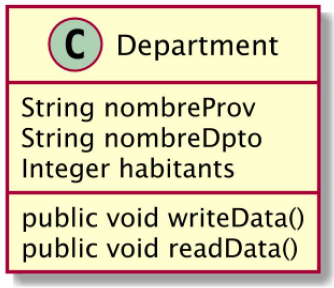
Query 4

Esta query devuelve los departamentos de una provincia determinada con una cantidad total de habitantes menor a *tope* que se pasa por parámetro.

- **Mapper <Integer, CensoTuple> -> <Integer, Department>:**
  - Devuelve como key un integer que representa al hashCode del Department, el cual se forma con la combinación del nombre de la Provincia y el nombre del departamento (para diferenciar los departamentos con el mismo nombre en distintas

provincias). El value devuelto es un Department con un 1 como total de habitantes.

- **Reducer <Integer, Department> -> Department:**
  - Para un departamento con un determinado hash, simplemente suma la cantidad de habitantes. Al finalizar devuelve una nueva instancia de Department con los datos totales.

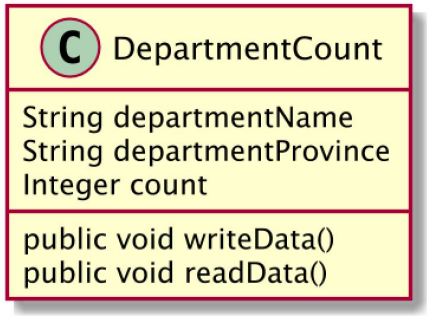


Para realizar de forma eficiente esta query en un principio se optó por implementar un KeyPredicate que validaba si se debía enviar o no el dato basado en la provincia ingresada. Luego, nos dimos cuenta que era mucho más eficiente filtrar el dato directamente en el parser y no agregar al mapa el valor leído si no coincidía con la provincia ingresada por el usuario. Si bien puede violar la privacidad de las capas, la eficiencia obtenida es mucho mayor de esta manera, por lo tanto se la dejó como implementación final. Igualmente se dejó el filtro luego del Map-Reduce por si se desea realizar otra implementación o utilizar un mapa ya existente. Al final de la misma se realiza este filtro, de la cantidad de habitantes y el ordenamiento de forma paralela.

Query 5

Esta query devuelve los pares de departamentos que tienen la misma cantidad de cientos de habitantes.

- **Mapper <Integer, CensoTuple> -> <Integer, DepartmentCount>:**
  - Devuelve como key un integer que representa al hashCode del DepartmentCount, el cual se forma con la combinación del nombre de la Provincia y el nombre del departamento (para diferenciar los departamentos con el mismo nombre en distintas provincias) y como value un DepartmentCount que posee el nombre del departamento, el nombre de la Provincia y un 1 representando el número de habitantes en ese departamento.
- **Reducer <Integer, DepartmentCount> -> DepartmentCount:**
  - Para el mismo DepartmentCount suma la cantidad total de habitantes. Al finalizar devuelve una nueva instancia de DepartmentCount con el total de habitantes por departamento.



Cómo nos parecía innecesario un Map-Reduce para realizar la combinación de los elementos se decidió implementar el mismo de la siguiente manera: Implementar un mapa que contiene como key los “cientos” de habitantes y como value una lista de los departamentos que contienen esa cantidad de cientos de habitantes. Luego iterando doblemente por cada una de esas listas (si tiene más de un departamento en ella) se obtienen todas las combinaciones posibles de departamentos de la misma.

Problemas encontrados

Durante el desarrollo del trabajo se encontraron diversos problemas y se tomaron diferentes decisiones respecto a ellos:

1. La lectura y transmisión de datos del dataset con 1.000.000 tuplas era muy lento, por lo tanto para realizar el análisis de performance se decidió realizar por un subset de datos de este archivo, que contenía 100.000 tuplas. También se decidió agregar un flag adicional -Dreuse que, para sacar el promedio luego de varias ejecuciones, reutiliza el mapa ya completo al momento de la lectura, permitiendo ahorrar el tiempo de transmitir el mapa.
2. Se tuvo inconvenientes al momento de utilizar los combiners en múltiples nodos en el laboratorio. Si bien estos funcionaban sin problemas al utilizarse en local, e incluso se notaba una reducción del tiempo de proceso, al momento de ejecutar con nodos montados en las computadoras del laboratorio se producían errores en los datos de tipo String que en el archivo de salida se mostraban como “null”. A causa de este inconveniente dejamos desactivados los combiners por defecto, pero si se desean utilizar se puede utilizar el flag -Dcombiners, ya que decidimos dejar igualmente la implementación de los mismos.

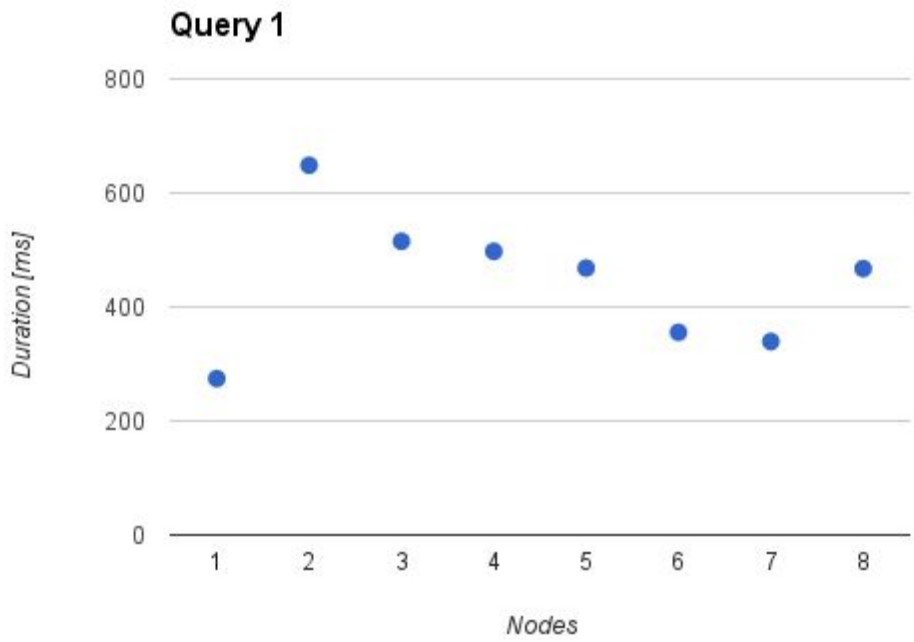
Análisis de performance

Se realizó un análisis de performance para cada query donde se varió la cantidad de nodos en el cluster de Hazelcast, con el objetivo de encontrar el número óptimo de nodos. Para esto se utilizó:

- Un archivo CSV de 100.000 tuplas
- En cada computadora de la red se instanció 1 nodo Hazelcast.
- Se promediaron 5 corridas
- Las corridas se hicieron una tras otra en un loop
- Se cargaron los datos en el mapa antes de cada query y se reutilizó para el resto

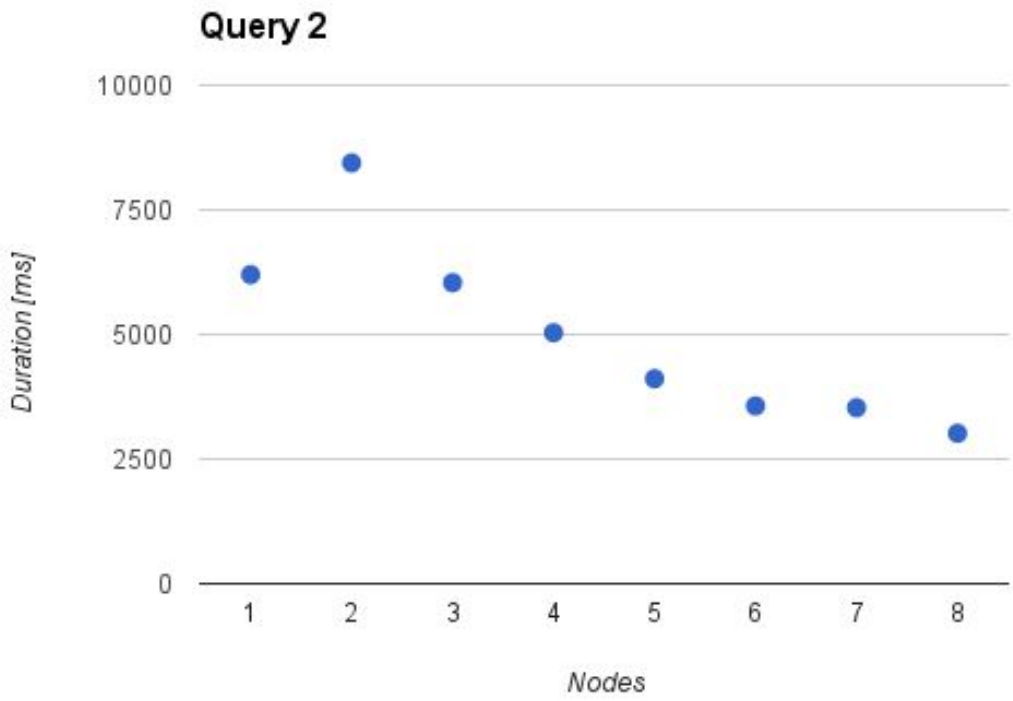
Query 1

Nodos	Tiempos					Promedio
1	523	285	185	210	169	**274,40**
2	860	636	573	556	618	**648,60**
3	687	512	452	415	510	**515,20**
4	652	414	404	596	422	**497,60**
5	786	436	368	395	356	**468,20**
6	643	290	307	287	251	**355,60**
7	604	285	273	251	284	**339,40**
8	796	437	365	367	372	**467,40**



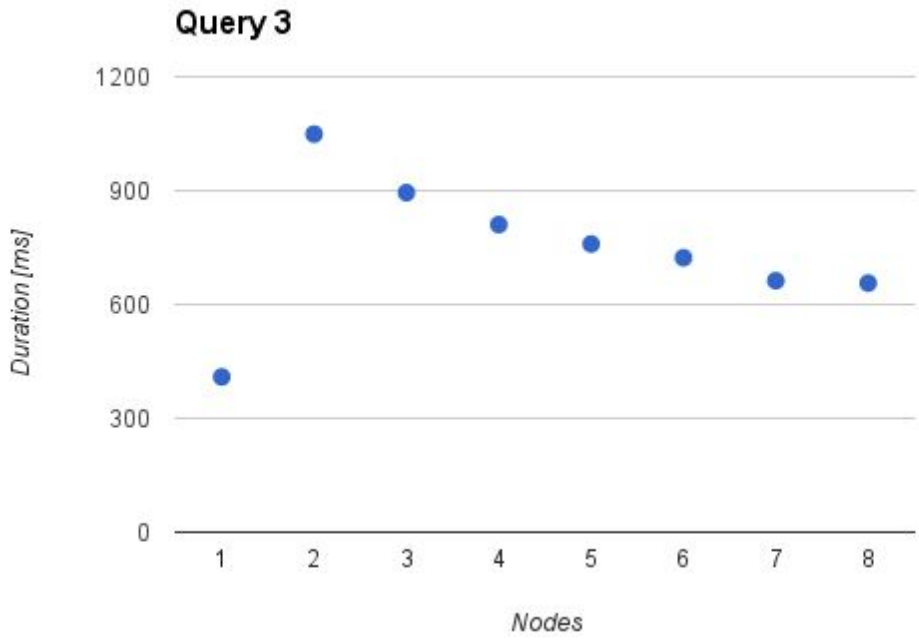
Query 2

Nodos	Tiempos					Promedio
1	6823	6196	5987	6125	5874	**6201,00**
2	8814	8382	8112	8577	8331	**8443,20**
3	6827	6793	5752	5421	5399	**6038,40**
4	5430	5061	5401	4534	4749	**5035,00**
5	4372	3958	4011	3992	4231	**4112,80**
6	4228	3547	3514	3268	3279	**3567,20**
7	3753	3721	3244	3101	3839	**3531,60**
8	3652	3158	2879	2876	2520	**3017,00**



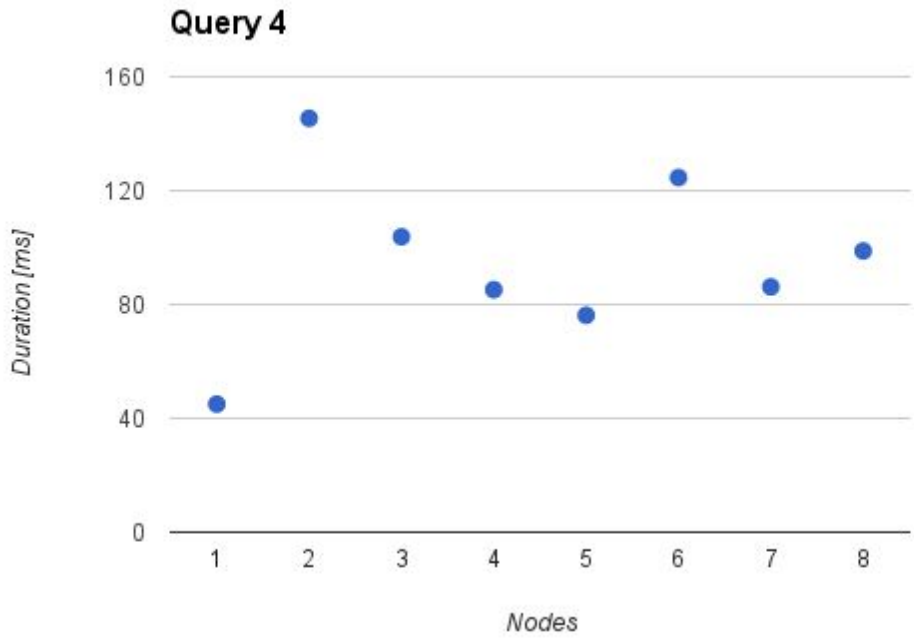
Query 3

Nodos	Tiempos					Promedio
1	832	404	329	235	246	**409,20**
2	1354	1095	950	923	923	**1049,00**
3	1094	874	893	840	772	**894,60**
4	1130	701	729	787	706	**810,60**
5	1193	646	672	645	641	**759,40**
6	1055	635	719	581	626	**723,20**
7	986	638	567	549	573	**662,60**
8	1045	551	624	542	520	**656,40**



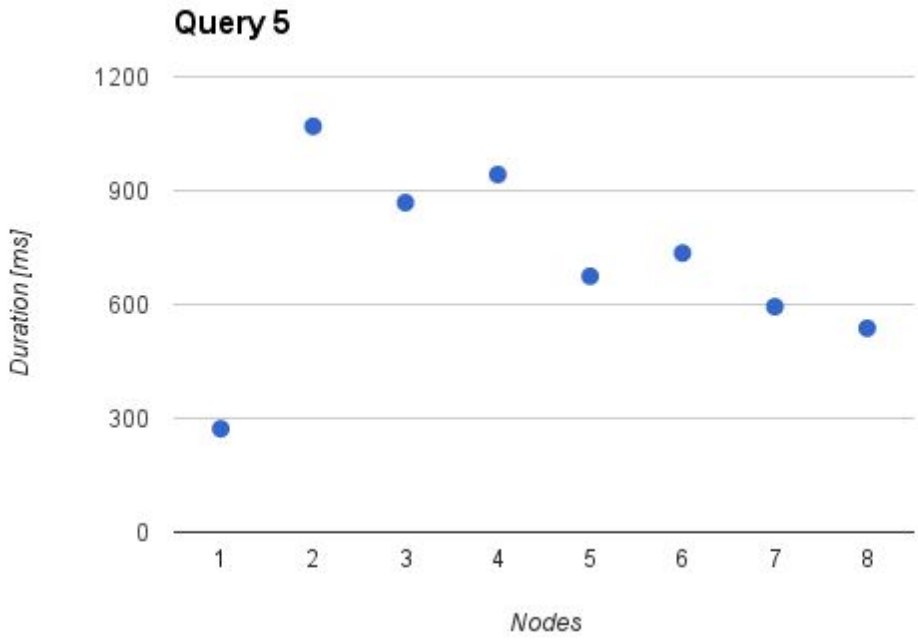
Query 4

Nodos	Tiempos					Promedio
1	116	26	28	29	26	**45,00**
2	167	150	134	133	143	**145,40**
3	142	92	102	92	91	**103,80**
4	128	77	70	82	69	**85,20**
5	128	59	58	69	67	**76,20**
6	112	301	59	80	71	**124,60**
7	197	54	58	70	52	**86,20**
8	281	49	63	47	54	**98,80**



Query 5

Nodos	Tiempos					Promedio
1	437	325	200	192	209	272,60
2	1275	1052	983	1091	946	1069,40
3	869	1077	803	767	826	868,40
4	942	892	835	866	1177	942,40
5	733	672	613	619	735	674,40
6	953	958	692	488	587	735,60
7	617	569	548	756	481	594,20
8	628	495	519	483	562	537,40



Conclusiones

En los resultados se puede observar como para todas las queries, utilizando 1 nodo resulta ser la corrida más rápida, luego se observa que el tiempo crece mucho y comienza a decaer. Esto se debe a que el costo de distribuir la información a los nodos es mayor que la del cálculo que se realiza.
La única excepción se da en la query 2, la cual se observa su mejor performance con 8 nodos, esto se debe a que dicha query realiza 2 map-reduce en lugar de 1, lo cual hace que la distribución de los cálculos sea más importante que en los otros casos.
También se observó que correr repetidamente las queries hace que el cálculo se realice cada vez más rápido bajo las mismas condiciones, esto creemos que se debe a optimizaciones automáticas de la JVM.

Anexo

Debido a las dificultades y velocidades para correr en el laboratorio se decidió ejecutar las queries sobre el archivo de 1.000.000 tuplas en una una máquina local, aprovechando la posibilidad de utilizar combiners y poder comparar los tiempos. Las pruebas se ejecutaron sobre una sola computadora con un procesador Intel i7 3612-QM 2.1Ghz y 8GB de memoria RAM, y los resultados son promedios en milisegundos luego de realizar 5 ejecuciones.

	Query 1	Query 2	Query 3	Query 4	Query 5
1 Nodo	937,40	183033,80	1151,40	23,40	1507,60
1 Nodo con combiners	946,20	159552,00	1405,60	18,60	2550,00
8 Nodos con combiners	1045,00	Error*	3225,40	177,80	2552,4**

*La query 2 presenta fallas de memoria al momento de ejecutar 8 nodos sobre una sola computadora.
**La query 5 presenta fallas en los resultados, devolviendo null en los nombres de las provincias, a pesar de esto las combinaciones parecen ser las correctas. (Tiempo sin combiners: 34603ms)

Si bien se está ejecutando en una sola máquina y la principal funcionalidad de los combiners es disminuir el tráfico sobre la red, nos pareció curioso la diferencia de tiempo en la query 2. También es necesario destacar que con 8 nodos se produjo un incremento de tiempo, pero se cree que es por estar ejecutando los 8 nodos sobre una misma computadora, ya que la memoria RAM estaba 100% durante la ejecución y se utilizaba la memoria swap.
En base a estos resultados obtenidos y los analizados previamente en el informe se cree que la query 2, la más compleja, puede ser aún mucho más eficiente ejecutándose en 8 o más nodos y la utilización de combiners si estos no produjeran un error en la salida.