Instituto Tecnológico de Buenos Aires

72.42 - Programación de Objetos Distribuidos

TPE 2 - Viajes en autos de aplicación en Nueva york

Campoli, Lucas - 63295

Coleur, Matías - 63461

Smirnoff, Sergio - 62256

Schvartz Tallone, Martina - 62560

Índice

1. Decisiones de diseño e implementación de los servicios

1.1 Implementación de la API

Para la implementación de los trabajos MapReduce, todas las queries siguen la misma estructura. Se definen entonces las clases (ordenadas por orden de procesamiento):

- TripRowQX: clase definida por cada query X, la cual posee los campos del CSV que la query necesita como entrada para operar (de forma que no se carguen todos los datos innecesariamente).
- Mapper: emite un par clave-valor con la información necesaria para resolver la query
- ReducerFactory: instancia el reducer que se encarga de procesar los value que tengan la misma clave emitida por el mapper
- Combiner: optimiza el tráfico de la red al combinar los datos en uno solo dentro de un mismo nodo en caso de que se necesite enviar a otro nodo
- Collator: post-procesamiento utilizado comúnmente para ordenar o transformar los objetos con la información relevante para el resultado de la query y su impresión

Existen queries que definen sus propias keys o values como objetos también, pero en general todas implementan las clases anteriormente mencionadas.

1.1.1 Query 3: Precio promedio de viaje por barrio de inicio y compañía

Al principio se pensó el *mapper* y *reducer* de esta consulta de forma muy básica, pasándole todos los componentes del csv y filtrando los que se necesitaran en el mapper. También se realizaban las consultas a un mapa distribuido de consultas. Luego se realizaron las modificaciones necesarias para que se pudiera filtrar en el cliente como se hablará más adelante, eliminando así la consulta distribuida de zonas y que al mapper no le llegaran datos que no iba a necesitar. Se implementó una clase Pair para la consulta que tuviera de clave la compañía y el barrio por el cual se estaba buscando, así se podría agrupar los datos necesarios. Esa clase pair implementó comparable para luego en el *collator* poder usar su *compare* para ordenar respetando los requerimientos del enunciado. Se implementó finalmente una clase *record* para poder entregar las líneas finales para imprimir a un archivo.

1.1.2 Query 5: Total de millas YTD por compañía

Para la implementación de esta consulta, se solicitaba el total de millas viajadas YTD por cada compañía. Desde un principio, era evidente que la clave del mapa a utilizar era la tupla **<compañía, año, mes>** por lo que se definió **TotalMilesKey**. Una vez establecido eso,

la unica decision de diseño a tomar consistió en como calcular el YTD de cada mes y año, ya que las dos opciones eran:

- Una de las opciones de diseño consideradas fue el desnormalizar los datos mediante una **expansión** en la fase de Mapeo. Por cada registro de viaje, se emitían **múltiples pares clave-valor**, replicando la métrica del viaje para todos los meses desde el mes del viaje hasta diciembre de ese año para luego sumarlas. Por ejemplo, si el viaje se realizaba en Julio de 2025, se emite el valor de las millas del viaje para las claves de Julio a Diciembre de 2025.
- La segunda opción era realizar un **MapReduce inicial** para calcular el total de millas viajadas en cada año y mes por cada compañía, y luego en un Collator añadirles el total de millas viajadas en los meses posteriores de ese año.

Finalmente se optó por la **segunda opción**, ya que en los resultados solo tenían que aparecer los valores para los meses en los que hubo por lo menos un viaje, y en la primera opción se emitían valores para todos los meses sin distinguir si había por lo menos un viaje en ese mes o no. Además que en la primera opción se emitían muchísimas más claves que en la segunda, lo cual se traduce a más información repetida en memoria y viajando por la red.

Otra aspecto a considerar es que el **Collator** era **necesario en las dos opciones** ya que se solicitaban los resultados ordenados, por lo que en la segunda opción solo había que agregar la lógica de sumar los valores de meses anteriores

1.1.3 Alternativas de diseño en común

1.1.3.1 Combiners

Se decidió implementar **Combiners** en todas las queries a **excepción de la tercera**, ya que por su naturaleza de retornar el promedio, se dificulta el cálculo entre nodos. La mejora implica que la gran mayoría de comparaciones se hagan dentro de cada nodo, por lo que disminuye el tráfico de red y por consiguiente, el tiempo de ejecución final.

1.1.3.2 Clases Mapper/Reducer

En un principio, se analizó una **alternativa** para poder reutilizar mappers/reducers entre queries, que consistía en tener clases genéricas para ambos tipos. Tomando como ejemplo la query 1 y la query 5, en ambas se realizaba una lógica similar: se arma la key y se emite como valor un número. La idea era poder implementar eso en una clase **CountMapper** genérica que recibiera por parámetro qué transformación aplicar al TripRowQX (value in para el mapper) para poder extraer la key out y luego que recibiera también qué value de salida enviar para esa key.

Rápidamente se **descartó** esto debido a que el hecho de enviar funciones por parámetro requería que estas fueran serializables y que cada una de las queries las definiera,

por lo que en cuestiones de diseño se optó por dejarlo como se había planteado al inicio, es decir que exista un mapper/reducer particular para cada query.

1.1.3.3 String vs. Integer

Se evaluó usar **Pair<Integer, Integer>** como clave para las queries 1 y 3, aprovechando que los enteros son de tamaño fijo y rápidos de comparar en la fase de shuffling. Sin embargo, este enfoque presentaba problemas: el desempate por igualdad de total de viajes requería los nombres de las zonas (no los IDs), obligando al Collator a consultar un mapa distribuido. Además, el Mapper también necesitaba acceder a ese mapa para validaciones, generando dos pedidos de red por cada línea procesada.

Para evitar esta sobrecarga de red, se decidió cambiar la clave a **Pair<String, String>**, usando directamente los nombres de las zonas. Aunque transferir strings es más costoso por su tamaño variable y mayor, se determinó que era preferible a los múltiples pedidos de red. Esta decisión, combinada con optimizaciones para la Query 4 (ver sección Optimización), permitió realizar el mapeo de ID a nombre durante la carga inicial de datos, eliminando por completo la necesidad de que el mapper acceda al mapa distribuido de zonas.

1.1.3.4 Incremento de tiempo timeout de HeartBeat

Se incrementó el tiempo de **heartbeat** de 1 minuto a 5 minutos para que cuando nosotros estamos cargando la información en los nodos hazelcast tenga tiempo de enviar un heartbeat además de toda la información junta por los threads que cargan el mapa.

Luego de otras pruebas nos dimos cuenta que no era necesario este incremento ya que por las implementaciones anteriores de carga de datos estaban generando que el heartbeat no llegara, realizando una carga en batch dejó que llegara correctamente esta señal haciendo que no haga falta esta modificación. Se volvió al tiempo default de 1 minuto.

1.2 Implementación del client

1.2.1 Clase Client Abstracta

Se implementó una clase **Client** abstracta, la cual implementa las principales funciones en común para todas las consultas como inicialización de variables, conexión al cluster Hazelcast, carga de datos, logging e impresión a archivo. Define las funciones abstractas necesarias para ser definidas en los clientes que la extiendan. Cada cliente de las distintas consultas, extiende de esta clase abstracta e implementa únicamente lo necesario para cada consulta: **executeMapReduce**, donde define el trabajo mapReduce con el Mapper, Reduce, Combiner y Collator correspondiente; **getCsvHeader**, donde se define el header del csv de respuesta; y finalmente se invoca el **run**, pasando como argumentos los map y filtros necesarios para la carga de datos inicial en el mapa distribuido.

Esta clase cuenta con toda la lógica compartida en los clientes y permite definir a cada clase que lo herede lo necesario para realizar el trabajo del mapReduce sobre el dataset de viajes y zonas.

1.2.2 Carga de datos de zonas

El método **loadZonesData** se ejecuta una única vez al inicio para leer **zones.csv** y cargarlo por completo en un HashMap local en la memoria del cliente, usando el LocationID como clave. Se optó por esta estrategia en lugar de un mapa distribuido de Hazelcast porque el archivo de zonas es **pequeño y cada registro contiene poca información**.

Esta decisión **acelera** significativamente la **carga de datos**: al tener este mapa en la memoria local, el TripBatchLoader puede resolver los nombres de las zonas de forma instantánea y sin ningún acceso a la red, antes de enviar los datos de viajes al clúster. Esto elimina por completo la necesidad de que los Mappers o Reducers que corren en el clúster accedan a esta información.

Somos conscientes de que este enfoque es ineficiente si la información de las zonas necesitará ser modificada (habría que actualizar múltiples valores en el mapa distribuido). Sin embargo, **para los fines solicitados en el trabajo práctico**, donde los datos se cargan una vez para ser consultados, esta estrategia es suficiente y **acelera drásticamente la carga**, eliminando por completo la necesidad de que los Mappers en el clúster accedan a esta información. Ver sección Optimización para información acerca de esto y comparaciones de tiempo.

1.2.3 Carga de datos de viajes

Para la carga de datos del archivo de viajes (**trips.csv**), se implementó una estrategia de **Producer-Consumer** optimizada. El **Producer** es el hilo principal, que lee el archivo secuencialmente línea por línea mediante un **Stream y un Iterator**. Este hilo no procesa las líneas individualmente, sino que las acumula en un **Batch** hasta alcanzar un tamaño predefinido. Al llenarse el batch, se crea una copia de esta lista y se la envuelve en una nueva tarea TripBatchLoader, la cual se delega a un pool de hilos para su procesamiento asíncrono.

El **Consumer** es un ThreadPoolExecutor configurado manualmente para controlar la carga y evitar desbordes de memoria. Se define un pool con un número fijo de hilos y, de forma crucial, una cola de trabajo acotada **ArrayBlockingQueue**. Para gestionar la saturación, se utiliza la política de rechazo **CallerRunsPolicy**. Esta política genera contrapresión: si la cola de tareas está llena, el hilo principal es forzado a ejecutar la tarea él mismo, pausando temporalmente la lectura del archivo y permitiendo que el sistema se auto-regule.

Finalmente, cada tarea **TripBatchLoader** es ejecutada por un hilo del pool. Para minimizar la sobrecarga de red y la contención en el **IMap** distribuido, la tarea primero procesa sus líneas y las almacena en un HashMap local. Solo cuando ha procesado todo su batch, realiza una única llamada de red para enviar el lote completo de datos al clúster de Hazelcast. Un **AtomicInteger** garantiza que cada registro tenga una clave única y thread-safe a través de todas las tareas en ejecución.

1.2.4 Logueo de tiempos

Para cumplir con los requisitos de monitoreo y análisis de rendimiento, se decidió implementar un mecanismo de logueo de tiempos dedicado, independiente del log principal de la aplicación. Esta decisión se materializó configurando un FileHandler específico, asegurando que todos los mensajes enviados al **timeLogger** se centralizan en un archivo de salida único **timeX.csv** como se solicita en el enunciado. Dado que los formatos de log estándar no proveían la granularidad o estructura solicitada, se optó por crear un Formatter personalizado. Este formateador anónimo define un DateTimeFormatter específico con un patrón para capturar los tiempos con precisión de milisegundos. Finalmente, se sobreescribe el método **format** para construir manualmente cada línea de log, incluyendo campos clave como el timestamp de alta precisión, el nivel del log, el nombre del hilo de ejecución y el mensaje, generando así un archivo de tiempos con un formato limpio y consistente, listo para su análisis

1.2.5 Tests

Se realizaron tests para las 5 consultas con un extracto de 2000 líneas del csv otorgado por la cátedra. Estos tests validan el funcionamiento básico de la consulta y verifica que todas terminen y realicen lo que deben hacer con los resultados esperados. Se utiliza 1 nodo únicamente para los tests.

2. Análisis de tiempos de ejecución

2.1 Tiempos por Nodo

2.1.1 Explicación del sistema Utilizado

Se realizará un análisis del comportamiento de la implementación cuando se incorporan más nodos. Cada nodo utilizará 4 gigas de ram como límite. El estudio se realizará sobre la consulta 4 pasando siempre los mismos parámetros, es decir, consultando por el borough “Manhattan”. Las primeras realizaciones, con 1 nodo, 2 nodos y 3 nodos, se realizaron en una sola máquina ya que la cantidad disponible de memoria RAM lo permitía.

Luego los restantes, 4 nodos y 5 nodos, se realizaron en 2 máquinas con conexión con un router y cables LAN versión 5E. Se distribuyó de la siguiente manera:

4 Nodos: Se utilizaron 2 nodos en cada máquina y una de las 2 máquinas ejecutó el cliente.

5 Nodos: Se utilizaron 2 nodos en una máquina con el cliente y 3 nodos en la otra máquina.

En todos los casos se utilizó el archivo csv con 20 millones de líneas.

2.1.2 Resultados

2.1.3 1 Nodo

Se tuvieron los siguientes resultados:

	Carga de datos (s)	MapReduce (s)
1er Intento	11	38
2do Intento	10	49
3er Intento	10	38

Se evidencia en la tabla que, en promedio, la carga de datos fue de 10.33 segundos y el MapReduce fue de 41.67 segundos, haciendo que el total son 52 segundos de corrida.

2.1.4 2 Nodos

Se obtuvieron los siguientes resultados:

	Carga de datos (s)	MapReduce (s)
1er Intento	15	29
2do Intento	15	31
3er Intento	14	29

Se evidencia en la tabla que, en promedio, la carga de datos fue de 14.67 segundos y el MapReduce fue de 29.67 segundos, haciendo que el total son 44.34 segundos de corrida.

2.1.5 3 Nodos

Se obtuvieron los siguientes resultados:

	Carga de datos (s)	MapReduce (s)
1er Intento	15	30
2do Intento	18	27
3er Intento	16	26

Se evidencia en la tabla que, en promedio, la carga de datos fue de 16.33 segundos y el MapReduce fue de 27.67 segundos, haciendo que el total son 44 segundos de corrida.

2.1.6 4 Nodos

Se obtuvieron los siguientes resultados:

	Carga de datos	MapReduce
1er Intento	2 min 27 s 285 ms	22 s 551 ms
2do Intento	2 min 19 s 267 ms	22 s 45 ms
3er Intento	2 min 19 s 304 ms	22 s 218 ms

Se evidencia en la tabla que, en promedio, la carga de datos fue de 2 minutos y 27 segundos y el MapReduce fue de 22.27 segundos, haciendo que el total son 2 minutos y 49 segundos de corrida.

2.1.7 5 Nodos

Se obtuvieron los siguientes resultados:

	Carga de datos	MapReduce
1er Intento	2 min 34 s 552 ms	22 s 352 ms
2do Intento	2 min 34 s 562 ms	19 s 305 ms
3er Intento	2 min 33 s 725 ms	21 s 279 ms

Se evidencia en la tabla que, en promedio, la carga de datos fue de 2 minutos y 34 segundos y el MapReduce fue de 20.67 segundos, haciendo que el total son 2 minutos y 55 segundos de corrida.

2.1.8 Análisis de los resultados

Con estos resultados podemos concluir que la carga de los datos del csv toma más tiempo a medida que se aumenta la cantidad, especialmente si se involucra la RED en el medio. Con los resultados que obtuvimos la carga de datos en la misma máquina tarda alrededor de 15 segundos pero cuando se involucra la red y más nodos esto aumenta a alrededor de 150 segundos lo cual es un x10 más lento. Con el uso de más nodos se nota que gradualmente va el MapReduce reduciendo su tiempo de ejecución debido a estar distribuido. Llegamos a la conclusión que para resolver este caso, lo más balanceado sería utilizar 3 nodos, ya que da un balance entre tiempo de carga de datos que es lo que más demora y una ejecución del mapReduce razonable, permitiendo un buen nivel de paralelismo.

2.2 Uso de Combiner

Se utilizará como base 3 nodos en una misma computadora asignados como máximo 4G de RAM. La modificación del combiner se hará sobre la *Query 1* ya que es en la que creemos que tendrá mejor rendimiento ya que funciona parecido al *word count* visto en clase.

2.2.1 Tiempos sin Combiner

Se realizaron 5 tiradas de la ejecución de la consulta 1 sin combiner y estos fueron los resultados:

	Carga de datos	MapReduce
1er Intento	14 s 899 ms	8 min 13 s 432 ms
2do Intento	15 s 391 ms	7 min 28 s 585 ms
3er Intento	16 s 12 ms	7 min 36 s 161 ms
4to Intento	16 s 502 ms	7 min 11 s 867 ms
5to Intento	27 s 944 ms	8 min 50 s 904 ms

Obtuvimos en promedio una carga de datos de 18 s con 150 ms y un mapReduce promedio de 7 min, 52 seg y 190 ms dando un total de 8 minutos, 10 segundos y 340 milisegundos.

2.2.2 Tiempos con Combiner

Se realizaron 3 tiradas de la ejecución de la consulta 1 con combiner y estos fueron los resultados:

	Carga de datos	MapReduce
1er Intento	15 s 96 ms	5 min 40 s 492 ms
2do Intento	14 s 796 ms	6 min 13 s 324 ms
3er Intento	19 s 724 ms	6 min 39 s 33 ms

Obtuvimos en promedio una carga de datos de 16 s con 539 ms y un mapReduce promedio de 6 min, 10 seg y 950 ms dando un total de 6 minutos, 27 segundos y 489 milisegundos.

2.2.3 Comparativa CON vs SIN combiner

Configuración	Carga de datos	MapReduce	Tiempo total
SIN combiner	18 s 150 ms	7 min 52 s 190 ms	~8 min 10 s
CON combiner	16 s 539 ms	6 min 10 s 950 ms	~6 min 27 s
Mejora	~1.6 segundos menos	~1 min 41 s	~1 min 43 s

El uso del combiner reduce significativamente el tiempo de MapReduce en aproximadamente 101 segundos (casi 2 minutos), representando una mejora del 21.4% en el procesamiento en la consulta 1.

2.3 Optimización

La estructura del archivo de zonas y de trips, junto con el output requerido por cada query, obligaba a hacer un mapeo de **LocationId** al nombre de la zona en algún momento de la línea de ejecución.

En consecuencia de la decisión de diseño elegida para las claves (es decir, la utilización de una clave conformada por dos Strings), el acceso al mapa distribuído de zonas se realizaba únicamente en el mapper correspondiente. Esto igualmente implicaba un aumento en los tiempos de ejecución debido a que se debe hacer el llamado por red igualmente. La implementación se muestra a continuación:

```
public class DelayPerBoroughZoneMapper implements Mapper<Integer, TripRowQ4,
String, Pair<String, Long>>, HazelcastInstanceAware {
   private transient IMap<Integer, ZonesRow> zonesMap;
   private final String desiredBorough;

   public DelayPerBoroughZoneMapper(String desiredBorough){
       this.desiredBorough = desiredBorough;
   }

   @Override
   public void map(Integer integer, TripRowQ4 tripRowQuery4, Context<String,
Pair<String, Long>> context) {
       ZonesRow PUZoneRow = zonesMap.get(tripRowQuery4.getPULocationID());
       ZonesRow DOZoneRow = zonesMap.get(tripRowQuery4.getDOLocationID());

            if(PUZoneRow != null && DOZoneRow != null &&
PUZoneRow.getBorough().compareTo(desiredBorough) == 0) {
           String PUZone = PUZoneRow.getZone();
           String DOZone = DOZoneRow.getZone();

           context.emit(PUZone, new Pair<>(DOZone,
tripRowQuery4.getDelayInSeconds()));
       }
   }

   @Override
   public void setHazelcastInstance(HazelcastInstance hazelcastInstance) {
       this.zonesMap = hazelcastInstance.getMap("zones");
   }
}
```

Para la implementación mostrada, se analizaron los tiempos de ejecución para tres corridas, probando con un subconjunto (de 2 millones de líneas) del archivo trips existente:

	Carga de datos	MapReduce
1er Intento	3 min 15 s 977 ms	1 min 10 s 999 ms
2do Intento	3 min 19 s 584 ms	1 min 9 s 974 ms
3er Intento	3 min 22 s 471 ms	1 min 9 s 629 ms

El cambio consistía entonces en evitar cargar las zonas a un mapa distribuído, utilizando un mapa en memoria para poder hacer el mapeo directamente en el cliente, y que el mapa distribuído de trips contenga los nombres de las zonas y no los ids. Se analizaron los tiempos de ejecución bajo las mismas condiciones que el anterior:

	Carga de datos	MapReduce
1er Intento	1 min 16 s 688 ms	1 s 377 ms
2do Intento	1 min 18 s 111 ms	1 s 76 ms
3er Intento	1 min 18 s 287 ms	964 ms

Se puede observar una clara mejoría para el segundo caso de implementación.

Configuración	Carga de datos	MapReduce	Tiempo total
Mapa distribuído	3 min 19 s 344 ms	1 min 10 s 200 ms	~4 min 29 s 544 ms
Mapa en memoria	1 min 17 s 695 ms	0 min 1 s 139 ms	~1 min 18 s 834 ms
Mejora	~2 min 1 s 649 ms	~1 min 9 s 61 ms	~3 min 10 s 710 ms

Decidimos entonces quedarnos con la segunda implementación, que es la que actualmente se encuentra en el trabajo. Hacer la carga de esa forma además permite hacer un pre-filtrado por las condiciones que cada query precise, definiendo una función **filter**. En el caso de la query 4 esto nos permite reducir la cantidad de líneas que el MapReduce debe procesar, ya que solo se cargan las líneas cuya zona pertenezca a el barrio indicado.

3. Potenciales puntos de mejora

Partition Awareness

Una de las mejoras que se le podría aplicar al proyecto es la utilización de de PartitionAware y la utilización de MultiMap. Esto modificaría el guardado de información de cada nodo, ya que no se guardaría a partir del hash, sino que se cargaría por la key que nosotros decidamos en cada caso.

Al hacer esto, garantizamos que todos los registros que comparten la misma clave de partición se almacenan físicamente en la misma partición y, por lo tanto, en el mismo nodo. A su vez se elimina casi por completo la fase donde los mappers envían datos por la red al reducer.

Igualmente, esto generaría un desbalance en el guardado de cada nodo, por lo que perdería eficiencia en la distribución de información. Existirían potencialmente nodos con una carga de datos mayor, aumentando a su vez la carga de procesamiento en cada nodo.

Combiner en Query 3

Actualmente, la query 3 no tiene un Combiner, ya que por la estructura de la misma, calcula el precio promedio de los viajes. Agregando un combiner, se perdería el “peso” de cada nodo, dado que en cada uno hay una cantidad de entradas distintas.
La solución sería refactorizar la query, y que el reducer no reciba el double del precio del viaje, sino que reciba un pair<double,double> con la suma de los precios y la cantidad total recibidas por el combiner. De esta forma, el reducer obtendría cada par de cada nodo, posibilitando ahi si el cálculo del promedio.

Strings en lugar de TripRowX

En lugar de usar clases separadas para cada query, en los casos de las queries 1, 3 y 5 se podría usar como clave de los Mappers strings armados ya en formato output, o sea para la query 1 sería “pickupZone;dropoffZone” y appendear el valor correspondiente de trips. Esto es debido a que transmitir strings por la red es mucho más eficiente que transmitir objetos y además se aprovecharía el ordenamiento natural de los mismos, evitando así implementar clases comparables para el Collator.

Mejora Query 1

Otra de las mejoras que podríamos implementar es mejorar la eficiencia del mapReduce de la consulta 1 para que termine en un tiempo razonable como el resto de las consultas. Actualmente, todas las veces que corremos las consultas con el csv completo no llega a terminar en un tiempo razonable. Para un csv con ⅔ de la información total la consulta termina sin problemas en un promedio de 7 minutos totales, pero cuando inyectamos el csv completo ha estado mas de 20 minutos de mapReduce sin poder terminar por alguna razón que desconocemos.