ReduceMapFast: Análisis de conceptos

Principalmente durante las primeras etapas de diseño y desarrollo, e inmersos en una vorágine de código y bugs inexplicables, solemos perder un poco el foco sobre las características de la solución que pretendemos alcanzar. El objetivo de este post es plantear una serie de características sobre las cuales se basa la solución planteada en este trabajo práctico y ver cómo los conceptos genéricos de desarrollo de software no se *escapan*, sino que más bien, *se esconden.*

Como bien nos enseñó Game of Thrones durante aquella semana en la que te viste las cuatro temporadas y te leiste los primeros tres libros, *los excesos son malos.* Y el concepto de *"divide y reinarás"* no se escapa de esta regla. Por más que la idea de encontrar pequeños problemas y modularizarlos sea algo que alentamos fervorosamente, **tenemos que recordar que cada uno de ellos es parte de un mismo objetivo común.** Si se me permite ser un poquito marketinero, podría decir que un propósito de este trabajo práctico es encontrar un nivel de *sinergia* sobre el cual todos estos módulos compartan una identidad definida.

Bajando un poquito a la práctica estos conceptos, lo que queremos decir con esto es que **la solución en sí plantea características generales.** Y basándonos en ellas sentamos un marco para poder encarar el diseño del sistema y ponemos un punto en común sobre el cual focalizar nuestro desarrollo.

Requerimientos del sistema

El enunciado del trabajo práctico nos plantea una solución ya modelada, y nos explica un poco cuáles son los componentes y qué hacen. Pero si hay una solución, entonces, *hay un problema*. En este caso particular, *nosotros necesitamos realizar análisis de grandes volúmenes de datos.*

Lo primero que necesitamos entonces, es un sistema que pueda realizar ese análisis. Como tranquilamente podríamos necesitar operar sobre varios cientos de Terabytes de datos, **nos conviene trabajar sobre un sistema distribuido, que realice el análisis en paralelo.** Y ahí nos encontramos con la Cenicienta de este cuatrimestre. Nuestra doncella tiene que poder aprovechar los beneficios del *paralelismo,* realizando *operaciones genéricas* sobre los

datos, para poder *distribuir la carga de trabajo en distintos puntos* y luego *unificar los resultados*. Pero hasta en los cuentos de Disney hay deadlines, y la princesa juega de visitante: es vital que llegue a un resultado *antes que la carroza se vuelva una calabaza, porque sino, se vuelve a casa sin un zapato -en Hollywood dijeron que el término "desaprobada", no vendía-.*

¿Cómo, entonces, aumentamos la performance de nuestro sistema? Bueno, algo contamos ahí arriba; Busquemos paralelizar:

De carroza a caravana: El camino al paralelismo

A riesgo de ganarnos la membresía de por vida del club de las tautologías, no podemos olvidar que para paralelizar, necesitamos que varios Nodos ejecuten en paralelo, operando en simultáneo y de forma asincrónica. Y aunque pueda parecer trivial, esto nos impone una restricción: **no nos puede importar el orden en el que los datos sean procesados.** Por ende, realmente nos es indistinto si el Nodo 3 opera antes que el Nodo 7, o si el Nodo 6 tiene cien tareas pendientes. Lo que nos interesa es poder contar con los resultados parciales de las operaciones entre bloques.

Esto lo logramos gracias a dos factores intrínsecos a la solución: La **inmutabilidad** de las estructuras sobre las que operamos y el **polimorfismo** en las operaciones que realizamos.

Para poder realizar operaciones de forma distribuida y *asincrónica*, necesitamos asegurarnos que si operamos los datos de un bloque, en el momento que sea, **bajo la misma operación, obtendremos los mismos resultados**. Ese nivel de **transparencia referencial** nos lo otorga el uso de un FileSystem de tipo WORM (Write Once, Read Many), en el que *la probabilidad de que se modifiquen los datos (y su metadata asociada) es tan, pero tan baja, que resulta despreciable.* Siendo extremadamente honestos, no somos los primeros en resolver problemas de transparencia referencial usando estructuras inmutables, sino que es un patrón muy común en sistemas que buscan aumentar su capacidad de respuesta (sensibilidad), escalables, con alta resistencia a fallos, y con necesidad de soportar alta tasa de procesamiento. Bah, reactivos, le dicen los hippies.

Otra cosa que aporta a la inmutabilidad del sistema es que *las operaciones de Map y Reduce no modifican los datos existentes, sino que generan y almacenan nuevos datos en otros bloques.*

Y ya que hablamos de las operaciones, podemos ver que estas tienen *una interfaz muy bien definida*. De ahí sale que nuestras operaciones *son por naturaleza, polimórficas.* Realmente

no sabemos si el Map toma patitos y los transforma en reactores nucleares, sólo nos interesa que los patitos entren por STDIN, y los reactores nucleares salgan por STDOUT.

Otra característica interesante que puede destacarse de las operaciones, puntualmente en Reduce, es su naturaleza **recursiva.** Como explica la especificación de la solución, vamos a estar utilizando Reduce sobre resultados que ya previamente fueron reducidos. Y eso, como mínimo, es bastante curioso.

Teniendo una carroza un poco más equipada, ahora ya con turbo y frenos ABS, podemos invitar a un par de amigos para que nos den una mano, y lo que antes era un único medio de transporte, de a poco se convierte en una caravana. *Cenicienta, chocha.*

Llegando antes de medianoche: Sincronización e inmutabilidad

Otra característica que viene casi regalada con la inmutabilidad del sistema, es que **nos simplifica enormemente la necesidad de sincronizar datos.** Como bien nos plantea Bernstein, existen ciertas condiciones para poder identificar la necesidad de sincronizar. *Y todas ellas se basan en la modificación de los datos.* Si los datos no son modificados, no hay necesidad de sincronizar el acceso a los mismos.

Esto para nuestra princesa significa que **la lectura a los bloques de datos no tiene necesidad de ser sincronizada**, y *todas las operaciones Map y Reduce que se realicen, pueden ser hechas en paralelo, sin necesidad de que sean sincronizadas.* Eso sí, **todavía existen algunos lugares en donde la sincronización es necesaria**, como las *estructuras administrativas internas del FileSystem,* pero eso ya no tiene forma de solucionarse, porque *para guardar un bloque, el FS tiene que mutar.*

Como vemos, la inmutabilidad del sistema nos regala, aparte de la posibilidad del paralelismo, una **sincronización implícita**, eliminando impedimentos que de otra forma tendríamos que encarar.

Del cuento al papel: Cerrando conceptos

De a poco, fuimos cubriendo algunos conceptos fundamentales que hacen que nuestro sistema pueda funcionar y devolver un resultado sin volverse una calabaza. Vimos la importancia y necesidad de la **transparencia referencial** al trabajar con sistemas **distribuidos** y **asincrónicos**. También analizamos la **inmutabilidad** como método para

poder solucionar este problema, y usamos operaciones **polimórficas** y **recursivas** para poder obtener un resultado final con buena **performance.**

Seguramente podríamos hacer un análisis mucho más profundo de todos estos conceptos, agregando otros que dejamos de lado, e interrelacionándolos mucho más. Pero la idea de este post es poder generar una base conceptual sobre la cual enmarcarse para *poder perseguir un objetivo común en la implementación del TP.*

Confiamos en que *comprender e incorporar los conceptos* es un incentivo a un desarrollo a conciencia, fomenta la motivación a realizarlo y nos permite divertirnos un poco en el proceso. *“Divertido es Disney”,* dicen grandes pensadores del Siglo XXI, pero como todo esto *lo hacemos por la Cenicienta,* supongo que está bien.