**Evidencia #3:**

**Implementación de un simulador de almacén automatizado**

**(segunda parte)**

**Diego Andrés Figueroa Peart**

A01660987

**Curso:**

Implementación de métodos computacionales

(TC2037.820)

**Profesores:**

Román Martínez Martínez

Alberto Oliart Ros

Valentina Narváez Terán

Germán Domínguez Solís

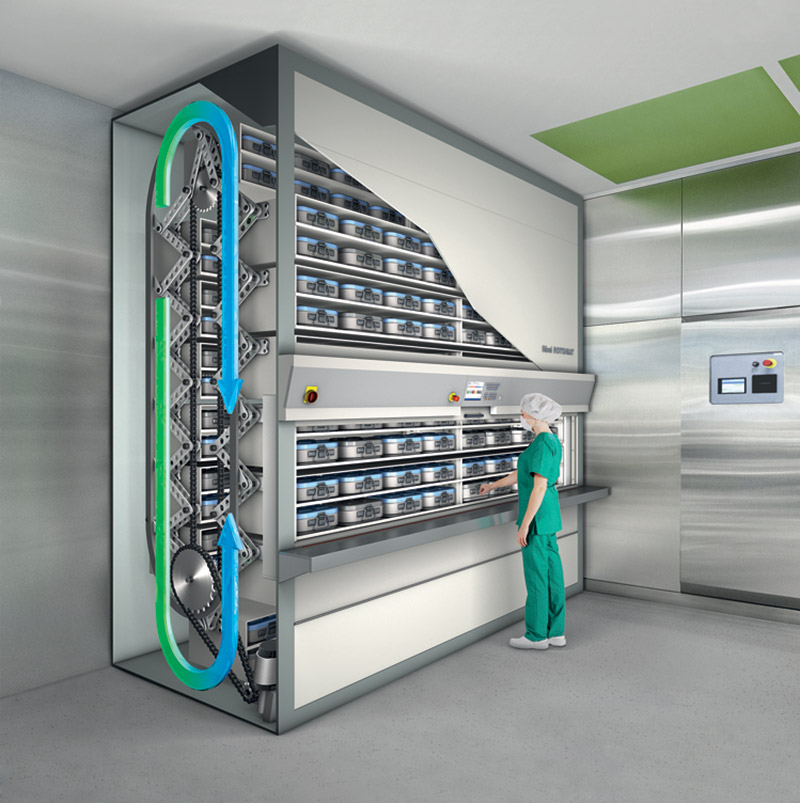
**Fecha de entrega:**

13 de junio de 2023

**Evidencia #3: Carrusel vertical paralelo**

En este tercer periodo, la situación problema consistió en modificar la entrega de la segunda evidencia para que esta funcione en el lenguaje de programación funcional Clojure, un lenguaje basado en Lisp pero construido en la plataforma de Java. Esto le permite utilizar los módulos existentes de Java pero con las ventajas de los paradigmas funcional y concurrente.

Recordando la síntesis de la situación problema del segundo periodo, esta consiste en elaborar un simulador de carrusel vertical de almacenamiento, el cual consiste en la siguiente estructura:



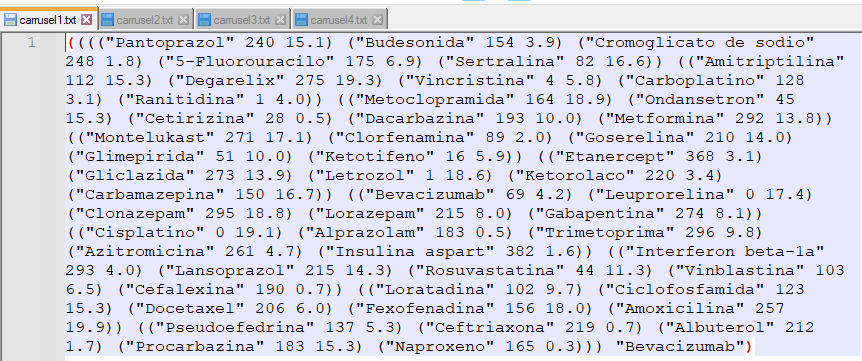
Ahora bien, si el simulador entregado como segunda evidencia de este curso funcionaba para simular un solo carrusel y realizar una lista de transacciones sobre él, ahora en Clojure, gracias a sus capacidades de utilizar el paradigma concurrente/paralelo, esta nueva versión del simulador podrá realizar varias listas de transacciones sobre distintos carruseles simultáneamente.

**Diseño de las estructuras de datos**

Al igual que en la entrega pasada, los carruseles tendrán la siguiente estructura utilizando listas anidadas para representar las filas y los productos individuales:

(((producto 1 cantidad precio) (producto 2 cantidad precio) (producto 3 cantidad precio)) ((producto 4 cantidad precio) (producto 5 cantidad precio) (producto 6 cantidad precio)) ((producto 7 cantidad precio) (producto 8 cantidad precio) (producto 9 cantidad precio)))

En esta ocasión, los carruseles serán almacenados en archivos de texto, uno para cada carrusel, los cuales tendrán los nombres “carrusel#.txt”, en donde # es el número indicador del carrusel. Estos estarán elaborados mediante un programa adicional, que generará combinaciones aleatorias de productos con precios y cantidades aleatorias también:



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

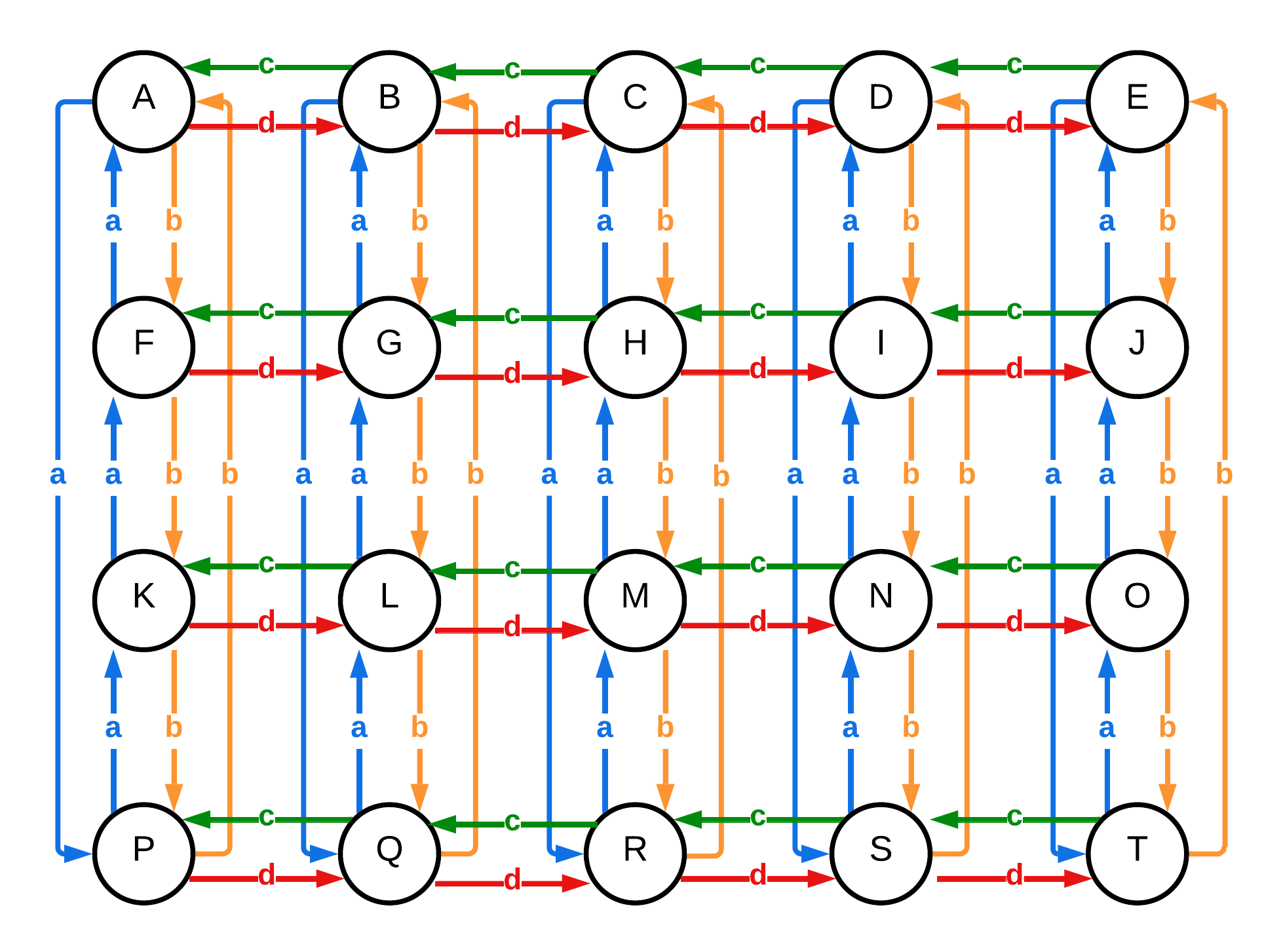
Los estados actuales de cada carrusel estarán adicionados a su respectivo archivo como el segundo elemento en una lista, el primero siendo el carrusel en si.

El movimiento del carrusel utilizará el mismo lenguaje que la entrega pasada; las funciones de movimiento serán las siguientes:

* a, hacia arriba.
* b, hacia abajo.
* c, hacia la izquierda.
* d, hacia la derecha

**Diseño del autómata**

La representación visual del automata que rige el comportamiento del carrusel es el mismo, sin embargo, ahora se ejecutarán múltiples versiones de este mismo al mismo tiempo para cada carrusel:



Las transacciones a realizar en cada carrusel serán almacenadas también en archivos únicos para cada carrusel, y también serán generadas automáticamente. Los nombres de los archivos serán “transacciones#.txt”, donde # corresponderá al número identificador del carrusel al que se le aplicarán las transacciones. Las transacciones posibles seguirán siendo las siguientes:

* **(a) –** El carrusel se moverá hacia arriba.
* **(b) –** El carrusel se moverá hacia abajo.
* **(c) –** El carrusel se moverá hacia la izquierda.
* **(d) –** El carrusel se moverá hacia la derecha.
* **(retirar cantidad {producto}) –** Si no se especifica producto, se retirará la cantidad del producto actualmente mostrado. Si se especifica un producto existente, el carrusel se moverá a la posición del producto especificado, y mostrará la ruta más eficiente para llegar a esta posición. Si se intentan retirar más de las existencias del producto, se retirará todo y se marcará un error.
* **(agregar cantidad {producto}) –** Al igual que la función retirar, si no se especifica un producto se agregará al producto actual, y si sí, se moverá el carrusel a la posición del producto utilizando la ruta más eficiente.

Ejemplos de archivos de transacciones:

A screenshot of a computer

Description automatically generated with medium confidence

Al finalizar la ejecución de todas las transacciones en todos los carruseles, se desplegarán en pantalla dos datos; el valor total de todo el inventario en todas los carruseles (calculado multiplicando todos los precios unitarios por las cantidades de los productos), y el top 10% de carruseles con mayor valor de inventario.

A picture containing text, screenshot, font

Description automatically generated

Adicionalmente, por cada transaccion que se realice en cada carrusel, se añadirá a un archivo “log#.txt” que almacenará todas las transacciones que se realicen en su respectivo carrusel, y finalmente se desplegarán los productos bajos en inventario (menores a 20 unidades) del respectivo carrusel.

A picture containing text, screenshot

Description automatically generated

**Mutabilidad de los datos**

Al igual que la entrega anterior, el simulador de carruseles es completamente dinámico y adaptable automáticamente a cualquier tamaño de carrusel; la única variable que el usuario deberá ajustar antes de ejecutarlo es el número de carruseles que se tienen. Fuera de esto, los carruseles pueden contener distintos productos, número de filas, número de columnas, etc.; mientras no varíe el número de columnas dentro del carrusel, y siga la estructura de datos anteriormente detallada, el código lo manejará sin problemas.

**Ventajas y desventajas**

La ventaja principal que se me ocurre al haber desarrollado este proyecto en un lenguaje de los paradigmas funcional y concurente/paralelo, como lo es Clojure, es la eficiencia en memoria y rapidez con la que es posible manejar grandes cantidades de datos gracias a la paralelización. Otra de las grandes ventajas de Clojure es que está construido sobre la plataforma de Java, permietiendole así utilizar el extenso arsenal de módulos y librerías que existen para este lenguaje.

Sin embargo, el sintaxis poco convencional de los lenguajes funcionales, combinado con su dificultad de instalación y ejecución de programas, especialmente en plataformas como Windows, me hace dudar de su practicidad en el campo real, y honestamente me encuentro poco seguro de volver a utilizar este lenguaje más allá de lo académico.

**Código**

; Evidencia #3

; Implementación de un simulador de un almacén automatizado (segunda parte)

; Diego Andrés Figueroa Peart A01660987

; Se establecen los parámetros básicos de cada carrusel.

(defn carrusel [n]

(first (read-string (slurp (str "carrusel" n ".txt")))))

(defn est-actual [n]

(second (read-string (slurp (str "carrusel" n ".txt")))))

(defn filas [n]

(count (carrusel n)))

(defn columnas [n]

(count (first (carrusel n))))

; Se guardan los cambios al carrusel.

(defn escribir [n txt]

(spit (str "carrusel" n ".txt") txt))

; El log de las transacciones se almacenará por carrusel.

(defn log [n txt]

(spit (str "log" n ".txt") (str txt "\n") :append true))

(defn coordenadas [lst col x y f]

(if (empty? lst) f

(if (<= y col)

(recur (rest lst) col x (+ y 1) (concat f (list (list (first (first lst)) (list x y)))))

(recur lst col (+ x 1) 1 f))))

(defn diccionario [n]

(coordenadas (apply concat (carrusel n)) (columnas n) 1 1 '()))

; Se busca un producto a partir de su nombre, se regresan sus coordenadas.

(defn buscar [producto n]

(let [x (filter #(= producto (first %)) (diccionario n))]

(if (nil? x) false

(second (first x)))))

; Se reciben las coordenadas del producto y se regresa su nombre, cantidad y precio.

(defn detalles-aux [lst y]

(if (not (= 1 y))

(recur (rest lst) (dec y))

(first lst)))

(defn detalles [lst x y]

(if (not (= 1 x))

(recur (rest lst) (dec x) y)

(detalles-aux (first lst) y)))

; Se cambia la cantidad del producto en el carrusel para luego sobreescribir el archivo.

(defn reemplazar-y [lst y r f]

(cond

(empty? lst) f

(= y 1) (recur (rest lst) (dec y) r (concat f (list r)))

:else (recur (rest lst) (dec y) r (concat f (list (first lst))))))

(defn reemplazar-x [lst x y r f]

(cond

(empty? lst) f

(= 1 x) (recur (rest lst) (dec x) y r (concat f (list (reemplazar-y (first lst) y r '()))))

:else (recur (rest lst) (dec x) y r (concat f (list (first lst))))))

(defn reemplazar [n x y r]

(reemplazar-x (carrusel n) x y r '()))

(defn arriba [x1 x2 lst filas]

(if (= x1 x2) lst

(if (= x1 1)

(recur filas x2 (concat lst (list 'a)) filas)

(recur (- x1 1) x2 (concat lst (list 'a)) filas))))

(defn abajo [x1 x2 lst filas]

(if (= x1 x2) lst

(if (= x1 filas)

(recur 1 x2 (concat lst (list 'b)) filas)

(recur (+ x1 1) x2 (concat lst (list 'b)) filas))))

; Se calcula la distancia entre el producto actual y el producto al que se desea llegar, se determina la ruta más rápida.

(defn distancia-y [y1 y2 lst]

(if (= y1 y2) lst

(if (> y1 y2)

(recur (- y1 1) y2 (concat lst (list 'c)))

(recur (+ y1 1) y2 (concat lst (list 'd))))))

(defn distancia [x1 y1 x2 y2 lst filas]

(cond

(< x1 x2) (if (> (- x2 x1) (/ filas 2))

(recur 0 y1 0 y2 (arriba x1 x2 lst filas) filas)

(recur 0 y1 0 y2 (abajo x1 x2 lst filas) filas))

(> x1 x2) (if (> (- x1 x2) (/ filas 2))

(recur 0 y1 0 y2 (abajo x1 x2 lst filas) filas)

(recur 0 y1 0 y2 (arriba x1 x2 lst filas) filas))

:else (distancia-y y1 y2 lst)))

; Se regresan los productos con menos de 20 unidades.

(defn productos-bajos [n]

(let [prods (filter #(> 20 (second %)) (apply concat (carrusel n)))]

(log n (str "Productos con menos de 20 unidades: " (list prods)))))

(defn valor-total [n]

(apply + (map #(\* (second %) (last %)) (apply concat (carrusel n)))))

; Movimiento hacia arriba.

(defn a [n]

(let [coords (buscar (est-actual n) n)]

(if (zero? (dec (first coords)))

(do

(escribir n (list (carrusel n) (first (detalles (carrusel n) (filas n) (second coords)))))

(log n (detalles (carrusel n) (filas n) (second coords))))

(do

(escribir n (list (carrusel n) (first (detalles (carrusel n) (dec (first coords)) (second coords)))))

(log n (detalles (carrusel n) (dec (first coords)) (second coords)))))))

; Movimiento hacia abajo

(defn b [n]

(let [coords (buscar (est-actual n) n)]

(if (< (filas n) (inc (first coords)))

(do

(escribir n (list (carrusel n) (first (detalles (carrusel n) 1 (second coords)))))

(log n (detalles (carrusel n) 1 (second coords))))

(do

(escribir n (list (carrusel n) (first (detalles (carrusel n) (inc (first coords)) (second coords)))))

(log n (detalles (carrusel n) (inc (first coords)) (second coords)))))))

; Movimiento hacia la izquierda.

(defn c [n]

(let [coords (buscar (est-actual n) n)]

(if (zero? (dec (second coords)))

(log n "Movimiento inválido.")

(do

(escribir n (list (carrusel n) (first (detalles (carrusel n) (first coords) (dec (second coords))))))

(log n (detalles (carrusel n) (first coords) (dec (second coords))))))))

; Movimiento hacia la derecha

(defn d [n]

(let [coords (buscar (est-actual n) n)]

(if (< (columnas n) (inc (second coords)))

(log n "Movimiento inválido.")

(do

(escribir n (list (carrusel n) (first (detalles (carrusel n) (first coords) (inc (second coords))))))

(log n (detalles (carrusel n) (first coords) (inc (second coords))))))))

(defn agregar

([cantidad n] ; Se agrega la cantidad al producto que se muestra actualmente.

(let [coords (buscar (est-actual n) n)

prod (detalles (carrusel n) (first coords) (second coords))

reem (list (first prod) (+ (second prod) cantidad) (last prod))]

(escribir n (list (reemplazar n (first coords) (second coords) reem) (est-actual n)))

(log n (str "Agregadas " cantidad " unidades de " (first prod) "."))))

([cantidad producto n] ; Se agrega la cantidad al producto especificado.

(let [origen (buscar (est-actual n) n)

destino (buscar producto n)]

(if (not destino)

(log n (str "Producto " producto " no encontrado."))

(let [prod (detalles (carrusel n) (first destino) (second destino))

reem (list (first prod) (+ (second prod) cantidad) (last prod))]

(escribir n (list (reemplazar n (first destino) (second destino) reem) producto))

(log n (str "Agregadas " cantidad " unidades de " (str (first prod)) "."))

(if (not (= origen destino))

(log n (str "Ruta: " (apply str (distancia (first origen) (second origen) (first destino) (second destino) '() (filas n)))))))))))

(defn retirar

([cantidad n] ; Se retira la cantidad del producto que se muestra actualmente.

(let [coords (buscar (est-actual n) n)

prod (detalles (carrusel n) (first coords) (second coords))

reem (list (first prod) (- (second prod) cantidad) (last prod))]

(if (> 0 (second reem))

(do

(escribir n (list (reemplazar n (first coords) (second coords) (list (first reem) 0 (last reem))) (est-actual n)))

(log n (str "Se intentaron retirar más de las existencias de " (first prod) ".")))

(do

(escribir n (list (reemplazar n (first coords) (second coords) reem) (est-actual n)))

(log n (str "Retiradas " cantidad " unidades de " (first prod) "."))))))

([cantidad producto n] ; Se retira la cantidad del producto especificado.

(let [origen (buscar (est-actual n) n)

destino (buscar producto n)]

(if (not destino)

(log n (str "Producto " producto " no encontrado."))

(let [prod (detalles (carrusel n) (first destino) (second destino))

reem (list (first prod) (- (second prod) cantidad) (last prod))]

(if (> 0 (second reem))

(do

(escribir n (list (reemplazar n (first destino) (second destino) (list (first reem) 0 (last reem))) producto))

(log n (str "Se intentaron retirar más de las existencias de " (first prod) ".")))

(do

(escribir n (list (reemplazar n (first destino) (second destino) reem) producto))

(log n (str "Retiradas " cantidad " unidades de " (first prod) "."))

(if (not (= origen destino))

(log n (str "Ruta: " (apply str (distancia (first origen) (second origen) (first destino) (second destino) '() (filas n)))))))))))))

; Se leerán y ejecutarán las transacciones línea por línea.

(defn main-aux [n transacciones]

(eval (concat (first transacciones) (list n)))

(if (not (empty? (rest transacciones)))

(recur n (rest transacciones))))

(defn main-p [n]

(main-aux n (read-string (slurp (str "transacciones" n ".txt")))))

(def num\_carruseles 50) ; El número de carruseles a procesar es configurable aquí.

(defn main []

(pmap #(main-p %) (range 1 (inc num\_carruseles))) ; Se procesan los carruseles de forma paralela.

(Thread/sleep 500)

(pmap #(productos-bajos %) (range 1 (inc num\_carruseles))) ; Se añade al log los productos con menos de 20 unidades.

(Thread/sleep 500)

(println "Valor total del inventario de todos los carruseles:" (apply + (map #(valor-total %) (range 1 (inc num\_carruseles))))) ; Se calcula el valor total del inventario de todos los carruseles.

(println "Top 10% de carruseles con mayor valor de inventario:" (take (/ num\_carruseles 10) (reverse (sort-by second (map #(list (str "Carrusel " %) (valor-total %)) (range 1 (inc num\_carruseles))))))))

(main)

**Pruebas realizadas**

Para demostrar la efectividad de la paralelización en Clojure, realicé dos pruebas generando 30 carruseles aleatorios y 30 listas de transacciones aleatorias también; una utilizando pmap y la otra con un map regular, midiendo el tiempo que se tomó cada una. Me aseguré que todos los carruseles se encontraran en exactamente el mismo estado inicial para que las condiciones de ejecución de ambas funciones fueran idénticas. Estos fueron los resultados, el primero con map y el segundo con pmap:

A picture containing text, screenshot, font

Description automatically generated

Aquí expongo algunos de los carruseles en esta prueba, sus respectivas listas de transacciones y sus archivos log:

Carrusel 3:

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

Carrusel 15:

A picture containing text, screenshot, font, software

Description automatically generated

A screenshot of a computer program

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with medium confidence

Carrusel 28:

A screen shot of a computer program

Description automatically generated with low confidence

A picture containing text, font, software, screenshot

Description automatically generated

A screenshot of a computer program

Description automatically generated with medium confidence

**Experiencia de aprendizaje**

Honestamente, esta evidencia me resultó increíblemente difícil de completar; tuve problemas en todos lados, desde instalar y correr Clojure en mi computadora, probar mi código en Replit, lograr que funcionara su integración con Visual Studio Code y Calva/Leiningen, y finalmente la lógica en sí de programar en paralelo. Fue un trabajo arduo y complejo pero me siento increíblemente orgulloso de mi mismo por haber logrado perseverar y solucionar la evidencia de forma completa.