Ejercicio 1

La ejecución del script se realiza mediante el comando:
node script.js

La ejecución de los test:
npm install
npm run test

En lugar de encadenar múltiples llamadas a .filter(), se implementa un índice de acceso rápido con la estructura stockIndex[key][zona]. Esto permite realizar búsquedas instantáneas con una complejidad de O(1), mejorando significativamente el rendimiento del proceso.

Ejercicio 2

Para evitar problemas de rendimiento, no conviene cargar grandes volúmenes de datos directamente en el entorno low-code. Lo ideal es usar paginación, carga bajo demanda y filtrado desde el backend.

La API debería tener un endpoint que devuelva una tabla con los atributos de key, idTienda, propuesta, tipoStockDesc, estadoStock y posicionCompleta. También debería soportar paginación, busqueda, ordenamiento y filtrado por estos campos, para integrarse bien con componentes dinámicos como las tablas interactivas.

Ejercicio 3

Si el JSON de prereparto fuera de 20GB, no sería viable seguir usando JSON.parse() para cargar todo en memoria o guardar el resultado en un array, ya que eso podría superar los límites de RAM y hacer fallar la aplicación. En ese caso, cambiaría el enfoque a uno basado en streaming, utilizando librerías como stream-json en Node.js, que permiten procesar los datos línea por línea sin cargar todo el archivo. Otra opción sería importar los datos en una base de datos y realizar el procesamiento mediante consultas SQL. Ambas soluciones permiten escalar el procesamiento manteniendo un uso de memoria bajo.

Ejercicio 4

Se puede agregar un desplegable para mostrar un esquema simplificado del almacén, que aparecería al lado de la tabla de datos. Al seleccionar una fila en la tabla, se marcaría con un punto la ubicación del artículo correspondiente en el esquema.

Además, cada tipo de zona puede tener un color distinto según el origen del producto (ZAR, MSR o SILO), lo que ayuda a identificar rápidamente de dónde viene cada artículo.