DOCUMENTAÇÃO

Teste prático - Projeto Classificatório Rocky

Objetivo do algoritmo:

A partir de um banco de dados NoSQL corrompido, *“broken-database.json”,* espera-se que possamos recuperar os dados perdidos, corrigindo o nome, preço e quantidade dos produtos. Além disso é necessário que validemos se a recuperação foi sucedida por meio de algumas etapas de ordenação e contagem dos dados.

Escolha da linguagem:

A linguagem escolhida foi o JavaScript, o motivo principal foi por já ter uma certa experiência com a mesma, o que facilitou o desenvolvimento do código.

Funcionalidades:

No desenvolvimento do algoritmo, foram criadas algumas funções para restaurar os dados corrompidos, sendo elas:

readJson: recebe o caminho absoluto do ***arquivo*** json e com o auxílio do módulo ***fs***, lê-se o mesmo e o retorna em forma de ***objeto***.

fixName: recebe os dados, o varre substituindo todas as ocorrências dos caracteres corrompidos de cada item pelos caracteres corretos por meio do método ***split*** (separa a string em pedaços pela string de busca) e ***join*** (junta os pedaços colocando a string de substituição entre eles).

fixPrice: recebe os dados, o varre convertendo o campo ***price*** de cada item para number por meio do método ***parseFloat*** caso o mesmo esteja em qualquer outro tipo de variável (a verificação é feita a partir do retorno do método ***typeof***).

fixQuantity: recebe os dados e o varre verificando se o campo ***quantity*** de cada item existe por meio do método ***hasOwnProperty***, caso não exista o mesmo é atribuído com o valor 0.

exportJson: recebe os dados a serem exportados e o nome que o arquivo terá (caminho absoluto). O objeto contendo os dados é convertido para uma string json serializável indentada (para poder ser salvo em disco) e em seguida por meio do módulo ***fs*** é armazenado no disco no caminho especificado.

sortData: recebe os dados, e primeiro os itens são ordenados pelo campo ***category*** em ordem alfabética, para isso, é comparado o valor da categoria de dois itens por vez (a e b), onde esse valor é convertido para letras minúsculas, pois o ***ascii*** de letras maiúsculas e minúsculas são diferentes o que causaria diferença na ordenação. O resultado dessa comparação é dividido em três valores, -1 para a < b, 1 para a > b e 0 para a = b e é retornado para o método ***sort*****,** que é responsável por realizar a ordenação em si dos itens de acordo com o valor passado. Após ordenar pela categoria, os itens de categorias iguais são organizados em ordem crescente pelo mesmo processo anterior. Finalizada as duas ordenações é mostrado os nomes dos itens ordenados.

quantityByCategory: recebe os dados (que precisam estar ordenados pela função ***sortData***), é declarado um vetor de objetos chamado ***stock*** que guarda a quantidade de itens por categoria, seu iterador ***stockIt*** (inicia em -1) e uma variável auxiliar ***categoryBuffer*** (inicia vazia) que guarda a categoria do item anterior do iterador principal. É analisado se a categoria do item atual é diferente do valor de ***categoryBuffer***, caso seja, é criado um objeto em ***stock*** na posição ***stockIt*** com o nome da categoria atual e com a quantidade zerada e ***stockIt*** é iterado em 1. Em seguida é atualizado o campo ***quantity*** do ***stock*** com a quantidade e o ***categoryBuffer*** com a categoria do item que está sendo verificado. O processo se repete para todos os itens do objeto de dados e no final retorna ***stock*** com as somas por categorias.

Tratamentos de erros:

Nas funções ***readJson*** e ***exportJson*** a adição de ***try catch*** foi uma estratégia para evitar que caso o caminho absoluto esteja incorreto ou não exista, o algoritmo não dê erro e pare de funcionar. Para as duas funções caso a operação tenha sucesso, é retornado 1 e para falha 0. Na utilização das mesmas esse retorno é usado para decidir se o banco de dados vai passar pela etapa de conserto e validação.

Nota adicional de complexidade:

Pelo fato do teste exigir a ordenação dos dados por categoria no passo 2. a), a complexidade da função ***quantityByCategory*** do passo 2. b) reduziu de algo próximo de ***O(quantidade-de-categorias * n)*** para ***O(n)*** pois os itens já estavam agrupados por categoria, não sendo necessário para cada item analisado, verificar se existem mais itens de tal categoria para somar, precisando somente armazenar a categoria anterior ao iterador principal para certificar que já foram analisados todos os produtos daquela categoria. Caso não fosse pedido a ordenação, para cada produto iria ser requerido buscar em ***stock*** se aquela categoria já existe, ou seja, seriam aproximadamente ***quantidade-de-produtos * quantidade-de-categorias*** iterações, que apesar de continuar linear seria uma perda de desempenho.