Relatório 2° Trabalho de implementação
Banco de Dados
Professor: Eduardo Almeida

Aluno: Erick Eckermann Cardoso GRR20186075

1. **Considerações gerais**

O trabalho implementa os algoritmos de teste de seriabilidade quanto ao conflito e de visão equivalente para transações em um banco de dados. Foi utilizada a linguagem de programação C para implementar toda a lógica e estruturas do algoritmo.

O código foi dividido em três bibliotecas e um arquivo fonte que contém a **main**, nomeado **find_conflicts**. As três bibliotecas criadas se chamam **Seriabilidade** (arquivo .h e .c), **VisaoEquivalente** (arquivo .h e .c) e **utils** (arquivo .h e .c). A biblioteca **Seriabilidade** guarda as funções necessárias para manipular o grafo utilizado no algoritmo de detecção de conflitos e as demais funções para o algoritmo. A biblioteca **VisaoEquivalente** armazena as funções para executar o algoritmo de visão equivalente. A biblioteca **utils** armazena as estruturas de dados utilizadas e as respectivas funções para manipulá-las.

2. **Especificidades**

Nesta seção irei abordar todos os pontos específicos do trabalho. Não despender tempo em detalhes minimalistas da lógica do código, ou em como são feitas as leituras.

2.1 Das estruturas de dados implementadas

Foram implementadas 4 estruturas no total. As operações do escalonamento são armazenadas em uma lista encadeada simples, ordenada pelo timestamp, onde cada nodo representa uma operação e armazena as características da operação(Timestamp, número da transação, operação e atributo)

Para representar o grafo foram necessárias duas estruturas, uma estrutura que representa o nodo(nodoT) que armazena uma transação e outra que representa uma lista de nodos(listaNodoT), implementada como uma lista encadeada simples. A nodoT armazena o número da transação e um ponteiro para uma lista de nodos, que representam as arestas. O grafo é representado como uma lista de nodos em que cada nodo é uma transação lida na entrada. Assim fica fácil de navegar pelo grafo, já que a partir do ponteiro grafo você pode acessar suas arestas e ir para outro nodo, que também terá suas arestas e assim por diante.

A outra estrutura representa um par Write Read para o algoritmo de visão equivalente. Ela armazena somente o timestamp da operação W e R respectivamente, ja que o timestamp indica uma operação única.

2.2 Da leitura das operações

Para a leitura das operações foi utilizado um vetor de commits que indica se todas as transações que foram lidas até então foram commitadas. Se não forem, ele lê, armazena no nodo de operaçẽos e inclui na lista. Para a leitura, eu assumi que o arquivo de entrada está ordenado por timestamp. Quando todas tiverem sido commitadas, então o escalonamento está pronto e armazenado na lista de operações, pronto para executar os algoritmos de detecção de conflito e visão equivalente.

2.3 Do algoritmo de detecção de ciclo no grafo

Para o algoritmo de reconhecimento de ciclos no grafo foi criado a função principal **buscaCicloGrafo**, que inicializa um array chamado pre e post para manter o registro da ordem de visitação dos nós. Em seguida, itera através de todos os nós do grafo e chama **DFSNodo** para cada nó que ainda não foi visitado.

A função **DFSNodo** realiza uma busca em profundidade (DFS) a partir do nó de entrada e atualiza o array pre com a contagem atual para manter o registro da ordem de visitação dos nós. Para cada aresta conectada ao nó, ela realiza uma chamada de DFS no nó de destino. Se o nó de destino ainda não foi visitado, a função chama a si mesma com o nó de destino como argumento. Se o nó de destino já foi visitado e o valor do seu post não foi definido, então ele retorna 1, indicando que há um ciclo no grafo. O valor do post de um nó é definido quando a busca DFS sobre esse nó é concluída, então se um nó já foi visitado e o valor do seu post não está definido, isso significa que há um ciclo.

2.4 Do algoritmo de visão equivalente

Utilizei a estrutura de dados par_wr que armazena os timestamps da operação write e read que forma o par do escalonamento original. Utiliza os timestamps para determinar se, no escalonamento novo, o par se manteve e a ordem de execução deles não foi alterada. Isso é possível de determinar somente com o timestamp pois ele representa uma operação única do escalonamento original, portanto a ordem desses pares precisam se manter no novo escalonamento.

Para realizar as permutações, eu utilizei um vetor de inteiros que armazena o número de cada transação. Esse vetor então é escalonado em todas as opções possíveis. Em cada escalonamento uma lista nova de operações é criada baseada na ordem em que o vetor está no escalonamento atual. A lista original e a lista nova então são passadas como referência para a função que confere se os dois escalonamentos são equivalentes.