Relatório
Trabalho 2 de Banco de Dados

GRR20190427 - João Lucas Cordeiro
GRR20196049 - Iago Mello Floriano

1 Introdução

Este é um relatório do segundo trabalho de Banco de Dados, onde implementamos um código que, dada uma entrada com alguns escalonamentos, determina se cada escalonamento é serializável ou não, tanto por conlfito, quanto por visão equivalente.

2 Leitura da Entrada

Nessa parte do algoritmo, usamos 4 estruturas: *activeTasks*, *currTasks*, *vars* e *varsIndex*.

O *activeTasks* é um conjunto com as transações do escalonamento atual que não deram commit; Se esse cojunto está vazio, o escalonamento acabou. O *currTasks* é um conjunto com as transações do escalonamento atual, ou seja, os vértices que serão usados nos algoritmos do teste. Já o *vars* é um vetor que guarda, em cada índice, o nome das variáveis e as operações realizadas nela. As operações são uma tupla da forma (*tempo*,*transação*,*tipo*). Enfim temos o *varsIndex*, um hashmap usado para facilitar as alterações no vetor vars.

3 Teste por conflito

O algoritmo que testa a serialização por conflito funciona assim:

Começamos criando um grafo vazio, *newg*. Então, para cada tarefa em *tasks*, adicionamos um nodo com o *id* igual ao *id* da tarefa no grafo *newg*. Daí, para cada variável em *vars*, passamos por todos os pares possíveis de operações naquela variável e as chamaremos de operações i e j. Testamos então se o tempo da operação i é menor que o tempo da operação j **e**, ou i é uma escrita e j também, ou uma das duas é uma escrita e a outra uma leitura. Se sim, adicionamos uma aresta da transação i para a transação j.

Depois de criarmos esse grafo, testamos se ele possui uma ordenação topológica. Se sim, o escalonamento é serializável por conflito, caso contrário, não é.

4 Teste de visão equivalente

O algoritmo que testa a serialização por visão equivalente funciona assim:

Começamos adicionando as tarefas T0 ($id == 0$) e a Tf ($id == -1$) na variável *tasks*. Então, para cada variável em *vars*, adicionamos duas operações: uma escrita feita pela tarefa T0 no tempo 0 e uma leitura feita pela Tf feito no *maior tempo + 1*. Criamos então o *vetorPares* que guardará pares de arestas e o *newg*, um grafo vazio. Assim como no teste por conflito, criamos um nodo para cada tarefa, dessa vez incluindo a T0 e a Tf.

Então, para cada variável em *vars*, passamos em cada par de operações possíveis nessa variável e as chamaremos de operações i e j. Se j é uma operação de leitura, i é uma de escrita e não há outra operação de escrita entre as duas, fazemos: adicionamos uma aresta do nodo i ao nodo j e, se existe uma operação k que também escreve na variável, fazemos: se a tarefa i é a T0, adicionamos uma aresta do nodo j ao k, se a tarefa j é a Tf, adicionamos uma aresta do nodo k ao i, e se não for nenhum desses casos, adicionamos este par de arestas ao *vetorPares*.

Depois disso, escolhemos um dos elementos (arestas) do par de cada índice do *vetorPares*. Então, adicionamos essas arestas ao grafo. Enfim, testamos se o grafo possui uma ordenação topológica. Caso não, removemos as arestas e tentamos com outra possibilidade de escolhas de arestas. Caso **alguma** das possibilidades possua uma ordenação topológica, o escalonamento é serializável por visão equivalente, caso contrário, não.

5 Teste de Ordenação Topológica

O algoritmo que testa a existência de ordenação topológica funciona assim:

Crie dois vetores vazios: S e L. Adicione todos os nodos com nenhuma aresta incidente em S. Enquanto S não for vazio, escolha um nodo n, o remova de S e o adicione em L. Então, para todo nodo v onde há uma aresta de n para v, remova esta aresta. Se v agora não possuir arestas incidentes, adicione v em S.

Depois disso, se L possuir número de elementos igual ao número de vértices no grafo, então ele possui uma ordenação topológica, caso contrário, ele não possui.