Skip List Concurrente

Pablo Yucra, *Pregrado UCSP*

Abstract—En este articulo explicaremos una estructura de datos concurrente como es el "lista de saltos concurrente", primeramente debemos entender que un lista de saltos es una estructura muy similar a una lista enlazada, la diferencia podría ser si nosotros imaginamos una autopista con muchos carriles, donde hay carriles de velocidad variable, para estos entonces llegaremos a recorrer más rápida la lista en caso de hacer operaciones ,sin embargo no tendremos todos los nodos, ya que nos saltearemos varios, y mientras vamos bajando a los carriles de menos velocidad tendremos más nodos, podríamos decir que la lista por si sola seria el carril 0 donde encontraremos todos los nodos, entonces teniendo en cuenta el concepto de lista de saltos, ahora podemos imaginar como hacer una en paralelo, para esto tenemos varias propuestas y para hacer esto mas sencillo de entender la idea es simplemente a trabajar un hilo por cada carril, aunque esto es más sencillo explicarlo en código. Entonces el algoritmo que presentaremos se basa básicamente en trabajar con la librería "thread" tratando de hacerlo mas eficiente que una lista de saltos básicamente común.

Index Terms—Computer Society, IEEE, IEEEtran, journal, LaTeX, paper, template.

1 Introducción

Estamos en una época donde el auge de la computación es muy grande, talvez el mas grande de todos sus tiempos, la computación tiene cabida en todos los aspectos importantes de la vida de las personas, nos atrevemos a decir que no se podría imaginar un presente sin computación actualmente, es por esto muy importante que el conocimiento de estructuras de datos para el desarrollo de sistema, que son aplicados para todas las áreas es necesario, ya que son muchos los casos en donde se puede plantear una solución, tomando en cuenta que esta no sería única si no por el contrario que pueden haber muchas mas , es entonces que nosotros hablamos de optimización de algoritmos o estructuras de datos avanzadas, en el caso de la lista de saltos , no es una excepción , encontraremos varios algoritmos de implementación en internet, sin embargo lo que debemos buscar es encontrar una solución optima para la actualidad, por otro lado también buscamos que este algoritmo sea simple, eficiente y por su puesto que trabaje en paralelo.

2 ¿Por qué Skip-list?

Para responder esta pregunta, es sencillo, a veces las personas complicamos la vida, tratando siempre de tomar cosas complejas, cosas difíciles, solo para demostrar que podemos, es obvio que podremos sin embargo el desarrollo no hace el camino ameno porque estamos mas centrados en la dificultad o en cumplir ese objetivo , que nos olvidamos de lo que realmente importa que es aprender, aprender paso a paso para que el desarrollo sea mas ameno, para poder lograr cosas mas grandes, es entonces que en esta ocasión nos centramos en una estructura de datos no tan compleja, para que de este modo podamos comprender estructuras más avanzadas, ahora centrándonos mas en que en la actualidad tenemos la disponibilidad de la concurrencia y esto lo hará más rápido.

Añadiendo podemos comparar una estructura de datos con otras como lo son los arboles balanceados, en este caso nuestra estructura de datos es una estructura probabilística, lo que se manifiesta que tenga un resultado malo por así decirlo en el peor de los casos, sin embargo, esto es muy improbable porque tenemos implementaciones en donde esto será casi imposible. También tenemos que mencionar que es más fácil balancear una estructura de datos probabilística que una de balance, es por eso que hacemos una comparativa para tener una lista de saltos como estructura prioritaria para problemas comunes. Para muchas aplicaciones las listas de saltos una herramienta muy común, ya que otras estructuras de balance son mas complicadas y se utilizan conocimientos más avanzados, sin embargo, la lista de saltos son simples y tienen un desempeño logarítmico muy bueno en comparación a estas.

3 PROPUESTA

La idea es como mencionábamos es tener una lista enlazada de varios niveles, donde en los demás niveles no tendremos todos los nodos, mientras el nivel es superior tendremos menos nodos pero se avanzara más rápido en la lista, esto nos sirve para las operaciones ya que mayormente siempre tenemos que encontrar el nodo antes de hacer cualquier cosa. Es entonces que tenemos esta estrategia para la lista de saltos, esto lo podemos observar en la (Fig.1). Como podemos observar tenemos una lista de saltos de 3 niveles desde el nivel 0 , o cuatro carriles como lo estábamos llamando en nuestro articulo , en el ultimo nivel 3 solo tenemos dos nodos mientras que vamos descendiendo y llegamos al nivel 0 que es donde tenemos todos los nodos completos. Esto nos ayudara mucho ya que al ser una estructura probabilística en un caso medio que es donde mayormente caerán las situaciones será muy optima.

- *Departamento de Ciencia de la Computación, Universidad Católica San Pablo, Arequipa, AQP, 0001.*
  E-mail: see http://cs.ucsp.edu.pe/contact.html
- *J Carlos Gutiérrez-Cáceres.*

Manuscript received December 10, 2022; revised December , 2022.

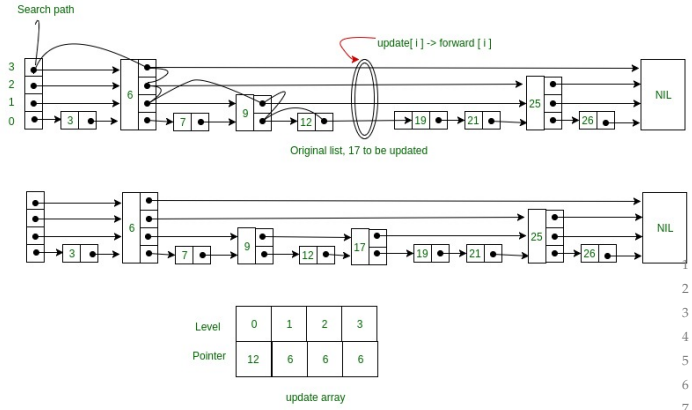

Fig 1. Skip List idea.

4 Propuesta concurrente

Para esta propuesta tenemos , básicamente aplicamos dos métodos los cuales nos ayudaran en todas las operaciones de la lista de saltos. Entonces tomamos en cuenta que la búsqueda se da por iteración de nodos y para esto es necesario trabajar con bloqueos, esto básicamente para lograr una optimización en cualquier búsqueda para posteriormente hacer la operación, después de esto para la eliminación lo único que haremos será que al eliminar un elemento lo que haremos será marcarlo antes de eliminarlo, esto básicamente para deshabilitarlo y hacer menos trabajo de lo necesario.

Entonces al tener estas operaciones con la librería "thread" tendremos que las operaciones como son, insertar,eliminar,buscar,etc, se podrán realizar en múltiples subprocesos. Ya que para la implementación utilizaremos un bloqueo manual para acceder a nuestros elementos y poder escribir sobre nuestra estructura, esto es muy importante para la escritura y lectura de nuestros datos.

```
class SkipList {
private:
    skipListNode* head;
    skipListNode* tail;
public:
    SkipList();
    SkipList(int max_elements, float probability);
    ~SkipList();
    int get_random_level();


    int find(int key, vector<skipListNode*>&
    predecessors, vector<skipListNode*>& successors)
    ;
    bool agregar(int key, string value);
    string buscar(int key);
    bool eliminar(int key);
    map<int, string> range(int start_key, int
    end_key);
    void display();
};
```

Listing 1. skipList

5 Implementacion del Nodo

La estructura nodo almacena una key y value. Entonces estos son un valor entre y una cadena. Este apuntara al siguiente nodo y cada nodo utiliza la key para bloquear el nodo cuando se esta modificando. Despues utilizaremos una variable atomica para indicar si se esta eliminando el nodo u otro elemento de nuestra estructura. Esta se utilizara para indicar si el nodo esta completamente vinculado a sus sucesores y predecesores o no es asi, nuestra variable miembro tiene todos los niveles en donde nuestro elemento esta disponible.

```
class skipListNode {
public:
    KeyValuePair key_value_pair;
    vector<skipListNode*> next;
    mutex node_lock;
    atomic<bool> marked = { false };
    atomic<bool> fully_linked = { false };
    int top_level;
    skipListNode();
    skipListNode(int key, int level);
    skipListNode(int key, string value, int level);
    ~skipListNode();
    int get_key();
    string get_value();
    void lock();
    void unlock();
};
```

Listing 2. skipListNode

5.1 Insertar

Antes de insertar un elemento en la lista de saltos, verificamos si el elemento ya está presente en la lista de saltos y si el nodo está marcado. Si el elemento ya está presente y el nodo no está marcado, no insertamos el elemento porque ya está presente en la lista de omisión. Si el elemento está presente y el nodo no está completamente vinculado, entonces esperamos hasta que esté completamente vinculado antes de insertarlo. Si el elemento está presente y el nodo está marcado, se está eliminando, por lo que esperamos e intentamos nuestro algoritmo de inserción completo nuevamente más tarde.

```
bool SkipList::agregar(int key, string value) {


    int top_level = get_random_level();


    vector<skipListNode*> preds(max_level + 1);
    vector<skipListNode*> succs(max_level + 1);

    for (size_t i = 0; i < preds.size(); i++) {
        preds[i] = NULL;
        succs[i] = NULL;
    }


    while (true) {


        int found = find(key, preds, succs);


        if (found != -1) {
            skipListNode* node_found = succs[found];

            if (!node_found->marked) {
                while (!node_found->fully_linked) {
                }
                return false;
```

```
            }
            continue;
        }


        map<skipListNode*, int> locked_nodes;


        try {
            skipListNode* pred;
            skipListNode* succ;


            bool valid = true;

            for (int level = 0; valid && (level <= top_level); level++) {
                pred = preds[level];
                succ = succs[level];


                if (!(locked_nodes.count(pred))) {
                    pred->lock();
                    locked_nodes.insert(make_pair(pred, 1));
                }


                valid = !(pred->marked.load(std::memory_order_seq_cst)) && !(succ->marked.load(std::memory_order_seq_cst)) && pred->next[level] == succ;
            }


            if (!valid) {
                for (auto const& x : locked_nodes) {
                    x.first->unlock();
                }
                continue;
            }

            skipListNode* new_node = new skipListNode(key, value, top_level);


            for (int level = 0; level <= top_level; level++) {
                new_node->next[level] = succs[level];
            }

            for (int level = 0; level <= top_level; level++) {
                preds[level]->next[level] = new_node;
            }


            new_node->fully_linked = true;


            for (auto const& x : locked_nodes) {
                x.first->unlock();
            }

            return true;
        }
        catch (const std::exception& e) {

            std::cerr << e.what() << '\n';
            for (auto const& x : locked_nodes) {
                x.first->unlock();
            }
        }
    }
}
```

```
};
```

Listing 3. skipListAgregar

6 ELIMINAR

Antes de eliminar un elemento de la lista de omisión, verificamos si el elemento está presente en la lista de omisión y si el nodo no está presente, regresamos. Si el elemento está presente, verificamos si está completamente vinculado y desmarcado, si no, intentamos eliminar algo nuevamente.

```

bool SkipList::eliminar(int key) {

    skipListNode* victim = NULL;
    bool is_marked = false;
    int top_level = -1;


    vector<skipListNode*> preds(max_level + 1);
    vector<skipListNode*> succs(max_level + 1);

    for (size_t i = 0; i < preds.size(); i++) {
        preds[i] = NULL;
        succs[i] = NULL;
    }


    while (true) {

        int found = find(key, preds, succs);


        if (found != -1) {
            victim = succs[found];
        }


        if (is_marked | (found != -1 && (victim->fully_linked && victim->top_level == found && !(victim->marked))))
        {

            if (!is_marked) {
                top_level = victim->top_level;
                victim->lock();
                if (victim->marked) {
                    victim->unlock();
                    return false;
                }
                victim->marked = true;
                is_marked = true;
            }


            map<skipListNode*, int> locked_nodes;

            try {
                skipListNode* pred;

                bool valid = true;

                for (int level = 0; valid && (level <= top_level); level++) {
                    pred = preds[level];


                    if (!(locked_nodes.count(pred))) {
                        pred->lock();
                        locked_nodes.insert(make_pair(pred, 1));
                    }

```

```
                    valid = !(pred->marked) && pred
    ->next[level] == victim;
                }

                if (!valid) {
                    for (auto const& x :
    locked_nodes) {
                        x.first->unlock();
                    }
                    continue;
                }

                for (int level = top_level; level >=
     0; level--) {
                    preds[level]->next[level] =
    victim->next[level];
                }

                victim->unlock();

                for (auto const& x : locked_nodes) {
                    x.first->unlock();
                }

                return true;
            }
            catch (const std::exception& e) {

                for (auto const& x : locked_nodes) {
                    x.first->unlock();
                }
            }

        }
        else {
            return false;
        }
    }
}
```

Listing 4. skipListNodeEliminar

7 BUSCAR

La búsqueda de un elemento en la lista de saltos se realiza recorriendo toda la lista de saltos en un nivel superior y bajando a niveles inferiores a medida que la búsqueda se acerca a la clave de búsqueda. Si se encuentra una clave, verificamos si el nodo está desmarcado y completamente vinculado. Si es así, entonces nuestra búsqueda es exitosa y devolvemos el valor asociado con la clave. Si el nodo está marcado o no está completamente vinculado, devolvemos falso ya que el nodo está marcado para su eliminación o no está completamente vinculado después de otras operaciones.

```
string SkipList::buscar(int key) {


    vector<skipListNode*> predecesores(max_level +
    1);
    vector<skipListNode*> sucesores(max_level + 1);

    for (size_t i = 0; i < predecesores.size(); i++)
     {
        predecesores[i] = NULL;
        sucesores[i] = NULL;
    }

    int found = find(key, predecesores, sucesores);

    if (found == -1) {
        return "";
    }

    skipListNode* curr = head;

    for (int level = max_level; level >= 0; level--)
     {
        while (curr->next[level] != NULL && key >
    curr->next[level]->get_key()) {
            curr = curr->next[level];
        }
    }

    curr = curr->next[0];

    if ((curr != NULL) && (curr->get_key() == key)
    && sucesores[found]->fully_linked && !sucesores[
    found]->marked) {
        return curr->get_value();
    }
    else {
        return "";
    }
}
```

Listing 5. skipListBuscar