Chapter 9

Exercise 9.1

Indicate whetever the following schedules can produce anomailes; the symbols c_i and a_i indicate the result (commit or abort) of the transaction.

1) $r_1(x)$, $w_1(x)$, $r_2(x)$, $w_2(y)$, a_1, c_2
2) $r_1(x)$, $w_1(x)$, $r_2(y)$, $w_2(y)$, a_1, c_2
3) $r_1(x)$, $r_2(x)$, $r_2(y)$, $w_2(y)$, $r_1(z)$, a_1, c_2
4) $r_1(x)$, $r_2(x)$, $w_2(x)$, $w_1(x)$, c_1, c_2
5) $r_1(x)$, $r_2(x)$, $w_2(x)$, $r_1(y)$, c_1, c_2
6) $r_1(x)$, $w_1(x)$, $r_2(x)$, $w_2(x)$, c_1, c_2

Sol:

1) The operation $r_2(x)$ reads the value written by $w_1(x)$, but the transaction 1 ends with an abort. This is a case of Dirty Read.
2) This schedule does not produce anomalies, because the two transactions refer to different objects.
3) This schedule does not produce anomalies because the transaction 1 does not write any objects.
4) This schedule produces an Update Loss.
5) This schedule does not produce anomalies.
6) This schedule does not produce anomalies.

Exercise 9.2

Indicate whether the following schedules are VSR.

1) $r_1(x), r_2(y), w_1(y), r_2(x), w_2(x)$
2) $r_1(x), r_2(y), w_1(x), w_1(y), r_2(x), w_2(x)$
3) $r_1(x), r_1(y), r_2(y), w_2(z), w_1(z), w_3(z), w_3(x)$
4) $r_1(y), r_1(y), w_2(z), w_1(z), w_3(z), w_3(x), w_1(x)$

Sol :

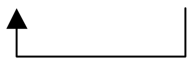
1) This schedule is not VSR, because the two serial schedule

S_1: $r_1(x), w_1(y), r_2(y), r_2(x), w_2(x)$ and

S_2: $r_2(y), r_2(x), w_2(x), r_1(x), w_1(y)$

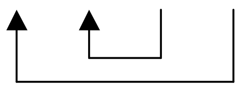
are not view-equivalent to the given schedule; they have a different READ-FROM relation.

2) $r_1(x), r_2(y), w_1(x), w_1(y), r_2(x), w_2(x)$

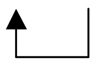


This schedule is not VSR because the schedules

S_1: $r_1(x), w_1(x), w_1(y)\ r_2(y), r_2(x), w_2(x)$



S_2: $r_2(y), r_2(x), w_2(x), r_1(x), w_1(x), w_1(y)$



have a different READ-FROM relation.

3) This schedule is VSR because it is view-equivalent to the serial schedule:

S : $r_2(y), w_2(z), r_1(x), r_1(y), w_1(z), w_3(z), w_3(x)$

4) This schedule is not-VSR, because it cannot exist a serial schedule with the same FINAL WRITE relation; note that transaction 1 has the final write on X and also a write on Z, while transaction 3 has the final write on Z and also a write on X.

Exercise 9.3

Classify the following schedules (as: Non-VSR, VSR, CSR). In the case of a schedule that is both VSR and CSR, indicate all the serial schedules equivalent to them.

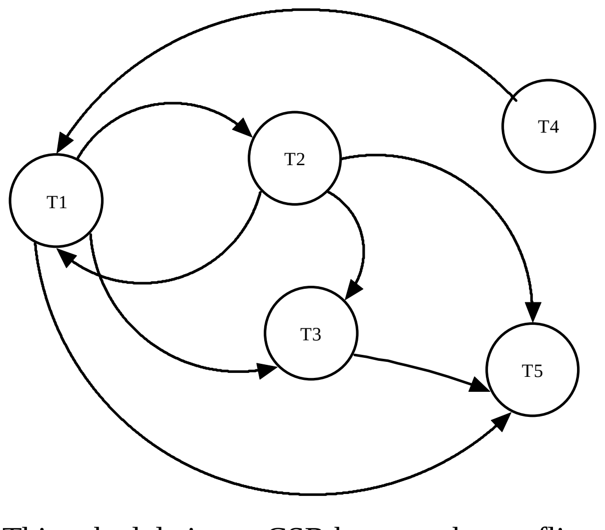
1) $r_1(x), w_1(x), r_2(z), r_1(y), w_1(y), r_2(x), w_2(x), w_2(z)$
2) $r_1(x), w_1(x), w_3(x), r_2(y), r_3(y), w_3(y), w_1(y), r_2(x)$
3) $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$
4) $r_1(x), r_3(y), w_1(y), w_4(x), w_1(t), w_5(x), r_2(z), r_3(z), w_2(z), w_5(z), r_4(t), r_5(t)$
5) $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), r_3(y), r_3(x), w_1(y), w_5(x), w_1(z), r_5(y), r_5(z)$
6) $r_1(x), r_1(t), r_3(z), r_4(z), w_2(z), r_4(x), r_3(x), w_4(x), w_4(y), w_3(y), w_1(y), w_2(t)$
7) $r_1(x), r_4(x), w_4(x), r_1(y), r_4(z), w_4(z), w_3(y), w_3(z), w_1(t), w_2(z), w_2(t)$

1) This schedule is both VSR and CSR, and it is conflict-equivalent to
S: $r_1(x), w_1(x), r_1(y), w_1(y), r_2(z), r_2(x), w_2(x), w_2(z)$

2) $r_1(x), w_1(x), w_3(x), r_2(y), r_3(y), w_3(y), w_1(y), r_2(x)$

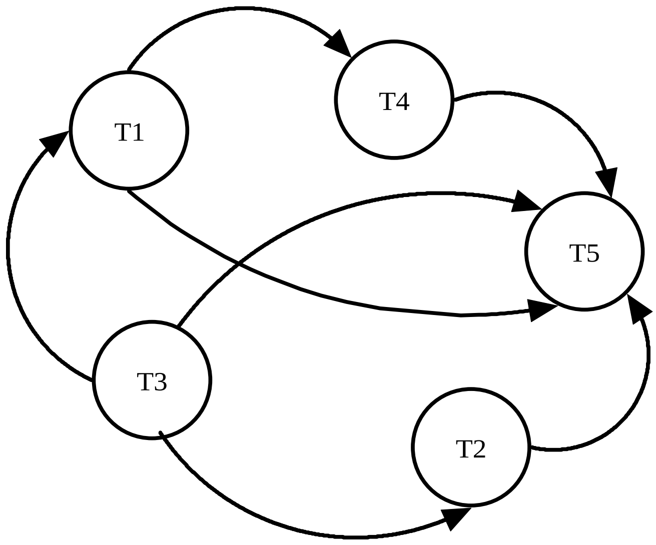
This schedule is Not-VSR. In a sequential schedule view-equivalent to this schedule the transaction 1 should follow the transaction 3 because of the FINAL WRITE on Y, but however it should precede the transaction 3 because of the READ FORM relation on X.

3) $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$



This schedule is not CSR because the conflict graph is cyclic.
This schedule is also Non-VSR, because, referring to X, the transaction1 should precede the transaction 2, and transaction 2 should precede the transaction 1 (the two transactions both read X before any write operation).

4) $r_1(x)$, $r_3(y)$, $w_1(y)$, $w_4(x)$, $w_1(t)$, $w_5(x)$, $r_2(z)$, $r_3(z)$, $w_2(z)$, $w_5(z)$, $r_4(t)$, $r_5(t)$



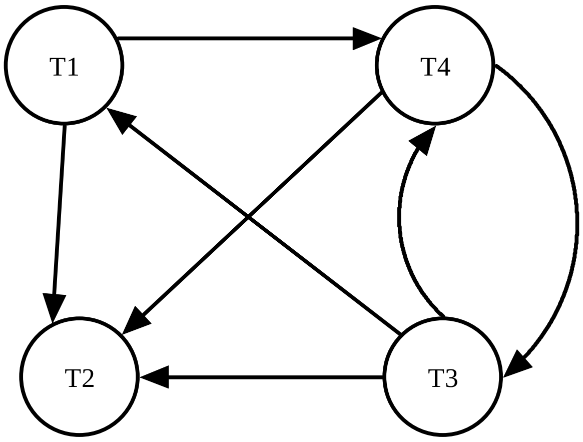
This schedule is both CSR and VSR.
The equivalent serial schedules are:

S_1: $r_3(y)$, $r_3(z)$, $r_1(x)$, $w_1(y)$, $w_1(t)$, $r_2(z)$, $w_2(z)$, $w_4(x)$, $r_4(t)$, $w_5(x)$, $w_5(z)$, $r_5(t)$
S_2: $r_3(y)$, $r_3(z)$, $r_2(z)$, $w_2(z)$, $r_1(x)$, $w_1(y)$, $w_1(t)$, $w_4(x)$, $r_4(t)$, $w_5(x)$, $w_5(z)$, $r_5(t)$

5) $r_1(x)$, $r_2(x)$, $w_2(x)$, $r_3(x)$, $r_4(z)$, $w_1(x)$, $r_3(y)$, $r_3(x)$, $w_1(y)$, $w_5(x)$, $w_1(z)$, $r_5(y)$, $r_5(z)$

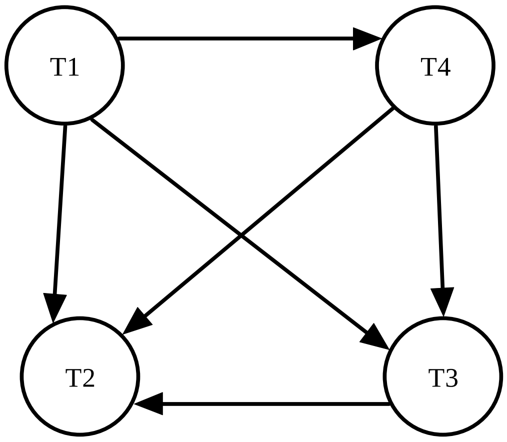
This schedule is Non-VSR; the transaction 1 and 2 both read and write X, but in this schedule they read X before any write operation, and so any sequential schedule with these transaction will have a different READ FROM relation.

6) $r_1(x)$, $r_1(t)$, $r_3(z)$, $r_4(z)$, $w_2(z)$, $r_4(x)$, $r_3(x)$, $w_4(x)$, $w_4(y)$, $w_3(y)$, $w_1(y)$, $w_2(t)$



This schedule is not CSR.
This schedule is also Non-VSR; the transaction 1 has the FINAL WRITE on Y, and so it should follow the transaction 4. But the transaction 4 writes X, and so it should follow the transaction 1.

7) $r_1(x), r_4(x), w_4(x), r_1(y), r_4(z), w_4(z), w_3(y), w_3(z), w_1(t), w_2(z), w_2(t)$



This schedule is both CSR and VSR.

The serial schedule equivalent is:
S: $r_1(x), r_1(y), w_1(t), r_4(x), w_4(x), r_4(z), w_4(z), w_3(y), w_3(z), w_2(z), w_2(t)$

Exercise 9.4

If the above schedules are presented to a schedule that uses two-phase locking, which transaction would be placed in waiting ? (Note that once a transaction is placed in waiting, its successive actions are not considered.)

Sol:

1) $r_1(x), w_1(x), r_2(z), r_1(y), w_1(y), r_2(x), w_2(x), w_2(z)$

No transaction in waiting.

2) $r_1(x), w_1(x), w_3(x), r_2(y), r_3(y), w_3(y), w_1(y), r_2(x)$

The transaction 1 and 3 are placed in waiting, but however this schedule can produce a deadlock : the action $r_2(x)$ must wait for the object x, locked by the transaction 1, and the transaction 1 is waiting for the object y, locked by the transaction 2.

3) $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$

The transactions 2, 3 and 5 are put in waiting, because of the lock on x.

4) $r_1(x), r_3(y), w_1(y), w_4(x), w_1(t), w_5(x), r_2(z), r_3(z), w_2(z), w_5(z), r_4(t), r_5(t)$

The transactions 1, 3, 4 and 5 are put in waiting. The transaction 1 must wait for y (locked by t_2), the transactions 4 and 5 must wait for x (locked by t_1) and the transaction 3 must wait for z (locked by t_2).

5) $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), r_3(y), r_3(x), w_1(y), w_5(x), w_1(z), r_5(y), r_5(z)$

The transaction 2, 3 and 5 are put in waiting. They must wait for x (locked by t_1).

6) $r_1(x), r_1(t), r_3(z), r_4(z), w_2(z), r_4(x), r_3(x), w_4(x), w_4(y), w_3(y), w_1(y), w_2(t)$

The transaction 2, 3 and 4 are put in waiting. t_2 and t_4 must wait for z (locked by t_3), and t_3 must wait for y (locked by t_1).

7) $r_1(x), r_4(x), w_4(x), r_1(y), r_4(z), w_4(z), w_3(y), w_3(z), w_1(t), w_2(z), w_2(t)$

The transaction 3 and 4 are put in waiting. They must wait for x and y, locked by t_1.

Exercise 9.5

Define the data structure necessary for the management of locking, for a non-hierarchical model with read repeatability. Implement in a programming language of your choice the functions **lock_r**, **lock_w**, and **unlock**. Assume that an abstract type ‘queue’ is available with the appropriate functions for the insertion of an element into a queue and for extracting the first element of the queue.

Sol:

Transaction are identified with a number, objects are identified with a string.

```
typedef struct resource {
     char *identifier;
     int free;       // 1 is free, 0 is busy
     int r_locked;   // 1 or more locked, 0 free
     int w_locked;   // 1 locked
     queue waiting_transaction;
}

typedef struct queue_element {
     int transaction;
     int type       // 0 read, 1 write
}

typedef *resource locktable;

locktable lt=new locktable [N];  // N is the number of resources


int lock_r (int transaction, char *resource) {
     int i=0;
     int end=0;
     while ((i<N) && (!end))
        if (strcmp(resource, lt[i])!=0) i++; else end=1;
     if (!end) return -1;        // resource not found
     if (lt[i].free)
        {  lt[i].free=0;
           lt[i].r_locked=1;
           return transaction;  }
        if (lt[i].r_locked)
           {  lt[i].r-locked++;
              return transaction;  }

        // the resource is w_locked

     queue_element q=new queue_element;
     q->transaction=transaction;
     q->type=0;
     in_queue(q,lt[i].queue);
     return 0;     }
```

```
int lock_w (int transaction, char *resource) {
     int i=0;
     int end=0;
     while ((i<N) && (!end))
        if (strcmp(resource, lt[i])!=0) i++; else end=1;
     if (!end) return -1;        // resource not found
     if (lt[i].free)
          { lt[i].free=0:
            lt[i].w_locked=1;
            return transaction;
          }
     queue_element q=new queue_element;
     q->transaction=transaction;
     q->type=1;
     in_queue(q,lt[i].queue);
     return 0;
}



int unlock (int transaction, char *resource) {
     int i=0;
     int end=0;
     while ((i<N) && (!end))
        if (strcmp(resource, lt[i])!=0) i++; else end=1;
     if (!end) return -1;        // resource not found

     if (lt[i].r_locked)
          { lt[i].r_locked--;
            if (lt[i].r_locked) return 0;
            if (is_empty(lt[i].queue))
              { lt[i].free=1;
                return 0;
              }
            queue_element q=out_queue(lt[i].queue);
             lt[i].w_locked=1;   //  transaction  in  queue  must  be  write-
transaction
            return q.transaction;
          }
     if (is_empty(lt[i].queue))
              { lt[i].free=1;
                return 0;
              }
        queue_element q=out_queue(lt[i].queue);
        if (!(q.type))    // read transaction in queue
          { lt[i].r_locked=1;
            lt[i].w_locked=0; }

        // now unlock extracts all the read-transaction from the queue;
        while (!(first_element(lt[i].queue)).type))
           { out_queue (lt[i].queue);
             r_locked++;
           }
        return q.transaction;     }
```

Exercise 9.6

With reference to the exercise above, add a timeout mechanism. Assume that we have available functions for getting the current system time and for extracting a specific element from a queue.

Sol:

The structure queue_element must be modified as follow:

```
typedef struct queue_element {
     int transaction;
     int type;
     int time;
}
```

The field time represents the instant in which the transaction is put in waiting.
The functions lock_r and lock_w must fill also this field before put in queue a transaction, using the instruction:

```
q.time=get_system_time();
```

The timeout mechanism is realised by a new function, check_time, which is called periodically by the system.

```
void check_time() {
     int now=get_system_time;
     for (int i=0:; i<N; i++)
      if (!is_empty(lt[i].queue))
        { int l=queue_length(lt[i].queue);
          for (int j=0; j<l; j++)
            { queue_element q=get_queue_element(lt[i].queue,j);
              if ((q.time+MAX_TIME)<now)
                 remove_from_queue(lt[i].queue,j);
            }
         }
}
```

Exercise 9.7

If the schedules described in Exercise 9.3 were presented to a timestamp-based scheduler, which transaction would be aborted ?

Sol:

1) $r_1(x), w_1(x), r_2(z), r_1(y), w_1(y), r_2(x), w_2(x), w_2(z)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
write(x,1)	Ok	WTM(x)=1
read(z,1)	Ok	RTM(z)=2
read(y,1)	Ok	RTM(y)=1
write(y,1)	Ok	WTM(y)=1
read(x,2)	Ok	RTM(x)=2
write(z,2)	Ok	WTM(z)=2

No transaction is aborted.

2) $r_1(x), w_1(x), w_3(x), r_2(y), r_3(y), w_3(y), w_1(y), r_2(x)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
write(x,1)	Ok	WTM(x)=1
write(x,3)	Ok	WTM(x)=3
read(y,2)	Ok	RTM(y)=2
read(y,3)	Ok	RTM(y)=3
write(y,3)	Ok	WTM(y)=3
write(y,1)	t_1 aborted	
read(x,2)	t_2 aborted	

3) $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
read(x,2)	Ok	RTM(x)=2
write(x,2)	Ok	WTM(x)=2
read(x,3)	Ok	RTM(x)=3
read(z,4)	Ok	RTM(z)=4
write(x,1)	t_1 aborted	
write(y,3)	Ok	WTM(y)=3
write(x,5)	Ok	WTM(x)=5
write(y,5)	Ok	WTM(y)=5
read(z,5)	Ok	RTM(z)=5

4) $r_1(x)$, $r_3(y)$, $w_1(y)$, $w_4(x)$, $w_1(t)$, $w_5(x)$, $r_2(z)$, $r_3(z)$, $w_2(z)$, $w_5(z)$, $r_4(t)$, $r_5(t)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
read(y,3)	Ok	RTM(y)=3
write(y,1)	t_1 aborted	
write(x,4)	Ok	WTM(x)=4
write(x,5)	Ok	WTM(x)=5
read(z,2)	Ok	RTM(z)=2
read(z,3)	Ok	RTM(z)=3
write(z,2)	t_2 aborted	
write(z,5)	Ok	WTM(z)=5
read(t,4)	Ok	RTM(t)=4
read(t,5)	Ok	RTM(t)=5

5) $r_1(x)$, $r_2(x)$, $w_2(x)$, $r_3(x)$, $r_4(z)$, $w_1(x)$, $r_3(y)$, $r_3(x)$, $w_1(y)$, $w_5(x)$, $w_1(z)$, $r_5(y)$, $r_5(z)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
read(x,2)	Ok	RTM(x)=2
read(x,3)	Ok	RTM(x)=3
read(z,4)	Ok	RTM(z)=4
write(x,1)	t_1 aborted	
read(y,3)	Ok	RTM(y)=3
read(x,3)	Ok	RTM(x)=3
write(x,5)	Ok	WTM(x)=5
read(y,5)	Ok	RTM(y)=5
read(z,5)	Ok	RTM(z)=5

6) $r_1(x)$, $r_1(t)$, $r_3(z)$, $r_4(z)$, $w_2(z)$, $r_4(x)$, $r_3(x)$, $w_4(x)$, $w_4(y)$, $w_3(y)$, $w_1(y)$, $w_2(t)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
read(t,1)	Ok	RTM(t)=1
read(z,3)	Ok	RTM(z)=3
read(z,4)	Ok	RTM(z)=4
write(z,2)	t_2 aborted	
read(x,4)	Ok	RTM(x)=4
read(x,3)	t_3 aborted	
write(x,4)	Ok	WTM(x)=4
write(y,4)	Ok	WTM(y)=4
write(y,2)	t_1 aborted	
write(t,2)	Ok	WTM(t)=2

7) $r_1(x)$, $r_4(x)$, $w_4(x)$, $r_1(y)$, $r_4(z)$, $w_4(z)$, $w_3(y)$, $w_3(z)$, $w_1(t)$, $w_2(z)$, $w_2(t)$

Operation	**Response**	**New Values**
read(x,1)	Ok	RTM(x)=1
read(x,4)	Ok	RTM(x)=4
write(x,4)	Ok	WTM(x)=4
read(y,1)	Ok	RTM(y)=1
read(z,4)	Ok	RTM(z)=4
write(y,3)	Ok	WTM(y)=3
write(z,3)	t_3 aborted	
write(t,1)	Ok	WTM(t)=1
write(z,2)	t_2 aborted	

Exercise 9.8

Consider both single-version and multi-version concurrency control based on timestamp for an object X. Initially WTM(X)=5, RTM(X)=7. Indicate the action of the scheduler in response to the following input:

r(x, 8), r(x, 17), w(x, 16), w(x, 18), w(x, 23), w(x, 29), r(x, 20), r(x, 30), r(x, 25).

Sol:

Single-Version:

Operation	**Response**	**New Values**
		WTM(x)=5
		RTM(x)=7
read(x,8)	Ok	RTM(x)=8
read(x,17)	Ok	RTM(x)=17
read(x, 16)	t_{16} aborted	
write(x,18)	Ok	WTM(x)=18
write(x,23)	Ok	WTM(x)=23
write(x,29)	Ok	WTM(x)=29
read(x,20)	t_{20} aborted	
read(x,30)	Ok	RTM(x)=30
read(x,25)	t_{25} aborted	

Multi-Version

Operation	**Response**	**New Values**	
		$WTM_1(x)=5$	
		RTM(x)=7	
read(x,8)	Ok	RTM(x)=8	
read(x,17)	Ok	RTM(x)=17	
read(x, 16)	t_{16} aborted		
write(x,18)	Ok	$WTM_2(x)=18$	
write(x,23)	Ok	$WTM_3(x)=23$	
write(x,29)	Ok	$WTM_4(x)=29$	
read(x,20)	Ok	RTM(x)=20	reads from x_2
read(x,30)	Ok	RTM(x)=30	reads from x_4
read(x,25)	Ok	RTM(x)=30	reads from x_3

Exercise 9.9

Define the data structure necessary for buffer management. Implement in a programming language of your choice the functions **fix**, **use** and **unfix**. Assume we have available the system functions described in section 9.3.4.

Sol:

```
typedef struct page {
     int *address;
     int valid;
     int modified;
     int in_use;
     int file_id;
     int block_num;
}

typedef struct file_open {
     int file_id;
     char *file_name;
     int size;
     int blocks[];  // each element refers to an element in page table
}

typedef page_table *page;
typedef file_table *file_open;

page_table pt=new page_table[N];
file_table ft=new file_table[M];


int fix(char *file_name, int block) {
     int i=0;
     int found=0;
     while (i<0 && !found)
      if (strcmp(file_name,ft[i].file_name)!=0) i++; else found=1;

     if (found && ((ft[i].blocks[block])!=-1) &&
           pt[ft[i].blocks[block]].valid )
                                    // -1 means that the block is
                                    // not loaded
       return (pt[ft[i].blocks[block]]).address;

     if (!found) {  //  file is open
          int id=open(file_name);
          int position=get_file_position() // find a free position in ft
          ft[position].file_id=id;
          strcpy(file_name, ft[position].file_name);
          ft[position].size=get_size(id);
          for(int k=0; k<ft[position].size; k++)
              ft[position].blocks[k]=-1;
     }
     // block is read

     int free=get_free_page(); // find a free page in pt
     read(ft[i].file_id, block, free.address);
     pt[free].valid=1;
     pt[free].modified=0;
     pt[free].file_id=ft[i].file_id
     pt[free].block_num=block;
     return pt[free].address;
}


void use (int *page) {
     int i=0;
     int found=0;
     while (i<N && !found)
          if (pt[i].address!=page) i++ else found=1;
     pt[i].in_use=1;
}

void unfix(int *page) {
     int i=0;
     int found=0;
     while (i<N && !found)
          if (pt[i].address!=page) i++ else found=1;
     pt[i].in_use=0;
     if (pt[i].modified) {
        pt[i].valid=0;
        int j=0;
             found=0;
        while (j<N && !found)
              if (ft[j].file:id!=pt[i].file_id) j++ else found=1;
        ft[j].block[pt[i].block_number]=-1;
     }
}
```

Exercise 9.10

Describe the warm restart, indicating the progressive building of the sets UNDO and REDO and the recovery action, given the following situation in the log:

DUMP, B(T_1), B(T_2), B(T_3), I(T_1, O_1, A_1), D(T_2, O_2, B_2), B(T_4), U(T_4, O_3, B_3, A_3),
U(T_1, O_4, B_4, A_4), C(T_2), CK(T_1, T_3, T_4), B(T_5), B(T_6), U(T_5, O_5, B_5, A_5), A(T_3), CK(T_1, T_4, T_5, T_6),
B(T_7), A(T_4), U(T_7, O_6, B_6, A_6), U(T_6, O_3, B_7, A_7), B(T_8), A(T_7), failure

Sol:

1) First of all the log is traced back until the first check-point record: CK(T_1, T_4, T_5, T_6).
The two sets UNDO and REDO are respectively:

UNDO= { T_1, T_4, T_5, T_6 } REDO={}

2) The log is traced forward, updating the two sets:

- B(T_7)	UNDO= { T_1, T_4, T_5, T_6, T_7 }	REDO={}
- A(T_4)	UNDO= { T_1, T_4, T_5, T_6, T_7 }	REDO={}
- B(T_8)	UNDO= { T_1, T_4, T_5, T_6, T_7, T_8 }	REDO={}
- A(T_7)	UNDO= { T_1, T_4, T_5, T_6, T_7 }	REDO={}

3) The log is traced back again, until the operation I(T_1, O_1, A_1), executing the following undo operation:

O_3=B_7
O_6=B_6
O_5=B_5
O_4=B_4
O_3=B_3
Delete O_1

4) The log is traced forward, but the REDO set is empty, and so no redo-operation will be executed.

Exercise 9.11

Assume that in the above situation a device failure involves the objects O_1, O_2 and O_3. Describe the cold restart.

Sol:

After the system restart, the log is traced back until the first DUMP record. The DUMP is accessed and the damaged parts are selectively copied.

Then the log is traced forward and all the actions regarding the damaged parts are applied:

Insert O_1=A_1
Delete O_2
O_3=A_3
Commit (T_2)
Abort (T_4)
O_3=A_7

Finally, the warm restart is applied.

Exercise 9.12

Consider a hash structure for storing tuples whose key fields contains the following names:

Green, Lovano, Osby, Peterson, Pullen Scofield, Allen, Haden, Harris, MacCann, Mann, Brown, Newmann, Ponty, Cobbham, Coleman, Mingus, Lloyd, Tyner, Hutcherson, Grant, Fortune, Coltrane, Sheep.

1) Suggest a hashing function with B=8 and F=4
2) Supposing B=40 and F=1, what is the probability of conflict ? And with B=20 and F=2 ?
3) With F=5 and B=7, what is the approximate length of the overflow chain ?

Sol:

1) A simple hashing function for the given names is:
   - For each character in the name, consider the corresponding number in alphabetical order (a=1, b=2...).
   - Sum all the numbers so obtained, and make the ‘modulo B’ division.

In this way we will obtain for each name a number between 0 and B-1.

Example:

Hash(Green)=(7+18+5+5+14) mod 8=1
Hash(Lovano)=(12+15+22+1+13+15) mod 8=6
Hash(Osby)=(15+19+2+25) mod 8=5
Hash(Peterson)=(16+5++20+5+18+19+15+13) mod 8=7

2) With B=40 and F=1 the probability of conflict is

$$p = 1 - T\left(\frac{1}{B}\right)\cdot\left(1-\frac{1}{B}\right)^{T-1} = 1 - 24\cdot\frac{1}{40}\cdot\left(\frac{39}{40}\right)^{23} = 0{,}6648$$

With B=20 and F=2

$$p = 1 - \sum_{i=1}^{F}\binom{T}{i}\cdot\left(\frac{1}{B}\right)^{i}\cdot\left(1-\frac{1}{B}\right)^{(T-i)} = 1 - 24\cdot\frac{1}{20}\cdot\left(\frac{19}{20}\right)^{23} - \binom{24}{2}\cdot\left(\frac{1}{20}\right)^{2}\cdot\left(\frac{19}{20}\right)^{22} = 0{,}4079$$

Note that this is the probability of having two or more collision in the same block, because each block contains 2 tuples, and 1 collision is admitted.

4) The length of the overflow chain may calculated as the weighted sum of probability of collisions:

$$l = \sum_{i=F+1}^{T} i\cdot\binom{T}{i}\cdot\left(\frac{1}{B}\right)^{i}\cdot\left(1-\frac{1}{B}\right)^{(T-i)} = 0{,}768$$

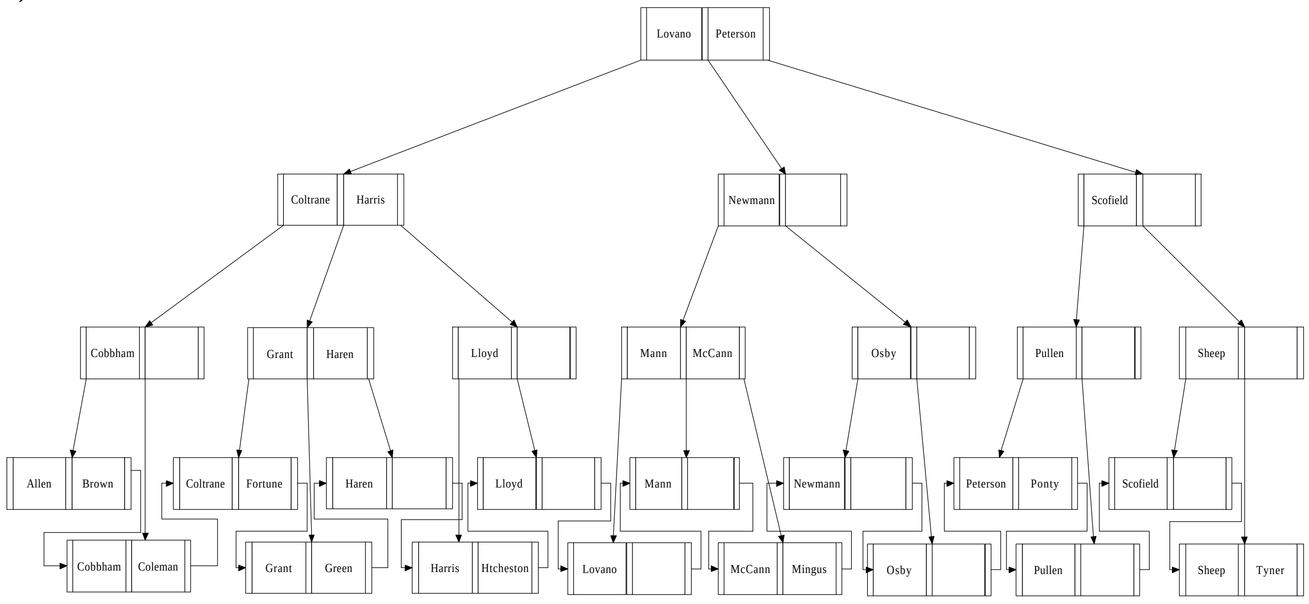
Exercise 9.13

Consider a B+ tree structure for storing tuples whose key field contains the data listed in the above exercise.

1) Describe a balanced B+ tree structure with F=2, which contains the listed data.
2) Introduce a data item that causes the split of a node at leaf level, and show what happens at leaf level and at the level above.
3) Introduce a data item that causes a merge of a node at leaf level, and show what happens at leaf level and at the level above.
4) Show a sequence of insertions that causes the split of the root and the lengthening of the tree.
5) Describe a B tree structure, with F=3, that contains the given data.

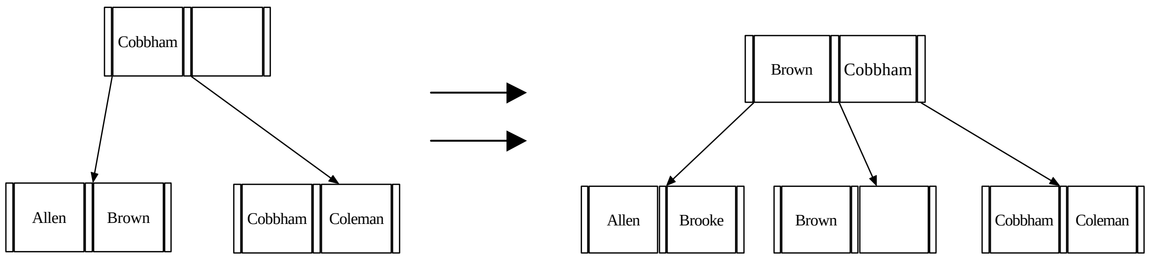
Sol:

1)

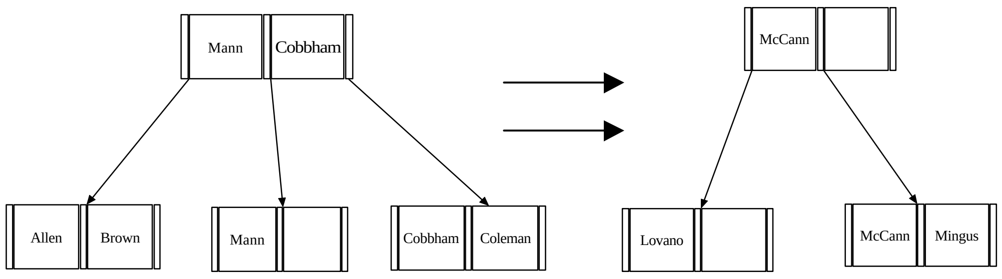


This B+ tree contains all the key values. It has 16 nodes at leaf level.

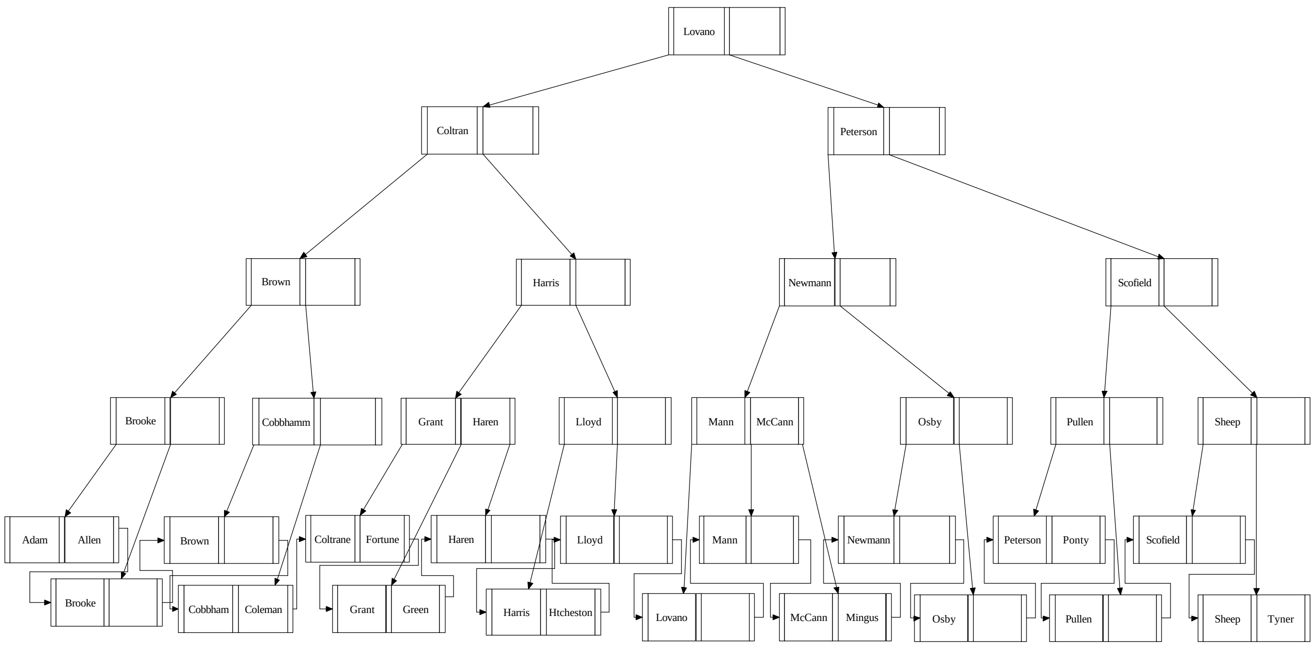
2) The introduction of value “Brooke” causes a split at leaf level, as illustrated in the following figures:



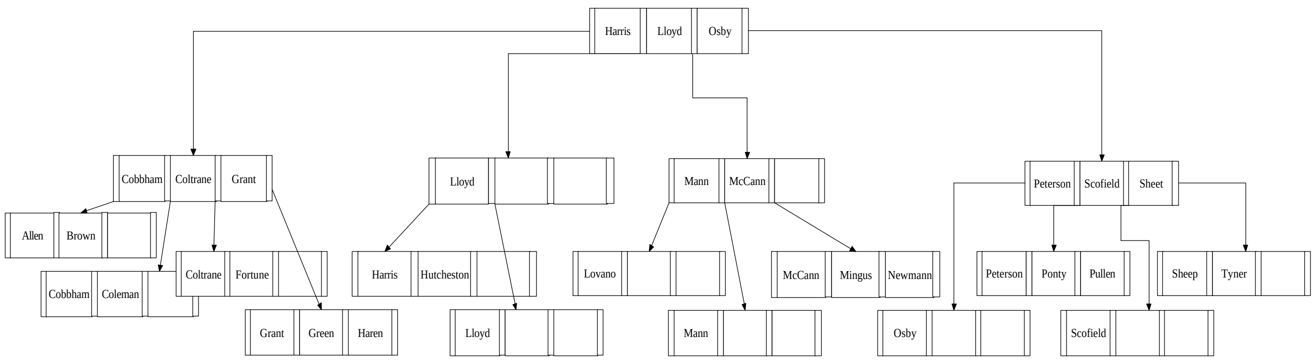
3) The cancellation of value “Mann” causes a merge of a node at leaf level:



5) The insertion of values “Brooke”, “Adam” causes a root split:



6) The following figure shows a B tree with the given data.



Exercise 9.14

Consider the database made up of the following relations:

Production (<u>ProdNumber</u>, PartType, Model, Quan, Machine)
OrderDetail (<u>OrderNumber</u>, <u>ProdNumber</u>)
Order(<u>OrderNumber</u>, Client, Amount)
Commission (<u>OrderNumber</u>, Seller, Amount)

Assume the following profiles

CARD(Production)=200000	Size(Production)=41
CARD(OrderDetail)=50000	Size(OrderDetail)=15
CARD(Order)=10000	Size(Order)=45
CARD(Commission)=5000	Size(Commission)=35
SIZE(ProdNumber)=10	VAL(ProdNumber)=200000
SIZE(PartType)=1	VAL(PartType)=4
SIZE(Model)=10	VAL(Model)=400
SIZE(Quan)=10	VAL(Quan)=100
SIZE(Machine)=10	VAL(Machine)=50
SIZE(OrderNumber)=5	VAL(OrderNumber)=10000
SIZE(Client)=30	VAL(Client)=400
SIZE(Amount)=10	VAL(Amount)=5000
SIZE(Seller)=20	VAL(Seller)=25

Describe the algebraic optimization and the computation of the profiles of the intermediate results for the following queries, which need to be initially expressed in SQL and then translated into relational algebra:

1) Find the available quantity of the product 77Y6878.
2) Find the machines used for the production of the parts sold to client Brown.
3) Find the clients who have bought from the seller White a box model 3478.

For the last two queries, which require the joining of three tables, indicate the ordering between joins that seem most convenient based on the size of the tables. Then describe the decision tree for the second query allowing for a choice of only two join methods.

Sol:

1) <u>SQL:</u>
```
select Quan
from Production
where ProdNumber=”77Y6878”
```

<u>Relational Algebra</u>

$$\Pi_{Quan}(ó_{ProdNumber=”77y6878”}(Production))$$

This query doesn’t need any algebraic optimization. If we indicate with T the table result of the query, the profile is:

CARD(T)=1	SIZE(Quant)=10
SIZE(T)=10	VAL(Quan)=1

2) <u>SQL :</u>

```
select Machine
from Production join OrderDetail on
          Production.ProdNumber=OrderDetail.ProdNumber
     join Order on OrderDetail.OrderNumber=Order.OrderNumber
where Client= ”Brown”
```

<u>Relational Algebra</u>

$\Pi_{Machine}(\sigma_{Client=”Brown’}$
$(Production\ _{ProdNumber=p}\quad _{o,p\leftarrow OrderNUmber,\ ProdNumber}(OrderDetail)$
$_{o=OrderNumber}\ Order\ ))$

Algebraic optimization: pushing down of selection and projections:

$\Pi_{Machine}\ (\ \Pi_{Pr}\ (\ \Pi_{OrderNumber}\ (\sigma_{Client=”Brown’}\ (Order))\quad _{OrderNUmber=Or}$
$_{Or,Pr\leftarrow OrderNumber,\ ProdNumber}\ (OrderDetail))$
$_{Pr=ProdNumber}\ (\ \Pi_{ProdNumber,\ Machine}\ (Production)\ )\ )$

Let $T_1 = \Pi_{OrderNumber}\ (\sigma_{Client=”Brown’}\ (Order))$

We have that:

$$CARD(T_1) = \frac{1}{VAL(Client)} \cdot CARD(Order) = 25$$
$$SIZE(T_1) = 5$$

The most convenient choice in ordering joins is (Order   OrderDetail)   Production

Let $T_2 = \Pi_{Pr}\ (\ T_1\quad OrderDetail)$

$$CARD(T_2) = CARD(T_1) \cdot CARD(OrderDetail)\frac{1}{VAL(OrderNumber)} = 125$$
$$SIZE(T_2) = 10$$

Note that the projection may be carried out together with the scan, and so does not need other intermediate results.

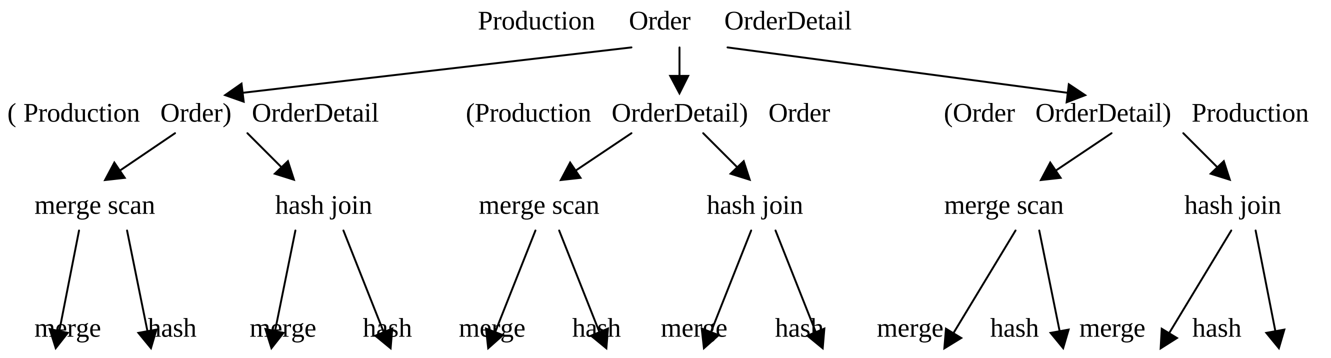
Let T$_3$= $\Pi_{\text{ProdNumber, Machine}}$ (Production)

$$CARD(T_3) = 200000$$
$$SIZE(T_3) = 20$$

Finally, let T$_4$= Π_{Machine} ($T_2 \quad\quad T_3$)

$$CARD(T_4) = Min(CARD(T_2) \cdot CARD(T_3) \cdot \frac{1}{VAL(\Pr odNumber)}, VAL(Machine)) = 50$$
$$SIZE(T_4) = 10$$

Decision tree:


Production Order OrderDetail
(Production Order) OrderDetail
(Production OrderDetail) Order
(Order OrderDetail) Production
merge scan
hash join
merge scan
hash join
merge scan
hash join
merge
hash
merge
hash
merge
hash
merge
hash
merge
hash
merge
hash

The first choice regards the order between joins, the second choice indicates the type of first join chosen, the last choice indicates the type of the second join.

3) <u>SQL:</u>

```
     select Client
     from Order join OrderDetail on
       Order.OrderNumber=OrderDetail.OrderNumber
          join Commission on
      Order.OrderNumber=Commission.OrderNumber
     where Seller='White' and ProdNumber='3478'
```

<u>Relational Algebra</u>

Π_{Client} ($\sigma_{\text{Seller='white'} \wedge \text{ProdNumber='3478'}}$ (Order $\quad_{\text{OrderNUmber=Or}}$

$\rho_{\text{Or}\leftarrow\text{OrderNuber}}$ (OrderDetail)

$_{\text{OrderNumber=Or2}} \quad \rho_{\text{Or2}\leftarrow\text{OrderNumber}}$ (Commission)))

Algebraic optimization:

Π_{Client} ($\Pi_{\text{OrdNumber}}$ ($\sigma_{\text{ProdNumber='3478'}}$ (OrderDetail) $\quad_{\text{OrderNumber=Or1}}$

(Π_{Or1} ($\sigma_{\text{Seller='White'}}$ ($\rho_{\text{Or1}\leftarrow\text{OrderNumber}}$ (Commission)))

$_{OrderNumber=Or2}$ ($\Pi_{Or2,Client}$ ($_{Or2\leftarrow OrderNumber}$ (Order)))

Let $T_{1=}$ $\sigma_{ProdNumber='3478'}$ (OrderDetail)

CARD(T$_1$)=0,25 *SIZE(T$_1$)=15*

Let T_2 = ($\Pi_{OrderNumber}$ ($\sigma_{Seller='White'}$ (Commission))

CARD(T$_2$)=200 *SIZE(T$_2$)=5*

Let $T_{3\,=}$ $\Pi_{OrdNumber}$ ($T_1 \quad T_2$)

$$CARD(T_3) = 0{,}25 \cdot 200 \cdot \frac{1}{10000} = 0{,}005$$

SIZE(T$_3$)=5

Let T_4= $\Pi_{OrderNumber,\ Client}$ (Order)

CARD(T$_4$)=10000 *SIZE(T$_4$)=35*

Let T_5= Π_{Client} ($T_3 \quad T_4$)

$$CARD(T_5) = 0{,}005 \cdot 10000 \cdot \frac{1}{10000} = 0{,}005$$

SIZE(T$_5$)=30

Note that the values *CARD(T$_i$)* could be also smaller than 1: *CARD* is only a statistic value.

Exercise 9.15

List the conditions (dimension of the tables, presence of indexes or sequential organization or of hashing) that make the join operation more or less convenient using the nested-loop, merge scan and hash methods. For some of these conditions, suggest cost formulas that take into account the number of input/output operations as a function of the average cost of the access operations involved (scans, ordering, index-based accesses).

Sol:

Dimension of the tables: if we are in a situation such that a table is very bigger than the other, it may be a good solution to make a nested-join using the greater table as external and the other table as internal. If the tables have the same size, the nested-join is convenient in presence of indexes or hashing in one of the tables. Otherwise is better to choice a merge-scan.

Hashing: the presence of a hashing function may suggest a hash-join or a nested loop. The choice depends from the size of the tables and the number of partitions produced by the hashing function.

Sequential structure: naturally, if the two tables have a sequential structure on the join attribute, the best choice is a merge-scan. If only a table has a sequential structure, the merge scan may be still a good choice if there are not other particular conditions (indexes, hashing).

Indexes: the presence of indexes in general suggests a nested-loop. However, if the table with the index is very small, or if the index is sparse, it could be better to not use the index and make a complete scan.

The cost of a nested loop join without indexes may be expressed as:

$C_{NL}=\text{scan}(T_1) + T_1 (\text{scan} (T_2))$

Where T_1 and T_2 are the number of tuples of the two tables, and scan(T) is the average cost of a scan operation

If there is a index on table 2

$C_{NL}=\text{scan}(T_1) + T_1 (\text{index} (T_2))$

A merge scan of non-sequential structures have a cost :

$C_{MS}= \text{order}(T_1) + \text{order}(T_2) + \text{scan}(T_1) + \text{scan}(T_2)$