Prise en main du code

Exercice 0

1. Un Nuplet[] est un Array des éléments Nuplet dans la Table.

Par exemple:

```
Nuplet[] test = new NupletInt(new byte[] {1,2,3});;
for(Nuplet n: test){
    System.out.println(n.toString()));
}
```

Résultat:

```
1 2 3
```

2. Dans notre code les données restent lorque le programme a finit de s'exécuter. Elles sont stockées dans le fichier `table2`.

Exercice 1

`impl/TableInt.java`

```
/*
* Applique la methode de parcours 'Fullscan' dans la TableInt.
*/
@Override
public Nuplet[] fullScan() {
        // TODO Auto-generated method stub
                /*
                 * Step 1: Importer tous des éléments dans une Collection
                 * Ici, j'utilise une Vector
                 */
                Vector<Nuplet> v = new Vector<Nuplet>();
                for(int i=0;i<this.size();i++){
                        Nuplet temp = this.get(i);
                        v.addElement(temp);
                }
                /*
                 * Step 2: Récupérer tous des éléments dans cette
Collection
                 * et les retourner
                 */
                Nuplet[] ret = new Nuplet[v.size()];
                for(int i=0;i<v.size();i++)
                        ret[i] = v.elementAt(i);
```

```
                return ret;
        }
```

Code Test

Main.java

```
//  Utilisation de fullScan
System.out.println("============");
System.out.println("Full Scan");
Nuplet[] fullscan = t.fullScan();
for(Nuplet n : fullscan)
        System.out.println(n.toString());
```

Et le résultat:

```
============
Full Scan
0       1       2       3       4       5       6       7       8       9
1       2       3       4       5       6       7       8       9       10
2       3       4       5       6       7       8       9       10      11
3       4       5       6       7       8       9       10      11      12
4       5       6       7       8       9       10      11      12      13
5       6       7       8       9       10      11      12      13      14
6       7       8       9       10      11      12      13      14      15
7       8       9       10      11      12      13      14      15      16
8       9       10      11      12      13      14      15      16      17
9       10      11      12      13      14      15      16      17      18
10      11      12      13      14      15      16      17      18      19
11      12      13      14      15      16      17      18      19      20
12      13      14      15      16      17      18      19      20      21
13      14      15      16      17      18      19      20      21      22
14      15      16      17      18      19      20      21      22      23
15      16      17      18      19      20      21      22      23      24
16      17      18      19      20      21      22      23      24      25
17      18      19      20      21      22      23      24      25      26
```

Opérateurs de base sans optimisation

Exercice 2

Dans la classe RestrictionInt, on a 3 opérateurs: Égalite, Supérieur et Inférieur. En term SQL, RestrictionInt est comme ca:

```
SELECT C
FROM R, S   ( R(A,B,C), S(D,E,F))))
WHERE B > 15 ou B < 100 ou B = 50
```

impl/RestrictionInt.java

D'abord, la method **egalite**:

```
public Nuplet[] egalite(Nuplet[] t, int att, Object v) {
                // TODO Auto-generated method stub
                Vector<Nuplet> value = new Vector<Nuplet>();
                for(Nuplet n: t) {
                        if((byte)(n.getAtt(att)) == (byte) v){
                                value.addElement(n);
                        }
                }
                Nuplet[] ret = new Nuplet[value.size()];
                for(int i=0;i<value.size();i++)
                        ret[i] = value.elementAt(i);
                return ret;
        }
```

Deuxièmement, la method **superieur**:

```
public Nuplet[] superieur(Nuplet[] t, int att, Object v) {
        // TODO Auto-generated method stub
        Vector<Nuplet> value = new Vector<Nuplet>();
        for(Nuplet n: t) {
                if((byte)(n.getAtt(att)) > (byte) v){
                        value.addElement(n);
           }
        }
        Nuplet[] ret = new Nuplet[value.size()];
        for(int i=0;i<value.size();i++)
                ret[i] = value.elementAt(i);
        return ret;
    }
```

Troisièmement, la method **inferieur**:

```
public Nuplet[] inferieur(Nuplet[] t, int att, Object v) {
                // TODO Auto-generated method stub
                Vector<Nuplet> value = new Vector<Nuplet>();
                for(Nuplet n: t) {
                        if((byte)(n.getAtt(att)) < (byte) v){
                                value.addElement(n);
                        }
                }
                Nuplet[] ret = new Nuplet[value.size()];
                for(int i=0;i<value.size();i++)
                        ret[i] = value.elementAt(i);
                return ret;
        }
```

Dans le ***main.java***:

```
// Utilisation de Restriction
    System.out.println("=============");
    System.out.println("Restriction");
    Restriction testRestriction = new RestrictionInt();
    System.out.println("Test Egalite");
    Nuplet[] testEgalite = testRestriction.egalite(fullscan, 3, (byte) 50);
    for(Nuplet n : testEgalite)
        System.out.println(n.toString());
    System.out.println("Test Superieur");
    Nuplet[] testSuperieur = testRestriction.superieur(fullscan, 3, (byte)
50);
    for(Nuplet n : testSuperieur)
        System.out.println(n.toString());
    System.out.println("Test Inferieur");
    Nuplet[] testInferieur = testRestriction.inferieur(fullscan, 3, (byte)
50);
    for(Nuplet n : testInferieur)
        System.out.println(n.toString());
```

Le resultat:

Test Egalite

```
Restriction
Test Egalite
47      48      49      50      51      52      53      54      55      56
```

Test Superieur

```
Test Superieur
48      49      50      51      52      53      54      55      56      57
49      50      51      52      53      54      55      56      57      58
50      51      52      53      54      55      56      57      58      59
51      52      53      54      55      56      57      58      59      60
52      53      54      55      56      57      58      59      60      61
53      54      55      56      57      58      59      60      61      62
54      55      56      57      58      59      60      61      62      63
55      56      57      58      59      60      61      62      63      64
56      57      58      59      60      61      62      63      64      65
57      58      59      60      61      62      63      64      65      66
58      59      60      61      62      63      64      65      66      67
59      60      61      62      63      64      65      66      67      68
60      61      62      63      64      65      66      67      68      69
61      62      63      64      65      66      67      68      69      70
62      63      64      65      66      67      68      69      70      71
63      64      65      66      67      68      69      70      71      72
64      65      66      67      68      69      70      71      72      73
65      66      67      68      69      70      71      72      73      74
66      67      68      69      70      71      72      73      74      75
67      68      69      70      71      72      73      74      75      76
68      69      70      71      72      73      74      75      76      77
69      70      71      72      73      74      75      76      77      78
70      71      72      73      74      75      76      77      78      79
71      72      73      74      75      76      77      78      79      80
72      73      74      75      76      77      78      79      80      81
73      74      75      76      77      78      79      80      81      82
74      75      76      77      78      79      80      81      82      83
75      76      77      78      79      80      81      82      83      84
76      77      78      79      80      81      82      83      84      85
77      78      79      80      81      82      83      84      85      86
78      79      80      81      82      83      84      85      86      87
79      80      81      82      83      84      85      86      87      88
```

Test Inferieur

```
Test Inferieur
0       1       2       3       4       5       6       7       8       9
1       2       3       4       5       6       7       8       9       10
2       3       4       5       6       7       8       9       10      11
3       4       5       6       7       8       9       10      11      12
4       5       6       7       8       9       10      11      12      13
5       6       7       8       9       10      11      12      13      14
6       7       8       9       10      11      12      13      14      15
7       8       9       10      11      12      13      14      15      16
8       9       10      11      12      13      14      15      16      17
9       10      11      12      13      14      15      16      17      18
10      11      12      13      14      15      16      17      18      19
11      12      13      14      15      16      17      18      19      20
12      13      14      15      16      17      18      19      20      21
13      14      15      16      17      18      19      20      21      22
14      15      16      17      18      19      20      21      22      23
15      16      17      18      19      20      21      22      23      24
16      17      18      19      20      21      22      23      24      25
17      18      19      20      21      22      23      24      25      26
18      19      20      21      22      23      24      25      26      27
19      20      21      22      23      24      25      26      27      28
20      21      22      23      24      25      26      27      28      29
21      22      23      24      25      26      27      28      29      30
22      23      24      25      26      27      28      29      30      31
23      24      25      26      27      28      29      30      31      32
24      25      26      27      28      29      30      31      32      33
25      26      27      28      29      30      31      32      33      34
26      27      28      29      30      31      32      33      34      35
27      28      29      30      31      32      33      34      35      36
28      29      30      31      32      33      34      35      36      37
29      30      31      32      33      34      35      36      37      38
30      31      32      33      34      35      36      37      38      39
31      32      33      34      35      36      37      38      39      40
32      33      34      35      36      37      38      39      40      41
33      34      35      36      37      38      39      40      41      42
```

Exercice 3

impl/ProjectionImpl.java

```java
/*
* @param Nuplet t
* @param atts: des columnes projections
*/
public Nuplet[] project(Nuplet[] t, int[] atts) {
    // TODO Auto-generated method stub
    NupletInt[] tab = new NupletInt[t.length];
    for (int i=0; i< t.length; i++) {
        tab[i] = new NupletInt(atts.length);
        for (int j=0; j < atts.length; j++) {
            tab[i].putAtt(j, t[i].getAtt(atts[j]));
        }
    }
    return tab;
```

Dans le ***Main.java***:

```
// Utilisation de Projection
System.out.println("=============");
System.out.println("Projection");
Projection testProjection = new ProjectionInt();
int[] atts = {1,2,3};
Nuplet[] testProj = testProjection.project(fullscan, atts);
for(Nuplet n : testProj)
    System.out.println(n.toString());
```

On crée une nouvelle table testProj qui contient 3 columnes 1,2,3 dans la table t. Cette table est utilisé pour des exercice suivants.

Le résultat:

```
Projection
1       2       3
2       3       4
3       4       5
4       5       6
5       6       7
6       7       8
7       8       9
8       9       10
9       10      11
10      11      12
11      12      13
12      13      14
13      14      15
14      15      16
15      16      17
16      17      18
17      18      19
18      19      20
19      20      21
20      21      22
21      22      23
22      23      24
23      24      25
24      25      26
25      26      27
26      27      28
27      28      29
28      29      30
```

Exercice 4

À partir de cet exercice, il faut ajouter une nouvelle méthod qui permet fusionner les deux nuplets.

```
public Nuplet joinTwoNuplet(Nuplet n1, Nuplet n2, int attributeMerge) {
   NupletInt merged = new NupletInt(n1.size() + n2.size() - 1);
```

```
        for (int i = 0; i < n1.size(); i++)
            merged.putAtt(i, n1.getAtt(i));
        for (int i = 0; i < attributeMerge; i++)
            merged.putAtt(n1.size() + i, n2.getAtt(i));
        for (int i = attributeMerge + 1; i < n2.size(); i++)
            merged.putAtt(n1.size() + i - 1, n2.getAtt(i));
        return merged;
    }
```

L'algorithme boucles imbriquées a pseudo-code comme ci-après:

```
for each row R1 in the outer table
    for each row R2 in the inner table
        if R1.join_column = R2.join_column
            return (R1, R2)
```

impl/JointureBI.java

```
public Nuplet[] jointure(Nuplet[] t1, Nuplet[] t2, int att1, int att2) {
    Vector<Nuplet> value = new Vector<Nuplet>();
    for(Nuplet n1: t1) {
        for(Nuplet n2: t2) {
            if((byte) n1.getAtt(att1) == (byte) n2.getAtt(att2)) {
                NupletInt tab = new NupletInt(n1.size() + n2.size() -1);
                tab = (NupletInt) joinTwoNuplet(n1, n2, att2);
                value.addElement(tab);
            }
        }
    }
    Nuplet[] ret = new Nuplet[value.size()];
    for(int i=0;i<value.size();i++)
        ret[i] = value.elementAt(i);
    return ret;
}
```

Dans le ***Main.java***:

```
// Utilisation de JointureBI
    System.out.println("JointureBI");
    Jointure testJointure = new JointureBI();
    Nuplet[] testJointureBI = testJointure.jointure(testProj, testProj2, 0,
0);
    for(Nuplet n : testJointureBI) {
        System.out.println(n.toString());
    }
```

Le résultat:

```
JointureBI
2       3       4       3       4
3       4       5       4       5
4       5       6       5       6
5       6       7       6       7
6       7       8       7       8
7       8       9       8       9
8       9       10      9       10
9       10      11      10      11
10      11      12      11      12
11      12      13      12      13
12      13      14      13      14
13      14      15      14      15
14      15      16      15      16
15      16      17      16      17
16      17      18      17      18
17      18      19      18      19
18      19      20      19      20
19      20      21      20      21
20      21      22      21      22
21      22      23      22      23
22      23      24      23      24
23      24      25      24      25
24      25      26      25      26
25      26      27      26      27
26      27      28      27      28
27      28      29      28      29
```

Amélioration des opérateurs

Exercice 5:

impl/JointureH.java

```
public Nuplet[] jointure(Nuplet[] t1, Nuplet[] t2, int att1, int att2) {
    // TODO Auto-generated method stub
    Vector<Nuplet> value = new Vector<Nuplet>();
    // Step 1: Hash phase
    HashMap<Object, Nuplet> hash = new HashMap<Object, Nuplet>();
    for (Nuplet n1 : t1) {
        hash.put(n1.getAtt(att1), n1);
    }
    // Step 2: Join phase
    for (Nuplet n2: t2) {
        Nuplet hash_test = hash.get(n2.getAtt(att2));
        if(hash_test != null) {
            NupletInt tab = new NupletInt(hash_test.size() + n2.size() -
1);
            tab = (NupletInt) joinTwoNuplet(hash_test, n2, att2);
            value.addElement(tab);
        }
    }
    Nuplet[] ret = new Nuplet[value.size()];
```

```
        for(int i=0;i<value.size();i++)
            ret[i] = value.elementAt(i);
        return ret;
    }
```

Dans le ***Main.java***:

```
System.out.println("JointureH");
Jointure testJointure_Hash_Join = new JointureH();
Nuplet[] testJointureH = testJointure_Hash_Join.jointure(testProj,
testProj2, 0, 0);
for(Nuplet n : testJointureH)
    System.out.println(n.toString());
```

Le résultat:

```
JointureH
2       3       4       3       4
3       4       5       4       5
4       5       6       5       6
5       6       7       6       7
6       7       8       7       8
7       8       9       8       9
8       9       10      9       10
9       10      11      10      11
10      11      12      11      12
11      12      13      12      13
12      13      14      13      14
13      14      15      14      15
14      15      16      15      16
15      16      17      16      17
16      17      18      17      18
17      18      19      18      19
18      19      20      19      20
19      20      21      20      21
20      21      22      21      22
21      22      23      22      23
22      23      24      23      24
23      24      25      24      25
24      25      26      25      26
```

Exercice 6

L'algorithm Sort Merge a le pseudo-code comme ci-après:

```
/* Step 1: Sorting */
sort R on R.A
sort Q on Q.B
/* Step 2: Merging */
r = first tuple in R
s = first tuple in S
```

```
while r != EOF and s != EOF do
    if r.A > s.B then
        s = next tuple in S after q
    else
    if r.A < s.B then
        r = next tuple in R after r
    else
     put (r,s) in the output relation

     r = next tuple in R after r
     s = next tuple in Q after q
endwhile
```

Parce que cette partie est trop long, donc je ne l'ai pas ajouté dans le rapport. J'explique quelques méthodes:

Méthode	Description
hasNext	n' = Next tuple in t after n. Par exemple: r = next tuple in R after r
triFusion	Appliquer l'algoritm Tri-fusion
isCheckEOF	cette méthode vérifie si t est EOF (dans le pseudo-code: r != EOF)

Le résultat:

```
JointureBI
2       3       4       3       4
3       4       5       4       5
4       5       6       5       6
5       6       7       6       7
6       7       8       7       8
7       8       9       8       9
8       9       10      9       10
9       10      11      10      11
10      11      12      11      12
11      12      13      12      13
12      13      14      13      14
13      14      15      14      15
14      15      16      15      16
15      16      17      16      17
16      17      18      17      18
17      18      19      18      19
18      19      20      19      20
```

```
JointureH
2       3       4       3       4
3       4       5       4       5
4       5       6       5       6
5       6       7       6       7
6       7       8       7       8
7       8       9       8       9
8       9       10      9       10
9       10      11      10      11
10      11      12      11      12
11      12      13      12      13
12      13      14      13      14
13      14      15      14      15
14      15      16      15      16
15      16      17      16      17
16      17      18      17      18
17      18      19      18      19
18      19      20      19      20
19      20      21      20      21
20      21      22      21      22
```

)

```
JointureS
2       3       4       3       4
3       4       5       4       5
4       5       6       5       6
5       6       7       6       7
6       7       8       7       8
7       8       9       8       9
8       9       10      9       10
9       10      11      10      11
10      11      12      11      12
11      12      13      12      13
12      13      14      13      14
13      14      15      14      15
14      15      16      15      16
15      16      17      16      17
16      17      18      17      18
17      18      19      18      19
18      19      20      19      20
19      20      21      20      21
```

Exercice : des plans d'exécution

On fait 6 plans d'exécution genre ca:

```
arbre 1 + BI => plan 1
arbre 2 + BI => plan 2
arbre 3 + BI => plan 3
arbre 1 + H => plan 4
arbre 2 + H => plan 5
arbre 3 + H => plan 6
```

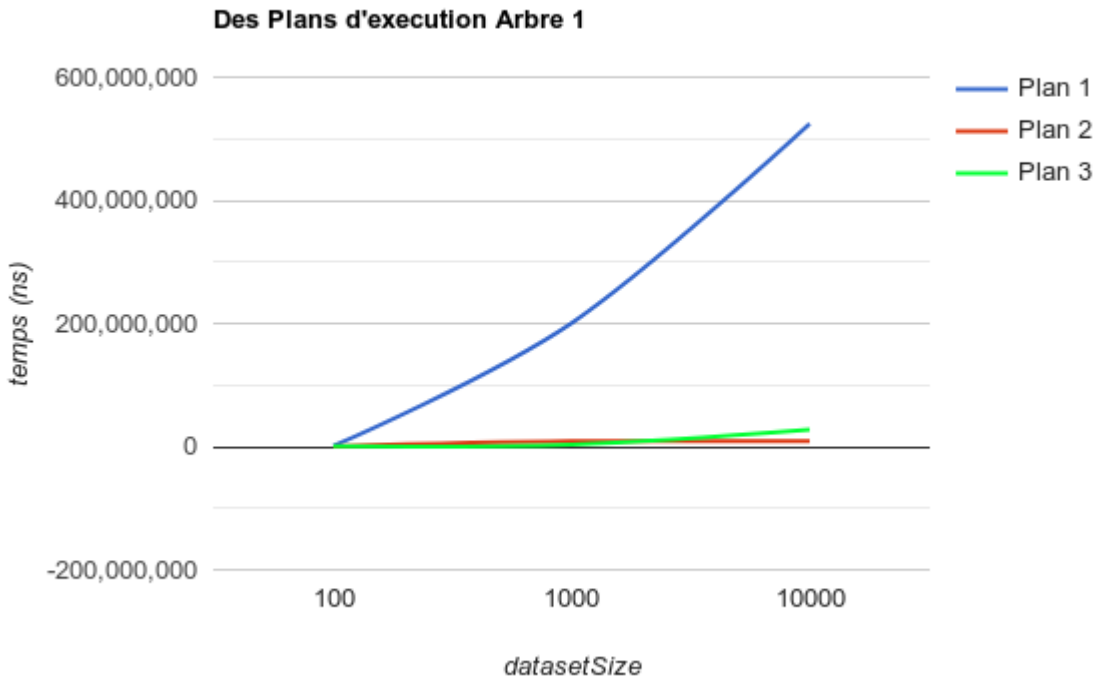
Pour mesure la temps d'excution, on lancera au début du programme:

```
final long start = System.nanoTime();
```

Et à la fin du programme:

```
final long end = System.nanoTime();
System.out.println("Temps d'exécution:: " + ((end_1 - start_1) ) );
```

Arbre 1:



Avec datasetSize = 100:

```
Plan 1
Temps d'exécution:: 1069264 ns
Plan 2
Temps d'exécution:: 151678 ns
Plan 3
Temps d'exécution:: 485548 ns
```

Avec datasetSize = 1000:

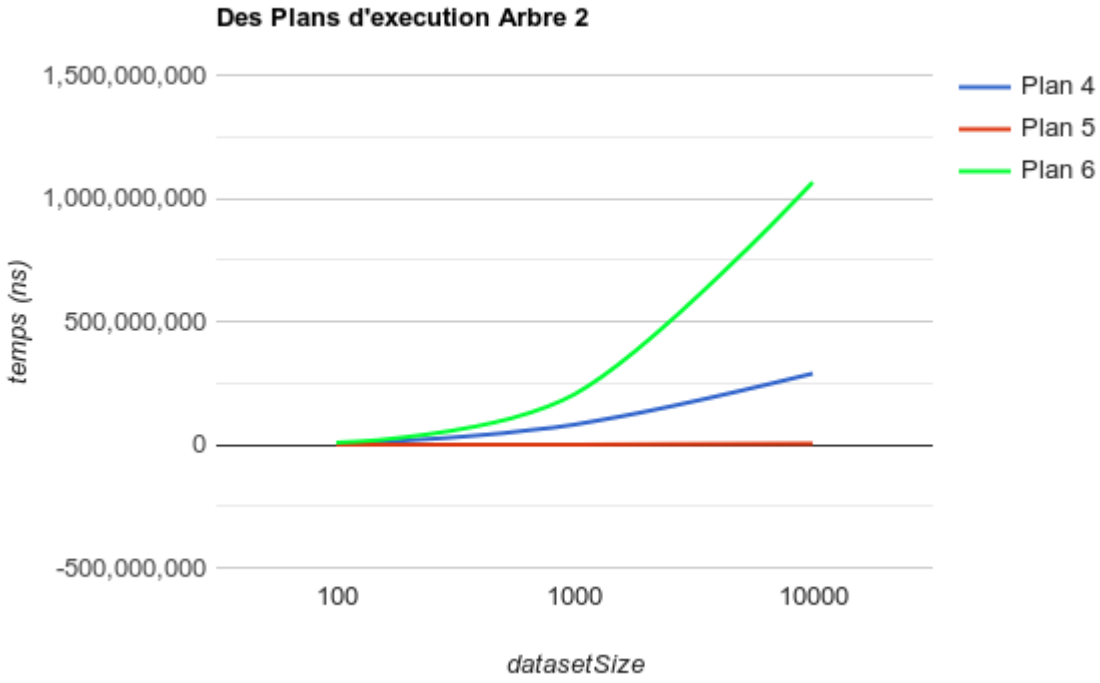
```
Plan 1
Temps d'exécution:: 200599705 ns
Plan 2
Temps d'exécution:: 8584955 ns
```

```
Plan 3
Temps d'exécution:: 3278174 ns
```

Avec datasetSize = 10000:

```
Plan 1
Temps d'exécution:: 524712129 ns
Plan 2
Temps d'exécution:: 8641058 ns
Plan 3
Temps d'exécution:: 27491182 ns
```

Arbre 2:



Avec datasetSize = 100:

```
Plan 1
Temps d'exécution:: 1122105 ns
Plan 2
Temps d'exécution:: 208815 ns
Plan 3
Temps d'exécution:: 8710989 ns
```

Avec datasetSize = 1000:

```
Plan 1
Temps d'exécution:: 82156042 ns
Plan 2
Temps d'exécution:: 635866 ns
Plan 3
Temps d'exécution:: 204428688 ns
```

Avec datasetSize = 10000:

```
Plan 1
Temps d'exécution:: 288411451 ns
Plan 2
Temps d'exécution:: 4460678 ns
Plan 3
Temps d'exécution:: 1065339674 ns
```

Mises à jour

Exercice 8:

Parce que cette partie est aussi trop long, donc je ne les ai pas ajouté dans le rapport. Je donne des résultats

D'abord, on crée une nouvelle table:

```
System.out.println("Création d'une table t1");
Table t1 = new TableInt("table1", nupletSize);
for(int i=0;i<datasetSize;i++){
    t1.put(tab[i]);
}
System.out.println("===== LECTURE VIA FULLSCAN =====");
for(Nuplet n : t1.fullScan())
    System.out.println(n.toString());
```

Ensuit, on ajoute une nouvelle Nuplet:

```
System.out.println("\n===== INSERT =====");
Nuplet newNuplet = new NupletInt(new byte[] {0,50,2,3,4,5,6,7,8,9});
System.out.println("\n t1.insert(nouveau_nuplet)");
t1.insert(newNuplet);
for(Nuplet n : t1.fullScan())
    System.out.println(n.toString());
```

Le résultat:

```
98      99      100     101     102     103     104     105     106     107
99      100     101     102     103     104     105     106     107     108
0       50      2       3       4       5       6       7       8       9
```

On met à jour une nouvelle value:

```
System.out.println("\n===== UPDATE =====");
System.out.println("\nt1.update(1, 50, 51);");
t1.update(1, (byte) 50, (byte) 51);
for(Nuplet n : t1.fullScan())
    System.out.println(n.toString());
```

Le résultat:

```
94      95      96      97      98      99      100     101     102     103
95      96      97      98      99      100     101     102     103     104
96      97      98      99      100     101     102     103     104     105
97      98      99      100     101     102     103     104     105     106
98      99      100     101     102     103     104     105     106     107
99      100     101     102     103     104     105     106     107     108
0       51      2       3       4       5       6       7       8       9
```

Enfin, on surrime le Nuplet dernier:

```
System.out.println("\n===== DELETE =====");
System.out.println("\nt1.delete(1, 51);");
t1.delete(1, (byte) 51);
for(Nuplet n : t1.fullScan())
    System.out.println(n.toString());
```

Le résultat:

```
95      96      97      98      99      100     101     102     103     104
96      97      98      99      100     101     102     103     104     105
97      98      99      100     101     102     103     104     105     106
98      99      100     101     102     103     104     105     106     107
99      100     101     102     103     104     105     106     107     108
```