Compte-rendu Projet SGBD

Perrin Matthieu - Gallay Jules

Décembre 2019

Sommaire

1 Présentation

Ce projet a pour but de comparer les coûts d'exécution de différents algorithmes de jointure dans des environnements variés. Le coût sera exprimé par le nombre de blocs lus et/ou écrits pour l'exécution des différents algorithmes, de plus il s'agira du coût réel d'exécution sur des tables que l'on aura créées.

2 Description des classes

Pour simuler ces différents tests, nous aurons besoins de briques de base que l'on utilisera pour calculer le coût des algorithmes. Il va s'agir des classes « Table », « Bloc », « Buffer », « Enregistrement », « Index » et « Segment ».

Ces classes fourniront les opérations nécessaires à leur mise en œuvre, des méthodes lectures/écritures seront mises à disposition par exemple.

Ci-dessous se trouve le diagramme de classe sur lequel nous nous sommes appuyés pour réaliser les différentes briques.

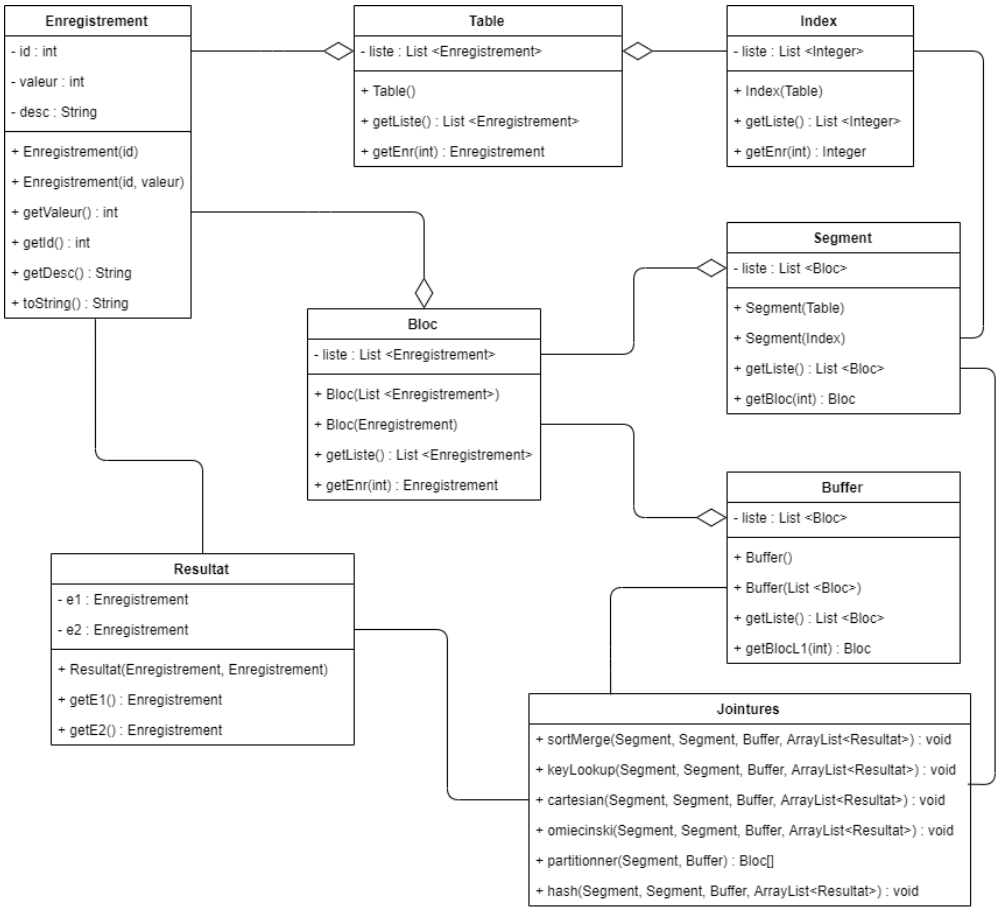

Diagramme de classe

2.1 Classe « Enregistrement »

La classe « Enregistrement » est une classe qui contiendra les valeurs que l'on comparera ensuite dans les algorithmes de jointure. Elle possède 3 attributs (id, desc et valeur) qui simule des informations que l'on peut inscrire dans un tuple. Les accesseurs ainsi que le constructeur n'ont rien de particulier, les valeurs sont définies par défaut aléatoirement et on créer un nouvel enregistrement en indiquant seulement l'id dans les paramètres. Un deuxième constructeur est disponible afin d'indiquer la valeur que l'on veut inscrire, et il est utilisé lors de la construction d'index sur l'attribut valeur.

```
//Constructeur
public Enregistrement(int p_id){
    this.id=p_id;

    this.valeur = 1 + (int)(Math.random() * 99);

    byte[] array = new byte[7]; // length is bounded by 7
    new Random().nextBytes(array);
    this.desc = new String(array, Charset.forName("UTF-8"));
}

public Enregistrement(int p_id, int p_valeur){
    this.id=p_id;
    this.valeur = p_valeur;
}
```

Cette classe implémente l'interface Comparable et la méthode CompareTo est surchargée sur l'attribut valeur. Cela permet de trier facilement les tables sur cet attribut.

```
@Override
    public int compareTo(Enregistrement e) {
      return this.valeur - e.getValeur();
    }
```

Une méthode toString est aussi implémenté où l'on affiche les attributs de la classe, l'un au-dessus des autres.

```
@Override
    public String toString(){
        String res = ""+this.id+"\nValeur : "+this.valeur+"\
    nDescription : "+this.desc;
        return res;
    }
```

2.2 Classe « Table »

Cette classe possède un seul attribut qui est une ArrayList d'Enregistrement. Un accesseur permet de récupérer la liste entièrement et un autre permet de récupérer l'enregistrement selon sa position dans la liste.

```
//Attribut
private List <Enregistrement> liste = new ArrayList<Enregistrement
    >();

//Getter
public List<Enregistrement> getListe() {
    return liste;
}

public Enregistrement getEnr(int i){
    return this.liste.get(i);
}
```

Le constructeur est par défaut et créer une liste de 100 enregistrements qui auront des valeurs aléatoires.

```
//Constructeur
public Table(){
    int i=0;
    for(i=0; i<100; i++){
        this.liste.add(new Enregistrement(i));
    }
}
```

2.3 Classe « Index »

La classe « Index » possède une ArrayList d'Enregistrement comme attribut. Cette liste représente la liste des valeurs des enregistrements d'une table. Les accesseurs sont les mêmes que pour la classe « Table » mais pour la liste d'Integer et non la liste d'Enregistrement. Le constructeur récupère les valeurs de la table mise en paramètre et les inscrit dans la liste que l'on a pour attribut.

```
// Constructeur
public Index(Table t){
    Iterator<Enregistrement> iter = t.getListe().iterator();
    while (iter.hasNext()) {
        Enregistrement element = iter.next();
        int valeur = element.getValeur();
        this.liste.add(valeur);
    }
}
```

2.4 Classe « Bloc »

Cette classe est similaire à la classe « Table », la seule différence est le constructeur qui prend en paramètre une liste d'enregistrement. Dans ce projet, un bloc peut contenir au maximum 10 enregistrements.

```
// Constructeur
public Bloc(List <Enregistrement> p_liste){
    this.liste=p_liste;
}
```

2.5 Classe « Segment »

La classe « Segment » possède un attribut ArrayList de Bloc avec des accesseurs pour récupérer la liste ainsi qu'un bloc selon sa position. Deux constructeurs sont mis à disposition, le premier avec une table en paramètre et le deuxième avec un index.

```
//Attribut
private List <Bloc> liste = new ArrayList<Bloc>();

//Getter
public List<Bloc> getListe() {
    return liste;
}

public Bloc getBloc(int i){
    return this.liste.get(i);
}
```

Ce constructeur permet de récupérer chaque enregistrement de la table pour ensuite créer un bloc avec une partie de cette liste d'enregistrement qui sera inséré dans le segment. Comme dit précédemment, un bloc ne contient que 10 enregistrements, donc il faut ajouter un nouveau bloc à la liste tous les 10 enregistrements

```
//Constructeur avec table
public Segment(Table t){
    int count = t.getListe().size();
    int i=0;
    int j=0;
    for (i=0; i<count; i=i+10){

        List <Enregistrement> li_enr = new ArrayList<Enregistrement
    >();
        for(j=0; j<10; j++){
            li_enr.add(t.getListe().get(i+j));
        }
        Bloc b = new Bloc(li_enr);
        this.liste.add(b);
```

```
    }
}
```

Pour l'index c'est le même principe sauf lorsque l'on remplit la liste d'enregistrement pour créer le bloc. Comme l'index ne possède que la valeur il nous faut aussi un id que l'on va renseigner en déclarant un nouvel enregistrement. La description quant à elle n'est pas initialisée, pour simuler un index qui ne porte que sur le champ valeur.

```
//Constructeur avec index
public Segment(Index in){
    int count = in.getListe().size();
    int i=0;
    int j=0;
    for (i=0; i<count; i=i+10){

        List <Enregistrement> li_enr = new ArrayList<Enregistrement
    >();
        for(j=0; j<10; j++){
            li_enr.add(new Enregistrement(i+j, in.getListe().get(i+
    j)));
        }
        Bloc b = new Bloc(li_enr);
        this.liste.add(b);
    }
}
```

2.6 Classe « Buffer »

Cette dernière brique est le buffer qui va être utilisé pour stocker les blocs lus après la jointure que l'on aura choisi d'effectuer. On y retrouve donc une arrayList de Bloc avec ces accesseurs.

```
//Attribut
private List <Bloc> liste;

//Getter
public List<Bloc> getListe() {
    return liste;
}

public Bloc getBlocL1(int i){
    return this.liste.get(i);
}

//Constructeurs
public Buffer(List <Bloc> l1){
    this.liste= l1;
}

public Buffer(){
    this.liste= new ArrayList<Bloc>();
}
```

2.7 Autres classes

Notre projet comporte d'autres classes dont la première est la classe « Resultat » qui possède 2 attributs « Enregistrement » que l'on va utiliser afin de récupérer les enregistrements qui ont une valeur commune lors des jointures.

```
//Attributs
private Enregistrement e1;
private Enregistrement e2;

//Getter
public Enregistrement getE1() {
    return e1;
}

public Enregistrement getE2() {
    return e2;
}

//Constructeur
public Resultat(Enregistrement en1, Enregistrement en2){
    this.e1=en1;
    this.e2=en2;
}
```

La deuxième est la classe « Jointures » qui possède tous les algorithmes que nous allons voir après.

On va y retrouver les 5 algorithmes de jointures : Key-lookup, Sort-merge, Produit Cartésien, Hashage et Omiecinski

On a décidé d'effectuer le tri avant la fonction et non à l'intérieur de celle-ci pour plus de facilité. On sélectionnera si les listes seront triées ou non dans le menu

3 Description des algorithmes

3.1 Sort-merge

L'algorithme Sort-merge est composé en 2 étapes : trié les deux relations sur le critère de jointure puis fusionner ces relations pour enfin calculer le coût.

On a décidé d'effectuer le tri avant la fonction et non à l'intérieur de celle-ci pour plus de facilité. On sélectionnera si les listes seront triées ou non dans le menu. Nous trions les relations grâce à la méthode sort de la classe List, par conséquent nous ne pouvons pas calculer le coût du tri, car les relations sont triées avant d'être « convertie » en segments et en blocs.

Notre fonction possède 4 paramètres : 2 segments, 1 buffer et une liste résultat. Premièrement on récupère la liste de bloc pour chaque segment et on déclare 2 variables pos1 et pos2 qui vont donner la position dans les blocs afin de pouvoir tout comparer. Les variables stock1 et stock2 serviront de stockage de cette position, pour vérifier si on entre dans un nouveau bloc. Elles vont nous permettre de calculer le coût de l'algorithme

```
//Attributs
List <Bloc> lb1 = s1.getListe();
List <Bloc> lb2 = s2.getListe();

int pos1=0;
int pos2=0;

int stock1=0;
int stock2=0;

//Boucle
...
```

Une boucle while est utilisé afin de pouvoir comparer chaque valeur des 2 segments. On récupère l'Enregistrement ainsi que la valeur de celui-ci pour les deux segments que l'on va ensuite comparer.

```
//Attributs
...

//Boucle
while (pos1 <= s1.getListe().size()*s1.getBloc(0).getListe().size()
    -1 && pos2 <= s2.getListe().size()*s2.getBloc(0).getListe().
    size()-1){

    Enregistrement enr1 = s1.getBloc(pos1/10).getEnr(pos1%10);
    Enregistrement enr2 = s2.getBloc(pos2/10).getEnr(pos2%10);

    int val1=s1.getBloc(pos1/10).getEnr(pos1%10).getValeur();
    int val2=s2.getBloc(pos2/10).getEnr(pos2%10).getValeur();

    //Test des valeurs
    ...

}
```

Si les deux valeurs sont égales alors on ajoute le couple de valeurs dans la liste Résultat, on stocke la position que l'on incrémente ensuite et pour finir on ajoute le bloc au buffer.

```
//Boucle
...

//Verification des resultat
if(val1 == val2){
    Resultat r = new Resultat(enr1, enr2);
    liste.add(r);
    stock2 = pos2/10;
    pos2++;
    if(pos2/10>stock2){
        b.getListe().add(s2.getBloc(stock2));
    }
}

//Suite des verifications
...
```

Sinon si la valeur du segment 1 est supérieur à celle du segment 2 alors on incrémente juste la position et on ajoute le bloc précédent au buffer si on change de bloc.Et inversement si la valeur de segment 2 est supérieur à celle du segment 1.

```
//Boucle
...

//Suite des verifications
else if(val1 > val2){
    stock2 = pos2/10;
    pos2++;
    if(pos2/10>stock2){
        b.getListe().add(s2.getBloc(stock2));
    };
}
else{
    stock1 = pos1/10;
    pos1++;
    if(pos1/10>stock1){
        b.getListe().add(s1.getBloc(stock1));
    }
}
```

Enfin on ajoute les derniers blocs au buffer si la position finale est supérieure à celle stockée en mémoire

```
//En dehors de la boucle
if(pos1/10>stock1){
            b.getListe().add(s1.getBloc((pos1-1)/10));
}
if(pos2/10>stock2){
            b.getListe().add(s2.getBloc((pos2-1)/10));
}
```

3.2 Key-lookup

L'algorithme Key-lookup se compose de boucles imbriquées pour comparer les tuples et les paramètres sont les mêmes que pour le premier algorithme. La première boucle récupère les valeurs du segment 1 qu'on ajoute au buffer puis la deuxième boucle récupère les valeurs du 2e segment qui sont aussi ajoutées au buffer.

```
//1ere boucle
for(i=0; i<s1.getListe().size()*s1.getBloc(0).getListe().size(); i
    ++){
    //recupere enregistrement
    Enregistrement enr1 = s1.getBloc(i/10).getEnr(i%10);
    int val1=s1.getBloc(i/10).getEnr(i%10).getValeur();
    //Stocke dans le bloc
    if(i/10>stock){
        b.getListe().add(s1.getBloc(stock));
    }

    //2e boucle
    for(j=0; j<s2.getListe().size()*s2.getBloc(0).getListe().size()
    ; j++){
        //Stocke dans le bloc
        if(j/10>stockj){
            b.getListe().add(s2.getBloc(stockj));
        }
        //recupere enregistrement
        Enregistrement enr2 = s2.getBloc(j/10).getEnr(j%10);
        int val2=s2.getBloc(j/10).getEnr(j%10).getValeur();

        //Verification des resultats
        ...
    }
    ...
}
```

Ensuite on compare les valeurs et si elles sont égales alors on les ajoute à la liste Résultat. On ajoute les derniers blocs à la fin comme pour Sort-merge.

```
//Boucle
...

//Verification des resultats
if(val1 == val2){
    Resultat r = new Resultat(enr1, enr2);
    liste.add(r);
}
```

3.3 Produit Cartésien

Le principe de cet algorithme est de charger plusieurs blocs de la première relation et un bloc de la deuxième pour ensuite le comparer avec les autres blocs. Cette étape est répétée plusieurs fois jusqu'à ce que tous les blocs soient lus.

Pour ce qui est de notre fonction, les paramètres sont toujours les mêmes avec plusieurs variables déclarées comme par exemple une variable pour connaître le nombre de blocs que l'on chargera en mémoire centrale pour la première relation (qu'on appelle "capa" dans notre code).

Une première boucle permet de répéter le nombre de fois nécessaire pour que tous blocs du segment 1 soit chargés et comparés.

```
//On repete size/capa fois (ici 2)
for(i=0; i<(s1.getListe().size())/capa; i++){

    //on charge les blocs a lire de r1
    lb1.clear();
    for(j=0; j<capa; j++){
        lb1.add(s1.getBloc(i*5+j));
    }

    //Boucle pour recuperer les autres blocs
    ...

}
```

Une deuxième boucle charge tous les blocs du segment 2 un à un pour les comparer avec les blocs du segment 1 chargés. Lorsque deux valeurs sont égales le traitement est similaire que les autres algorithmes.

```
//Boucle
...

//on parcours tous les blocs de r2 pour les comparer avec les 5
    blocs
for(k=0; k<s2.getListe().size()*s2.getBloc(0).getListe().size(); k
    ++){
    if(k/10>stock_k){
        b.getListe().add(s2.getBloc(stock_k));
    }

    //Recuperation des valeurs puis comparaison
    ...

}
```

3.4 Omiecinski

Cet algorithme ressemble au Sort-merge mais ici des indexes secondaires sont utilisés. Premièrement, on trie les indexes sur leurs attributs, ce qui est effectué avant de lancer la fonction, puis on effectue les comparaisons entre les valeurs.

La fonction possède deux boucles imbriquées qui permet de parcourir chaque bloc que l'on va ensuite traiter. La première boucle parcourt les éléments de la première relation.

```
//Boucle 1
 while(cpt1 < s1.getListe().size()*s1.getBloc(0).getListe().size()
    -1){

    Enregistrement enr1 = s1.getBloc(cpt1/10).getEnr(cpt1%10);
    Enregistrement enr2 = s2.getBloc(cpt2/10).getEnr(cpt2%10);
    int val1=s1.getBloc(cpt1/10).getEnr(cpt1%10).getValeur();
    int val2=s2.getBloc(cpt2/10).getEnr(cpt2%10).getValeur();

    //Boucle 2
    ...

    //Envoie des valeurs a la liste Resultat
    ...

}
```

La deuxième boucle parcourt non seulement les éléments de la relation 2 mais vérifie si les éléments sont différents.

```
//A l'interieur de la boucle 1
...

//Boucle 2
while(val1 != val2 && cpt2<s2.getListe().size()*s2.getBloc(0).
    getListe().size()-1){

    stock2=cpt2/10;
    cpt2++;
    if(cpt2/10>stock2){
        b.getListe().add(s2.getBloc(stock2));
    }
    val2=s2.getBloc(cpt2/10).getEnr(cpt2%10).getValeur();
    enr2 = s2.getBloc(cpt2/10).getEnr(cpt2%10);

}

//Envoie des valeurs a la liste Resultat
    ...
```

Si les valeurs sont égales alors on sort de la boucle pour les envoyer dans la liste Résultat.

```
//Boucle 2
...

//Envoie des valeurs a la liste Resultat
Resultat r = new Resultat(enr1, enr2);
liste.add(r);

stock1=cpt1/10;
cpt1++;
if(cpt1/10>stock1){
        b.getListe().add(s1.getBloc(stock1));
}
cpt2=0;
```

3.5 Hachage

3.5.1 Fonction partition

On crée d'abord un tableau de Blocs qui récupérera les enregistrements après partition puis on parcourt les éléments du segment mis en paramètre.

Chaque enregistrement est récupéré puis enregistré dans le tableau de Blocs, stocké à l'indice de leur valeur. Par exemple un enregistrement de valeur 47 sera dans le bloc à l'indice 47 dans le tableau. Une fois tout ceci effectué on renvoie le tableau de Blocs

```
Bloc tab[] = new Bloc[100];

//on stocke les enregistrements a l'indice de leur valeur
for(i=0; i<s1.getListe().size()*s1.getBloc(0).getListe().size(); i
    ++){

    if(i/10>stock){
        b.getListe().add(s1.getBloc(stock));
    }
    Enregistrement enr1 = s1.getBloc(i/10).getEnr(i%10);
    int val1=s1.getBloc(i/10).getEnr(i%10).getValeur();
    tab[val1]= new Bloc(enr1);

    stock=i/10;
}
if(i/10>stock){
            b.getListe().add(s1.getBloc((i-1)/10));
}

return tab
```

3.5.2 Fonction hash

Dans cette fonction on appelle la fonction ci-dessus pour chaque segment en paramètre puis une première boucle récupère un élément du premier et du deuxième tableau où chaque bloc est ajouté au buffer.

```
Bloc[] tab1= partitionner(s1,b);
Bloc[] tab2= partitionner(s2,b);

//Boucle 1
for(i=0; i<100; i++){
    Bloc b1 = tab1[i];
    Bloc b2 = tab2[i];

    b.getListe().add(b1);
    b.getListe().add(b2);

    //Recuperation de la taille du tableau b2 pour la deuxi me
    boucle
    ...
    //Boucle 2
    ...
}
```

Une deuxième boucle va ajouter les enregistrements du 2e bloc à une variable puis une dernière boucle récupère les enregistrements du 1er bloc.

```
//Boucle 1
...

//Boucle 2
for(j=0; j<sizeb2; j++){
    Enregistrement enr2 = b2.getEnr(j);

    //Recuperation de la taille du tableau b1 pour la derniere
    boucle
    ...
    //Boucle 3
    ...
}
```

Ces enregistrements seront envoyés dans la liste Résultat car les enregistrements partitionnés précédemment sont stockés à l'indice de leur valeur c'est-à-dire que deux enregistrements au même indice sont forcément égaux.

```
//Boucle 2
...

//Boucle 3
for(k=0; k<sizeb1; k++){
    Enregistrement enr1 = b1.getEnr(k);
    Resultat r = new Resultat(enr1, enr2);
    liste.add(r);
}
```

4 Interface

Pour pouvoir effectuer tous les tests nécessaires, nous avons décidé de faire une interface graphique avec NetBeans pour plus de facilité et de rapidité.

Cette interface se compose d'un menu à gauche où l'on peut sélectionner une table ou un index pour chaque relation. On peut aussi choisir si la relation sera triée ou non, sachant que la jointure Sort-merge et Omiecinski n'affichera aucun résultat si les deux relations ne sont pas triées.

Enfin on choisit le type de jointure que l'on veut parmi les 5 disponibles. 3 zones de textes afficheront les information concernant les relations, la première pour les éléments de la relation 1, la deuxième pour les éléments de la relation 2 et enfin la dernière pour les résultats après jointure.

5 Interprétation des résultats

Key-Lookup : 1010 blocs lus

Ce coût très important s'explique par le fait que l'on ne puisse pas faire facilement une sélection sur la relation R2 avec l'index secondaire car nos index sont modélisés comme des tables. Pour une jointure de ce type, avec notre algorithme, le coût est de Br1 + Tr1*Br2 soit 10+100*10=1010.

Sort-merge : 20 blocs lus

Ce coût paraît très attractif, mais il n'est pas représentatif de la réalité car nous n'incluons pas le coût du tri des 2 relations dans l'algorithme. Le tri est géré en amont et nous n'avons donc ici que le coût de la jointure des relations triées qui est de Br1+Br2 soit 20.

Produit Cartésien : 30 blocs

Ce coût est similaire au coût obtenu en condition réelle pour cet algorithme. Notre mémoire centrale accueille 5 blocs de R1 et les compare séquentiellement avec 1 blocs de R2, elle a donc une capacité de 6 blocs. On retrouve ce coût grâce à la formule du cours qui est (Br1/(M-1))*Br2 + Br1 soit ici (10/5)*10+10=30.

Hachage : 220 blocs lus

Ce coût peut paraître élevé, mais il est en fait cohérent avec les conditions réelles. Ce qui le fait autant monter est la fonction de partitionnement qui n'est pas optimisée car la liste de blocs qui en résulte a autant d'indices que de valeurs possibles pour l'attribut valeur d'un enregistrement.

Notre fonction de partitionnement a un coût de Br1, on l'utilise deux fois donc Br1+Br2 soit 20 dans notre cas.

Ensuite la fonction de jointure parcourt les deux tableaux de blocs partitionnées, donc ces tableaux font 100 cases, et le coût est donc de 200. Notre coût total est donc bien de 220.

Omiecinski : 20 blocs lus

Tout comme l'algorithme Sort-merge, ici le coût paraît attractif car nous ne comptons pas le tri des relations. Le tri est géré en amont et nous n'avons donc ici que le coût de la jointure des relations triées qui est de Br1+Br2 soit 20.

Ces résultats sont ceux observés avec nos choix de modélisation, ils ne reflètent donc pas forcément les coûts que nous pouvons observer sur un SGBD comme Oracle. Dans notre cas, l'algorithme du produit cartésien semble le plus attractif si l'on exclu les jointures sort merge et Omiecinski du fait du coût du tri absent.

A cause de nos choix de modélisation, l'algorithme key-lookup est à éviter absolument car il parcourt beaucoup trop de blocs.

Si nous parvenons à améliorer la fonction de partitionnement, l'algorithme de hachage peut devenir lui aussi très intéressant