Rapport du projet Bases de Données *Groupe Moteur*

TT6 : Soizic Geslin-Minh Le Hoai-Samy Fouilleux-Maxime Chambreuil : KiKiTeam

Plan du rapport :

Objectifs du projet
Résultat apres 15 semaines

Objectifs du groupe Moteur

Résultat apres 15 semaines

Des problèmes avec ou sans solutions

Les problèmes
Les solutions que nous avons trouvées

Conclusion générale :

Rapport du projet Bases de Données *Groupe Moteur*

TT6 : Soizic Geslin-Minh Le Hoai-Samy Fouilleux-Maxime Chambreuil : KiKiTeam

Objectifs du projet

- Concevoir un Système de Gestion de Base de Données
- Découvrir les problèmes liés à la conception d'un SGBD
- Découvrir les problèmes liés à une application multi-processus

Résultat apres 15 semaines :

- Nous avons conçu une idée, mais certainement pas un SGBD

- Parmi les problèmes liés a la conception d'un SGBD, et dans le cadre des processus indépendants, nous pouvons souligner les difficultés liées à :
    - la communication entre les processus,
    - la mise en place de spécifications
    - la mise en place d'une interface utilisateur (type phpMyAdmin, pgAcces…) pour exprimer une requête
    - la mise en place d'un formalisme pour la requête afin que tous les processus puissent exploiter cette requête
        - nécessité d'écrire une grammaire
    - l'écriture d'une grammaire lorsqu'on a défini un formalisme précis
    - la reconstruction dans une forme exploitable pour un programme de la requête découpée par la grammaire
    - la réécriture des opérateurs dérivés, tout en respectant le formalisme prévu précédemment
    - l'implémentation des opérateurs de base
    - la création de relations

- Quelles ont été nos erreurs principales ?
    - Nous nous sommes focalisés trop longtemps sur la grammaire, oubliant le reste
    - Nous avons discuté des heures sur les spécifications des fonctions de communication sans raison valable
    - Nous avons donc été arrêtés par les délais alors que nous avions découvert de nouvelles dimensions au projet
    - Nous avions délégué une personne responsable de la grammaire par TT qui est rapidement devenue responsable du projet par TT, la conséquence étant que les autres membres du TT n'ont pas toujours fourni un travail qui s'avérait nécessaire.

Objectifs du groupe Moteur

- Concevoir le moteur de résolution de requêtes
- Définir une grammaire avec les groupes SQL et Interface
- Définir un protocole de communication
- Générer un plan d'exécution

Résultat après 15 semaines :

- Le principe du moteur a été assimilé
- La grammaire n'est toujours pas au point
    - Elle ne gère pas les alias alors qu'ils sont nécessaires pour les requêtes SQL et pour les jointures

- Le protocole de communication avec l'interface et les opérateurs est prêt.
- Le plan d'exécution contient encore des erreurs trop importantes
    - On ne sait pas réécrire les opérateurs dérivés
    - La création des relations pose encore des problèmes

Des problèmes avec ou sans solutions

Les problèmes

La gestion des alias par la grammaire

Les alias actuellement mis en place dans la grammaire ne satisfait pas toujours pas le groupe SQL : ils ne peuvent pas traiter les jointures.

Le plan d'exécution du moteur

Pour bien comprendre le fonctionnement du moteur, et surtout trouver les problèmes sans en oublier, nous avons écrit l'algorithme qu'il suit.

Sur l'algorithme (copié ci-dessous), les problèmes auxquels nous avons fait face apparaissent en gras.

Projet base de données

```
**********************
Algorithme du moteur(dernière modification du mardi 18 juin)
********************
********************

#defined groupe_interface 1
#defined groupe_moteur 2
#defined groupe_unaire 3
#defined groupe_binaire 4
#defined groupe_Sql 5
#defined groupe_environnement 6

main()
      debutCommunication()
      moteur()

*************************************************************************
fonction moteur()
      RecevoirMessage(emetteur,groupe_moteur,message)
      Si ((emetteur=groupe_environnement) et (message=stop)) alors
               //dans le cas ou le groupe environnement nous envoie le
message ``stop''\\
               //on doit terminer le processus
              //apres avoir indiqué aux opérateurs de terminer leur
processus
               envoyerMessage(groupe_unaire,groupe_moteur,stop)
               envoyerMessage(groupe_binair,groupe_moteur,stop)
               finCommunication()
      Sinon
               //Emetteur doit etre interface
              arbreRequete <- decouperRequeteAvecGrammaire(requete)
              //on crée l'arbre qui nous servira a découper le travail
pour les opérateurs
              //maintenant que l'arbre est créé, il faut le réorganiser
pour pouvoir faire pédaler les
               //opérateurs
```

```
                //il faut réécrire les opérations composites que les
opérateurs n'offrent pas.

                reecrireArbreRequete(arbreRequete)

                //une fois l'arbre réorganisé, on peut le parcourir et faire
pédaler les opérateurs
                //il faut commencer par le bas, parce que ce sont les
opérations
                // qui n'ont pas besoin d'attendre un résultat avant d'être
effectuées
                //on va donc faire un parcours en largeur

                //c'est pendant le parcours en largeur qu'on envoie les
opérations aux opérateurs*
                 //unaires et binaires
                //en fait c'est la fonction parcourirLArbre qui fait tout ça
                //c'est aussi cette fonction qui va créer le nom de la
requête résultat a renvoyer.

                nomRelationResultat<-parcourirLArbre(arbreRequete)
                //a ajouter : traitement des erreurs
                                                          //(renvoi la
chaîne de caractère "0")

envoyerMessage(groupe_moteur,groupe_interface,nomDeLaRelationResultat)
                supprimerLesRelationsIntermediaires();
                moteur()
      fsi
fin de l'algo



********************************************************************************

les fonctions que nous utilisons dans l'algo
*****************************************

réécrire les requêtes pour pouvoir les faire exécuter par les opérateurs de
base
_____________________________________________________________________________

procédure reecrireArbreRequete(E/S : type abstrait Requete:arbreRequete)

Cette procédure prend en entrée et sortie un arbre de Requete, doit le
parcourir et le rendre entier avec des réécritures des requêtes qui ne
pourraient pas être exécutées telles quelles par les opérateurs.


*********************************
découper la requête de l'interface
_____________________________________

fonction decouperRequeteAvecGrammaire(Chaine de Caracteres:requete)-> type
Abstrait Requete

Cette fonction construit l'arbre d'opérations a partir de l'analyse
grammaticale de la chaîne de caractères 'requete'

************************************************************************
la fonction de traitement d'une opération par les opérateurs élémentaires
```

```
fonction traiterOperation(type Abstrait Operation:operationElementaire)-
>nomRelationResultat

     nomResultat<-obtenirRelationResultat(operationElementaire)

     numeroGroupeOperateur<-obtenirTypeOperation(operationElementaire)

     nomOperateur<-obtenirOperateur(operationElementaire)
     nomRelation1<-obtenirPremiereRelation(operationElementaire)
     nomRelation2<-obtenirDeuxiemeRelation(operationElementaire)
     listeCriteres<-obtenirListe(operationElementaire)

     requeteElementaire<-
concatener(nomOperateur+"("+nomRelation1+","+nomRelation2+","+listeCriteres
+")")

     creerUneRelationResultat(requeteElementaire);
     envoyerMessage(groupe_moteur,numeroGroupeOperateur,requeteElementaire)


recevoirMessage(numeroGroupeOperateur,groupe_moteur,nomRelationResultat)
     //ici aussi il faut traiter les erreurs.

     retourner nomRelationResultat

finfonction

fonction creerRelationResultat(type Abstrait Requete: requeteElementaire)


finfonction

**************************
obtenir le type d'opération
_____________________________

fonction obtenirTypeOperation(type abstrait
Operation:operationElementaire)-> entier

renvoie le numéro du groupe qui devra effectuer l'opération
a voir si on simplifie en revoyant direct 3 et 4, le problème c'est que
c'est moins évident pour quelqu'un qui n'a jamais vu le projet.

**********************************
crever les noms des relations résultat
_____________________________________

fonction creerUnNomDeRelationResultat()

Cette fonction créée un nom pour la relation résultat.
L'idée est pour l'instant d'avoir une variable à incrémenter pour éviter
que 2 relations aient le même nom en même tps, pour éviter d'écraser un
résultat que l'interface n'aurait pas encore l

****************************************
type abstrait nécessaire au traitement
_____________________________________

type abstrait : KiKiTeam
```

```
Type : KiKiTeam
Utilise : entier, Operation (type abstrait)
Opération :
         creerKiKiTeam : NULL -> KiKiTeam
         supprimerKiKiTeam : KiKiTeam -> NULL
         creerKiKiTeam : entier, operation -> KiKiTeam

         fixerNumero : entier, KiKiTeam -> KiKiTeam
         fixerOperation : operation, KiKiTeam -> KiKiTeam

         obtenirNumero : KiKiTeam -> entier
         obtenirOperation : KiKiTeam -> operation

Sémantique : a chaque opération est associée un numéro qui permettra de
déterminer l'ordre d'execution.

Exemple : entier : 1 .. Operation : KiKiTeam1

type abstrait : ListeDeKiKiTeam

Type : ListeDeKiKiTeam
Utilise : KiKiTeam
Opération :
         creerListeDeKiKiTeam : NULL -> ListeDeKiKiTeam
         ajouterUnKiKiTeam : ListeDeKiKiTeam, KiKiTeam -> ListeDeKiKiTeam
         obtenirPremierKiKiTeam : ListeDeKiKiTeam -> KiKiTeam
         obtenirListeSuivante : ListeDeKiKiTeam -> ListeDeKiKiTeam

************************************************************

fonction d'execution des opérations de l'arbre réécrit
_____________________________________________________

fonction parcourirLArbre(Entrée Type Abstrait Requete : requete, Sortie
String Resultat )
//Requete l'arbre réecrit
//apres avoir été traité par la grammaire, l'arbre DOIT respecter pour
chaque noeud :
//le fils gauche (resp fils droit) a le meme nom que la relation 1 (resp
relation 2) de l'opération (type abstrait opération)
//                  0 <-KikiTeam1
//                 / \
//                /   \
// KikiTeam2 -> 0     0 <-KiKiTeam3
//et obtenirPremiereRelation(KiKiTeam1)=KiKiTeam2
//   obtenirDeuxiemeRelation(KiKiTeam1)=KiKiTeam3
//Resultat contient le nom de la relation finale

parcourRecursif(requete,1,tableauDeKiKiTeam)
trierListeDeKiKiTeam(liste)
relationResultat <- executer(liste)
retourner relationResultat

finfonction

******************************************************

procédure de parcours de l'arbre
________________________________
```

```
procedure parcoursRecursif(entree : type abstrait requete:requete
,entier:numeroDuNoeudCourant , entree/sortie : listeDeKiKiTeam:liste)

  si (requete#NULL)


ajouterUnKiKiTeam(liste,creerKiKiTeam(numeroDuNoeudCourant,obtenirOperation
Racine(requete))

parcoursRecursif(obtenirOperationFilsGauche(requete),numeroDuNoeudCourant*2
,liste)

parcoursRecursif(obtenirOperationFilsDroit(requete),numeroDuNoeudCourant*2+
1,liste)

  finsi

fproc

//on peut remarquer que le tableau pourra avoir des trous si certaines
Requete  n'on qu'un fils.
//ce n'est pas grave il suffira de ne pas tenir compte des valeurs nulles
au moment du tri.

fonction executer(listeDeKiKiTeam :liste)
      traiterOperation(obtenirPremierElement(liste))
      executer(obtenirListeSuivante(liste))
finfonction
```

Les solutions que nous avons trouvées :

Problème :

```
arbreRequete <- decouperRequeteAvecGrammaire(requete)
```

Solution :

Nous nous sommes rendus compte du problème suffisamment tôt pour créér un type abstrait de données Requete qui permet de traiter facilement une requete

Problème :

```
reecrireArbreRequete(arbreRequete)
```

Solution :

Aucune solution viable n'a été envisagée pour la réécriture de l'arbre. Nous avons été capable de comprendre toute la dimension du problème assez tard. Le groupe des opérateurs binaires nous avait promis des règles de réécritures pour que nous n'ayons plus qu'à les appliquer, mais les règles proposées le jour de la présentation ne sont toujours pas valables. Pour la mise en pratique de ces règles de réécritures, il nous paraît possible de remplacer les nœuds du type abstrait de données Requete sans difficulté majeure.

Problème :

```
nomRelationResultat<-parcourirLArbre(arbreRequete)
```

Solution :

Le problème pour notre parcours d'arbre est que nous devons absolument commencer par les feuilles et remonter de proche en proche, sans jamais sauter une étape, puis executer les opérations exactement dans cet ordre.

Nous avons résolu ce problème en créant un type abstrait de données qui attribue des numéros à chaque nœud suivant sa position dans l'arbre. L'execution se fera ensuite selon l'ordre des numéros, on sera sûrs qu'on demandera toujours d'exécuter des opérations sur des relations intermédiaires déjà obtenues.

Nous avons détaillé l'algorithme de ce parcours ainsi que l'exécution des opérations.

Problème :

```
creerUneRelationResultat(requeteElementaire);
```

Solution :

Avant de créer une relation, nous devons connaître le schéma de cette relation. Avant de connaître le schéma d'une relation, nous devons connaître le schéma des relations impliquées, et nous pourrons déduire facilement le schéma de la relation résultat.

Le schéma résultat suit les règles (simples) :

Opérateurs Unaires :

Sélection : même schéma que la relation impliquée

Projection : schéma constitué des attributs sur lesquels sont effectué la projection

Opérateurs Binaires : (formalisme : opérateurBinaire(R1,R2)

Union et différence :

*condition : le schéma de R1 est identique à celui de R2

schéma du résultat : le schéma commun à R1 et R2

Produit cartésien : schémas constitué de tous les attributs de R1 et tous les attributs de R2.

Deux solutions pour connaître le schéma des relations impliquées dans l'opération ont été envisagées.

Première solution :

Lire le fichier dans lequel est stocké la métabase, puisqu'il stocke les structures des autres relations présentes dans la base.

Le format choisi pour ce fichier est a priori public, donc nous pouvons par l'intermédiaire de l'OS lire les tuples de ce fichier, choisir ceux qui correspondent aux relations qui nous intéressent, et en déduire leurs attributs.

Deuxième solution :

Faire une requête sur la relation métabase pour obtenir les attributs d'une relation.

La requête à effectuer pour obtenir le schéma de maRelation est :

```
selection(projection(metabase,nomAttribut,indiceAttribut,typeAttribut),nomRelation=maRelation)
```

Il faut au préalable prévoir les relations résultats (résultat intermédiaire de la projection,puis sélection) pour cette requête, mais on connaît sa structure puisque la première opération est une projection.
Il ne resterait donc plus qu'a lire les tuples obtenus, et pour cela il faut prévoir une fonction « lireTuple »

Dans les deux cas, si nous créons une relation, il faut mettre à jour la métabase. Pour cela, il faut donc que nous prévoyons les fonctions d'insertion de tuple, certainement récupérable chez les opérateurs.

Conclusion générale :

Les erreurs que nous avons commises

- Nous ne nous sommes rendus compte que trop tard de tout ce que le moteur devait faire en dehors de lire la requête proposée par l'interface.
  - Mi-avril, nous commencions à peine la communication inter-processus
  - Fin-mai, nous nous sommes penchés sur la façon de demander aux opérateurs d'exécuter
- Nous avons mal partagé le travail dans le groupe, voire délaissé certaines parties du travail
- Nous nous sommes trop focalisés sur certains détails oubliant des problèmes beaucoup plus importants, notamment au niveau du parcours de l'arbre alors que les règles de réécritures sont loin d'être prêtes et que le problème de la création de relation n'est pas résolu.