Groupe de Projet
Thomas Moulard
Taha Karim
Julien Assemat

Table des matières

Introduction

Présentation Générale du Projet [2 min]

Avancement Général du Projet

Avancement Général du Site Web

Avancement Prévu entre la Soutenance 2 et 3

Problèmes Rencontrés [1 min]

Groupe

Réalisation

Présentation des Tâches Individuelles [12 min]

Tâches effectuées par Taha

Tâches effectuées par Julien

Tâches effectuées par Thomas

Conclusion

1 Introduction

Une deuxième étape dans la création de notre projet *epiDatabase* vient maintenant de s'achever. A l'occasion de cette nouvelle partie de la conception du projet, nous devions dans un premier temps en terminer les fondements, déjà largement avancés lors de la première soutenance, puis dans un deuxième temps, nous avions à rendre le programme fonctionnel, notamment par la création du système de Parseur et Lexeur mais également par le début de la gestion des requêtes, lesquelles permettent de faire la liaison entre l'utilisateur et le programme après être passées par la partie d'analyse du programme (Parseur, Lexeur).

Pour cette deuxième soutenance, nous devions donc finir la conception du parsuer et lexeur mais aussi réaliser en partie la gestion des requêtes.

2 Présentation du Projet

2.1 Composants du Projet

La réalisation d'*epiDatabase* nécessitera la mise en relation de plusieurs composants tels que la gestion des fichiers, la grammaire des requêtes, la reconnaissance de cette grammaire, une application réseau faisant le lien avec tous les programmes dépendant du SGBD.

2.1.1 Gestionnaire de Fichiers

Le premier constituant, et surtout le plus important, est le gestionnaire de fichiers. De part sa structure, il nous permettra de traiter et de réorganiser de la meilleure manière possible l'ensemble des informations stockées. C'est à ce niveau-là qu'interviendront le résultat des requêtes reçues par l'application, à savoir suppression, ajout, modification ou encore sélection d'enregistrements. La structure sera plus amplement développée dans les sections individuelles.

2.1.2 Requêtes

Les requêtes sont en quelque sorte les commandes permettant la mise en relation entre l'application qui demande les informations et le SGBD qui va les traiter et les lui envoyer. Plus grossièrement, les requêtes permettent de savoir ce que veut l'application.

Comme cela a déjà été précisé dans le cahier des charges, les requêtes suivront la grammaire SQL, avec plus ou moins de simplifications sur certaines fonctionnalités. Ce langage n'a pas été choisi par hasard, en effet, il a largement démontré son efficacité parmi les plus grands SGBD de par sa logique et sa simplicité, qui font de cette grammaire un langage efficace.

2.1.3 Parseur

Le parseur n'est rien d'autre qu'un analyseur syntaxique. Pour notre projet, il servira à reconnaître la grammaire SQL, utilisée principalement pour les requêtes : en effet, il faut que le SGBD soit en mesure de comprendre les *ordres* que lui envoie le programme qui veut les informations.

Pour la réalisation de ce composant, nous utiliserons le programme Bison, qui, à partir de la définition de la grammaire qu'on lui donne, génère automatiquement les fichiers .c permettant de parser n'importe quelle requête syntaxiquement correcte.

2.1.4 Lexeur

Le lexeur est le constituant indispensable au parseur pour une application telle qu'Epi-Database. Couplé au parseur, il permet de reconnaître les motifs de la grammaire dans une requête par exemple et à partir de là, il exécute du code selon le motif trouvé. De même que précédemment, nous utiliserons le meilleur allié de Bison, à savoir Flex, dont nous aborderons la programmation et les principes dans les sections individuelles.

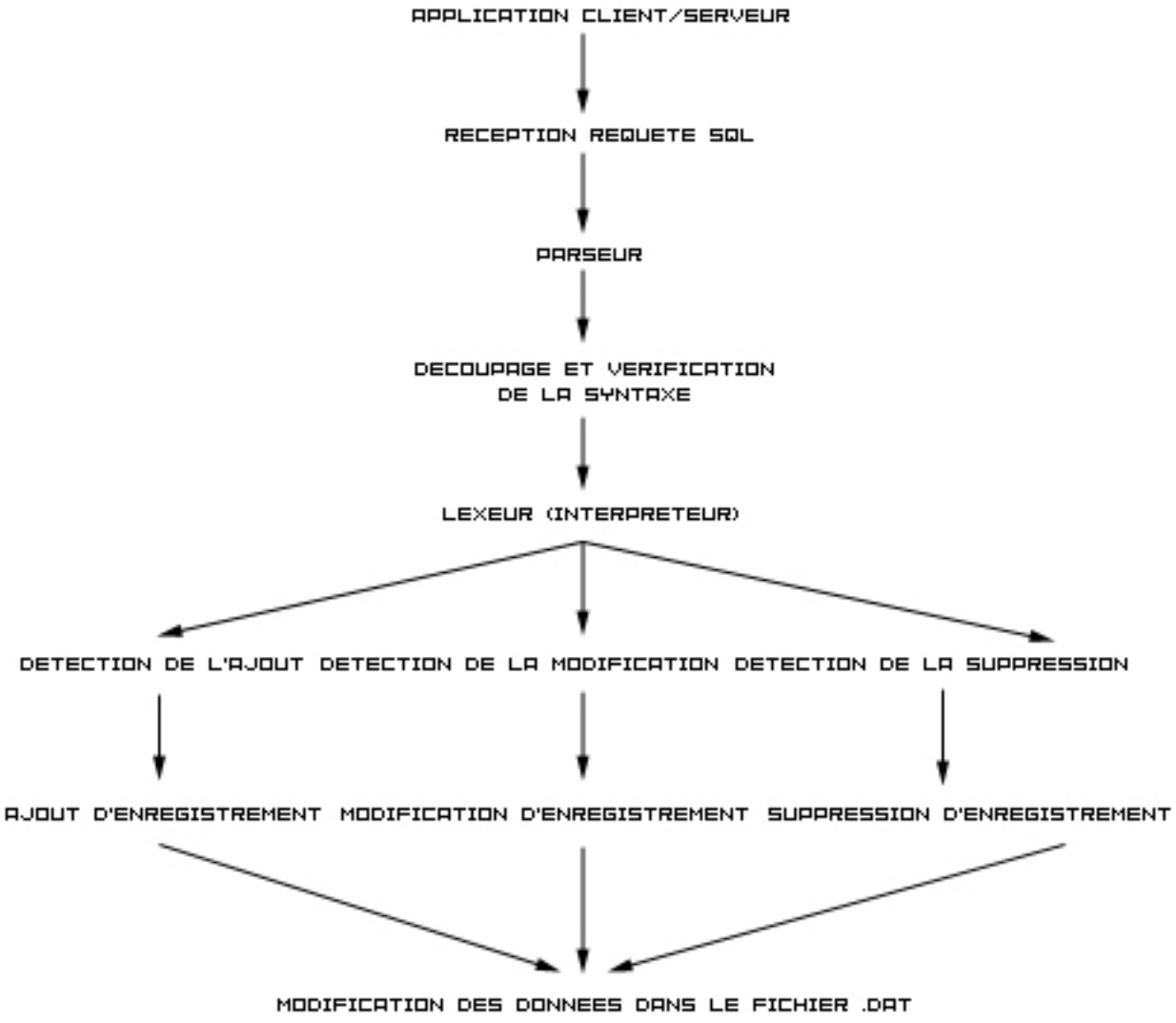
2.1.5 Application Client/Serveur

L'application Client/Serveur est chargée de faire le lien entre le SGBD et l'application qui demande les informations, le tout à l'aide de requêtes conformes à la grammaire SQL.

Dans un premier temps, le programme demandant des informations envoie une requête à l'application client/serveur qui elle, va rediriger vers le SGBD qui va effectuer les traitements.

Il est important de préciser qu'une base de données doit pouvoir traiter de nombreuses requêtes en quasi-simultanéité. Pour cela, une fois la requête reçue, l'accès a la base sera verrouillé le temps du traitement, puis déverrouillé une fois le traitement terminé et la requête suivante pourra être exécutée, tout cela afin d'éviter d'avoir des conflits au niveau des accès simultanés aux mêmes objets.

2.2 Schéma de Fonctionnement



3 Avancement Général

3.1 Avancement Général du Projet

Comme évoqué précédemment, l'avancement du projet en lui-même se fait sur deux axes majeurs.

Parseur & Lexeur

Initialement prévue sur les trois premières soutenances, la conception du Parseur et du Lexeur s'est avérée être primordiale dès la première soutenance. C'est pourquoi, dès celle-ci, le Parseur et le Lexeur régissant les reqûetes SQL ont été commencés très tôt et sont, à la fin de cette deuxième étape de développement, proche de leur fonctionnement intégral.

Nous pouvons donc considérer à l'heure actuelle ce composant principal d'*epiDatabase*, comme achevé. Toutes les requêtes prévues ou presque sont donc détectées, seules les liaisons entre les fonctions externes de traitement des données seront rajoutées au fur et à mesure.

Gestion des Requêtes - Partie 1

Le deuxième point principal pour l'aboutissement de cette deuxième soutenance concerne les premières gestions des requêtes, c'est-à-dire la relation entre les données et les requêtes SQL elles-mêmes. A l'heure actuelle, la première partie de la réalisation de cette tâche est fonctionnelle pour la gestion des requêtes des types suivants :

- `CREATE DATABASE ma_base`
- `CREATE TABLE ma_table(champ1 TYPE, champ2 TYPE)`
- `DELETE FROM ma_table`
- `INSERT INTO ma_table VALUES('valeur1','valeur2','valeur3')`
- `RENAME ma_table TO ma_table2`
- `SHOW COLUMNS ma_table`
- `DROP TABLE ma_table`
- `DROP DATABASE ma_base`

Les requêtes nommées ci-dessus sont reconnues et fonctionnent.

3.2 Avancement Général du Site

Quant au site, il est quasiment terminé. En effet, la deuxième et dernière grosse partie du site était toute la section d'administration du site, permettant à un utilisateur ayant les codes d'accès de poster une actualité qui s'affichera en première page mais également de

gérer les fichiers de la section "Téléchargements" du site (Rapports, Cahier des Charges...).

Bref, vous l'aurez compris, pour cette deuxième soutenance notre objectif était de rendre le site Web du projet opérationnel à 100%.

Le site est accessible depuis l'adresse Internet suivante :

http ://www.homerlan.com/epidatabase

3.3 Avancement Prévu pour la troisième Soutenance

Pour la troisième soutenance, le projet *epiDatabase* devrait apparaît comme quasi-fonctionnelà 100% à quelques détails près. Il s'agira en effet pour cette troisième étape de progresser considérablement dans la gestion des requêtes afin de terminer cette partie-là et pouvoir commencer à se consacrer à la mise en place de la liaison Client/Serveur.

Pour ce qui est de l'avancement du site Web, rien n'étant prévu dans le cahier des charges, nous comptons donc l'améliorer en proposant par exemple dans la section d'Administration un formulaire permettant non pas de déclarer les fichiers mais de les envoyer directement sur le serveur puis de les déclarer.

4 Problèmes Rencontrés

4.1 Groupe

Le seul problème réel que nous ayons rencontré était la redistribution des tâches, qui, pour la première soutenance, s'est avérée équilibrée, mais qui, pour la seconde, a été plus difficile à mettre en place et à maintenir.

4.2 Projet

A ce niveau-là, le principal obstacle rencontré fût celui de savoir comment représenter nos données, nos types... En effet, la plupart du temps, de réelles questions se posent.

Cependant, avec les précédentes recherches que nous avions entreprises, ce problème a pu être contourné et le développement du projet a suivi son cours normal.

5 Tâches Individuelles

5.1 Tâches réalisées par Taha

1. **Première soutenance + Rappels :**

   – **Introduction :**
   Dans la première soutenance, j'ai fait de la recherche sur tout ce qui est systèmes de gestion de bases de données j'avais présenté MySql, historique, et surtout le système de fichiers sur lequel on s'est basé pour notre projet. D'autre part, j'avais commencé à coder quelques fonctions utiles, pour ma deuxième soutenance . Pour cette soutenance, mon but, c'est de m'occuper de tous ce qui est manipulation de fichiers, à savoir, les fichiers.epi, les fichiers.data, et les fichiers.key qu'on verra a la fin de cette partie.

   – **Rappel de la mise en place du systeme de fichiers :**
   Comme nous l'avons présenté dans la première soutenance, notre système de fichier est le suivant :

   ```
   fichier.epi : stocker les structures des tables .

   fichier.key : stocker les clefs de chaque table .

   fichier.data: stocker les données de chaque table.
   ```

   De cette façon on a un système de fichiers bien structuré, et très facile à exploiter. Pour plus de détails voir section Julien Assémat.

2. **Nouvelles fonctions, commandes :**
   Dans cette partie je vais présenter toutes les nouvelles fonctions que j'ai ajoutées au moteur epiEngine , ainsi que les liaisons entre mes fonctions et le Parseur que l'on a mis en place (voir section Thomas Moulard)

- **Fonction INSERT :**
  Cette fonction permet d'ajouter une donnée, dans une table, et ainsi la création d'un fichier .data ou seront stockées les données.
- Exemple de code :

```
#la fonction insert# :
```

```
void insert(struct s_sql_query *query, struct s_sql_result
*result) {

   FILE *f;
   char filename[S_TABLE+strlen(EXTENSION_DATA)-1];
   p_table tbl;

   printf("\t Insert into table %s\n",query->tblname);
  *filename=0;

   strcpy(filename, PATH_DATA);
   strcat(filename, query->dbname);
   strcat(filename, "\\");
   strcat(filename, query->tblname);
   strcat(filename, EXTENSION_DATA);

   f = fopen(filename, "r");
   if(!f)
   {
    f = fopen(filename, "w");
    tbl = query->table;
     while(tbl !=NULL)
     {
       fwrite(&tbl->column.name, sizeof(struct s_column),1, f);
       tbl=tbl->next;
     }

     fclose(f);
     result->error_code=MSG_OK;
   }
   else
   {
     fclose(f);
     if(query->if_not_exists)
        result->error_code=MSG_OK;
     else
        result->error_code=MSG_TABLE_EXISTS;
   }
}

#liaison avec le parser#
```

```
  #~sql.y#

/* === INSERT === */ insert: TOKINSERT TOKINTO table_ident
TOKVALUES PARENTHESEOPEN .. .. insert_values PARENTHESECLOSE {
   insert(&gl_query, &gl_result);
} ;

insert_values: /* empty */ | insert_values IDENT VIRGULE {
    addFieldTable(&ParserTemp, $2, 0);
    gl_query.table = ParserTemp;
    gl_query.nbcols++;
} | insert_values IDENT {
    addFieldTable(&ParserTemp, $2, 1);
    gl_query.table = ParserTemp;
    gl_query.nbcols++;
} ;

  #~sql_tab.c#

case 53: #line 228 "sql.y" {
   insert(&gl_query, &gl_result);
;
    break;}
case 55: #line 236 "sql.y" {
    addFieldTable(&ParserTemp, yyvsp[-2].string, 0);
    gl_query.table = ParserTemp;
    gl_query.nbcols++;
;
    break;}
case 56: #line 242 "sql.y" {
    addFieldTable(&ParserTemp, yyvsp[-1].string, 1);
    gl_query.table = ParserTemp;
    gl_query.nbcols++;
;
    break;}
```

De cette maniere, on a defini à la fois la commande INSERT, et la fonction qui va avec, et qui créé en même temps, notre fichier.data, ou l'on stockera nos données. La commande est

`INSERT INTO ma_table VALUES(donée1, donnée2).`

- **Fonction DELETE :** La fonction DELETE permet de supprimer une donnée dans

une table passée en paramètre, il s'agit donc de suppression de données, alors on se place dans notre fichier.data, et on cherche la donnée, et on la supprime, une fois de plus tout en tenant compte de la relation entre notre fonction et le parseur, et tout qui va avec.
La commande est

```
DELETE FROM ma_table WHERE predicat.
```

Chaque ligne a son prédicat, si il est à vrai, la ligne sera supprimée .

- **Fonction UPDATE :**
  Cette fonction permet la mise à jour de la table , par exemple

```
UPDATE ma_table SET nom = 'valeur'
```

  Elle permet de mettre le champ 'nom' à la valeur 'valeur'.
- Exemple de code :

```
#~sql_tabl.c#

case 69: #line 300 "sql.y" { update(&gl_query, &gl_result); ;
    break;}
case 70 : #line 307 "sql.y" {
 while(gl_query.table!= NULL)
 strcpy(gl_query.tblname, yyvsp[-2].string);
 gl_query.table = gl_query.table->next
;
    break;}
```

3. **Conclusion :**
   Pour cette soutenance, on a pratiquement fini le parseur, le lexeur, le système de fichiers et les fonctions principales INSERT, DELETE, UPDATE qui vont avec, donc tout va bien pour le moment, on est toujours en avance, on espère garder ce rythme jusqu'à la fin de l'année, même si on a perdu un 4ème membre, à 3 on peut dire qu'on s'en sort plutôt bien.

5.2 Tâches réaplisées par Thomas

Comme à la première soutenance, mes efforts se sont portés sur les deux axes essentiels de notre projet : l'interpréteur SQL et epiEngine, notre moteur de base de donnée.

5.2.1 Interpréteur SQL

L'analyse sémantique n'a quasiment pas évolué depuis la première soutenance, par contre notre analyseur syntaxique a fait l'objet de la majeure partie de nos efforts.

Gestion des listes de valeurs

L'un des gros problèmes que nous avons rencontré lors de la première soutenance est la gestion des listes de valeurs comme la liste des colonnes lors de la création d'une table. Ainsi tous les types de syntaxe étaient reconnus qu'il y ait ou pas de virgule entre les colonnes. Ce problème a été résolu pour toutes les listes lors de cette période via la solution suivante :

```
column_definition_commalist:
  column_definition
| column_definition_commalist VIRGULE column_definition ;
```

On notera au passage que toutes les règles récursives que nous introduisons sont récursives à gauche afin de faciliter le travail de bison et de générer un interpréteur plus fiable.

Analyse de requêtes complexes

Avec la progression de notre travail, il devient nécessaire d'analyser des requêtes de plus en plus complexes : il faut désormais stocker les données contenues dans les requêtes, interpréter les conditions, etc.

La reconnaissance et le stockage des valeurs contenues dans la requête Notre premier problème est un problème de TYPE. En effet les colonnes peuvent avoir différents types : TINYINT, MEDIUMINT, CHAR, FLOAT, etc. et notre interprêteur lexical ne les différencie qu'en trois types de base : entier, flottant ou chaîne de caractères.
Ainsi tous les problèmes de détermination de type, de transtypage doivent être gérés manuellement. De plus si les types ne sont pas du tout compatibles, il faut s'en apercevoir et réagir en conséquence ! Tous ces problèmes de gestion de types "à la main" n'ont pas été anticipés et ont constitués un problème important de cette seconde période.

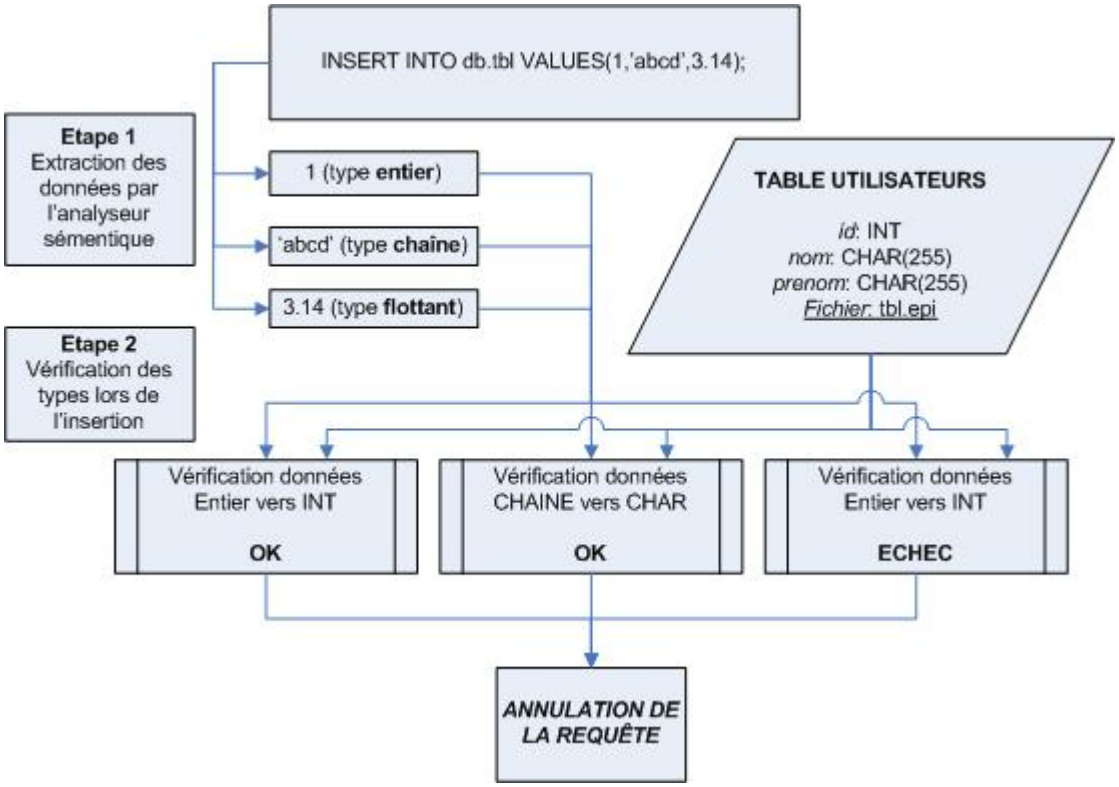


Fig. 1 – Stockages des informations de la requête et vérification des types

Le problème a donc été de pouvoir stocker des données quelque soit son type. Nous avons donc dû faire une large utilisation des unions via la définition d'un type "universel" :

```
typedef enum { VIDE = 0, FLOTTANT, ENTIER, CHAINE, COLONNE_REF }
e_basic_type; union u_data {
   double            floatNumber;
   int               number;
   char              * string;
   char              * col_ref; /* ONLY USED IN SCALAR, NOT ATOM ! */
};

struct s_univdata {
   union u_data data;
   e_basic_type type;
};
```

Ce type permet à l'interprêteur de stocker temporairement les données de la requête afin de les passer au moteur de base de donnée. Le type énuméré permet de savoir quel est le type de notre donnée.
Notre interpréteur a donc été modifié en conséquence pour pouvoir gérer ce type de donnée :

```
%union
{ double floatNumber; int number; char *string; char subtok;

struct s_univdata data; } [...]
%type <data>  scalar_exp atom literal column_ref
[...] literal:
  STRING  { $$.data.string=strdup($1); $$.type=CHAINE;   }
| FLOAT   { $$.data.floatNumber=$1;    $$.type=FLOTTANT; } |
INTEGER { $$.data.number=$1;         $$.type=ENTIER;   } ;
```

Le symbôle $$ permet de déterminer la donnée que l'état renvoit. La structure s_univdata nous permet de renvoyer d'état en été n'importe quel type ! epiEngine la récupère ensuite soit sous la forme d'un paramètre directement, soit via un stockage dans une liste chaînée.

Les conditions "WHERE" Les clauses de conditions "WHERE" sont particulièrement difficiles à gérer : en effet il existe de nombreuses variantes et elles peuvent être très longues et complexes ! Il a donc tout d'abord fallu simplifier le problème en le décomposant. Il y a 5 types de conditions :

1. sous-conditions reliées par un ET
2. sous-conditions reliées par un OU
3. sous-condition précédée d'un NON
4. sous-condition entre parenthèses
5. prédicat

Un prédicat est une règle spécifique que l'on doit respecter, il y en existe de 6 sortes :

1. la comparaison de deux scalaires ($id < 5$)
2. le bornage d'une valeur (BETWEEN)
3. la recherche de valeurs avec joker (LIKE)
4. recherche de valeurs nulles/non-nulles
5. la recherche d'un ensemble de valeurs (IN)
6. les sous-requêtes

Actuellement, seul les tests de compaisons sont supportés.
Les scalaires correspondent à une expression complet pouvant contenir des noms de colonnes, des valeurs litérales et toutes les opérations et fonctions supportées par epiDatabase.
Exemple : id+(MD5(auteur)*3+2)/4 (ici id et auteur sont les colonnes d'une table)
Il est donc nécessaire d'analyser et de calculer les valeurs de ces expressions tout en tenant compte des problèmes de type.
Ce problème fonctionne actuellement dans des cas précis mais nous sommes loin de gérer la totalité des cas, notamment l'interpréteur n'a pour l'instant aucun moyen de connaître la valeur d'une colonne !
Cela fera parti des défis à relever pour les soutenances à venir.

5.2.2 epiEngine

Modification de "CREATE TABLE"

Même si la base de la fonction de création de table "CREATE TABLE" a déjà été présentée en première soutenance, celle-ci s'est vu considérablement améliorée.
En effet elle gère maintenant l'enregistrement des types de donnée, des tailles, des flags de chaque colonne. Le parseur est déjà près pour gérer d'autres fonctionnalités telles que les clés pour optimiser les recherches et les contraintes sur les colonnes pour préserver la cohérence des données.

```
/* === CREATE TABLE === */ create_table: TOKCREATE TOKTABLE
if_not_exists table_ident PARENTHESEOPEN
column_definition_commalist PARENTHESECLOSE {
```

```
createtable(&gl_query,&gl_result); } ;
column_definition_commalist:
  column_definition
| column_definition_commalist VIRGULE column_definition ;
column_definition:
  IDENT { AddColumnDefinition($1, &gl_query); }
column_type opt_column_options_list | table_constraint_definition
; column_type: TOKTINYINT { SetColumnDefinitionType(TYPE_TINYINT,
&gl_query);
  SetColumnDefinitionSize(1, &gl_query)
} [...] ; column_type_options: /* empty */ | TOKUNSIGNED
TOKZEROFILL { SetColumnDefinitionFlag(FLAG_UNSIGNED |
FLAG_ZEROFILL,1,&gl_query); } | TOKUNSIGNED {
SetColumnDefinitionFlag(FLAG_UNSIGNED,1,&gl_query); } |
TOKZEROFILL { SetColumnDefinitionFlag(FLAG_ZEROFILL,1,&gl_query);
} ; opt_column_options_list: /* empty */ | opt_column_options_list
opt_column_options ; opt_column_options:
  TOKNULL
{ SetColumnDefinitionFlag(FLAG_NULL,1,&gl_query); } | NOT TOKNULL
{ SetColumnDefinitionFlag(FLAG_NULL,0,&gl_query); } [...] |
TOKDEFAULT atom { SetColumnDefaultValue($2, &gl_query);
FREE_DATA($2); } [...] ; table_constraint_definition: [...] ;
```

Les passages qui se répètent pour tous les types ainsi que les comportement interprétés mais non-gérés par epiEngine ont été supprimés.
En comparaison de la première soutenance, l'interpréteur a beaucoup évolué et est devenu bien plus complexe.

Parmi les nouveautés, on peut remarquer la gestion des flags via un unique champ CHAR et un système de masques. Le char flag est décomposé selon le schéma suivant :

-	UNSIGNED	ZEROFILL	BINARY	ASCII	UNICODE	NULL

On peut donc via des masques mettre à 0 ou 1 n'importe quel(s) bit(s) ou bien les récupérer comme l'illustre la ligne suivante de l'interpréteur :

```
SetColumnDefinitionFlag(FLAG_UNSIGNED |
FLAG_ZEROFILL,1,&gl_query);
```

On positionne ici les bits UNSIGNED et ZEROFILL à 1.

Modification de "SHOW COLUMNS"

La requête "SHOW TABLE" a été renommée "SHOW COLUMNS" afin de rester compatible avec les SGBD existants.
Elle affiche désormais toutes les informations concernant une table comme le montre le screenshot ci-dessous :

```
CREATE DATABASE db;
        Create DB (error if exists) db
CREATE TABLE tbl (id INT NOT NULL DEFAULT 1, username CHAR, date BIGINT NULL, PR
IMARY KEY (id), UNIQUE (date));
        Create table (error if exists) tbl
SHOW COLUMNS FROM tbl;
        Show columns tbl
=== STRUCTURE - tbl - ===
id 4(2) [0]Default value: 1
username 17(255) [0x1]
date 7(4) [0x1]
=== FIN STRUCTURE ===
DROP TABLE tbl;
        Drop table (error if not exists): tbl
DROP DATABASE db;
        Drop database (error if not exists): db
-
```

Fig. 2 – Exemple de session epiDatabase

Comme vous pouvez le voir ci-dessus, nous gérons désormais le nom, le type, les flags de chaque colonne ainsi que la valeur par défaut.
D'autres modifications interviendront par la suite comme la gestion des index.

Modification de "DROP DATABASE"

Cette fonction supprime désormais toutes les tables qu'elle contient afin de pouvoir supprimer des bases non-vides.

Renommage des tables

Le renommage de table s'effectue via la requête suivante :

```
RENAME TABLE tbl TO tbl; RENAME TABLE db.tbl TO db.tbl;
```

Elle permet non seulement de renommer des tables mais également de changer la base

de donnée à laquelle elle appartient.

Suppression des données

La suppression des données permet simplement de vider le fichier data pour l'instant. En effet même si les clauses de condition sont déjà analysées et gérées, nous n'avons pas eu le temps de mettre les deux parties en relation.

Voici un exemple de requête correctement executées actuellement :

```
DELETE * FROM tbl;
```

5.2.3 Gestion des signaux

La gestion des signaux permet de quitter proprement la base de donnée même en cas de "segmentation fault" ou bien d'interruption du programme via le clavier (CTRL+C). Actuellement il ne s'agit que d'afficher du texte et de libérer des ressources, mais cela peut avoir des grandes conséquences à l'avenir lorsqu'il faudra interrompre proprement les connexions en cours via les sockets ou bien pour gérer les transactions ou encore éviter les corruption de la base de donnée en cas de bug.

5.2.4 Conclusion

Points positifs

Le premier point positif est qu'une fois que le réseau sera implémenté, notre projet sera utilisable ! La suppression et l'insertion de données, même grossière permet déjà de l'utiliser pour des applications extrêmement simples. Nous avons surmonté la plupart des problèmes rencontrés durant cette période et finalement résolu tous les points qui restaient en suspens après la première soutenance. Je suis donc très confiant quant à l'évolution du projet.

Points négatifs

Toutefois, il reste encore quelques points qui nous posent du soucis : le projet semble de plus en plus difficile et demande de plus en plus de connaissances. Même si l'apprentissage de la bibliothèque standard et de POSIX a permis de nous donner pas mal d'éléments de réflexion, il reste encore beaucoup de difficultés à surmonter. De plus, tout ce qui concerne la gestion des threads et du réseau que nous allons devoir implémenter le sera sans aucun cours ni connaissance préalables (tout du moins en C).
De plus les points non-résolus de cette seconde période s'annoncent très difficiles à résoudre : la gestion des types "manuellement" est très lourde à programmer et l'interprétation com-

plète des clauses de condition s'annonce longue même si elle est déjà bien commencée.

Bilan de la seconde période

Le bilan de cette seconde période est très positif une fois de plus : le planning est respecté, même si nous devons nous en écarter pour implémenter certaines fonctionnalités non-prévues (gestion des types). Nous avons réussi à programmer toutes les actions de base que l'on peut attendre d'une base de donnée : insertion/suppression des bases de donnée, tables et données. Le projet devient donc petit à petit utilisable ce qui est positif, cependant on se rend également compte qu'il se compléxifie beaucoup. J'espère donc que nous pourrons tenir le rythme et arriver à produire à la soutenance finale une application complète fonctionnant à l'aide de notre base de donnée !

5.3 Tâches réaslisées par Julien

Pour ma part, en ce qui concerne cette deuxième étape du développement d'*epiDatabase*, mes tâches se situaient sur plusieurs niveaux. En premier lieu, il était question de progresser au maximum dans la conception du site Web relatif au projet pour arriver à un niveau de fonctionnement optimal. Dans un deuxième temps, il était question pour moi d'adapter le Parseur déjà largement mis en place par Thomas pour le relier aux fonctions externes et pour l'amener au traitement des données proprement dites. Et enfin pour clôturer cette étape de conception, le Rapport de Soutenance.

5.3.1 Interventions sur le Parseur / Lexeur

Quasiment développement dans sa totalité par Thomas, le Parseur nécessite d'être rallié à un interprêteur ou Lexeur pour pouvoir appliquer un certain traitement directement à nos données en fonction de la requête reçue par le Parseur.

Tout d'abord, il a s'agit pour moi d'amener une petite modification au Parseur en lui-même. En effet, il ne reconnaissait que les reqûetes SELECT de ce type :

```
SELECT champ1 FROM ma_table
```

C'est-à-dire qu'il ne savait reconnaître que la sélection d'un seul champ ou de tous mais pas de deux, trois ou quatre champs différents. Les modifications apportées ont donc été les suivantes :

```
select_list_ident: IDENT
    {
        AddColumnToList(&gl_query,$1,"","");
    }
| select_list_ident VIRGULE IDENT
    {
        AddColumnToList(&gl_query,$3,"","");
    }
;
```

Par cette modification, le Parseur est donc capable à l'heure actuelle de détecter les requêtes SELECT de ce type :

```
SELECT champ1, champ2, champ3 FROM ma_table
```

5.3.2 Requête SELECT

Actuellement la requête SELECT implémentée permet d'afficher tous les champs sélectionnés dans la requête. Dans le code montré ci-dessus apparaît la fonction *AddColumnToList*. Elle permet tout simplement d'ajouter les champs sélectionnés à une liste chaînée présente dans la structure *s_ sql_ query*.

Une fois tous les champs de la requête récupérés, la première chose à faire et de vérifier que tous les champs demandés dans la requêtes existent réellement dans la table. Pour cela, j'utilise la fonction suivante :

```
int IsInTable(p_lfield List1, p_lfield List2) {
    p_lfield Temp;

    Temp = List1;
    while(List2 != NULL)
    {
            List1 = Temp;
            while(List1 != NULL)
            {
                if(strcmp(List1->field_name, List2->field_name) == 0)
                    break;
                else
                    List1 = List1->next;
            }
            if(List1 == NULL)
                return 0;
            List2 = List2->next;
    }
    return 1;
}
```

Le type *p_ lfield* est un pointeur sur un élément contenant entre autres le nom du champ. Ainsi on utilise une double-boucle qui parcourt la liste des champs sélectionnés (*List2*) et qui vérifie que chacun de ces champs est bel et bien un champ de la table. Tous les champs de la table en question sont localisés dans la variable *List1*. La fonction retourne 1 si tous les champs sont corrects, 0 sinon.

Toujours dans cette même partie, il m'a fallu créer une partie de la fonction permettant de récupérer tous les champs d'une table. Cette même fonction a été complétée par la suite par Thomas pour la gestion des tailles et des types.

```
void LoadFieldsTable(p_lfield * FieldsList, char *DatabaseName,
    char * TableName, struct s_sql_result *result) {
    p_lfield newField, Temp;
    FILE *f;

    char filename[S_TABLE+strlen(EXTENSION_TABLE)-1];
    struct s_column col;

    *filename = 0;
    strcpy(filename, PATH_DATA);
    strcat(filename, DatabaseName);
    strcat(filename, "\\");
    strcat(filename, TableName);
    strcat(filename, EXTENSION_TABLE);
    f = fopen(filename, "r");
    if(f)
    {
        while(!feof(f))
        {
           *col.name=0;
           fread(&col,1,sizeof(struct s_column),f);

           if(*col.name != 0)
           {
                newField = calloc(1, sizeof(struct s_lfield));
                strcpy(newField->field_name, col.name);
                newField->next = NULL;

                if(*FieldsList == NULL)
                {
                    *FieldsList = newField;
                }
                else
                {
                    Temp = *FieldsList;
                    while(Temp->next != NULL)
                        Temp = Temp->next;
                    Temp->next = newField;
                }
           }
        }
```

```
        result->error_code = MSG_OK;
        fclose(f);
    }
    else
        result->error_code = MSG_TABLE_NOT_EXIST;
}
```

Elle ouvre donc le fichier dédié à la table recherchée et lit au fur et à mesure les données qui nous intéressent, et plus exactement le nom des champs.

Enfin, dernière étape pour la requête de type SELECT, la récupération des données et accessoirement leur affichage.

Il faut savoir que les données relatives à une table sont stockées dans le fichier *ma_ table.data* et que ces données sont stockées dans le fichier de manière variable, c'est-à-dire avec une taille différente pour chaque champ. Ainsi pour la lecture de ces mêmes données, il nous faut d'abord récupérer les données de la table considérée et pour chaque colonne (champ) en lire un nombre de bits bien précis, variable en fonction de la taille du champ.

5.3.3 Requête INSERT

La requête INSERT, correspondant à la requête d'insertion, permet simplement d'ajouter un enregistrement à la table, sachant qu'un enregistrement est représenté par un structure contenant les différentes données.

Pour cela, il faut dans un premier temps intervenir de la même manière que précédemment au niveau du Parseur pour que chacune des données entrées dans la reqûete soit récupérée. On utilise donc le code suivant au sein du Parseur :

```
insert_values:
  atom
    {
        AddData($1, &gl_query);
        gl_query.nbcols++;
    }
| TOKNULL
    {
        AddDataVide(&gl_query);
        gl_query.nbcols++;
    }
;
```

De cette manière, on récupère à la fois les données et le type de ces données de manière à les comparer avec celui des champs de la structure de la table pour s'assurer que l'on n'ajoute pas un entier dans un champ dédié à accueillir une chaîne ou inversement. A chaque fois que l'une de ces données est détectée, elle est ajoutée à une liste chaînée avec son type et son contenu.

Puis la procédure se termine par l'insertion au coeur même du fichier correspondant à la table considérée des données récupérées.

5.3.4 Site Web - epiDatabase

Enfin, dernière grosse tâche en ce qui me concerne pour cette deuxième soutenance, la fin de la réalisation du Site Web.

Après avoir réalisé la mqeutte graphique, l'intégration HTML de cette dernière ainsi que le début de l'intégration des premiers scripts PHP, il m'a fallu pour aboutir à une application Web fonctionnelle autant que possible à la mise en place d'une section d'administration pour gérer, à partir de cette dernière, une majeure partie du site.

Pour cela, j'ai décidé d'utiliser une base de données mySQL reliée au site par l'intermédiaire de PHP. En voici une illustration, à savoir celle de l'affichage sur la première page de la dernière actualité postée.

```
<?
$SQL = "SELECT * FROM epiNews ORDER BY Date DESC Limit 0,1";
$result = mysql_db_query("$nom_bdd" , $SQL , $connexion);
while($resultat = mysql_fetch_array($result))
{
   $Date = gmdate("d.m.Y", $resultat['Date']);
   $Titre = $resultat['Titre'];
   $Corps = str_replace("\n", "<br>", $resultat['Corps']);
}
?>
```

Par ce traitement, on récupère la date d'écriture, le titre et le corps de la brève postée. Il ne reste plus qu'à l'afficher sur la première page du site par l'intermédiaire d'un code HTML.

De la même manière, on gère l'affichage des fichiers.

L'espace d'administration permettant de gérer ces différents points se présente sous forme sécurisée par un formulaire comportant un nom d'utilisateur et un mot de passe, tous deux stockés dans la base de données. Une fois les identifiants vérifiés et validés, une

session PHP est lancée sur le serveur pour une durée de 20 minutes (durée par défaut). Ainsi, l'utilisateur a accès aux suppressions et ajouts des différents éléments du site.

5.3.5 Conclusion

Tout comme lors de la première soutenance, de bonnes impressions malgré les nouvelles notions abordées qui ne m'ont pas toujours semblées évidentes et qui ont donc requis un réel travail de recherche et de réflexion à ce niveau.

6 Conclusion

En définitive, pour terminer cette deuxième grosse étape dans le développement du projet *epiDatabase*, nous pouvons nous satisfaire du travail accompli, d'une part par la façon de sa réalisation mais également par la quantité. En effet, toutes les tâches prévues dans le cahier des charges ont été accomplies et certaines prévues pour la troisième et quatrième soutenance ont été commencées.

La troisième étape du développement devrait permettre au projet de devenir réellement opérationnel.