1 Travaux réalisés 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.

1.1 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.

1.1.1 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.

1.1.2 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.

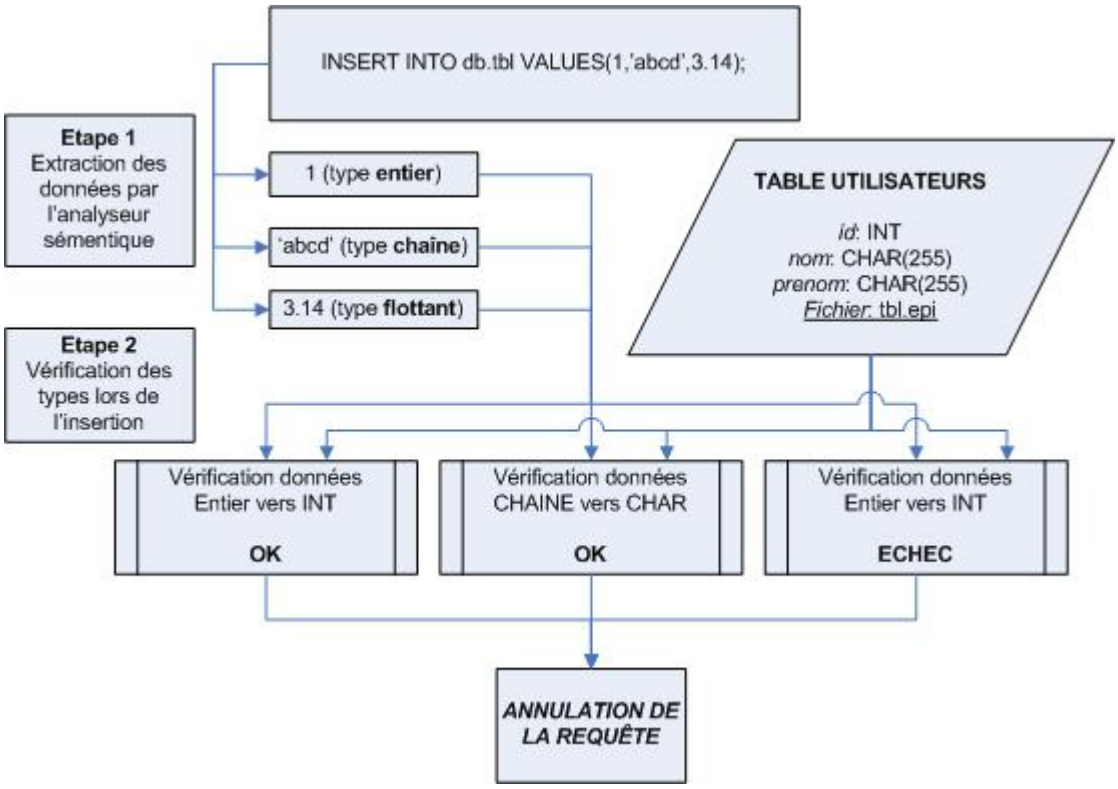

FIG. 1 – Stockages des informations de la requête et vérification des types

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.

1.2 epiEngine

1.2.1 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.

1.2.2 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 :

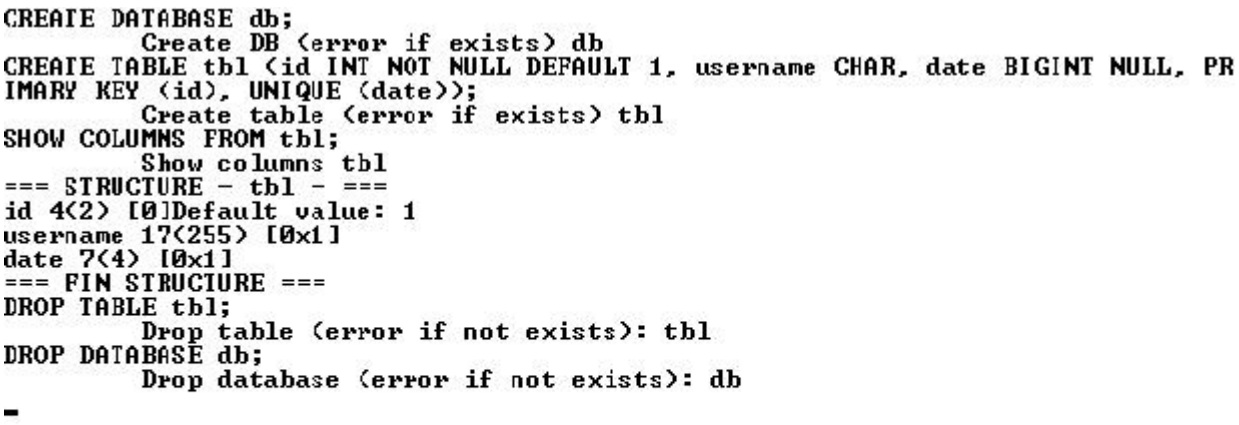

FIG. 2 – Exemple de session epiDatabase

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.

1.2.3 Modification de "DROP DATABASE"

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

1.2.4 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.

1.2.5 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;
```

1.3 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.

1.4 Conclusion

1.4.1 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.

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

1.4.3 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 !