UNIVERSITÉ Clermont Auvergne | ISIMA

LIMOS
LABORATOIRE D'INFORMATIQUE,
DE MODÉLISATION ET D'OPTIMISATION DES SYSTÈMES

1 Rue de la Chebarde
63178 Aubière

Rapport d'élève de master

Stage de M1

Filière : Informatique

Traitements des réponses d'une requête dans un environnement avec incertitude

Présenté par : **Martin BENITO-RODRIGUEZ**

Responsable ISIMA : Annegret WAGLER — **28 juin 2021**

Responsable entreprise : Lhouari NOURINE — **3 mois**

Campus des Cézeaux . 1 rue de la Chébarde . TSA 60125 . 63178 Aubière CEDEX

1. Remerciements

Je remercie mon tuteur entreprise Lhouari NOURINE, son doctorant Simon VILMIN et Jean-Marc Petit, chercheur au CNRS et Annegret WAGLER mon tuteur ISIMA pour leur aides, leurs conseils et pour m'avoir accompagné durant ce stage.

2. Table des matières

3. Table des figures et illustrations

4. Résumé

Le but de ce stage de recherche est d'étudier la faisabilité de la requête select dans un environnement avec incertitude, ainsi que le classement des réponses de la requête select dans un environnement avec incertitude.

Les programmes ont été codé avec le système de gestions de base de données PostgreSQL. Le développement a été fait sur l'IDE ATOM et les tests ont été faits sur le compilateur en ligne SQLliteonline.

A ce jour, les programmes fonctionnent correctement mais le temps d'exécution peut être long sur des données de tailles importantes en utilisant la table de fonctions de comparaisons.

Mots-clés : SGBD, PostgreSQL, SQL, fonction de comparaison, treillis, égalité, NULL

5. Abstract

The goal of this research internship is to study the feasibility of the select query in an environment with uncertainty, and the ranking of the responses of the select query in an environment with uncertainty.

The programs have been coded with the PostgreSQL database management system. The development was done on the IDE ATOM and the tests were done on the SQLliteonline online compiler.

To date, the programs work correctly but the execution time can be long on large datasets using the table of comparison functions.

Keywords : DBMS, PostgreSQL, SQL, comparability function, lattice, equality, NULL

6. Introduction

Ce stage est un stage de recherche au Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS) situé sur le site des Cézeaux. Il s'inscrit dans un processus à long terme pour le traitement de l'incertitude dans SQL. Ce trvail s'insère dans l'axe Modèles et Algorithmes de l'Aide à la Décision(MAAD) du LIMOS et dans le thème Algorithmique, Graphes, Complexité (AGC).

Les requêtes incertaines sont des requêtes sur des bases de données incomplètes. Elles sont présentes dans de nombreux domaines de l'activité humaine. Et les manières dont SQL les interprètent peuvent être très contres-intuitives.

La manière dont SQL traite les données avec l'égalité peut ne pas nous convenir. Le problème est particulièrement visible en médecine, où des expressions comme « très élevé », « élevé », « normal », « faible », « très faible » sont couramment utilisées pour décrire différents états. Jusqu'à présent, l'utilisateur est obligé de préciser dans la clause where comment filtrer les données, pour éviter cela, un de nos objectifs est de relaxer l'égalité pour l'utiliser dans la clause select.

Le second objectif est de trouver quels tuples d'une table sont les plus proches d'un tuple de référence avec l'égalité que nous avons nous-même défini.

Nous présenterons d'abord le LIMOS et l'égalité en SQL avant de présenter la méthodologie et les technologies adoptées. Puis nous verrons l'opérateur d'égalité de SQL relaxé ainsi qu'une méthode pour créer ses fonctions. Enfin nous présenterons la fonction pour chercher des tuples semblablent avant de conclure.

7. I-Contexte du stage et du problème

7.1 Présentation du LIMOS

Le Laboratoire d'informatique, de modélisation et d'optimisation des systèmes (LIMOS) est une unité mixte de recherche (UMR 6158) qui est situé à Aubière.

Il est rattachée à l'Institut des Sciences de l'Information et de leurs Interactions (INS2I) du Centre national de la recherche scientifique (CNRS) mais aussi à l'Institut des Sciences de l'Ingénierie et des Systèmes (INSIS).

Il est également rattaché à l'École des Mines de Saint-Étienne (EMSE) et à l'Université Clermont Auvergne (UCA).

Le LIMOS collabore avec quelques entreprises pour le développement ou la recherche comme Yansys, une entreprise éditrice de progiciels médicaux ou encore Michelin, pour la modélisation de pneumatique.

Le travail du laboratoire s'articule autour de 3 axes :
-Axe MAAD : Modèles et Algorithmes de l'Aide à la Décision.
-Axe SIC : Systèmes d'Information et de Communication
-Axe ODPL : Outils Décisionnels pour la Production et la Logistique.

Aujourd'hui, il est composé de 208 membres dont :
-95 enseignants-chercheurs et chercheurs
-77 doctorants
-8 postdocs
-9 chercheurs associés
-5 ATER
-14 Ingénieurs, Techniciens et Administratifs

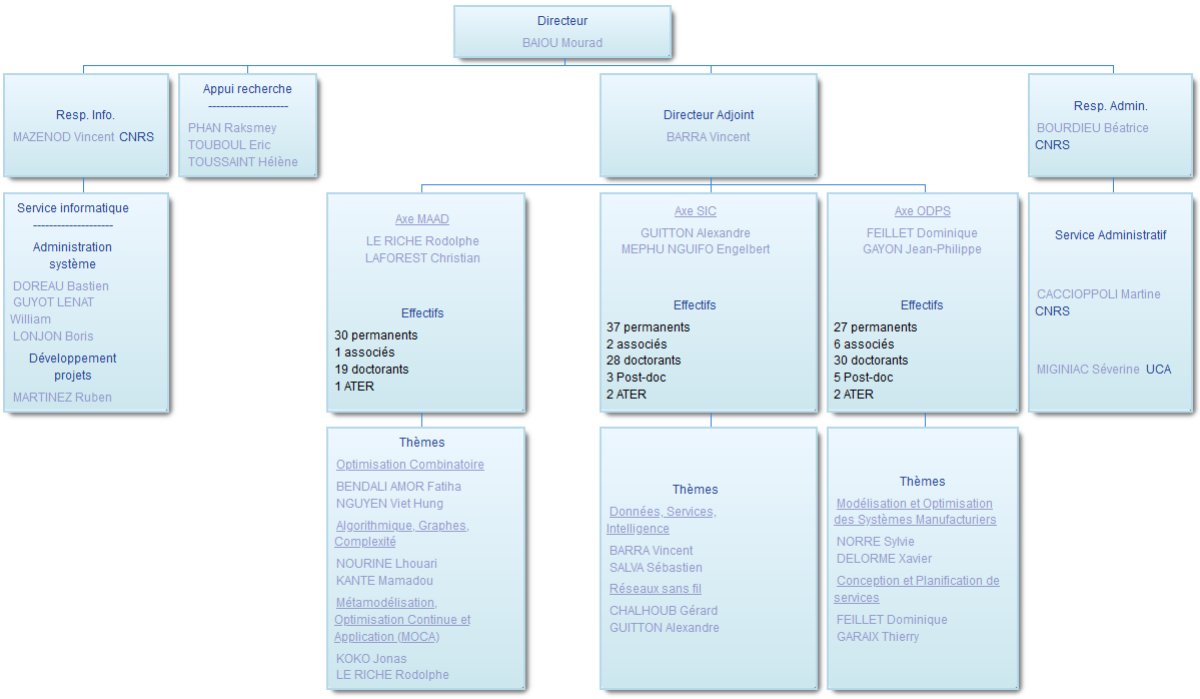

FIGURE 7.1 – Organigramme du LIMOS

Le LIMOS a publié 3803 documents depuis 1985 dont 60 depuis le début de l'année 2021.

7.2 L'égalité en SQL

SQL utilise une logique a 3 valeurs de vérité : vrai, faux et NULL. Manipuler des données qui sont vrais ou fausses se fait de manière assez intuitive, mais les choses se compliquent si un NULL entre en jeu.

Mes critiques portant sur le traitement des données incomplètes par SQL sont basées sur l'article [2]

La logique de SQL est basée sur la logique trivaluée de Kleene dont voici la table de vérité :

$\wedge$	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

$\vee$	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

	$\neg$
t	f
f	t
u	u

FIGURE 7.2 – Table de vérité de la logique trivaluée de Kleene
t signifie true pour vrai, f signifie false pour faux et u signifie unknow pour NULL. Extrait de [2]

La façon dont les opérateurs logique se comportent avec un NULL n'est pas forcément celle dont l'utilisateur a besoin.

Pour savoir si une valeur est null, une manière instinctive serait d'écrire `val=NULL`, cette opération renverrai vrai si val est null, sinon elle renverrai faux. Mais en réalité, cette opération renvoie NULL. Pour tester si une valeur est NULL, il faut écrire `val is NULL` ou bien `val is not NULL`.

En SQL, les opérateurs tel que la somme, le produit, la différence, etc... avec au moins une opérande NULL renvoient NULL.

D'un point de vue théorique, une réponse est correcte si il s'agit d'une réponse certaine, c'est à dire que peut importe l'interprétation que l'on a des données incomplètes, le résultat ne changera pas. Or le résultat d'une requête peut-être certain d'un point de vue théorique, mais pas dans la pratique comme dans l'exemple ci-dessous.

Voici une table Payments :

PAYMENTS

cid	*oid*
c1	o1
c2	o2

FIGURE 7.3 – Table Payments
Extrait de [2]

Et voilà une requête :

```
SELECT cid FROM Payments
  WHERE oid = 'o2' OR oid <> 'o2'
```

FIGURE 7.4 – Requête sur Payments
Extrait de [2]

Si on analyse la requête, on se rend compte que peu importe les valeurs de la colonne oid, tous les tuples de la table Payments devront êtres renvoyés. La réponse certaine dans ce cas est {c1, c2}, alors que si on remplace o2 par NULL, la requête renverra uniquement c1.

Nous avons vu qu'une requête peut renvoyer un résultat qui ne correspond pas à nos attentes. Le premier objectif est de trouver un moyen d'exécuter des requêtes select dans une base de données incomplètes qui correspond à nos attentes. Le second objectif est de trouver

quels tuples d'une table sont les plus semblable à un tuple de référence.

Mais jusqu'ici le problème que nous avons soulevé est l'interrogation des bases de données incomplètes. Mais on pourrait aussi critiquer l'égalité car même si ses opérandes ne sont pas NULL, l'égalité pourrait se comporter d'une manière qui ne nous convient pas. Dans certains domaines, 2 valeurs peuvent êtres considérées égales alors que pour SQL ce n'est pas le cas. C'est notamment le cas dans des domaines tel que la physique ou la biologie où l'on traite des données expérimentales. Je peux par exemple dire que si 2 personnes ont une différence de taille inférieur ou égal à 1 millimètre, elles font la même taille. Si on raisonne comme cela, seuls les experts des domaines peuvent vraiment dire ce qu'est l'égalité, tandis que le programmeur l'ignore.

Une idée pour résoudre ce problème a été donné dans [1] où les chercheurs utilisent un treillis de vérité. Parmi ces 3 chercheurs, l'un est mon tuteur entreprise, mon travail sera de mettre en pratique la théorie de leur article.

Un treillis de vérité est une structure mathématiques composé de sommets reliés entre eux par des arêtes, tel que chaque sommet correspond à une valeur de vérité. Plus le sommet est haut dans le treillis, plus sa valeur est proche de vrai.

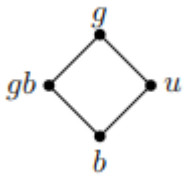

FIGURE 7.5 – Exemple d'un treillis de vérité
Chaque valeur est écrite en abrégée : g pour good, gb pour good or bad, u pour unknown, b pour bad. Extrait de [1]

Pour utiliser les treillis de vérité dans le cas de notre problème, on assigne à chaque attribut dans le schéma de relation un treillis de vérité, ainsi qu'une fonction de comparaison. Cette fonction possède 2 arguments et renvoie une valeur de vérité du treillis de l'attribut.

$$f_A(x,y) = \begin{cases} good & \text{if } x = y \text{ or } x,y \in [0,2[\\ good\ or\ bad & \text{if } x,y \in [2,5[, x \neq y \\ unknown & \text{if } (x,y) \text{ or } (y,x) \in [0,2[\times[2,5[\\ bad & \text{otherwise.} \end{cases}$$

FIGURE 7.6 – Exemple d'une fonction de comparaison
Extrait de [1]

Maintenant que ce système est mis en place, il nous reste une dernière chose à faire : assigner une valeur de vérité booléenne aux valeurs de vérités du treillis de vérité. On utilise pour cela une interprétation de notre treillis de vérité.

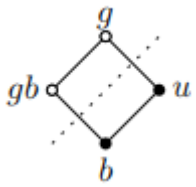

FIGURE 7.7 – Exemple d'une interprétation de treillis
Les sommets situés au dessus de la ligne de pointillés sont considérés comme vrais, les autres comme faux. Extrait de [1]

Grâce à ce système, nous avons une base théorique pour redéfinir l'égalité, c'est sur elle que l'on va s'appuyer.

Bien sûr, l'interprétation est destinée à être modifiée en fonction de nos besoins, de même que la fonction de comparaison.

Durant tous le reste du rapport j'utiliserai une même table avec ses fonctions de comparaisons et ses interprétations comme exemple :

r	T. level (A)	Gender (B)	W. size (C)
t_1	1.4	F	73
t_2	1.5	F	null
t_3	3.2	M	72
t_4	3.5	F	76
t_5	40	F	100

FIGURE 7.8 – Une relation r
Extrait de [1]

$$f_A(x,y) = \begin{cases} \textit{good} & \text{if } x = y \text{ or } x, y \in [0,2[\\ \textit{good or bad} & \text{if } x, y \in [2,5[, x \neq y \\ \textit{unknown} & \text{if } (x,y) \text{ or } (y,x) \in [0,2[\times[2,5[\\ \textit{bad} & \text{otherwise.} \end{cases}$$

$$f_B(x,y) = \begin{cases} \textit{true} & \text{if } x = y \\ \textit{different} & \text{if } x \neq y \end{cases}$$

$$f_C(x,y) = \begin{cases} \textit{correct} & \text{if } x = y \neq \textsf{null} \text{ or } x, y \in [70,80] \\ \textit{unknown} & \text{if } u = \textsf{null} \text{ or } v = \textsf{null} \\ \textit{incorrect} & \text{otherwise.} \end{cases}$$

FIGURE 7.9 – Fonctions de comparaison des attributs de r
Extrait de [1]

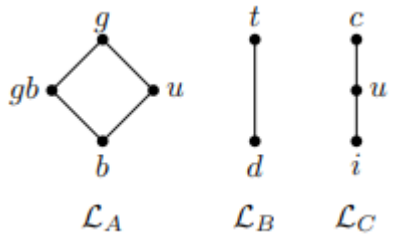

FIGURE 7.10 – treillis de vérité des attributs r
Extrait de [1]

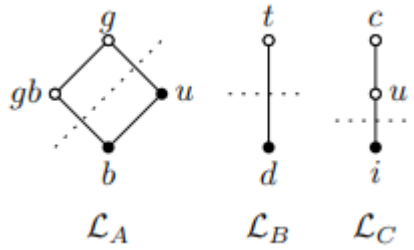

FIGURE 7.11 – Interprétation des attributs des treillis de vérité de r
Extrait de [1]

8. II-Méthodologie et outils

8.1 Mis en oeuvre du projet

Durant tout le long du stage, j'ai été accompagné par Lhouari NOURINE, chercheur au LIMOS, et son doctorant Simon VILMIN. Les premières réunions se faisaient en présentielles au LIMOS, tandis que les dernières se faisaient en distancielles avec ZOOM pour pouvoir communiquer avec Jean-Marc PETIT, chercheur au CNRS à l'Université de Lyon qui se joignit a notre équipe. Je rappelle que je dois mettre en pratique ce que ces 3 chercheurs ont établi. Enfin, Gabriel Eychene lui aussi en stage de recherche au LIMOS s'est joint à nos réunions car il doit récupérer mon travail pour pouvoir travailler. Pour chaque réunion en présentielles, je devais rédiger un compte rendu de ce que j'avais fait. Pour chaque réunion en distancielles, je préparais des diapositives pour présenter mon travail.

Au début du stage, Lhouari NOURINE m'a dit les principaux objectifs, mais sans planification. L'évolution se fera petit à petit en fonction de mon avancement, peut-être aurai-je le temps de faire des choses en plus ou peut-être n'aurai-je pas le temps de boucler les objectifs à temps. Finalement, j'ai atteint les 2 objectifs principaux : la création des opérateurs et la fonction abstract. Et j'ai eu le temps de créer le programme FunctionTable qui était optionnel. Le diagramme de Gantt ci-dessous correspond au travail réalisé et est donc le seul diagramme de Gantt car il n'existe pas de diagramme de Gantt prévisionnel.

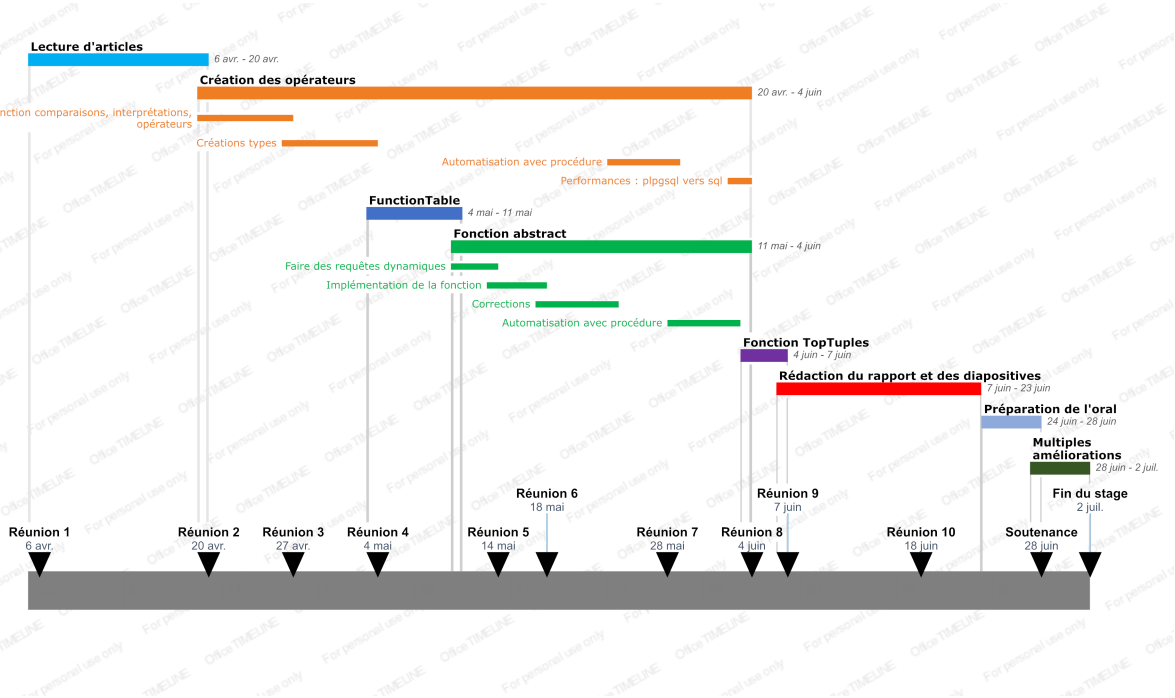

FIGURE 8.1 – Diagramme de Gantt

8.2 Technologies utilisées

Les codes que j'ai écrit ont été implémenté avec le SGBD PostgreSQL que je ne connaissais pas. Mais d'après Simon VILMIN, il était facile de créer des opérateurs. J'ai donc du apprendre a manipuler ce SGBD qui est libre et gratuit.

Dans un premiers temps, j'ai voulu l'utiliser sur mon ordinateur sous Windows 10. Je l'ai installé, mais je n'ai pas réussi a le lancer avec la console, j'ai donc installé PgAdmin qui permet d'avoir une interface pour travailler sur postgreSQL, mais je n'ai pas réussi à le lancer non plus. J'ai alors travaillé sur internet avec des sites qui permettent de faire des requêtes en ligne. J'ai d'abord utilisé le site ExtendsClass (https ://extendsclass.com/postgreSQL-online.html), puis j'ai changé pour SQLiteonline (https ://SQLiteonline.com/) car la fenêtre pour écrire les requêtes était bien plus grande et il y avait un mode sombre, ce qui rendait le travail beaucoup plus confortable.

Pour utiliser ce SGBD, je me suis surtout appuyé sur la documentation sur le site officiel de PostgreSQL (https ://www.postgreSQL.org/docs/current/), il arrivait quelquefois que la documentation ne réponde pas à mes interrogations, j'ai utilisé le moteur de recherche Google pour trouver des solutions. Dans quasiment tous les cas, je trouvais une réponse sur le forum stackoverflow (https ://stackoverflow.com/).

Il me fallait également un éditeur pour écrire mon code source. J'ai opté pour l'éditeur de code source Atom qui est un environnement de développement libre et gratuit. Il est

considéré comme un des meilleurs éditeurs de source en raison de son nombre impressionnant de packages open-source qui permettent de personnaliser l'interface à volonté.

En ce qui concerne les tests, j'ai utilisé la commande `explain analyse` pour obtenir le temps d'exécution d'une requête. Puisque mes requêtes étaient lancées sur un site en ligne, mes résultats obtenus sont à prendre avec des pincettes car le nombre d'utilisateur altère les capacités du serveur.

9. Création des opérateurs

Nous avons vus dans l'introduction un contexte théorique pour relaxer l'égalité. On a pour cela besoin d'une fonction de comparaison, d'un treillis de vérité et d'une interprétation de ce treillis sur les attribus dont nous voulons modifier l'égalité.

Le SGBD PostgreSQL permet de surcharger les opérateurs, c'est à dire que l'on peut ajouter un nouvel opérateur avec le nom d'un opérateur existant. Si plusieurs opérateurs (ou fonctions) ont le mêmes noms, alors c'est grâce aux types des opérandes que le SGBD déterminera quel opérateur ou fonction utiliser.

Un premier problème qui se pose est que des attributs vont (très probablement) avoir le même type, et à moins qu'ils possèdent tous la même fonction de comparaison, on ne pourra pas créer un opérateur pour chaque attribut. Plusieurs options s'offrent à nous pour régler ce problème : on peut par exemple créer un type pour chaque attribut, puis on créera un opérateur dont les opérandes sont du type que l'on a créé. Une autre méthode serait de créer un opérateur différent pour chaque attribut en changeant de symbole d'opérateur. C'est finalement la première méthode que j'ai adopté.

Avec PostgreSQL, il existe plusieurs manières de créer des types, une méthode simple pour répondre au problème est l'utilisation de types composites. Ce sont des types auxquels ont peut assigner plusieurs variables, un peu comme les structures en C. On va donc ici créer un type composite avec une seule variable à l'intérieur et cela pour chaque attribut.

```
create type level_comp AS (val numeric);
create type gender_comp AS (val char);
create type size_comp AS (val float);
```

Pour nous y retrouver, j'ai pris des conventions : le nom du nouveau type viendra de la concaténation entre le nom de l'attribut de base avec « _comp ». L'unique variable dans le type composite se nommera « val » et son type sera le type de l'attribut de base.

Puisque nous avons créé de nouveaux types, il faut créer une nouvelle table en remplaçant les types de la table de base avec les nouveaux types. Un défaut de l'approche que j'ai choisi (création de nouveaux types) est que les requêtes que nous ferons ne seront pas adressées à

la table de base, mais sur une table que nous devons créer nous-même.

```
create table r_comp (level level_comp, gender gender_comp,
size size_comp);
```

J'ai là aussi posé des conventions : le nom de la nouvelle table viendra de la concaténation entre le nom de la table de base avec « _comp ». Puis chaque attribut de cette nouvelle table se nommera comme la table de base, mais avec les types composites que nous avons créé.

Il faut maintenant remplir cette nouvelle table. Une valeur d'un type composite s'écrit sous la forme `row(val)`, ainsi, au lieu d'écrire ça :

```
insert into r values (1.4, 'F', 73);
insert into r values (1.5, 'F', null);
```

Nous écrirons ça :

```
insert into r_comp values (row(1.4), row('F'), row(73));
insert into r_comp values (row(1.5), row('F'), row(null));
```

Maintenant que la table `r_comp` est prête, nous allons créer l'opérateur et pour cela il faut tout d'abord écrire la fonction de comparaison.

Mais avant, il faut trouver un moyen pour implémenter la notion de treillis de vérité. La méthode retenue est de remplacer les valeurs de vérités (comme good, bad, etc...) par des mots binaires. Pour un treillis donné, tous ses mots binaires auront le même nombre de bits. Si une valeur de vérité est directement supérieur à une autre (c'est à dire, séparées par une seule arête), alors la valeur supérieure aura un seul bit de différent à de valeur inférieure : ce bit sera a 1 pour la valeur supérieure et à 0 pour la valeur inférieure. Donc plus on monte dans le treillis (plus les valeurs sont proches de vrais), plus les mots contiennent de 1.

Notre treillis de base est comme ça :

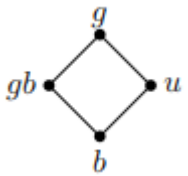

FIGURE 9.1 – Exemple d'un treillis de vérité
Chaque valeur est écrite en abrégée : g pour good, gb pour good or bad, u pour unknown, b pour bad. Extrait de [1]

Nous allons le représenter comme ceci :

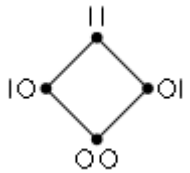

FIGURE 9.2 – Représentation du treillis précedent en treillis binaire

Ainsi, au lieu de renvoyer les valeurs de vérité du treillis, les fonctions de comparaisons renverront des mots binaires. Ces mots binaires seront ensuite interprétés pour obtenir des valeurs booléennes.

Voici la fonction de comparaison que l'on veut implémenter :

$$f_A(x,y) = \begin{cases} good & \text{if } x = y \text{ or } x,y \in [0,2[\\ good\ or\ bad & \text{if } x,y \in [2,5[, x \neq y \\ unknown & \text{if } (x,y) \text{ or } (y,x) \in [0,2[\times[2,5[\\ bad & \text{otherwise.} \end{cases}$$

FIGURE 9.3 – La fonction de comparaison de l'attribut level
Extrait de [1]

Et voici son implémentation en SQL :

```
create or replace function flevel(x numeric, y numeric) returns bit
AS $$
    select
        case
        when (x=y) or ((0<=x and x<2) and (0<=y and y<2)) then b'11'
        when ((2<=x and x<5) and (2<=y and y<5)) then b'10'
        when ((0<=x and x<2) and (2<=y and y<5)) or
        ((2<=x and x<5) and (0<=y and y<2)) then b'01'
        else b'00'
    end
$$ language SQL;
```

Dans un premier temps j'ai écrit mes fonctions de comparaisons en plpgSQL, mais je me suis rendu compte que mes requêtes étaient très lentes. En passant du langage SQL au langage plpgSQL, les fonctions de comparaisons en SQL étaient environs 50 fois plus rapides que les fonctions en plpgSQL.

Une fois la fonction de comparaison écrite, il faut écrire la fonction d'interprétation qui en fonction du mots binaire renverra vrai ou faux.

Nous voulons encoder cette interprétation :

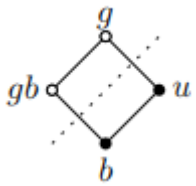

FIGURE 9.4 – Exemple d'une interprétation de treillis
Les sommets situés au dessus de la ligne de pointillés sont considérés comme vrais, les autres comme faux. Extrait de [1]

```
create or replace function interprlevel(b bit) returns boolean as $$
    select
    case
      when b='11' or b='10' then true
      else false
    end
$$ language SQL;
```

Avant de créer l'opérateur il y a un problème à régler : il faut pouvoir faire en sorte que l'utilisateur puisse utiliser l'opérateur de différentes manières : attribut=constante, constante=attribut et attribut=attribut. Pour cela nous sommes contraints de créer 3 opérateurs en fonction du type des opérandes et donc de créer 3 fonctions supplémentaires, une par opérateur. Ces nouvelles fonctions s'écriront sous la forme « gattribut » et porterons toutes le même nom car les opérandes sont différents à chaque fois ce qui permet d'éviter les confusions.

Voilà par exemple la fonction pour l'opérateur attribut=constante.

```
create or replace function glevel(x level_comp, y numeric) returns boolean
        select interprlevel(flevel(x.val, y));
$t$ language SQL;
```

Et voici son opérateur :

```
create operator = (
        function = glevel,
        leftarg = level_comp,
        rightarg = numeric,
```

```
        negator = !=  ,
        commutator = =,
        restrict = eqsel ,
        join = eqjoinsel ,
        merges
);
```

Il faut écrire la fonction `glevel` et son opérateur pour les 3 cas possibles. A noter qu'il est plus prudent pour l'opérateur entre les 2 attributs de l'écrire `==` car dans certains cas que je ne comprends pas, il y a une confusion entre les opérateurs.

Mais après avoir écrit toutes ces fonctions, nous avons relaxé l'égalité uniquement sur un seul attribut. Écrire toute ces informations pour chaque attribut peut être long sur des tables importantes.

J'ai alors créé une procédure pour automatiser la création des types, de la table, des fonctions et des opérateurs.

Cette procédure se nomme `createtable_comp` et prend en paramètre une chaîne de caractères qui doit prendre la forme de tables séparées par des virgules si il y a plus d'une table. Les types, tables et fonctions sont créés en utilisant du SQL dynamique.

Avant de lancer la procédure, il faut s'assurer d'avoir chargé les fonctions de comparaisons et d'interprétations sur les attributs qui nous intéressent, mais nous ne sommes pas obligés d'écrire ces 2 fonctions pour tous les attributs. Car même si avoir des fonctions de comparaisons est utile, certains attributs de nos tables n'en on pas besoins, peut-être que l'égalité normale convient très bien a certains attributs.

Donc lors du lancement de la procédure, la procédure va analyser chaque attribut de chaque table et si il existe pour cet attribut une fonction de comparaison ET une fonction d'interprétation, le nouveau type composite est créé. C'est pour cela qu'il est important de respecter les conventions de nommage des fonctions de comparaisons et d'interprétations.

Après la création des types, on créé la nouvelle table `table_comp` qui a comme types des attributs le type créé si les 2 fonctions existent ou le type de base dans le cas contraire. Par exemple si j'ai uniquement chargé mes fonctions de comparaisons et d'interprétations pour les attributs level et size, voici la requête dynamique lancée par la procédure lors de la création de la table `table_comp` :

```
create table r_comp (level level_comp, gender char(1),
size size_comp);
```

On peut voir que gender a conservé son type de base et que level et size ont tous deux un nouveau type.

Il nous reste enfin a remplir la table en mettant soit des valeurs composites, soit les valeurs de bases. Si on conserve l'exemple de la table créée ci-dessus, les premiers tuples insérés dans la table seront :

```
insert into r_comp (row(1.4), 'F', row(73));
insert into r_comp (row(1.5), 'F', row(null));
```

Maintenant nous pouvons enfin faire des requêtes sur cette tables :

```
call createtable_comp('r');
select * from r_comp where level=1 and gender='F';
```

Je rappelle que les 2 opérateurs dans la clause where n'ont pas la même origine. Le égal dans `gender='F'` est l'opérateur de base car les gender dans la table `r_comp` sont des `char(1)`, tandis que le = de `level=1` est un opérateur créé automatiquement par la procédure.

Voici les performances de mes opérateurs comparées à l'opérateur d'égalité de base. Attention, les 2 opérateurs ne retournent pas forcément le même nombre de lignes.

	Table de base		Table de comparaison	
Requêtes précédées par «select * from <table> where»	**# Lignes**	**Performances**	**# Lignes**	**Performances**
level=5;	0 / 5	0,035 ms	0 / 5	0,045 ms
level=1.4;	1 / 5	0,055 ms	2 / 5	0,075 ms
r.level=s.level;	1 / 5	0,180 ms	9 / 5	0,350 ms
sepal_l=0;	0 / 150	0,065 ms	0 / 150	0,150 ms
sepal_l=5;	10 / 150	0,060 ms	83 / 150	0,170 ms
r.sepal_l=s.sepal_l;	900 / 150	0,700 ms	12820 / 150	25,000 ms

FIGURE 9.5 – Performances des requêtes sur le table de base et sur la table comp

On voit que les performances des nouveaux opérateurs sont assez convenables comparées aux performances des opérateurs classiques.

On pourrait reprocher à mon modèle plusieurs choses : tout d'abord, les requêtes ne sont pas lancées sur la table de base mais sur une table qu'on créer. Ensuite l'opérateur pour comparer 2 attributs est '==', et non '='.

Pour faire évoluer ce programme, je pourrais créer tous les opérateurs classiques : !=, <, >, <=, >=. Je pourrais tenter aussi d'améliorer les performances.

10. La table functionTable

Les programmes que je présente sont destinés a êtres utilisés par des experts de domaines spécifiques. Or ces experts n'ont pas forcément de compétences en informatique mais ils doivent tout de même écrire des fonctions de comparaisons. J'ai donc créer un programme qui permet à un utilisateur d'encoder des fonctions dans une table appelée functionTable.

Chaque tables avec des données possède sa propre functionTable, par exemple, si nous avons les tables r et iris, nous pouvons aussi avoir functionTable_r et functionTable_iris.

Pour rappel, voici nos fonctions de comparaisons :

$$f_A(x,y) = \begin{cases} \textit{good} & \text{if } x = y \text{ or } x, y \in [0,2[\\ \textit{good or bad} & \text{if } x, y \in [2,5[, x \neq y \\ \textit{unknown} & \text{if } (x,y) \text{ or } (y,x) \in [0,2[\times[2,5[\\ \textit{bad} & \text{otherwise.} \end{cases}$$

$$f_B(x,y) = \begin{cases} \textit{true} & \text{if } x = y \\ \textit{different} & \text{if } x \neq y \end{cases}$$

$$f_C(x,y) = \begin{cases} \textit{correct} & \text{if } x = y \neq \textsf{null} \text{ or } x, y \in [70,80] \\ \textit{unknown} & \text{if } u = \textsf{null} \text{ or } v = \textsf{null} \\ \textit{incorrect} & \text{otherwise.} \end{cases}$$

FIGURE 10.1 – Les fonctions de comparaisons de r
Extrait de [1]

Et voici l'encodage dans la table functionTable_r :

name	num	x	y	result
flevel	0	y || [0,2[	x || [0,2[	11
flevel	1	[2,5[&& not y	[2,5[&& not x	10
flevel	2	[0,2[	[2,5[	01
flevel	3	[2,5[	[0,2[	01
flevel	4	other	other	00
fgender	0	y	x	1
fgender	1	not y	not x	0
fsize	0	y && not null || [70,80]	x && not null || [70,80]	11
fsize	1	null	not null	10
fsize	2	not null	null	10
fsize	3	other	other	00

FIGURE 10.2 – La table functionTable_r

Analysons cette table, elle possède 5 colonnes : name, num, x, y et result. La colonne name correspond au nom de nos fonctions, ici la table r possède 3 fonctions de comparaisons : flevel, fgender et fsize. La 2e colonne « num » assigne un numéro à chaque lignes sachant qu'un numéro est unique pour une même fonction de comparaison, qu'il s'incrémente à chaque ligne et qu'il commence à 0. Les colonnes x et y correspondent aux conditions des 2 paramètres qu'il faut écrire en chaîne de caractères. Pour cela, il faut utiliser les opérateurs logiques « ||, » avec des expressions comme « y », « not y » ou des intervalles. Par exemple, la condition de x du premier tuple de fsize « y not null || [70,80] » signifie que la condition est satisfaite si x est égal à y et que x est non nul ou qu'il appartient à l'intervalle [70,80]. On peut si besoin modifier le programme pour ajouter d'autres types de conditions comme la différence maximale entre 2 valeurs.

Enfin, la colonne result correspond à ce que doit renvoyer la fonction quand les 2 conditions sur x et y sont vrais.

On peut remarquer qu'on peut tout à fait écrire plusieurs tuples pour renvoyer une même valeur, par exemple les num 2 et 3 de flevel, ou encore les num 1 et 2 de fsize.

Une fois la table chargé dans le SGBD, nous devons écrire des fonctions qui se nomment « flevel », « fgender » et « fsize ». Leur résultat dépendra du contenu de la table functionTable.

```
create or replace function flevel(a numeric, b numeric) returns
bit as $$
        select result
        from functiontable_r
        where name='flevel' and
        num=search(name, 'functiontable_r', a::text, b::text);
$$ language SQL;
```

La fonction `search` va parcourir la table `functionTable_r` à la recherche d'un tuple pour lequel les 2 conditions x et y sont vrais. Ce tuple est identifié par son num. La fonction flevel renvoie ensuite le result correspondant au num. Puisque les tuples sont parcourus dans l'ordre, il est important de placer les `other` à la fin.

Les 2 autres fonctions sont bâties sur le même modèle, les seules choses à changer sont le nom de la fonction ainsi que le « flevel » dans la clause where.

Une fois la table `functionTable` et les fonctions de comparaisons créées, nous pouvons comme dans la partie précédentes utiliser la procédure `createtable_comp` pour créer les opérateurs.

	Table de base		Table de comparaison		Table de comparaison et functionTable
Requêtes précédées par «select * from <table> where»	**# Lignes**	**Performances**	**# Lignes**	**Performances**	**Performances**
level=5;	0 / 5	0,035 ms	0 / 5	0,045 ms	25,000 ms
level=1.4;	1 / 5	0,055 ms	2 / 5	0,075 ms	14,000 ms
r.level=s.level;	1 / 5	0,180 ms	9 / 5	0,350 ms	39,000 ms
sepal_l=0;	0 / 150	0,065 ms	0 / 150	0,150 ms	220,000 ms
sepal_l=5;	10 / 150	0,060 ms	83 / 150	0,170 ms	220,000 ms
r.sepal_l=s.sepal_l;	900 / 150	0,700 ms	12820 / 150	25,000 ms	timeout

FIGURE 10.3 – Performances de l'opérateur d'égalité personnalisé avec ou sans functionTable comparé avec l'opérateur d'égalité de base

On voit que l'opérateur de la table de comparaison avec la functionTable est plusieurs centaines de fois plus lent que l'opérateur sans la functionTable.

Pour faire évoluer mon programme, la toute première chose a faire est de trouver un moyen d'améliorer les performances qui sont vraiment très mauvaises. Une fois ceci fait, je pourrais écrire une procédure pour générer automatiquement les fonctions de comparaisons qui se ressemblent toutes. Cette procédure n'aura qu'a parcourir la table `functionTable_r` et pour chaque name différent, elle créerait une fonction. Je pourrais aussi essayer de supprimer la colonne num pour la remplacer par un index qui se créé automatiquement avec une procédure. Cela éviterais à l'utilisateur de remplir la colonne num, mais les calculs supplémentaires pourrais diminuer les performances. De plus, puisque cette fonctionnalité est censée être utilisée par des non-informaticiens, je pourrais aussi créer une table pour les fonctions d'interprétations.

11. La table abstraite et la fonction topTuples

Dance ce chapitre, nous verrons comment trouver quels sont les tuples les plus similaires dans une table à un tuple de référence.

Nous avons vu dans la partie sur la création des opérateurs que nous pouvons comparer 2 valeurs pour obtenir un mot binaire. Nous pouvons alors comparer 2 tuples de cette manière.

r	T. level (A)	Gender (B)	W. size (C)
t_1	1.4	F	73
t_2	1.5	F	null
t_3	3.2	M	72
t_4	3.5	F	76
t_5	40	F	100

FIGURE 11.1 – Une relation r
Extrait de [1]

$$f_A(x,y) = \begin{cases} good & \text{if } x = y \text{ or } x, y \in [0,2[\\ good \text{ or } bad & \text{if } x, y \in [2,5[, x \neq y \\ unknown & \text{if } (x,y) \text{ or } (y,x) \in [0,2[\times[2,5[\\ bad & \text{otherwise.} \end{cases}$$

$$f_B(x,y) = \begin{cases} true & \text{if } x = y \\ different & \text{if } x \neq y \end{cases}$$

$$f_C(x,y) = \begin{cases} correct & \text{if } x = y \neq \textsf{null} \text{ or } x, y \in [70,80] \\ unknown & \text{if } u = \textsf{null} \text{ or } v = \textsf{null} \\ incorrect & \text{otherwise.} \end{cases}$$

FIGURE 11.2 – Fonctions de comparaison des attributs de r
Extrait de [1]

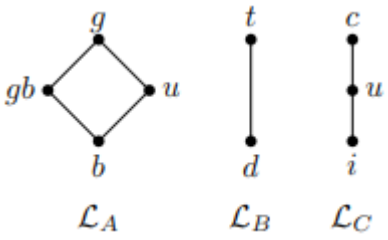

FIGURE 11.3 – treillis de vérité des attributs r
Extrait de [1]

Si nous considérons le couple de tuples (t1,t2) de r, nous pouvons comparer ces deux tuples en comparant chacun de leur attributs.

Nous avons $fr(t1,t2) = (fa(1.4,1.5), fb(F,F), fc(73,null)) = (g,t,u)$. (g,t,u) est appelé un tuple abstrait.

Puisque nous avons décidé de représenter les valeurs de vérités des treillis par des mots binaires, alors chaque élément d'un tuple abstrait est un mot binaire. Ici (g,t,u) correspond à (11,1,10).

A partir de là, nous pouvons générer une table abstraite, c'est à dire que l'on peut comparer une table avec un tuple pour obtenir une table avec le même nombre de ligne et de colonne que la table de base. Mais où la valeur placé sur la ligne i et la colonne j est un mot binaire qui vient de la comparaison entre la valeur de la ligne i et la colonne j de la table de base et la valeur de la colonne j du tuple.

Par exemple, nous pouvons comparer la table r avec le tuple (1, F, 70).

level	gender	size
11	1	11
11	1	10
01	0	11
01	1	11
00	1	10

FIGURE 11.4 – table abstraite r avec le tuple (1, F, 70)

Pour obtenir une telle table, nous utilisons une fonction `abstract_r` qui prend comme argument une chaîne de caractère avec des égalités séparées par une virgule. Dans l'exemple que j'ai montré : `abstract_r('level=1, gender="F", size=70')`. Puisque c'est une fonction qui renvoie une table, la requête nécessaire pour obtenir cette table est donc `select * from abstract_r('level=1, gender="F", size=70')`.

Mais nous ne sommes pas obligés de donner une valeur à chaque attributs. Si nous lancons la requête : `select * from abstract_r('level=1, size=70')`, la colonne gender sera remplie de NULL.

Puisque la fonction s'appelle `abstract_r`, on devine que cette fonction ne peut s'utiliser que sur la table r. Ainsi, chaque table aura sa propre fonction `abstract`.

Dans un premier temps, j'avais pensé à créer une fonction `abstract` qui prend en paramètre 2 chaînes de caractères : la première chaîne correspond à la table, et la deuxième aux conditions. Seulement, les fonctions qui retournent une table ont besoins de noms de colonnes et de types de colonnes. Sachant cela, je n'ai pas trouvé de moyens pour faire une fonction qui peut renvoyer des tables avec un nombre de colonne variable, des noms de colonnes variables et des types de colonnes variables. J'ai donc choisit de faire une fonction par table.

Une fois la table abstraite obtenue, il faut trouver quels tuples sont les plus proches du tuple de référence. C'est le travail de mon camarade Gabriel Eychene qui prend une table avec des mots binaires et grâce à sa fonction skyline, renvoie les meilleurs tuples.

Cependant les tuples qu'il renvoie sont les tuples sous formes binaires et nous voulons obtenir les tuples originaux. J'ai donc créer une fonction `topTuples_r` qui s'utilise de la même manière que `abstract_r`, mais renvoie une table avec les tuples venant de r les plus proches des conditions spécifiées en paramètre.

La requête `select * from topTuples_r('level=1, size=70')` renverra la table suivante :

level	gender	size
1.4	F	73

FIGURE 11.5 – Tuple le plus proche de (level=1,size=70) dans r

Pour créer la fonction `topTuples_r`, j'appelle une procédure `create_topTuples` qui a un comportement semblable à la procédure `createtable_comp`. La procédure prend un argument une chaîne de caractère qui sont les tables séparées par une virgule et génère une fonction topTuples par table.

La commande suivante `call create_topTuples('r, iris')` créer les fonction `topTuples_r` et `topTuples_iris`.

select * from abstract_r('level=1, size=70');	22,000 ms
select * from toptuples_r('level=1, size=70');	60,000 ms
select * from abstract_iris('sepal_l=5');	20,000 ms
select * from abstract_iris('sepal_l=5, sepal_w=3, petal_l=3.8, petal_w=1.2');	20,000 ms
select * from topTuples_iris('sepal_l=5');	90,000 ms
select * from topTuples_iris('sepal_l=5, sepal_w=3, petal_l=3.8, petal_w=1.2');	90,000 ms

FIGURE 11.6 – Performances des fonctions abstract et topTuples

On remarque que pour les fonctions abstract, le temps d'exécution est toujours d'environ 20 millisecondes, peu importe la taille des données. On voit aussi que le nombre de conditions n'a pas d'impact sur les performances. Concernant les fonctions topTuples, la taille des données a un impact, mais celui-ci n'est pas très violent. Et comme pour les fonctions abstract, le nombre de conditions ne modifie pas les performances.

Pour améliorer ce programme, on pourrais utiliser d'autres opérateurs que l'égalité dans les fonctions abstract ou topTuples comme la négation, l'infériorité, la supériorité, etc...

12. Conclusion

Nos objectifs étaient de relaxer l'égalité d'une requête select et de classer des données d'une table avec la nouvelle égalité. Nous avons réussi à générer nos propres opérateurs d'égalité pour lancer des requêtes qui se comportent comme nous le souhaitons. De plus, nous avons pu créer des fonctions qui permettent de trouver les tuples d'une table les plus proches d'un tuple idéal.

Ces objectifs ont été atteints et un objectif bonus a pu être complété avec la table FunctionTable. Les 3 programmes fonctionnent convenablement.

Les difficultés ont surtout été d'ordre techniques car je ne connaissais pas le SGBD, mais grâce au forum stackoverflow, j'ai réussi à implémenter les fonctionnalités nécessaires.

Ce stage m'a appris à utiliser le SGBD PostgreSQL qui est un des SGBD les plus utilisé. De plus, ce fut ma toute première expérience importante centré sur les bases de données, cela m'a permis de mieux comprendre SQL, notamment le fonctionnement de select. C'était aussi la toute première fois que j'utilisai l'environnement de développement Atom et son système de packages. Enfin, ce stage m'a appris à mieux lire un article de recherche.

Maintenant que les objectifs du projets ont été atteints, il reste de nombreuses pistes pour améliorer les programmes. Je me suis surtout concentré sur l'égalité, que ce soit pour la création des opérateurs ou pour la table abstraite, mais on pourrait généraliser ça à l'ensemble des opérateurs logiques comme la négation, l'infériorité, etc... Puisque les programmes ne sont pas destinés à être spécialement utilisé par des informaticiens, nous pourrions partir d'un fichier csv ou xml pour créer les fonctions nécessaires. Enfin, certaines performances peuvent et doivent êtres améliorer.

13. Références bibliographiques

Bibliographie

[1] Simon Vilmin Lhouari Nourine, Jean-Marc Petit. Towards declarative comparabilities : application to functional dependencies. 2021.

[2] Leonid Libkin Etienne Toussaint Marco Console, Paolo Guagliardo. Coping with incomplete data : Recent advances. pages 33–47, Jun 2020.