SQLf : extension de PostgreSQL au traitement de requêtes flexibles

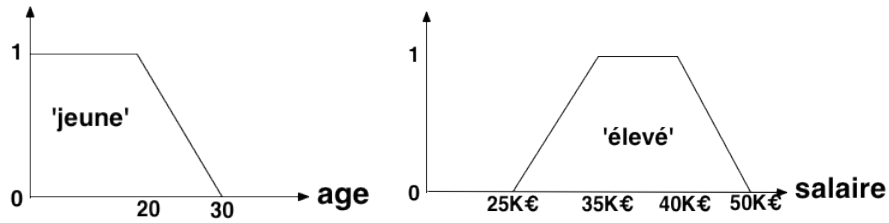
Introduction

Ce travail est réalisé dans le cadre d'une collaboration avec l'équipe IRISA/Pilgrim et l'association PostgreSQL-France.

L'objectif principal de ce travail est de permettre l'interrogation d'un système de gestion de bases de données relationnel (SGBDR) par des requêtes prenant compte de conditions fondées sur les ensembles flous. L'application réalisée modifie le plan d'exécution de requêtes du SGBDR PostgreSQL (version 9.1) en se fondant sur son code original.

SQLf : présentation

SQLf s'appuie sur le fait que les ensembles flous modélisent des qualificatifs du langage naturel qui sont par nature imprécis et subjectifs. Ainsi l'âge d'une personne qui est précisément décrite par un attribut numérique *âge* dans une base de données peut servir à caractériser des propriétés telles que *jeune* ou *âgé.* Ces deux qualificatifs correspondent à des notions vagues et subjectives modélisées dans SQLf par des ensembles flous, chacun étant caractérisé par une fonction d'appartenance. Cette fonction d'appartenance qualifie par un score normalisé dans l'intervalle [0,1] l'appartenance d'un tuple à ce prédicat.



Ainsi, contrairement à SQL qui repose sur une logique booléenne, un tuple appartiendra de manière graduelle à l'ensemble des réponses. Ce score d'appartenance offre directement une information utile pour le classement qualitatif des réponses vis-à-vis de la requête flexible. Outre, la prise en compte de prédicats flous, SQLf introduit des modificateurs qui expriment des adverbes de la langue naturelle, tels que *très*, *assez*, *peu*, etc..

Limites des modifications déjà réalisées

La première version de PostgreSQL-f souffre de plusieurs défauts qui limitent son développement, sa maintenance et sa distribution. Les fonctionnalités déjà implémentées sont réparties dans de nombreux fichiers qui aboutissent à une version spécifique du serveur PostgreSQL qui se différencie nettement de la version originale. Il devient très difficile de faire évoluer indépendamment les versions de PostgreSQL et son extension au traitement de requêtes flexibles. Par ailleurs, les modifications du module « executor » perturbe le fonctionnement normal de PostgreSQL au point de ne plus pouvoir gérer convenablement l'administration des bases de données (notamment avec l'outil pgadmin).

Vers une externalisation des fonctionnalités

De manière à faciliter la maintenance et la distribution du code, les fonctionnalités propres à SQLf sont ici externalisées au sein d'une extension. PostgreSQL met en effet à disposition des outils pour construire des extensions, appelées PGXS (des exemples sont situés dans le dossier contrib), permettant à de simples extensions d'être construites sur un serveur déjà installé. PGXS peut être destiné aux extensions qui incluent du code C mais aussi à celles composées de code SQL, de procédures PL/PGSQL ou PL/PYTHON.

La création et le chargement de notre extension reposent sur les commandes suivantes :

```
CREATE EXTENSION SQLf ;
LOAD 'sqlf' ;
```

Pour le Lot 1, cette extension définit des prédicats flous, des moyens pour filtrer ces prédicats et des opérateurs (IS, AND, OR et AND).

Prise en compte de prédicats flous (Lot 1)

SQLf reprend la structure du bloc d'interrogation de SQL tout en introduisant de la gradualité partout où elle présente un intérêt. Ainsi le bloc d'interrogation de SQLf est le suivant :

```
select [distinct] [n | t | n, t] attributs from relations where condfloue;
```

La clause **where** peut ainsi contenir des conditions booléennes (*département = 'finance'*) et des conditions floues (âge *is 'jeune'*). Le résultat d'une telle requête n'est plus une relation classique, mais une relation floue où chaque tuple est associé à un degré de satisfaction des conditions spécifiées dans la clause **where**.

Définition de prédicats flous

Un prédicat flou (e.g. âge *is 'jeune'*) peut être représenté comme une fonction trapézoïdale associée à un degré de satisfaction défini comme un nombre flottant. La relation résultat alors doit contenir les tuples pour lesquels le degré de satisfaction MU du prédicat est strictement supérieur à un seuil ALPHA.

En PL/PGSQL , la signature de cette fonction générique est la suivante :

```
CREATE OR REPLACE FUNCTION trapezoidalFuzzyPredicate(val real, support1 real, core1 real, core2 real, support2 real) RETURNS real
```

Ce prédicat générique permet de définir d'autres prédicats flous tels que jeune(âge) ou élevé(salaire). En faisant appel à trapezoidalFuzzyPredicate, la fonction create_predicate(predicate_name TEXT, support1 REAL, core1 REAL, core2 REAL, support2 REAL), simplifie ainsi la création de prédicats flous :

```
SELECT create_predicate('jeune', 0, 0, 20, 30);
```

équivaut à :

```
CREATE OR REPLACE FUNCTION jeune(val REAL) RETURNS REAL AS '
SELECT trapezoidalFuzzyPredicate($1, 0, 0, 20, 30);' LANGUAGE SQL;
```

Ces prédicats flous sont destinés à être employés dans la clause WHERE de requêtes SQL telles que :

```
SELECT * FROM employes WHERE jeune(age)
```

Une telle fonction implémentant un terme flou retourne un degré de satisfaction normalisé dans l'intervalle [0,1]. Ce degré est ensuite attaché à chaque tuple par l'intermédiaire d'un attribut score qui est ajouté si celui-ci n'existe pas encore.

Intégration de prédicats flous dans la clause *WHERE* (lot 1-a)

L'évaluation de la clause WHERE d'une requête SQL s'appuie normalement sur la logique booléenne pour effectuer la sélection des tuples de la relation résultats. Le degré de satisfaction d'un prédicat flou renvoie un nombre réel qui ne correspond pas au booléen habituellement nécessaire pour aboutir à l'évaluation de clause WHERE sans erreur. Afin d'intégrer ces prédicats flous dans la clause WHERE, nous avons envisagé deux stratégies d'évaluation différentes qui reposent sur la conversion entre booléens et nombre flottants.

Prédicats flous : prise en charge native

Pour traiter nativement des nombres flottants dans une clause WHERE, une modification importante de la stratégie d'évaluation est nécessaire. Ces modifications concernent des fonctions natives présentes dans de nombreux fichiers sources C, ce qui limite la maintenance, le développement et la distribution de l'extension SQLf. Par ailleurs, des effets de bords indésirables ont été observés sur le comportement normal de PostgreSQL.

Stratégie de conversion avec un opérateur de CAST(float as boolean)

Un résultat analogue peut être obtenu en utilisant uniquement des opérations définies en PL/PGSQL.

Nous pouvons tout d'abord proposer une conversion de valeurs booléennes vers des nombres flottants :

```
CREATE OR REPLACE FUNCTION bool2fuzzy(BOOLEAN)
RETURNS FLOAT LANGUAGE SQL AS 'SELECT $1::int::FLOAT';
CREATE CAST (BOOLEAN AS FLOAT) WITH FUNCTION bool2fuzzy(FLOAT) AS IMPLICIT;
```

Cette conversion est utilisée par la suite par les opérateurs de conjunction et disjunction OR et AND.

Nous définissons également une procédure pour convertir en booléen le degré de satisfaction MU d'un prédicat flou :

```
CREATE OR REPLACE FUNCTION fuzzy2bool(mu FLOAT) RETURNS boolean LANGUAGE SQL AS 'SELECT set_mu(mu);SELECT mu > get_alpha()';
```

Cette conversion est appliquée implicitement par un opérateur de CAST, juste avant l'évaluation d'un prédicat flou terminal dans la clause WHERE :

```
CREATE CAST (real AS boolean) WITH FUNCTION fuzzy2bool(real) AS IMPLICIT;
```

Pour des besoins ultérieurs, la valeur réelle associée au prédicat MU est stockée dans une variable à l'aide de la fonction set_mu(real).

Affectation et stockage de variables (alpha, k, norme, quantifier)

L'affectation de variables avec set_var(val) dans une clause SELECT n'aurait lieu qu'après la l'exécution de la clause WHERE. Il serait alors nécessaire d'exécuter deux fois la requête pour avoir la bonne valeur à affecter pour la variable. En conséquence, les fonctions set_var(val) doivent être employées dans la clause FROM, comme l'indique ces deux requêtes équivalentes :

```
SELECT set_alpha(0.5) ; SELECT * FROM employes WHERE age ~= 'jeune';
SELECT * FROM employes, set_alpha(0.5) WHERE age ~= 'jeune';
```

Le langage Pl/PGSQL ne propose pas de mécanismes de stockage natif de variables. Lorsque l'on envisage un stockage dans des tables SQL, il faut se confronter à la nécessité de maintenir un jeu de variables propre à chaque session utilisateur. Pour contourner cette difficulté, nous proposons deux solutions de stockage alternatives reposant sur les langages PL/Python et C.

Dictionnaire de variables globales en PL/Python

Une solution simple est d'utiliser un dictionnaire global (GD) défini en pl/python qui offre des mécanismes de stockage de variables pour chaque utilisateur/connexion contrairement à pl/pgsql. Cependant, chaque connexion au SGBD est associée à une session python, ce qui risque de gaspiller la mémoire d'un serveur devant supporter plusieurs milliers de connexions.

Accès et modification de variables en langage C

En vue d'une optimisation, le stockage des variables a été implémenté en C. Deux niveaux de définition sont alors nécessaires pour faire correspondre les fonctions C et leur appel en PG/PLSQL. Par exemple, La fonction set_alpha est définie en langage C (fichier « sqlf.c ») de la manière suivante :

```
Datum set_alpha(PG_FUNCTION_ARGS){
    ALPHA = PG_GETARG_FLOAT8(0);
    PG_RETURN_FLOAT8(ALPHA);
}
PG_FUNCTION_INFO_V1(set_alpha);
```

Cette fonction est appelée par la procédure PL/PGSQL (fichier « sqlf.sql ») correspondante:

```
CREATE OR REPLACE FUNCTION set_alpha(alpha FLOAT) RETURNS FLOAT
AS '$libdir/sqlf', 'set_alpha'
LANGUAGE C STRICT;
```

Opérateurs

La prise en compte de la notation « post-fixée » définie dans SQL_f pourrait reposer une étape de transformation syntaxique définie dans le parseur de requêtes de PostgreSQL. Cependant, les mots clefs et les opérateurs SQL de PostgreSQL sont définis au sein d'une grammaire Bison figée lors de la compilation du code source. La modification de cette grammaire pour contourner l'usage normal des opérateurs mène à des conflits lors de l'analyse syntaxique de requêtes.
Pour des questions de portabilité, nous avons préféré définir ces opérateurs en dehors de la grammaire Bison à l'aide de la commande CREATE OPERATOR. Le nom de ces opérateurs est restreint à une séquence de caractères spéciaux (`+ - * / < > = ~ ! @ # % ^ & | ` ?`). Du fait de cette restriction, nous avons choisi d'employer les symboles « ~= », « && », « || » et « ! » pour représenter les opérateurs IS, AND, OR et NOT.

Prédicats flous et modifieurs : notation post-fixée avec IS

L'opérateur SQL ~= emploie une notation post-fixée pour matérialisé par l'opérateur IS :

```
SELECT * FROM employe WHERE age ~= 'jeune'
```

L'opérateur =~ est également capable de prendre en compte l'emploi de modifieurs tels que 'très' :

```
SELECT * FROM employe WHERE age is 'tres jeune'
```

La fonction ISf définit =~ avec deux arguments :

```
CREATE OPERATOR ~= (LEFTARG = REAL, RIGHTARG = VARCHAR, PROCEDURE =
isF);
FUNCTION ISf(att REAL, funcs VARCHAR) RETURNS REAL
```

Dans la fonction ISf, chaque token de la chaîne de caractères est évaluée comme une procédure PL/PLSQL à l'aide de l'instruction EXECUTE (analogue à EVAL dans d'autres langages de programmation). Cette procédure effectue les conversions suivantes :

```
age ~= 'jeune' -> ISf(age, 'jeune') -> jeune(age)
age ~= 'tres jeune' -> ISf(age, 'tres jeune') -> tres(jeune(age))
```

Conjonctions et disjonctions avec AND et OR

Les scores retournés par chaque condition floue ou booléenne sont combinés suivant l'expression fournie dans la clause **where**. Notons que d'un point de vue théorique, la sémantique des opérateurs **and** et **or** sont définis dans la logique floue comme des normes et co-normes. Comme le montre la requête suivante, nous les avons fait correspondre aux opérateurs SQL && et ||, utilisables comme alternatives aux AND et OR classiques :

```
SELECT * FROM employes WHERE age ~= 'jeune' && salaire ~= 'eleve';
```

L'emploi d'éléments de type boolean avec && et || engendre une conversion implicite vers des nombres flottants vers des booléen à l'aide d'un opérateur de CAST qui appelle la procédure bool2-fuzzy(BOOLEAN) .

```
SELECT * FROM employes WHERE age ~= 'jeune' && salaire > 25000;
```

Par ailleurs, les opérateurs && et || sont associés aux fonctions C fuzzy_and_operator et fuzzy_or_operator qui réalisent le calcul de la norme/co-norme.

Paramétrage de la norme

Les fonctions C fuzzy_and_operator et fuzzy_or_operator effectuent le calcul des normes/co-normes selon la valeur de la variable C « char * norm ». Elle peut prendre les valeurs 'zadeh', 'probabiliste', 'lukasiewicz' et 'weber' et peut être paramétrée à l'aide de set_norm(n) dans la clause FROM d'une requête SQL :

```
SELECT * FROM set_norm('probabiliste'), employes
WHERE fuzzy_and_operator(age ~= 'jeune', salaire ~= 'eleve');
```

De manière alternative, on aussi peut écrire :

```
SELECT set_norm('probabiliste') ;
SELECT * FROM employes WHERE fuzzy_and_operator(age ~= 'jeune',
salaire ~= 'eleve');
```

Le calcul des différentes normes est implémenté en langage C, comme par exemple pour la disjonction probabiliste :

```
float8 or_probabiliste(float8 a, float8 b){
    return a + b - a * b;
}
```

Priorité des opérateurs

L'utilisation d'expressions contenant plusieurs opérateurs (~=, &&) soulève des problèmes au niveau de l'évaluation d'expressions telles que :

```
age ~= 'jeune' && salaire ~= 'elevé'
```

qui est évaluée comme

```
elevé(fuzzy_and_operator(jeune(age), salaire))
```

Ce comportement peut être évité en ajoutant des parenthèses pour délimiter les sous-expressions :

```
(age ~= 'jeune') && (salaire ~= 'elevé')
```

L'instruction CREATE OPERATOR ne permet pas de définir des règles de priorité pour spécifier la précédence des opérateurs : le comportement de la précédence est codé en dur dans l'analyseur syntaxique.

It is not possible to specify an operator's lexical precedence in CREATE OPERATOR, because the parser's precedence behavior is hard-wired.

http://www.postgresql.org/docs/current/static/sql-createoperator.htm

Filtrage de prédicats flous

La clause FROM peut être complétée par des paramètres indiquant la quantité de résultats attendus (k) et/ou la qualité, c'est-à-dire le degré de satisfaction minimal attendu (alpha). Voici un exemple d'une telle requête :

```
SELECT age, mu FROM set_k(4), set_alpha(0.1), employes
WHERE age ~= 'jeune' && salaire ~= 'eleve';
```

Filtrage « alpha cut »

Dans la définition précédente de la fonction fuzzy2bool(float), un filtrage « alpha cut » est effectué pour ne conserver que les prédicats ayant un degré de satisfaction MU > ALPHA. On stocke à cette occasion la valeur réelle attachée au prédicat flou dans la variable mu. La valeur de mu est alors accessible dans la clause SELECT à l'aide de la fonction get_mu() :

```
SELECT *, get_mu() as mu FROM employes;
```

Cette vue doit être définie pour chaque table.

Filtrage du nombre de résultats

Nous rencontrons actuellement des difficultés pour limiter le nombre de résultats d'une requête en fonction de la variable globale K. L'utilisation d'une vue pour encapsuler l'instruction `LIMIT get_k()` engendre des effets de bord sur la valeur de la variable MU. Pour le moment, le nombre de résultats d'une requête est réduit par l'employer des instructions `ORDER BY get_mu() LIMIT get_k()` dans la requête elle-même :

```
SELECT set_k(4), set_alpha(0.1);
SELECT age, get_mu() as mu FROM, employes
WHERE age ~= 'jeune' && salaire ~= 'eleve'
ORDER BY get_mu() LIMIT get_k();
```

Quantificateurs flous (Lot 2)

Les quantificateurs flous sont un moyen de spécifier des conditions floues. Une proposition quantifiée floue peut être déclarée dans une requête à l'aide de l'opérateur « la_plupart » :

```
SELECT * FROM employes
WHERE la_plupart(age = 'jeune', dept = 'finance',
                 salaire ~= 'élevé', prime ~= 'moyenne',
                 ancienne ~= 'importante');
```

Parmi les différentes interprétations possibles de ces quantificateurs, nous avons implémenté les méthodes de calcul Zadeh, Yager-CTA et Yager-OWA. Le choix de la méthode de calcul est paramétrable dynamiquement à l'aide de la fonction set_quantifier(q) dans clause FROM (1) de la requête ou dans la clause SELECT d'une requête séparée (2).

```
        1. SELECT * FROM set_quantifier('owa'), employes
WHERE proposition_quantifiee;
        2. SELECT set_quantifier('cta');
SELECT * FROM employes WHERE proposition_quantifiee;
```

La méthode de calcul choisie pour « la_plupart » repose sur la valeur de la variable C QUANTIFIER qui est accessible avec la fonction get_quantifier(). Pour mieux comprendre, l'utilisation de propositions quantifiées floues est illustrée par un exemple.

Exemple

Considérons la proposition quantifiée floue suivante ;

```
la_plupart(age is jeune, dept = 'finance',
           salaire is élevé, prime is moyenne,
           ancienne is importante);
```

Soit les tuples avec leurs degré de satisfaction des 3 prédicats J (jeune), F (finance), E (élevé), M (moyenne) et I (importante)

t1 : J=0.5 ; F=1 ; E=0.2 ; M=0.4 ; I=0.6
t2 : J=0.1 ; F=1 ; E=0.5 ; M=0 ; I=0.1
t3 : J=0 ; F=0 ; E=0.9 ; M=1 ; I=0
t4 : J=0.2 ; F=0 ; E=1 ; M=0.1 ; I=0.5

Soit la définition du quantificateur relatif la_plupart x → x*x :

```
CREATE OR REPLACE FUNCTION most_of(x FLOAT) RETURNS FLOAT AS $$
SELECT $1 * $1;
$$ LANGUAGE SQL;
```

ZADEH

Pour le quantificateur Zadeh, il faut faire la moyenne des degrés de satisfaction puis interpréter ce résultat par le quantificateur 'most_of' :

μ(most t1) = most((0.5 + 1 + 0.2 + 0.4 + 0.6) / 5) = most(0,54) = 0,2916
μ(most t2) = most((0.1 + 1 + 0.5 + 0 + 0.1) / 5) = most(0,34) = 0,1156
….

Ce calcul est effectué par la fonction `pl/pgsql` zadeh(VARIADIC xs FLOAT[]).

YAGER CTA

Pour le quantificateur Yager-CTA, on utilise la formule suivante :

μ(X) = sup_{1 <= i <= n} min(μ_q(i/n),μ_A(xi))

Où :

- μ(X) est le degré à calculer ;
- n est le nombre de prédicats à agréger ;
- mu_q(i/n) est l'interprétation de i par la définition du quantificateur (on suppose ici que i est un quantificateur relatif) ;
- μ_A(xi) est le degré de satisfaction du prédicat.

Pour réaliser ce calcul, la fonction pl/python cta(VARIADIC X FLOAT[]) réalise tout d'abord un ordonnancement des degrés de satisfaction μ(xi) par ordre décroissant. Par exemple pour t1, on a 1 >= 0.6 >= 0.5 >= 0.4 >= 0.2. On applique ensuite la formule ci-dessus pour calculer le degré μ(X).

Ainsi, pour t1 on obtient :

sup(min(muQ(1/5), 1), min(muQ(2/5), 0.6), min(muQ(3/5), 0.5) min(muQ(4/5), 0.4),
min(muQ(5/5), 0.2))
= sup(min(0.04, 1), min(0.16, 0.6), min(muQ(0.36, 0.5) min(muQ(0.64), 0.4), min(muQ(1), 0.2))
= 0.4

Pour t2 on obtient :

sup(min(muQ(1/5), 1), min(muQ(2/5), 0.5), min(muQ(3/5), 0.1) min(muQ(4/5), 0.1),
min(muQ(5/5), 0))
= sup(min(0.04, 1), min(muQ(0.16), 0.5), min(muQ(0.36), 0.1) min(muQ(0.64), 0.1), min(muQ(1),
= 0.16

YAGER OWA

Un opérateur OWA est défini de la manière suivante

OWA(x1…xn ; w1…wn) = \sum_{1 <= i <= n} ^ w_i * x_ki

Où :

- x1 .. xn sont les degrés de satisfactions des prédicats à agréger ;
- les x_ki sont les degrés de satisfactions des prédicats classés par ordre décroissant ;
- w1 .. wn sont les poids d'importance ;

Le quantificateur Yager-OWA est défini par la fonction PL/PYTHON owa(VARIADIC xs FLOAT[]) RETURNS FLOAT. Cette fonction effectue tout d'abord le calcul des poids wi à partir de la définition du quantificateur utilisé :

wi = μ_Q(i) – μ_Q(i-1) pour les quantificateurs absolus et
wi = μ_Q(i/n) – μ_Q((i-1)/n) pour les quantificateurs relatifs

Donc dans notre cas :

w1 = μ_Q (1/5) = 0.04
w2 = μ_Q (2/5) – μ_Q (1/5) = 0,16 – 0,04 = 0,12
w3 = μ_Q (3/5) – μ_Q (2/5) = 0,36 – 0,16 = 0,2
w4 = μ_Q (4/5) – μ_Q (3/5) = 0,64 – 0,36 = 0,28
w5 = μ_Q (1) – μ_Q (4/5) = 1 - 0,64 = 0,36

La fonction classe ensuite les degrés de satisfaction μ(xi) par ordre décroissant (x_ki). Par exemple pour t1, on a 1 >= 0.6 >= 0.5 >= 0.4 >= 0.2. En appliquant la formule ci-dessus, on obtient alors :

$$1*0.04 + 0.6 * 0.12 + 0.5*0.2 + 0.4*0.28 + 0.2*0.36 = 0.04 + 0.072 + 0.1 + 0.112 + 0.072 = 0,396$$

Pour t2 on obtient :

$$1*0.04 + 0.5 * 0.12 + 0.1*0.2 + 0.1*0.28 + 0.0*0.36 = 0.04 + 0.06 + 0.02 +0.028= 0.148$$

Le quantificateur doit correspondre à une fonction croissante. Cependant, la prise en compte de fonctions non monotones devrait être envisageable dans définition du quantificateur.

Opérateurs graduels (Lot 3)

En SQL, l'opérateur IN est utilisé comme une fonction d'appartenance. En SQLf, une fonction d'appartenance floue peut se définir à l'aide d'opérateurs graduels (~ et in~) associés à des fonctions de distance μ_~(a,b) entre des attributs a et b.

```
SELECT * FROM employe WHERE age ~ 50;
SELECT * FROM employe WHERE departement in~ ('accounting', 'finance',
'R&D');
```

On suppose que le domaine de définition de a et b est connu. Par exemple, pour le domaine 'département', on peut définir la distance μ('comptabilité', 'marketing') comme étant plus petite que μ_~('comptabilité', 'r&d'). Bien que le calcul de ces distances soit déterminé par le nom des champs manipulés par l'opérateur, PostgreSQL ne donne pas le moyen de connaitre les noms des colonnes associés à ces champs.

Distances et types associés à des domaines de définition

Pour contourner ce problème, nous associons des types qui représentent le domaine de définition des attributs pour lesquels on cherche à mesurer une distance. Chaque attribut pour lequel une distance est définie est alors encapsulé dans un type spécifique pour lequel on considère également un score mu :

```
CREATE TYPE age AS (value float, mu float);
```

Pour des besoins ultérieurs (requêtes imbriquées), la création d'éléments de ce type prend compte de la valeur du degré de satisfaction get_mu() dans la propriété mu.

```
CREATE OR REPLACE FUNCTION age(float) RETURNS age AS
'SELECT ROW($1,get_mu())::age;'
LANGUAGE SQL IMMUTABLE;
```

Un opérateur de CAST est alors employé pour simplifier la conversion avec l'opérateur « :: » :

```
CREATE CAST (FLOAT AS age) WITH FUNCTION age(float);
SELECT age::age FROM employes;
```

Nous pouvons ensuite associer chaque types à une fonction distance spécifiques. On définit une fonction distance s'appliquant par défaut aux éléments de type FLOAT:

```
distance(a,b)=1 - min(a,b)/max(a/b)
```

Pour le type 'age', la distance peut être représentée comme une fonction trapézoïdale :

```
CREATE OR REPLACE FUNCTION distance(age, age)RETURNS FLOAT AS
'SELECT trapezoidalFuzzyPredicate($1.value, $2.value - 50.0,
$2.value, $2.value, $2.value + 50.0);' LANGUAGE SQL;
```

Le calcul de distance(age ::age, 50 ::age) donne alors :

μ=1 si age == 50 ; μ= 0.9 si age == 45 ou age == 55 ; μ=0.8 si age == 40 ou age == 60...

Opérateur graduel ~

On peut faire correspondre chaque fonction « distance » définie pour un type à un opérateur graduel ~ spécifique :

```
CREATE OPERATOR ~ (LEFTARG = age,
                   RIGHTARG = age,
                   PROCEDURE = distance);
```

Le calcul de la distance sur le domaine 'age' peut alors être effectué dans une requête :

```
SELECT * FROM employes WHERE age::age ~ 50::age;
```

Opérateur IN~

En SQLf, l'opérateur IN~ est calculé comme la distance maximale en un attribut a et les éléments d'un ensemble E :

```
μ_in(a,E)= sup_{b \in support(E)} μ_~(a,b)
```

L'opérateur SQLf IN~, représenté par le symbole SQL <~, fait appel à la fonction mu_in qui réalise le calcul :

```
CREATE OPERATOR <~ (LEFTARG = anyelement, RIGHTARG = anyarray, PRO-
CEDURE = mu_in);
```

On peut ensuite exécuter des requêtes du style :

```
SELECT * FROM employe WHERE department::departement <~  ARRAY['ac-
counting', 'finance', 'R&D']::department[];
SELECT * FROM v_employes WHERE age::age <~  ARRAY[25.0, 35.0]::age[];
```

Requêtes imbriquées (Lot 4)

En SQL, l'opérateur *IN* permet de tester l'appartenance d'une valeur dans une relation résultant de l'exécution d'une requête imbriquée. En SQLf , l'appartenance d'une valeur à une relation floue (requête imbriquée flexible) est calculée par l'opérateur IN_f de la manière suivante :

```
mu_in_f(a,E)= sup_{b \in support(E)} (min(distance(a,b),µ_E(b)))
```

Cette mesure calcule la distance maximale entre l'élément att et les éléments de la requête flexible imbriquée E en fonction du degré de satisfaction µ_E(b) des tuples b de E.

L'opérateur IN_f, représenté par le symbole SQL **<~~**, est associé à la fonction mu_in_f pour l'exécution de requêtes du style :

```
SELECT * FROM table WHERE att::typeX <~~ requête_imbriquée;
```

A notre connaissance, PL/PGSQL ne permet pas de définir une fonction d'appartenance mu_in_f ayant une requête imbriquée comme argument. Pour le moment, cette requête est stockée dans un tableau :

```
ARRAY(SELECT age::age FROM employes WHERE age ~= 'jeune')
```

Les éléments de ce tableau sont exploités au sein de la fonction PL/PGSQL mu_in_f (att ANYNONARRAY, sqlf_query ANYARRAY) comme l'indique ces deux requêtes équivalentes :

```
SELECT * FROM employes WHERE mu_in_f(age::age, ARRAY(select age::age
from employes where age ~= 'jeune'));

SELECT * FROM employes WHERE age::age <~~ ARRAY(SELECT age::age FROM
employes WHERE age ~= 'jeune');
```

Il est important de noter que le type spécifié pour l'attribut 'att' doit être le même que celui des éléments de la requête imbriquée sqlf_query. L'intérêt de spécifier ce type est double. D'une part, il joue un rôle dans le choix de la fonction de distance à employer comme dans le cas des opérateurs graduels. D'autre part, ce type permet de prendre en compte le degré de satisfaction mu des tuples de la requête imbriquée flexible sqlf_query. En effet, pour chaque tuple examiné dans dans une requête imbriquée, la création d'éléments spécifiques à un domaine (e.g. type age) engendre le stockage du degré de satisfaction mu d'un prédicat flou.

Conclusion

L'extension SQLf définit un cadre général pour définir et manipuler des requêtes flexibles au sein du SGBDR PostgreSQL. Ces requêtes présentent des conditions décrites par des prédicats flous caractérisés par leur degré de satisfaction. Sa prise en compte implique une adaptation du plan d'exécution des requêtes.

Fonctionnalités

Les résultats d'une requête flexible reposent sur le calcul de prédicats flous assemblés à l'aide d'opérateurs et fonctions paramétrables. Différentes normes et co-norme peuvent ainsi définir les opéra-

teurs de conjonction/disjonction (&&, ||). Plusieurs choix (Zadeh, CTA-Yager, OWA-Yager) sont également possibles pour le calcul de propositions quantifiées. Par ailleurs, les opérateurs graduels (~, IN~) introduisent des mesures de distance adaptées aux domaines de définition des données. Des mécanismes sont mis en place pour mesurer le degré d'appartenance (IN_f) opérant sur des requêtes imbriquées. Enfin, les résultats issus de ces requêtes peuvent être filtrés quantitativement (k) et qualitativement (alpha).

Limitations et améliorations envisageables

Syntaxe :

- Le nom des opérateurs est limité à des séquences de caractères spéciaux (~=><!&| ...) qui ne correspondent pas aux noms demandés dans le cahier des charges.
- Les noms des prédicats doivent être délimités par des *single quotes* car les noms de fonctions ne peuvent pas être des arguments.
- Pour le moment, on ne peut pas définir de prédicats homonymes (height is low, mileage is low).
- En pl/pgsql, l'affectation avec le mot clé SET ne semble pas adaptée à nos usages. La définition de de variables GUC nécessiterai de maintenir des jeux de variables spécifiques à des sessions utilisateurs.

Affectation des variables :

- L'affectation des variables doit se faire dans une clause FROM plutôt que dans la clause SELECT car la clause SELECT est évaluée après la clause FROM.

Filtrage avec alpha et k :

Une vue « flexible » doit être définie sur chaque table pour spécifier les clauses ORDER BY get_mu(). L'instruction LIMIT get_k() ne fonctionne pas dans une vue : il faut la spécifier à la fin de la requête générale. Pour contourner ce problème, deux alternatives sont envisagées. La première emploie une méthode de classement ('`RANK() OVER(ORDER BY mu) as sqlf_rank`'). La seconde fait appel à l'instruction C '`SPI_exec(query, K)`' qui permet de limiter le nombre de résultats retournés par une requête. Ces deux techniques de filtrage n'ont pas encore été testées.

Opérateurs graduel et requêtes imbriquées :

- Des types spécifiques à chaque domaine doivent être définis pour garantir un choix de distance approprié. Des instructions SPI utilisées notamment dans le module PGXN colname (http://api.pgxn.org/src/colnames/) pourraient être utiles pour connaitre le nom des colonnes à manipuler.
- Une requête imbriquée doit être décrite par une chaîne de caractères.

Implémentations et documentations manquantes :

- Moyenne géométrique, harmonique avec pondération éventuelle.

- Ré-implémentation de certaines fonctions PL/Python et PL/PGSQL en C.
- Manque de références pour compléter cette documentation.