Université Aboubakr Belkaid – Tlemcen

École doctorale STIC option SIC

Mémoire de magister en informatique

Thème

Étude de la relaxation de requêtes dans un contexte flexible

par Amine BRIKCI-NIGASSA

a_brikci@mail.univ-tlemcen.dz

Soutenu devant le jury :

Président : Pr Mohammed BENABDELLAH, Faculté de Technologie, Université de Tlemcen.

Directeur du Mémoire à l'université de Tlemcen : Dr Mohammed-El-Amine CHIKH, Maître de Conférences A, Faculté de Technologie, Université de Tlemcen.

Directeur du Mémoire à l'étranger : Dr Allel HADJALI, Maître de Conférences, Université de Rennes 1 (France).

Examinateur : Dr Fedoua DIDI, Maître de Conférences A, Faculté des Sciences, Université de Tlemcen.

Invité : Dr Abdelkarim BENAMMAR, Maître de Conférences B, Faculté des Sciences, Université de Tlemcen.

Remerciements

Le présent travail n'aurait pu voir le jour sans l'aide précieuse de M. Allel HADJALI, Maître de Conférences à l'université de Rennes 1 (France), que je remercie pour sa patience, sa disponibilité et les nombreux conseils qu'il n'a pas hésité à me prodiguer.

Je remercie également Dr Mohammed-El-Amine CHIKH, Maître de Conférences à la faculté de Technologie de l'université de Tlemcen, pour son aide et pour m'avoir soutenu et encouragé tout au long de la réalisation de ce projet.

Que Pr Mohammed BENABDELLAH, Maître de Conférences à la faculté de Technologie de l'université de Tlemcen, trouve aussi l'expression de mon entière reconnaissance pour l'honneur qu'il me fait d'accepter de présider le jury.

Que Dr Fedoua DIDI, Maître de Conférences à la faculté des Sciences de l'université de Tlemcen, qui n'a pas hésité à examiner ce travail dans des délais si rapides, accepte l'expression de ma profonde reconnaissance.

Que Dr Abdelkarim BENAMMAR, Maître de Conférences à la faculté des Sciences de l'université de Tlemcen, trouve ici toute ma gratitude d'avoir bien voulu faire partie du jury malgré ses nombreuses occupations.

Je tiens de même à remercier mon ami Djallal MIDOUNI, Maître Assistant au département d'informatique de la faculté des Sciences de Tlemcen, pour son aide précieuse et son soutien.

Je ne peux oublier d'exprimer également ma gratitude aux membres de ma famille pour leur patience et pour m'avoir soutenu durant toutes ces épreuves.

Amine BRIKCI-NIGASSA

Résumé

L'une des priorités lors de la conception de systèmes de gestion de bases de données est de rendre la conversation avec la machine plus abordable pour l'utilisateur profane. L'un des problèmes majeurs qu'il peut rencontrer est celui des réponses vides, dont le traitement s'inscrit dans un domaine qui étudie les subtilités des échanges conversationnels : celui des réponses *coopératives*.

L'accès aux données distantes étant souvent incomplet, il semble pertinent de traiter ce problème en agissant sur la requête, en atténuant l'exigence de ses critères. Sa *relaxation* pourra permettre de récupérer des réponses qui bien que proches de ces conditions avaient été écartées car elles ne les satisfont pas strictement, le but étant d'obtenir un ensemble non-vide de réponses.

Notre travail s'inscrit dans le cadre de l'étude des requêtes flexibles, à prédicats graduels et représentés par des ensembles flous. Les tuples sélectionnés ne satisfont pas obligatoirement *pleinement* les conditions comme avec les requêtes classiques, et sont ordonnés selon leur *degré de satisfaction* à ces conditions, ce qui revient à les discriminer selon les *préférences* de l'utilisateur.

L'aspect coopératif est introduit par une technique de relaxation basée sur la *proximité relative*. Les éléments qui ne satisfont pas (du tout) la condition d'un prédicat mais sont dans un *voisinage* proche pourront être récoltés grâce à cette *relation de tolérance*. Ainsi, les prédicats constituant la requête infructueuse sont transformés un par un de manière itérative afin d'aboutir à une requête qui renvoie un ensemble non vide de réponses sans trop s'éloigner de la requête initiale.

Les combinaisons de transformations successives constituent un treillis de requêtes relaxées. Afin d'optimiser son parcours, ce treillis pourra être élagué en utilisant la notion des MFS *(Minimally Failing Subqueries)*. La meilleure des requêtes à réponse non vide obtenues après ce parcours sera ensuite choisie grâce à la mesure de la *distance de Hausdorff*, employée pour comparer ses prédicats flous avec ceux de la requête initiale.

La mise en œuvre de l'approche présentée a été effectuée à travers un prototype écrit en Java. Son interface permet de tester des requêtes flexibles et de réaliser des expérimentations pour estimer l'efficacité de l'approche étudiée sur des bases de données du monde réel.

Mots-clés : Bases de données, requêtes flexibles, ensembles flous, requêtes coopératives, relaxation de requêtes.

Université Aboubakr Belkaid – Tlemcen

École doctorale STIC option SIC

Mémoire de magister en informatique

Thème

Étude de la relaxation de requêtes dans un contexte flexible

par Amine BRIKCI-NIGASSA

a_brikci@mail.univ-tlemcen.dz

Soutenu devant le jury :

Président : Pr Mohammed BENABDELLAH, Faculté de Technologie, Université de Tlemcen.

Directeur du Mémoire à l'université de Tlemcen : Dr Mohammed-El-Amine CHIKH, Maître de Conférences A, Faculté de Technologie, Université de Tlemcen.

Directeur du Mémoire à l'étranger : Dr Allel HADJALI, Maître de Conférences, Université de Rennes 1 (France).

Examinateur : Dr Fedoua DIDI, Maître de Conférences A, Faculté des Sciences, Université de Tlemcen.

Invité : Dr Abdelkarim BENAMMAR, Maître de Conférences B, Faculté des Sciences, Université de Tlemcen.

Table des matières

Introduction

En constante expansion, l'Internet donne accès à un nombre important de bases de données, variées par leur nature, leur contenu et la manière dont elles sont implémentées. Qu'elles soient à visée scientifique, bibliographiques, pour la réservation des billets de transport ou qu'elles contiennent des données financières, leur mise en œuvre peut être *complexe* et employer les dernières avancées en matière d'informatique distribuée, mais elle est de plus en plus *transparente* pour les utilisateurs.

Les profils de ces derniers évoluent également : avec la démocratisation de l'informatique et de l'accès aux réseaux, actuellement beaucoup d'utilisateurs profanes interrogent ces bases de données. C'est pourquoi l'une des priorités lors de la conception de systèmes de gestion de bases de données est de rendre la conversation avec la machine plus abordable pour l'utilisateur.

Il faut pour cela considérer les obstacles qu'il peut rencontrer. L'un des problèmes majeurs est celui des réponses vides : la requête posée retourne un ensemble vide de réponses, ce qui provoque souvent une frustration, d'autant plus que la raison de cet échec n'est pas toujours facile à déterminer. Le traitement des réponses vides s'inscrit dans un domaine qui étudie les subtilités des échanges conversationnels : celui des réponses *coopératives*.

Parmi les raisons pour lesquelles l'exploitation des sources d'informations à travers le réseau mondial n'est pas triviale et induit quelques difficultés, citons le fait que l'utilisateur a seulement un accès *indirect* aux données. En effet, on ne peut parcourir toute la base de données cible, mais uniquement les tuples qui satisfont les requêtes soumises au système. Partant de ce constat, une solution qui nous semble pertinente au problème des réponses vides est d'agir sur la requête elle-même. On modifiera pour cela les conditions qui y sont contenues pour atténuer son exigence. Cette *relaxation* de la requête pourra permettre de récupérer des réponses qui se rapprochent de ces conditions mais avaient été auparavant

écartées car elles ne les satisfont pas strictement, le but étant d'obtenir un ensemble non-vide de réponses.

Notre travail s'inscrit dans le cadre de l'étude des requêtes flexibles. Ces requêtes contiennent des prédicats graduels, représentés par des ensembles flous. Ainsi, les tuples de la base de données interrogée n'ont pas besoin de satisfaire *pleinement* les conditions pour être sélectionnés comme avec les requêtes ordinaires (à prédicats booléens). Les résultats obtenus sont ordonnés selon le *degré de satisfaction* aux conditions de la requête, ce qui revient à les discriminer selon les *préférences* de l'utilisateur.

C'est dans ce contexte que l'on s'efforcera d'introduire l'aspect coopératif par l'étude d'une technique de relaxation basée sur la notion de *proximité relative*. En effet, les éléments qui ne satisfont pas (du tout) la condition d'un prédicat mais sont dans un *voisinage* proche pourront être récoltés grâce à cette *relation de tolérance*. Ainsi, les prédicats constituant la requête infructueuse sont transformés un par un de manière itérative afin d'aboutir à une requête qui renvoie un ensemble non vide de réponses sans trop s'éloigner de la requête initiale.

Les combinaisons de transformations successives constituent un treillis de requêtes relaxées, qu'il faudra parcourir à la recherche de solutions satisfaisantes. Afin d'optimiser son parcours, le treillis pourra être élagué en utilisant la notion des MFS *(Minimally Failing Subqueries)*, c'est-à-dire des sous-requêtes minimales qui échouent de la requête initiale.

Une distance sémantique devra ensuite être évaluée afin de ne garder que la meilleure des requêtes à réponse non vide obtenues à la suite du parcours du treillis, c'est-à-dire la plus proche sémantiquement parlant de la requête initiale. Parmi les différentes possibilités, nous avons choisi la *distance de Hausdorff*, connue notamment dans le domaine du traitement d'images où elle est utilisée pour

mesurer la ressemblance de courbes ou de formes quelconques, et qui est souvent employée pour la comparaison d'ensembles flous.

Notre travail s'articule en cinq chapitres :

- Le *chapitre 1* présente une classification des différents aspects de réponses coopératives, avec les techniques de résolution du problème des réponses vides dans le cadre des requêtes booléennes ordinaires.
- Dans le *chapitre 2*, nous introduisons le paradigme des requêtes flexibles, en rappelant d'abord la notion d'ensemble flou imaginée par L. Zadeh au milieu des années 1960, ainsi que leur application aux langages d'interrogation de bases de données relationnelles.
- Au *chapitre 3,* nous aborderons les approches traitant les réponses vides dans le cadre des requêtes flexibles. Les techniques de relaxation utilisées dans notre travail y sont présentées. Pour cela, la notion de relation de tolérance y sera d'abord introduite.
- Dans le *chapitre 4*, nous proposons une mesure de proximité sémantique entre les requêtes basée sur la distance de Hausdorff afin de déterminer la meilleure relaxation.
- Le *chapitre 5* décrit la mise en œuvre de notre travail et son expérimentation sur une base de données test.

Chapitre 1 :

Réponses coopératives

Il y a quelques années, l'évolution des systèmes de gestion de bases de données semblait guidée par la recherche sur les nouveaux paradigmes. Aujourd'hui, les bases de données objet sont encore à l'étude et le modèle relationnel inventé par Codd en 1970 n'est pas près de disparaître [Codd 70]. Au lieu de cela, de nombreux axes de recherche sur les bases de données actuelles sont orientés vers les moyens permettant de les rendre plus intelligentes, la priorité étant souvent la satisfaction de l'utilisateur.

L'intelligence artificielle essaie de reproduire le comportement humain afin que l'utilisateur bénéficie des avantages qui le caractérisent. Dans le domaine des bases de données, on s'intéresse notamment au caractère coopératif de l'interlocuteur, qui fait défaut dans les systèmes actuels, pour lesquels une réponse correcte doit s'en tenir strictement à ce qui est demandé par la requête. Ce n'est pas le cas pour l'être humain qui, par exemple, à la question « Avez-vous l'heure ? » ne se contentera pas de répondre « Oui » mais donnera l'heure !

1.1 Les maximes conversationnelles

La branche de la linguistique appelée *pragmatique* s'intéresse aux éléments du langage dont la signification ne peut être comprise qu'en connaissant le contexte.

Les critères utilisés par certains auteurs pour guider leurs recherches sur les réponses coopératives ont été établis par le linguiste philosophe américain H.P. Grice. Les théories de la pragmatique s'appuient sur la distinction qu'il fait entre le *sens pour le locuteur* et le sens proprement linguistique. Les notions de *présupposition*, *d'inférence*, de sens *non-dit* ou *implicite* doivent obligatoirement être considérées dans une conversation coopérative.

Grice introduit dans son étude de la conversation [Grice 75] un principe dictant les propriétés fondamentales du comportement coopératif qui font qu'une

réponse[1] est *correcte, non-déroutante* et *utile*. Il décompose ce *principe de coopération* (*cooperative principle*[2]) en *maximes* :

- **La maxime de qualité** nécessite de s'assurer de la *validité* d'une réponse avant de la donner. Plus particulièrement, on fera attention à ne pas sous-entendre une information que l'on sait fausse. Il peut y avoir un risque que l'utilisateur soit *dérouté* en déduisant une fausse assomption de la réponse à sa requête. Si c'est le cas, le système devra éliminer ce risque en dotant sa réponse de précisions complémentaires. [Kaplan 82]

- **La maxime de quantité** dicte qu'une réponse doit être aussi *instructive* que requis, pas plus. Elle ne devra pas être plus détaillée qu'il n'est nécessaire. Cela est particulièrement important avec les grandes bases de données, dont la taille atteint parfois des proportions astronomiques. La résolution par les systèmes coopératifs du problème des *réponses pléthoriques* est un axe de recherche d'actualité [Bosc *et al.* 08].

- **La maxime de pertinence** explique qu'une réponse doit être pertinente, en particulier pour l'utilisateur. Cela dépendra donc des buts et intentions de celui-ci, qui peuvent par exemple être déduits de son profil ou modèle.

- **La maxime de manière** stipule que l'on doit éviter les réponses obtuses et les ambiguïtés, les significations multiples étant source de mauvaises interprétations. La réponse devra donc être brève, succincte et précise.

1 Nous appliquons ses maximes aux *réponses* mais Grice parle en fait de toute *contribution* à une conversation

2 *« Make your conversational contribution such as is required, at the stage at which it occurs, by the accepted purpose or direction of the talk exchange in which you are engaged »* [Grice, 1975]

1.2 Classification des techniques coopératives

Dans son passage en revue des systèmes de bases de données coopératifs, A. Motro suggère une classification des techniques basée sur les étapes du processus itératif selon lequel se déroule typiquement l'interaction d'un utilisateur avec un SGBD (*figure 1.1*) [Motro 00].

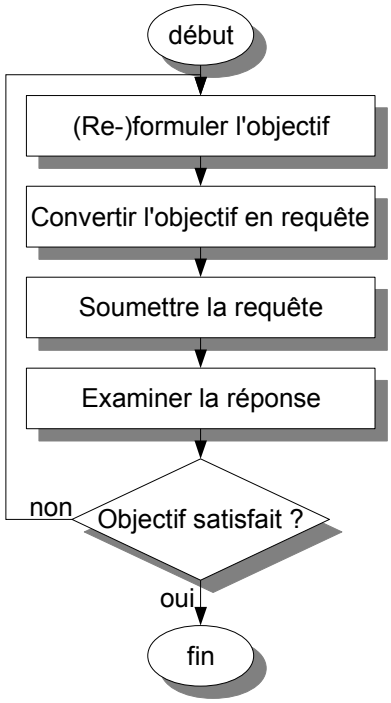


Figure 1.1 : Processus itératif de l'interaction avec le SGBD (d'après [Motro 00])

1.2.1 Formulation des objectifs

La coopération à cette étape permet essentiellement d'aider l'utilisateur à comprendre ses besoins. Certains outils très sophistiqués permettent de parcourir les données (*browsers*) en suivant par exemple les liens entre les différentes relations. Ainsi l'utilisateur, en examinant le contenu concret, peut mieux cerner la sémantique de la base de données et la formulation de ses besoins lui sera plus facile.

Pourtant, n'ayant qu'un comportement passif, ces techniques ne peuvent prétendre être coopératives. Cependant, certaines d'entre elles font exception : celle qui consiste à enregistrer les activités de navigation qui se répètent pour en déduire des requêtes donnant les mêmes résultats plus efficacement [D'Atri 89] ou bien celle du système « *query-by-browsing* » qui propose à l'utilisateur d'évaluer la pertinence des tuples qu'il lui présente afin d'en inférer le motif correspondant à l'objectif [Dix 95].

1.2.2 Conversion de l'objectif en requête

À ce niveau, l'intérêt recherché est principalement de guider l'utilisateur novice pour lequel le formalisme du langage de requête employé est trop complexe, grâce à un outil de *construction interactive de requête*. Ce sera typiquement une interface graphique (un *assistant* ou *wizard)* qui découpera cette construction en petites étapes.

Motro [Motro 00] a écarté de son étude les interfaces de *query-by-example* (*QBE*, requêtes par l'exemple) mais il semble bien que c'est dans cette catégorie que l'on peut citer les outils de construction de requêtes « visuels » de logiciels grand public comme « Microsoft Access » ou ses équivalents libres « LibreOffice Base » et « Kexi ». Ces outils permettent de construire, par de simples clics, des requêtes plus ou moins complexes, dont l'équivalent en SQL est généré automatiquement *(figure 1.2*).

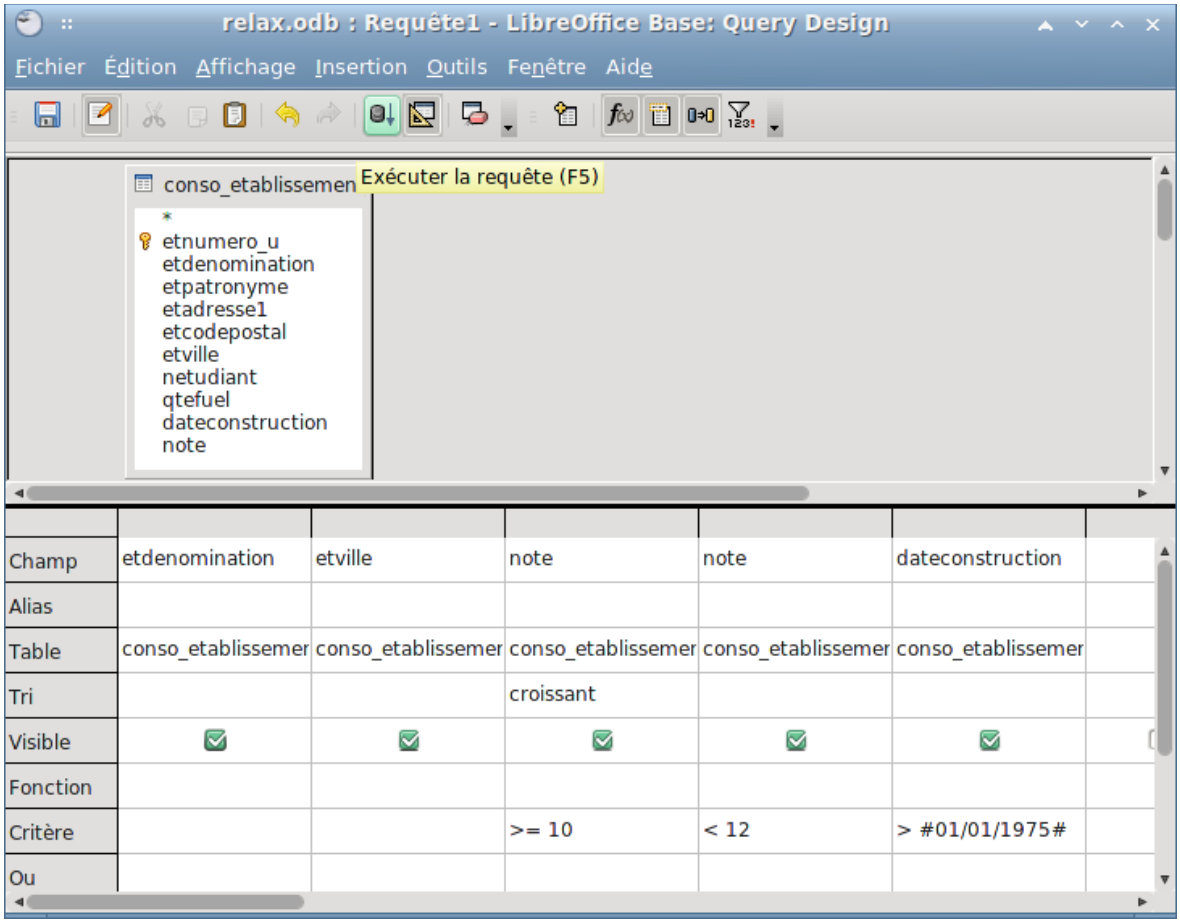


Figure 1.2 : Interface *Query-By-Example* de LibreOffice Base

1.2.3 Détection des anomalies dans les requêtes apparemment correctes

Une fois que l'on s'est assuré que l'utilisateur sait bien ce qu'il veut et qu'il l'a correctement formulé dans sa requête, le risque qui prédomine est la persistance de déficiences insidieuses dans les objectifs exprimés. Ces problèmes peuvent se produire si l'utilisateur a une mauvaise perception du contenu de la base de données. Dans ces cas, les *présuppositions* reflétées dans la requête peuvent s'avérer erronées. Le rôle d'un système coopératif est donc de détecter de telles erreurs et de les corriger. Selon [Motro 00], les méthodes utilisées pour cela peuvent être divisées en quatre groupes :

1. détection et simplification des objectifs/requêtes inutilement *complexes*,

2. détection et modification des objectifs/requêtes *insatisfiables*,

3. détection et modification des objectifs/requêtes *incomplètes*,

4. annotation et explication des réponses.

a. Requêtes complexes

L'excès de complexité d'une requête quand celle-ci peut être simplifiée n'est pas toujours le reflet du manque de maîtrise de l'utilisateur qui l'a formulée à l'égard du langage d'interrogation utilisé. En effet, la *redondance* qu'elle comporte est souvent due à une *conception erronée* de la sémantique de la base de données de la part de l'utilisateur.

Par exemple, soit une base de données avec une contrainte qui requiert que tous les professeurs aient un doctorat. Considérons la requête « la liste des professeurs ayant un doctorat ». L'utilisateur qui l'a soumise n'a pas conscience de la complexité inutile de cette requête, due à la redondance introduite par la condition « ayant un doctorat ». Il apparaît évident qu'il croit qu'il existe des professeurs sans doctorat, ce qui est une *fausse présupposition*. Un comportement coopératif ne se suffira pas à répondre en donnant la liste de tous les professeurs, car cela irait à l'encontre de la *maxime de qualité* de Grice : ignorer la fausse présupposition reviendrait à la *confirmer*. Un interlocuteur humain pourrait répondre dans ce cas « voyons, mais tous les professeurs ont un doctorat ! »[3] ce qui a pour but de la *corriger*. Se contenter de donner la liste de tous les professeurs ferait croire qu'il en existe d'autres et serait anormal.

Avant même de se soucier de la coopération, la simplification des requêtes complexes est un cas particulier de l'*optimisation des requêtes*. Le traitement d'une requête dont les redondances ont été supprimées permet en effet de réduire la complexité du traitement sur la base de données. L'aspect coopératif sera apporté en ajoutant au résultat de la requête simplifiée les précisions nécessaires pour corriger la conception erronée de l'utilisateur.

3 C'est une réponse intensionnelle (voir p.24)

b. Requêtes insatisfiables

Le problème des requêtes insatisfiables est l'un des aspects de l'une des situations ayant reçu le plus d'attention récemment (et qui inclura la nôtre) : le problème des *réponses vides*[4]. En effet, Motro a défini une requête insatisfiable comme une requête qui retournerait une réponse vide pour toute instance de la base de données, i.e. qui ne pourra *jamais* avoir de réponse (quel que soit le contenu de la base).

La détection de telles requêtes est intimement liée à celle des réponses complexes, puisqu'elle aussi met en jeu l'aspect sémantique de la base de données. La connaissance étant représentée par les *contraintes d'intégrité*, il suffit de confronter celles-ci avec la requête pour trouver la contradiction en cause.

Comme précédemment, c'est un mécanisme logique d'optimisation de requêtes qui pourra aisément s'en occuper. Une fois la contradiction détectée, au lieu d'une réponse stricte « il n'y a pas de réponse », la réponse coopérative « il *ne peut pas* y avoir de réponse » sera retournée, accompagnée de la contrainte d'intégrité qui explique pourquoi la fausse présupposition de l'utilisateur est erronée.

Reprenons l'exemple précédent : si l'utilisateur demande cette fois « la liste des professeurs de plus de 50 ans n'ayant pas de doctorat ». Encore une fois, une réponse stricte, c'est-à-dire vide, serait *déroutante.* En effet, elle pourrait amener l'utilisateur à *relaxer* sa première condition, i.e. soumettre sa requête en modifiant progressivement l'âge à chaque fois. Indiquer à l'utilisateur que sa requête *ne peut pas* être satisfaite à cause de la contrainte d'intégrité « tous les professeurs ont un doctorat » est une réponse coopérative qui éviterait la perte de temps pour l'utilisateur et la surcharge du système par des requêtes inutiles.

4 Le problème des réponses vides ne se limite pas aux requêtes insatisfiables mais constitue aussi une importante partie des *requêtes incomplètes* comme nous allons le voir

c. Requêtes incomplètes

L'insatisfaction de l'utilisateur par une réponse stricte engendre souvent le besoin de requêtes supplémentaires. Dans l'exemple précédent, aucune n'aurait pu aboutir à une réponse convenable. Dans le cas des requêtes *incomplètes,* la requête est satisfiable (elle n'est pas incompatible avec la sémantique de la base) mais ne contente pas l'utilisateur.

Le rôle des techniques coopératives de cette catégorie est d'anticiper efficacement les requêtes qui s'ensuivent.

i) Réponses partielles

On retrouve ici le problème des *réponses vides*, comme précédemment avec les requêtes insatisfiables, la différence étant que l'absence de résultat est lié à l'*état* (les données) actuel de la base de données et non à sa *sémantique* (les contraintes d'intégrité). Autrement dit, la base pourrait contenir des données qui satisfont la requête mais ce n'est pas le cas actuellement. Il n'y a pas de contradiction avec les contraintes d'intégrité.

Cependant, cette situation est également due aux *présuppositions erronées*. En effet, l'utilisateur qui soumet une requête s'attend habituellement à recevoir une réponse (sinon pourquoi demander ?).

Une requête est toujours associée à la présupposition que des données correspondantes existent. Une réponse vide montre que la présupposition était fausse.

Voici un exemple typique[5] de conversation avec un SGBD donné par Kaplan [Kaplan 82] :

Q1: Quels sont les étudiants qui ont une note éliminatoire aux examens du deuxième semestre ?

5 Légèrement modifié par nos soins

R1: Aucun.

Connaissant la grande difficulté des examens et n'ayant trouvé aucun résultat intéressant, l'utilisateur pose une question plus générale :

Q2: Quels sont les étudiants qui ont échoué à un examen du deuxième semestre ?

R2: Aucun.

L'utilisateur dépité aborde la question sous un autre angle :

Q3: Combien de personnes ont réussi tous les examens du deuxième semestre ?

R3: Zéro.

Bizarre : personne n'a échoué mais personne n'a réussi... L'utilisateur finit par comprendre :

Q4: Quels sont les examens du deuxième semestre ?

R4: Aucun.

Kaplan qualifie ce comportement non-coopératif de *stonewalling*[6]. La réponse R2 qui signifie « personne n'a échoué » est interprétée comme « il y a eu des examens et personne n'a échoué ». Si la question Q2 était posée à un interlocuteur humain (donc coopératif), la réponse vide aurait obligatoirement signifié que tous les étudiants ont réussi aux examens ; ce n'est pas le cas ici. On voit donc bien que ces réponses vides sans aucune explication supplémentaire enfreignent la maxime de qualité de Grice : elles constituent une sorte de « mensonge par omission »[7].

En outre, Kaplan explique que si à une requête constituée de plusieurs prédicats correspond une présupposition comme nous l'avons dit, l'existence de réponse aux requêtes *plus générales* constituées d'un *sous-ensemble* de ces prédicats est une conviction encore plus certaine pour l'utilisateur.

6 En anglais, le verbe *to stonewall* signifie « donner des réponses évasives, refuser de coopérer en fournissant des informations ».

7 Parmi les définitions de *mentir* dans le dictionnaire « Le Petit Larousse », on trouve « Tromper par de fausses apparences ».

Pour lui, quand une présupposition[8] échoue (i.e. la requête ne retourne aucun résultat), la réponse la plus appropriée est de corriger les fausses présuppositions correspondant aux *plus petites sous-requêtes* associées.

En effet, dans l'exemple, l'interlocuteur humain comprendrait que l'utilisateur fait la fausse présupposition « il y a eu un examen au deuxième semestre » et aurait stoppé net le dialogue dès la première réponse en détrompant l'utilisateur. Cette fausse présupposition correspond à la requête Q4, qui est une sous-requête de la première.

Notons que dans certains cas il est utile de répondre par des réponses vides. En effet, supposons que le niveau de tous les étudiants ait été tel qu'on n'ait jamais vu de note éliminatoire. Il aurait été plus juste de répondre « il n'y a pas eu d'examen ce semestre et il n'y a jamais eu de note éliminatoire ». Ces réponses vides aux sous-requêtes Q4 et « quels sont les étudiants qui ont déjà eu une note éliminatoire ? » sont *significatives*.

Plus tard, Godfrey, reprenant le raisonnement de Kaplan, proposa un algorithme de recherche de ces « sous-requêtes échouantes minimales » ou *MFS (minimal failing subqueries*) en étudiant sa complexité [Godfrey 97].

Motro lui pousse plus loin la réflexion et remarque que des requêtes plus générales peuvent être construites non seulement à partir des sous-requêtes mais aussi en généralisant chaque prédicat.

Par exemple, la requête « quels sont les examens de cette année ? » est plus générale que Q4 (ce n'est pas une sous-requête). Si elle retournait une réponse vide, il serait plus approprié de corriger la présupposition associée en retournant « il n'y a pas eu d'examen cette année » à la requête initiale Q1.

Motro définit par conséquent une *présupposition erronée maximale,* MEP (*maximal erroneous presupposition*), comme correspondant à une requête *Q*

8 Kaplan parle en fait de *présomptions*, notion plus générale que les *présuppositions*. Comme Motro et par souci de simplification, nous ne ferons pas cette distinction.

donnant une réponse vide et telle que toutes les requêtes plus générales que *Q* donnent des réponses non-vides. Godfrey appelle cette requête une *MGQ (maximally generalized failing query)* et note que les MFS sont un cas particulier de MGQ où les prédicats sont généralisés à l'extrême [Godfrey 97].

Motro conclut que seules les MEP sont significatives et qualifie les réponses vides à ces MGQ[9] d'*authentiques* (*genuine empty answers*).

Motro ne s'arrête pas là toutefois et observe que, bien qu'elles soient authentiques, les réponses vides aux MGQ n'en déçoivent pas moins l'utilisateur. Celui-ci cherchera le plus souvent à assouplir certaines conditions de sa requête pour obtenir les données qui s'y rapprochent au mieux. Il serait coopératif d'anticiper les requêtes qui pourraient s'ensuivre. Or, ces requêtes sont précisément les requêtes *plus générales* que l'on aura testées pour déterminer chaque MGQ.

Finalement, la réponse la plus coopérative sera donc formée des réponses vides authentiques correspondant aux MEP et d'un choix des requêtes plus générales qui auront des chances de satisfaire l'utilisateur puisqu'elles correspondront aux requêtes les plus proches de la sienne pour lesquelles il existe au moins une réponse.

Le dialogue de l'exemple pourra alors devenir :

Q1: Quels sont les étudiants qui ont une note éliminatoire aux examens du deuxième semestre ?
R : Il n'y a jamais eu de note éliminatoire.
Il n'y a pas eu d'examen au deuxième semestre.
Voulez-vous connaître les étudiants qui ont une note inférieure à 10 aux examens de cette année ?

9 Motro n'utilise pas le terme MGQ. Nous utiliserons les termes de Godfrey pour simplifier et ne pas avoir à parler à chaque fois des « requêtes correspondant aux MEP ».

Les méthodes employées pour généraliser la requête originale dépendent bien sûr du modèle de données. Notamment, les conditions portant sur les valeurs littérales peuvent nécessiter une connaissance des valeurs possibles et de leur ordre (*hiérarchie*) afin de pouvoir les relaxer.

ii) Réponses augmentées

L'anticipation des requêtes est également possible dans le cas où les données stockées sont groupées par sujet, comme dans [Cuppens, Demolombe 88] où le modèle entité-relation est muni de relations *d'agrégation* et de *généralisation*. Les requêtes sur une partie d'un sujet sont alors fréquemment suivies d'autres requêtes complémentaires.

Prenons pour exemple une requête originale qui demande *la liste des vols du matin de Londres à Paris*. La réponse stricte donnera uniquement les vols avec leur numéro, la compagnie et l'heure de départ. Une réponse coopérative prévoira d'y ajouter les informations *apparentées* susceptibles d'intéresser l'utilisateur afin de lui éviter de formuler d'autres requêtes :

- les attributs manquants : non seulement ceux de l'entité ciblée par la requête (heures d'arrivées, prix, disponibilité), mais aussi ceux des entités participant aux relations d'agrégation avec cette entité (les moyens de transport de l'aéroport à la ville de destination, les hôtels à proximité) ;

- les résultats obtenus par une relaxation des spécifications de la requête (la liste des vols du début d'après-midi) ; cette technique est en ce point comparable à celles utilisées dans le cas des réponses vides ;

- les tuples représentant d'autres objets qui sont des instances des entités participant aux relations de généralisation avec le sujet de la requête ; ce qui revient à une suggestion des alternatives comparables (les départs de train par exemple).

On voit que dans cette technique les tables des réponses sont augmentées en y ajoutant de nouvelles lignes et de nouvelles colonnes.

On remarquera que l'augmentation des réponses peut entraîner une contradiction avec la maxime de quantité de Grice si l'on tombe dans l'abus.

1.2.4 Annotation et explication : les méta-réponses

Cette catégorie de techniques fournit des informations supplémentaires à l'utilisateur, mais cette fois ce ne sera pas des données mais des informations sur les données. Il ne s'agit pas de corriger une quelconque déficience provenant de l'utilisateur comme précédemment : les méta-réponses permettent de commenter les réponses fournies par la base de données. Elles incluent principalement les *réponses intensionnelles* ainsi que les *estimations de qualité*.

a. Réponses intensionnelles

L'intension[10] d'une base de données désigne notamment :

- son *schéma* : l'ensemble des définitions des structures de données
- ses *contraintes d'intégrité* : conditions sur les valeurs que peuvent prendre les données
- ses *vues* (ou *relations dérivées*) : définitions de structures dérivées des structures de base

Elle se distingue de *l'extension* qui est l'ensemble des valeurs qui occupent les structures de données et correspond aux *instances*.

Une autre catégorie d'informations intensionnelles, mais qui ne fait pas partie de la base de données, est constituée par les requêtes. Leur traitement nécessite l'utilisation de l'intension mais leurs réponses n'étant que des ensembles d'instances satisfaisant les conditions de la requête, elles ont toujours été de nature purement extensionnelle. Après l'interrogation de la base de données, la seule

10 Certains auteurs, comme [Gardarin 00], écrivent *intention*.

description intensionnelle des valeurs obtenues en réponse à laquelle l'utilisateur a accès est donc la requête elle-même. Les techniques de cette catégorie partent du principe que certaines des informations intensionnelles contenues dans la base de données peuvent intéresser l'utilisateur qui a reçu une réponse à sa requête puisqu'elles peuvent donner des explications sur cette réponse et compléter sa sémantique.

Diverses approches existent et sont hétérogènes. Selon celle employée, une réponse intensionnelle possible à une requête sur *le personnel dont le salaire est supérieur à 50 000 DA,* par exemple, pourra comporter :

- *Tous les gérants seniors et les ingénieurs, sauf Linda (une gérante junior qui gagne plus que ce montant) et Anis (un ingénieur qui gagne moins que ce montant).*
- *Les gérants senior et leurs superviseurs.*
- *La réponse donnée inclut tous les ingénieurs.*
- *Tous les employés inclus dans la réponse sont senior.*

b. Estimations de qualité

Le prototype de système coopératif *Panorama* développé par Motro [Motro 96] permet de fournir non seulement des informations intensionnelles mais aussi des indications sur la qualité des résultats obtenus par une requête. Il utilise pour cela les méta-informations contenues dans une méta-base de données associée à la base de données qui permettent d'estimer si les données sont *justes, complètes* (exhaustives), *vides* ou *admissibles*.

C'est ainsi que pourra être reproduit le comportement coopératif qui consiste à compléter ses propres réponses par des commentaires sur leur qualité. Par exemple, un interlocuteur humain qui fournirait une liste des librairies à Washington pourrait ajouter l'une des précisions suivantes :

- *« Cette liste est parfaite, faites moi confiance »*
- *« Il peut y avoir d'autres librairies dont je n'ai pas connaissance »*

- *« Je suis sûr de toutes ces librairies, sauf de la dernière, qui a pu être convertie en un vidéoclub »*

1.2.5 Correspondance des requêtes aux objectifs

Dans certaines situations, la requête posée par l'utilisateur semble correcte et ne comporte aucune erreur décelable, mais ne retourne pas les résultats escomptés. L'objectif de l'utilisateur et l'expression formelle de la requête sont tous deux convenables dans ces cas, mais ils ne correspondent pas. Cette situation s'apparente à celle rencontrée en programmation lorsque le programme une fois exécuté s'avère syntaxiquement correct mais comporte des erreurs logiques qui lui font faire une toute autre tâche que celle pour laquelle il a été écrit.

Il est évident que ces problèmes sont difficiles à déceler. En effet comment connaître l'objectif que l'utilisateur a à l'esprit pour pouvoir le comparer avec celui de la requête formulée ? La première solution qui peut être proposée fait participer l'auteur de la requête. Celui-ci fournira en même temps que sa question des indications sur les résultats attendus. Celles-ci permettent, si elles sont en contradiction avec la réponse obtenue de déceler certaines de ces « erreurs logiques ». Ces indications peuvent être :

- des intervalles de cardinalité ; par exemple, la requête est erronée si la réponse est vide ;
- des données connues qui doivent obligatoirement être retrouvées dans les résultats ;
- des données qui doivent obligatoirement *ne pas* être retrouvées dans les résultats.

Exemple: La requête demandant la liste des employés touchant plus de 50 000 € peut être accompagnée des indications « la réponse doit comporter au moins cinquante employés, et l'un d'entre eux doit être Betty ». Si la réponse ne satisfait pas l'une de ces conditions, alors l'erreur provient d'une mauvaise formulation soit de la requête soit des indications.

La deuxième solution qui peut être employée pour déceler l'éventualité de telles erreurs utilise l'historique pour comparer la requête avec celles déjà posées au système depuis sa mise en route. Si cet historique est suffisamment conséquent, et si la requête soumise ne ressemble à aucune de celles qui ont déjà été formulées, c'est fort probablement que l'utilisateur s'est trompé. Cela ne permet cependant que d'émettre des « suspicions » et cela ne fonctionne pas si la requête erronée reste dans le cadre des requêtes habituellement formulées.

1.3 Le problème des réponses vides

Pour traiter les requêtes qui retournent un ensemble vide de réponses, plusieurs approches visent à les *relaxer*. Le processus de *relaxation* permet d'obtenir une requête plus générale, pour corriger la présence d'une fausse présupposition. L'idée est de tenter d'obtenir des résultats plus généraux ou bien appartenant au voisinage proche de ceux qui auraient dû être retournés par la requête initiale (si elle n'avait pas échoué). Pour cela, il faudra « assouplir » les conditions de recherche trop restrictives pour permettre de récolter des informations supplémentaires en résultat.

Les approches ayant inspiré la démarche adoptée dans notre travail sont essentiellement celles de Motro et de Godfrey:

1. *L'approche de Motro (relaxation par généralisation)* consiste à transformer les conditions de la requête en les remplaçant par de plus générales [Motro 86][Motro 90].

2. *L'approche de Godfrey (relaxation par suppression)* vise à supprimer des parties de la requête afin d'obtenir une sous-requête moins contraignante [Godfrey 97]

Par l'aspect coopératif apporté dans les deux cas, on veut aider l'utilisateur à éliminer les présuppositions erronées soit en fournissant une explication sur la

raison de l'échec de sa requête soit en l'assistant pour transformer celle-ci afin qu'elle n'échoue plus.

Considérons une requête conjonctive booléenne Q de la forme:

$$Q = A_1 \wedge \ldots \wedge A_N$$

où chaque A_i est une condition atomique. On dira que Q' est une *sous-requête* de Q si et seulement si

$$Q' = A_{S1} \wedge \ldots \wedge A_{Sm}$$

et $\{s_1, \ldots, s_m\} \subset \{1, \ldots, N\}$.

Il est clair que si la sous-requête Q' de Q retourne une réponse vide, alors la requête Q ne peut retourner elle même qu'une réponse vide.

1.3.1 Approche de Motro

Pour résoudre le problème des requêtes à réponse vide, Motro [Motro 86] proposa une approche qui combine l'idée de relaxation de ces requêtes en requêtes plus générales et celle de recherche des fausses présuppositions. Les requêtes généralisées (qui sont des présuppositions logiques de la requête originale) sont obtenues en remplaçant certaines des conditions de la requête par de plus générales. Cette généralisation peut être effectuée en affaiblissant les conditions mathématiques et/ou en éliminant les conditions non mathématiques.

Considérons par exemple la requête Q_1 qui demande les employés satisfaisant les conditions suivantes : age ≤ 30, genre = féminin et salaire ≥ 40 k€. Le treillis des généralisations de Q_1 est représenté en *figure 1.3*, où la notation (x, y, z) désigne la présupposition qu'il peut y avoir des employés d'âge inférieur à x, de genre y, et de salaire annuel d'au moins z. Le symbole '*' indique toute valeur (la condition correspondante ne peut pas être généralisée davantage). Q_4 = (30, F, 39) est une présupposition pour Q_1 = (30, F, 40) où la condition "salaire ≥ 40 k€" est relaxée en "salaire ≥ 39 k€". Selon cette approche, une sous-

requête peut être vue comme une généralisation extrême : certaines des conditions ont été éliminées (elles sont considérées comme toujours vraies).

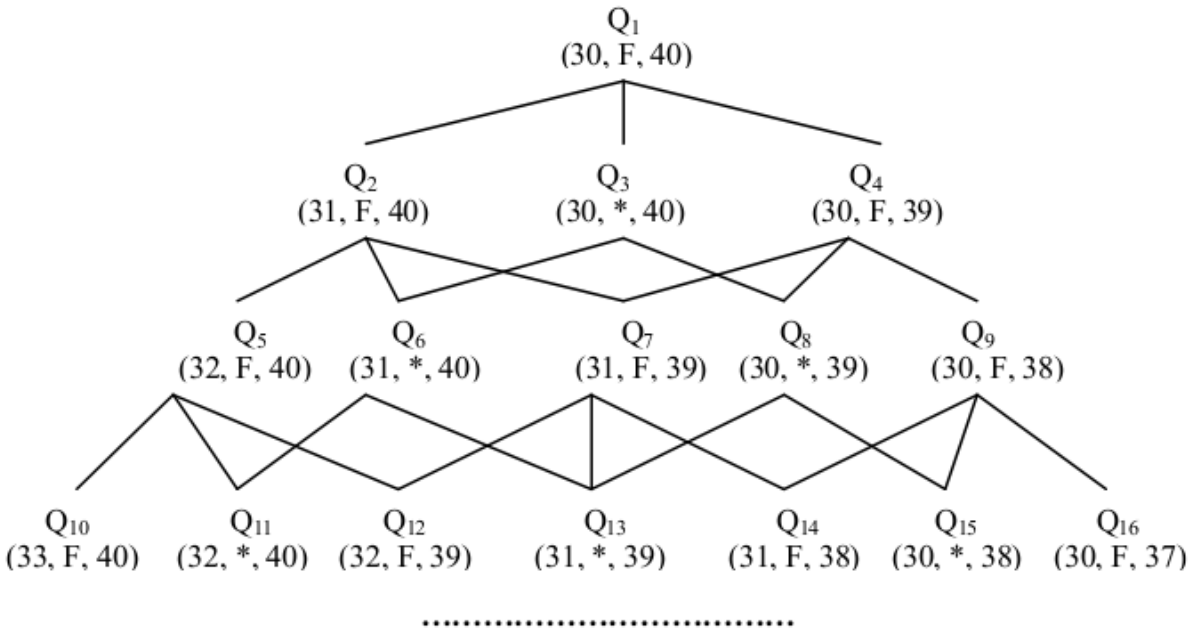


Figure 1.3 : Treillis de requêtes généralisées

Une des raisons de l'échec d'une requête est la présence de fausses présuppositions. Dans ce cas, au lieu de ne renvoyer qu'un ensemble vide de réponses, il est plus coopératif de fournir à l'utilisateur des informations sur les fausses présuppositions de la requête soumise. Pour extraire les présuppositions de requêtes et vérifier leur adéquation par rapport à la base de données, Motro proposa un outil, appelé SEAVE. En partant du fait que la fausse présupposition qui n'a pas de fausses présuppositions est responsable de l'échec de la requête, SEAVE peut retourner l'ensemble de toutes les fausses présuppositions significatives (les présuppositions telles que toutes les présuppositions plus générales produisent un ensemble non vide de réponses).

Définition : Soit Q une requête infructueuse (à réponse vide). Q*, une généralisation infructueuse de Q, est dite *maximale* si et seulement si aucune de ses généralisations ne retourne une réponse vide.

Toutes les présuppositions fournies par SEAVE sont des généralisations infructueuses maximales, ou *MGQ* (*Maximally Generalised failing Queries*). L'intérêt des MGQ est qu'elles sont plus significatives et fournissent un moyen de relaxation à l'utilisateur. Elles fournissent une explication de l'échec ou une forme d'assistance pour rendre la requête fructueuse.

Voici une version légèrement modifiée de l'algorithme SEAVE :

Entrée : p : une requête ;
1. **Fonction** seave(p, **var** Mgq) : **booléen**;
2. **début**
3. **si** test(p) **alors retourner vrai**
4. **sinon**
5. **début**
6. est_Mgq := **vrai**;
7. ens := mgp(p);
8. **pour** q **dans** ens **faire**
9. est_Mgq := est_Mgq **et** seave(q, Mgq);
10. **si** est_Mgq **alors**
11. Mgq := Mgq ∪ {p};
12. **finsi**;
13. **retourne faux**
14. **fin**;
15. **finsi**;
16. **fin**;

Sortie : Mgq: l'ensemble de MGQ de la requête p

Algorithme 1.1 : Une version légèrement modifiée de SEAVE [Bosc et al. 08a]

- *test(p)* évalue la présupposition *p* par rapport au contenu de la base de données : retourne *vrai* si *p* produit des réponses, *faux* si *p* échoue ;
- *mgp(p)* retourne l'ensemble des présuppositions immédiatement plus générales pour *p*. Par exemple dans la *figure 1.3*, $mgp(Q_2) = \{Q_5, Q_6, Q_7\}$.

Cette approche requiert d'imposer une certaine forme de limite, par exemple, une limite supérieure de *k* étapes de relaxation. En outre, la taille de l'étape de relaxation doit également être spécifiée. Notons bien que pour identifier

les MGQ, il peut être nécessaire d'évaluer plusieurs requêtes successives sur la base de données. Pour réduire le coût de telles évaluations, Motro propose d'abord de trouver la requête qui correspond à la borne inférieure, c'est-à-dire celle obtenue en affaiblissant chaque condition atomique par *k* étapes, puis d'évaluer cette requête sur la base de données et sauvegarder les relations résultantes. Ainsi, toutes les requêtes successives seront évaluées non plus sur toute la base de données distante mais sur les résultats ainsi stockés localement. Malgré tout, le principal inconvénient de SEAVE demeure son coût de calcul élevé, provenant du calcul et de l'évaluation d'un grand nombre de présuppositions.

1.3.2 Approche de Godfrey

Reprenant les travaux de Kaplan [Kaplan 82], Godfrey considère que toute sous-requête est une présupposition de la requête initiale [Godfrey 97]. Par exemple, soit *Q* une requête composée de trois conditions atomiques (en utilisant la même notation que [Godfrey 97]) :

$$Q = a(x) \wedge b(x) \wedge c(x)$$

Le treillis de sous-requêtes associées à *Q* est schématisé en *figure 1.4* (chaque sous-requête est marquée avec le nombre de réponses qu'elle produit. Σ_{DB} indique le nombre de tuples dans la base de données interrogée).

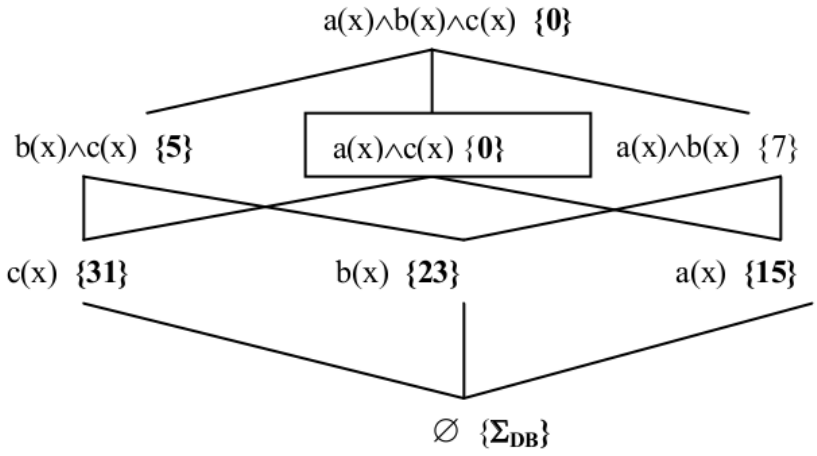


Figure 1.4 : Treillis de sous-requêtes

(Le symbole ∅ indique une sous-requête vide).

Une requête réussit (c.-à-d. est fructueuse : donne des réponses en résultat) si toutes ses sous-requêtes réussissent également. Quand une requête échoue (c.-à-d. est infructueuse : ne donne aucune réponse en résultat), l'approche de Godfrey suggère d'identifier les présuppositions (les sous-requêtes) qui échouent (puisqu'il est plus utile de rapporter l'échec de la sous-requête que l'échec de la requête elle-même).

Définition : Soit Q une requête infructueuse, Q^* une sous-requête de Q infructueuse est *minimale* si et seulement si aucune de ses sous-requêtes n'échoue.

Les présuppositions intéressantes sont celles qui sont minimales : ce sont les *MFS* (*Minimal Failing Subqueries* : sous-requêtes minimales qui échouent). Par conséquent, le système ne rapporte pas seulement l'ensemble de réponses vide mais également la cause de l'échec de la requête en identifiant une ou plusieurs MFS (remarquons qu'une MFS n'est pas unique, il peut y en avoir de nombreuses).

Partant de l'observation suivante : si une requête $Q = A_1 \wedge \ldots \wedge A_N$ échoue et la sous-requête $Q' = Q - \{A_i\}$ réussit, alors toute MFS de Q doit contenir l'atome A_i, Godfrey proposa un algorithme efficace pour trouver une MFS d'une requête conjonctive composée de N prédicats [Godfrey 97] (voir l'*algorithme 1.2*). Cet algorithme, qui procède en profondeur d'abord et de haut en bas, est polynomial et se déroule en $O(N)$. La fonction *test*(Q) de l'*algorithme 1.2* consiste à évaluer la requête Q sur la base de données de façon que :

- si la requête échoue, *test* retourne *vrai*,
- sinon, si la requête donne un ensemble non-vide de réponses, *test* retourne *faux*

Entrée : Top: la requête d'origine.

```
1.  fonction une_mfs_rapide(Top, Mfs) : booléen
2.  début
3.      si test(Top) alors
4.          début
5.            une_mfs_vrai(Top, Mfs, ∅);
6.            retourne vrai;
7.          fin
8.      sinon retourne faux
9.      finsi;
10. fin;
```

Sortie : Mfs: une mfs de la requête Top

```
1.  procédure une_mfs_vrai(Ens, Mfs, Core)
2.  début
3.      si Ens = ∅ alors Mfs := Core;
4.      sinon
5.          début
6.            choisir Ele ∈ Ens;
7.              si test((Ens − Ele) ∪ Core) alors
8.                  une_mfs_vrai(Ens − {Ele}, Mfs, Core);
9.              sinon une_mfs_vrai(Ens − {Ele}, Mfs, Core ∪ {Ele});
10.           finsi;
11.         fin;
12.     finsi;
13. fin;
```

<u>Algorithme 1.2 :</u> Algorithme de recherche d'une MFS en *N* étapes [Godfrey 97]

Il a été prouvé que le problème de trouver toutes les MFS d'une requête est NP-Complet et donc intraitable. Cependant, on peut trouver *k* MFS, pour tout *k* fixé, dans un temps polynomial, grâce à l'algorithme optimal ISHMAEL, décrit en détail dans [Godfrey 97] et dont voici les caractéristiques de base :

- Il minimise le nombre d'appels à *test* :

  i) il n'évalue jamais la même requête deux fois ;

  ii) si une requête a été soumise et évaluée comme vide, aucune super-

requête ne sera alors soumise ;

iii) si une requête a été soumise et évaluée comme non-vide, aucune sous-requête ne sera alors soumise ;

- Il exploite convenablement la « décomposabilité » du processus d'énumération : toutes les MFS d'une sous-requête infructueuse sont également des MFS de la requête originale.

Comme on le verra par la suite, l'approche de relaxation de requêtes flexibles que nous utiliserons combine l'idée de généralisation de requête (utilisée par Motro) et la notion de MFS.

La généralisation de requête est accomplie par une transformation appropriée et vise à modifier les contraintes floues d'une requête en variantes moins restrictives, tandis que la notion de MFS est utilisée pour rechercher une requête fructueuse dans le treillis des requêtes modifiées [Bosc *et al.* 08a].

Chapitre 2 :

Requêtes flexibles

2.1 Introduction

La quête vers une nouvelle génération de SGBD a amené certains à proposer des solutions pour pallier le manque de flexibilité des systèmes actuellement disponibles. Il faut toutefois s'interroger sur le sens du terme flexible, qui peut être, selon les auteurs, d'ordre syntaxique (rigidité de l'utilisation) ou sémantique (fonctionnalités disponibles).

Nous définirons la notion de flexibilité comme la possibilité pour le SGBD de fournir des réponses *discriminantes*, particulièrement par le biais de *requêtes imprécises* c'est-à-dire dont *l'interprétation* est flexible. Les réponses sont alors ordonnées selon le degré de pertinence qui indique *à quel degré* la condition est satisfaite. Cela implique une relation de *préférence* entre les éléments obtenus, qui constituera un ordre total ou partiel. Un prolongement de ce raisonnement mène aux travaux concernant les réponses coopératives, dont le but est d'éviter les résultats vides (voir notamment à ce propos [Bosc *et al.* 04]).

Pour introduire cette imprécision dans les requêtes, plusieurs approches sont proposées. Nous en retiendrons trois **(Figure 2.1)**. La première divise chaque requête en deux parties : une sélection booléenne ordinaire et une composante permettant le classement des résultats (cas notamment des *requêtes à préférences* comme celles du langage *Preference SQL* [Kiessling 01]). Dans la deuxième, les conditions imprécises sont traduites en conditions booléennes se rapportant à des intervalles au lieu de valeurs uniques.

Nous nous intéresserons à la troisième approche : l'interprétation des conditions imprécises par des *ensembles flous*. Pour le classement des résultats, elle utilise, comme précédemment, une notion de distance des n-uplets par rapport à la condition imprécise. Cependant, les ensembles flous constituent un outil *plus précis* et naturellement plus adéquat. C'est pourquoi nous commencerons par un rappel sur la théorie des ensembles flous.

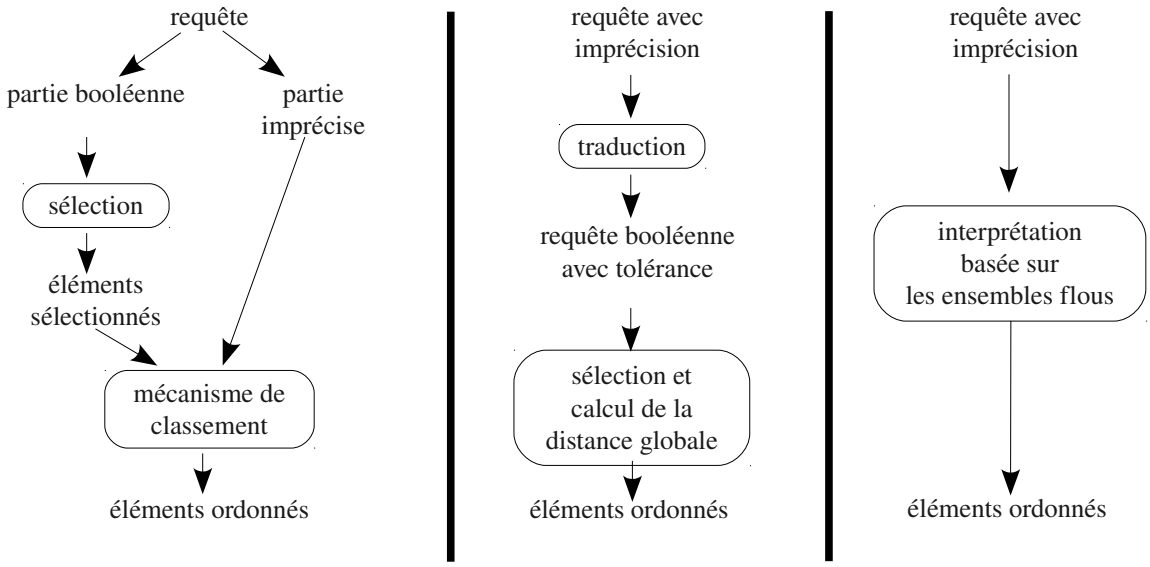


Figure 2.1 : Trois approches pour les requêtes imprécises (d'après [Bosc 92])

2.2 Rappels sur la théorie des ensembles flous

2.2.1 Notion d'ensemble flou

Les ensembles flous ont été définies par L. Zadeh en 1965 comme une extension de la théorie des ensembles dans le but de décrire des ensembles dont les limites ne sont pas définis de manière précise. Le degré d'appartenance $\mu_A(x)$ d'un élément x à l'ensemble flou A est dans l'intervalle unité [0, 1]. C'est ainsi que la transition entre la pleine appartenance ($\mu_A(x)$=1) et sa non-appartenance ($\mu_A(x)$=0) se fait d'une manière graduelle.

Exemple : Soit l'ensemble flou représentant les valeurs de température *proches de 37°C*, le degré d'appartenance de 36.6°C pourra avoir la valeur de 0.7 par exemple, alors qu'une température entre 36.9°C et 37.1°C aura un degré de 1, et une température inférieure ou égale à 36°C aura elle un degré nul.

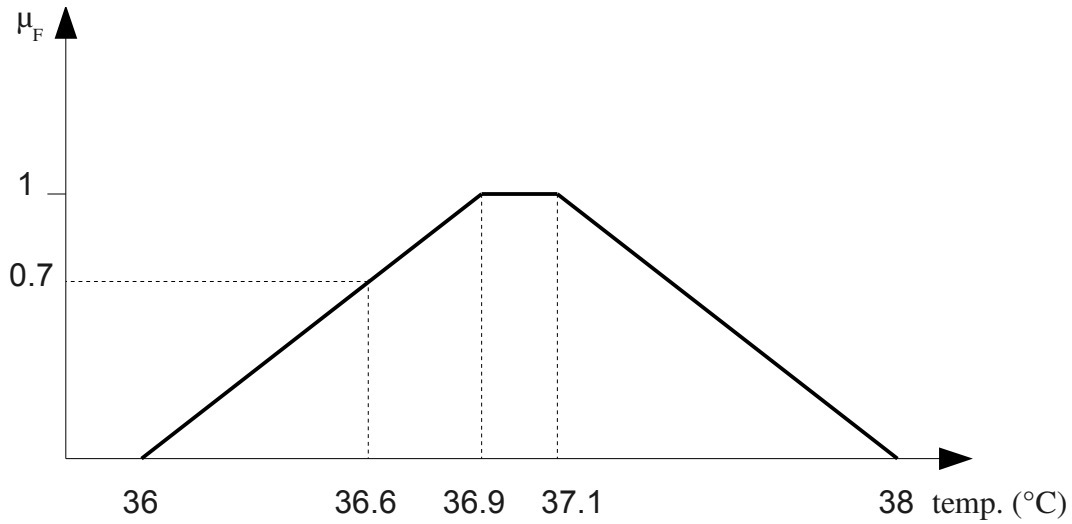


Figure 2.2 : Fonction d'appartenance trapézoïdale de

***F* = « *proches de 37°C* » = (36, 38, 0.9, 0.9)**

Un ensemble flou F dans le référentiel U est caractérisé par une fonction d'appartenance μ_F: $U \rightarrow [0, 1]$, où $\mu_F(u)$ représente le degré d'appartenance de u à F.

Caractéristiques d'un ensemble flou

Deux ensembles nets (non-flous) sont d'un intérêt particulier lors de la définition d'un ensemble flou F :

- le *noyau* : $C(F) = \{u \in U / \mu_F(u) = 1\}$
- le *support* : $S(F) = \{u \in U / \mu_F(u) > 0\}$

Le plus souvent, il est commode de définir les *intervalles flous* par une *fonction d'appartenance trapézoïdale* (*f.a.t.*[11]), notée (A, B, a, b), pour laquelle le noyau est [A, B] et le support est]A-a, B+b[. Les valeurs intermédiaires croissent (à gauche du noyau) et décroissent (à droite) de manière linéaire :

11 En anglais *t.m.f.* (*trapezoidal membership function*).

$$\mu_F(u)=\begin{cases}0 \text{ si } u\le A-a \text{ ou } u\ge B+b,\\ 1 \text{ si } A\le u\le B,\\ 1+(u-A)/a \text{ si } A-a\le u\le A,\\ 1-(B-u)/b \text{ si } B\le u\le B+b.\end{cases}$$

Ce ne sont pas les seules représentations possibles (les « côtés » peuvent avoir une forme parabolique ou utiliser une fonction gaussienne notamment), mais pour ne pas compliquer inutilement les calculs, nous utiliserons ce type de fonctions pour notre travail.

Dans l'exemple précédent, $C(F)$ = [36.9, 37.1] et $S(F)$ =]36, 38[(*figure 2.2*).

La *hauteur* d'un ensemble flou est la plus grande valeur prise par sa fonction d'appartenance : $h(F) = \sup_{u\in U} \mu_F(u)$. Quand elle est égale à 1, on dira que l'ensemble flou est *normalisé.*

Un ensemble flou peut être *discret*. Si son support est fini, toutes les valeurs de sa fonction d'appartenance peuvent être énumérées. Sa *cardinalité* (ou son *cardinal*) est le résultat de l'addition de ces valeurs[12]. On notera par exemple F = {0.1 / a, 0.5 / b, 0.3 / c} pour désigner l'ensemble flou discret dont le support est constitué des trois éléments a, b et c, de degrés d'appartenance respectifs 0.1, 0.5 et 0.3, et dont la cardinalité est de 0.9 (F n'étant pas normalisé, son noyau est l'ensemble vide).

Une *coupe de niveau* α (ou *α-coupe*) de l'ensemble flou *F,* notée F_α, est un ensemble ordinaire défini par

$$F_\alpha=\{u\in U / \mu_F(u) \ge \alpha\}$$

α étant un réel pris dans l'intervalle]0, 1].

2.2.2 Opérations sur les ensembles flous

Les opérations sur les ensembles flous sont généralement fondées sur les notions de *norme* et de *co-norme triangulaires*.

12 Dans le cas des ensembles flous continus, la cardinalité est calculée par *l'intégrale* de la fonction d'appartenance.

Les normes et co-normes triangulaires sont des opérateurs définis de [0,1] × [0,1] vers [0,1] vérifiant les propriétés suivantes :

- commutativité : $op(a,b) = op(b,a)$
- associativité : $op(a,op(b,c)) = op(op(a,b),c)$
- monotonie non-décroissante : si $a \geq c$ et $b \geq d$, alors $op(a,b) \geq op(c,d)$
- un élément neutre n : $op(a,n) = op(n,a) = a$, l'élément neutre devant être n=1 pour la norme et n=0 pour la co-norme.

La paire norme/co-norme d'opérateurs op_1/op_2 la plus utilisée est celle de Zadeh :

$$op_1(a,b) = \min(a,b) \text{ ; } op_2(a,b) = \max(a,b)$$

Il en existe cependant une infinité, et nous pourrons citer en particulier :

- $op_1(a,b) = ab$; $op_2(a,b) = a + b - ab$ (produit et somme probabilistes)
- $op_1(a,b) = \max(a + b - 1, 0)$; $op_2(a,b) = \min(a + b, 1)$ (Lukasiewicz)

Soit A et B deux ensembles flous définis dans l'univers U. Les opérations classiques de la théorie des ensembles peuvent à présent être étendues aux ensembles flous grâce aux opérateurs op_1 et op_2 précédemment définis, qui permettent d'obtenir leurs fonctions d'appartenance respectives. Celles-ci sont décrites dans le tableau ci-dessous :

Opération	**Fonction d'appartenance (générale)**	**Fonction d'appartenance (opérateurs de Zadeh)**
Intersection	$\forall x \in U,\ \mu_{A \cap B}(x) = op_1(\mu_A(x),\mu_B(x))$	$\mu_{A \cap B}(x) = \min(\mu_A(x),\mu_B(x))$
Union	$\forall x \in U,\ \mu_{A \cup B}(x) = op_2(\mu_A(x),\mu_B(x))$	$\mu_{A \cup B}(x) = \max(\mu_A(x),\mu_B(x))$
Complément	$\forall\ x \in U,\ \mu_{\bar{A}}(x) = 1 - \mu_A(x)$	$\mu_{\bar{A}}(x) = 1 - \mu_A(x)$
Différence	$\forall\ x \in U,\ \mu_{A-B}(x) = op_1(\mu_A(x), \mu_{\bar{B}}(x))$ (car $A - B = A \cap \bar{B}$)	$\mu_{A-B}(x) = \min(\mu_A(x), \mu_{\bar{B}}(x))$

On pourra vérifier par ailleurs que les lois de De Morgan sont respectées :

$$A \cap B = \overline{\bar{A} \cup \bar{B}} \quad \text{et} \quad A \cup B = \overline{\bar{A} \cap \bar{B}}$$

2.2.3 Relations floues

a. Définition

Une relation R définie sur des ensembles flous X_1, X_2, ..., X_n est un sous-ensemble flou du produit cartésien $X_1 \times X_2 \times ... \times X_n$ de ces ensembles, caractérisé par une fonction d'appartenance μ_R. Celle-ci quantifie le degré de la liaison (graduelle ou imprécise) entre les éléments de ces ensembles [Bouchon-Meunier 07].

b. Composition de relations floues

La composition de deux relations floues R sur $X \times Y$ et R' sur $Y \times Z$ est la relation floue RoR' définie sur $X \times Z$ dont la fonction d'appartenance est définie par :

$$\forall (x,z) \in X \times Z,\ \mu_{RoR'}(x, z) = \sup_{y \in Y} \min(\mu_R(x, y), \mu_{R'}(y, z))$$

Remarque : cette définition est celle de la *composition max-min*, qui utilise comme norme triangulaire l'opérateur min, les *compositions max-T* utilisant un autre opérateur T sont plus rarement utilisées.

2.2.4 Prédicats flous

Les ensembles ordinaires permettent la définition de prédicats booléens. D'une manière analogue, des prédicats (ou conditions) graduels peuvent être associés aux ensembles flous. Souvent, les prédicats flous élémentaires correspondent à des adjectifs dans le langage naturel, tel que *jeune*, *grand*, *cher* ou *bien payé*. Ils peuvent être modélisés comme des fonctions (de forme habituellement triangulaire ou trapézoïdale) d'un ou plusieurs domaines vers l'intervalle unité. Un prédicat élémentaire peut également comparer deux attributs en utilisant non seulement les opérateurs usuels (égalité, supériorité, etc.), mais aussi des opérateurs graduels tels que « *plus ou moins égal* » ou « *nettement*

supérieur à ». Il est possible de modifier (en l'affaiblissant ou en le renforçant) le sens d'un prédicat donné en utilisant un modificateur qui est généralement associé à un adverbe (tel que « *très* », « *plus ou moins* », « *relativement* », « *vraiment* »). Par exemple, « *très cher* » est plus restrictif que « *cher* » et « *assez haut* » est moins exigeant que « *haut* ».

2.3 Requêtes flexibles

2.3.1 Principes

Afin d'obtenir un résultat qui ne soit pas un ensemble plat mais plutôt une variété discriminante d'éléments classés selon leurs degrés de satisfaction, les **requêtes à préférences** [Borzsonyi *et al.* 01 ; Chomicki 03 ; Hadjali *et al.* 08] expriment que certaines valeurs sont préférées aux autres. Chaque élément de la réponse pourra ainsi être comparé avec les préférences définies dans la requête. On pourra donc restreindre le résultat soit à ceux qui sont suffisamment satisfaisants soit aux *k* meilleurs éléments.

Les **requêtes flexibles** [Christiansen *et al.* 97 ; De Calmès *et al.* 03 ; Bosc *et al.* 04] sont des requêtes dans lesquelles les préférences de l'utilisateur peuvent être exprimées. Les prédicats graduels (comme « *jeune* », « *bien payé* », « *autour de 30* », etc.) correspondant à des termes vagues du langage naturel, constituent les fondements de ce type de requêtes. Le cadre formel des ensembles flous offre un outil approprié pour représenter ce type de prédicats et ainsi pour interpréter les requêtes flexibles. La réponse à une requête est donc un ensemble d'éléments auxquels un degré de satisfaction est attaché (plus le degré est élevé plus l'élément est satisfaisant).

Pour illustrer la notion de requête flexible, considérons le cas d'une personne souhaitant interroger une base de données sur les restaurants pour choisir un « menu à prix abordable », dans un restaurant « de préférence chinois » et « proche du centre-ville ». Il faut avant tout non seulement définir les termes

utilisés dans la requête qui véhicule les préférences afin d'obtenir des réponses discriminées mais aussi l'importance relative attribuée à chacun des trois critères (cuisine, prix, lieu). Concernant la définition des termes, on peut, par exemple, considérer que « de préférence chinois » sera compris comme : chinois complètement préféré, vietnamien ou indien acceptable, japonais s'il n'y a pas de meilleure possibilité. Ainsi *Restaurant_chinois* peut être modélisé par l'ensemble flou :

$$\{1/chinois,\ 0.8/vietnamien,\ 0.75/indien,\ 0.4/japonais\}.$$

Notons que de telles distinctions ne peuvent être considérées dans un cadre purement booléen, où un prédicat ne peut être que satisfait ou pas (chinois et japonais par exemple ne peuvent pas être comparés) et où toutes les conditions ont la même importance.

Considérons un second exemple. Soit la relation *Employés* dont le schéma est *R(num, nom, salaire, âge, ville_habitée)*. Supposons qu'un utilisateur soit intéressé par la recherche des employés *jeunes* et *bien payés*. La requête à formuler s'écrit $Q = jeune \wedge bien\_payé$ où les prédicats flous *jeune* et *bien_payé* sont représentés par les fonctions d'appartenance données en *figure 2.3*.

Dans ce cadre les préférences peuvent être exprimées à deux niveaux distincts : à l'intérieur des conditions élémentaires et aussi dans l'agrégation de ces conditions. Dans le premier cas, l'objectif est d'exprimer que certaines valeurs sont plus adéquates que d'autres, et dans le second cas, les préférences traduisent des niveaux d'importance associés au conditions élémentaires.

En résumé, l'utilisation des ensembles flous dans les requêtes est motivée par plusieurs raisons, en particulier :

- le fait qu'il est très bien adapté à l'interprétation des termes linguistiques, qui constituent un moyen commode pour un utilisateur d'exprimer ses préférences,

- le fait que la théorie des ensembles flous repose sur une hypothèse de commensurabilité, qui permet de combiner plusieurs préférences sur divers attributs de façon à obtenir au final un pré-ordre complet sur les réponses.

Dans notre travail, c'est le cadre des ensembles flous qui est utilisé comme un outil pour supporter l'expression des préférences. L'utilisateur ne spécifie pas des conditions nettes, mais floues, dont la satisfaction peut être considérée comme une question de *degré*. En conséquence, le résultat d'une requête n'est plus un ensemble plat d'éléments mais est un ensemble d'éléments différenciés selon leur satisfaction globale aux contraintes floues apparaissant dans la requête.

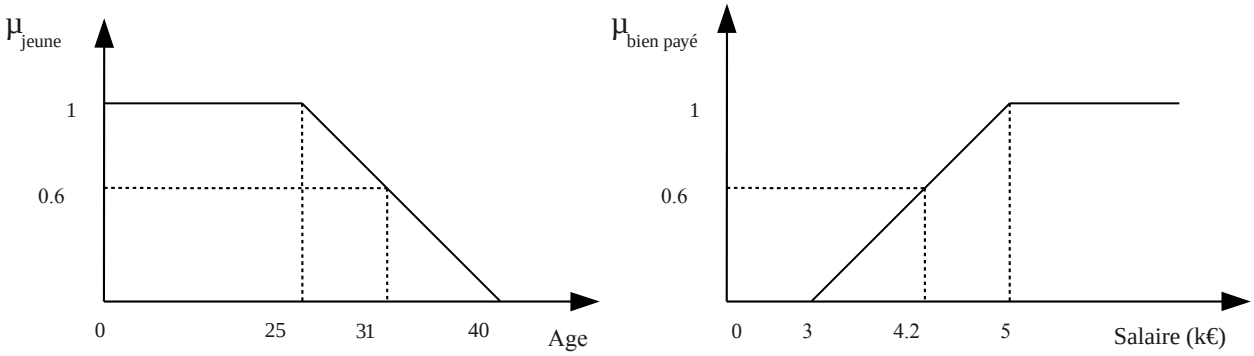


Figure 2.3 : Prédicats flous *jeune* et *bien-payé* ($\mu_{jeune}(31) = 0.6$ et $\mu_{bien\text{-}payé}(4.2) = 0.6$).

Dans l'exemple typique de requête flexible « rapporter les employés qui sont *jeunes* et *bien-payés* », *jeune* et *bien-payé* sont les prédicats graduels représentés grâce aux ensembles flous illustrés en *figure 2.3*. Ici, le prédicat « *jeune* » est modélisé par la *f.a.t.* (A, B, a, b) = (0, 25, 0, 15) où [A, B] = [0, 25] représente le noyau de ce prédicat et [A–a , B+b] = [0, 40] en représente le support.

2.3.2 Extension de l'algèbre relationnelle

L'introduction de la théorie des ensembles flous dans l'algèbre relationnelle implique l'utilisation de *relations floues*, qui, par analogie avec les relations usuelles, sont définies comme des sous-ensembles flous du produit cartésien des domaines. Toute relation floue *r* est constituée de n-uplets pondérés, notés *μ/t*, où

μ représente le degré d'appartenance du n-uplet t à la relation, i.e. le degré de compatibilité avec le concept véhiculé par r. Les bases de données interrogées ne contenant que des relations ordinaires, ces relations initiales sont en fait des cas particuliers de relations floues où tous les poids des n-uplets sont égaux à 1. Notons que les poids rattachés aux n-uplets dans ce modèle ne font pas du tout référence à des données imprécises ou mal connues (comme celles stockées dans les *bases de données possibilistes* par exemple) mais correspondent à l'idée de graduation dans le respect de la condition flexible.

Les opérations usuelles de l'algèbre relationnelle peuvent être directement étendues aux relations floues. Pour cela, d'un côté l'on considère les relations floues comme des ensembles flous, et de l'autre côté l'on introduit les prédicats graduels avec les booléens dans les opérations adéquates (sélection et jointure en particulier).

Soit r et s deux relations floues définies sur les mêmes domaines D_1, ..., D_k, les opérations ensemblistes vues précédemment s'appliquent de la même manière grâce aux opérateurs norme et co-norme, que l'on notera respectivement $\top$ et $\perp$ cette fois ci :

- union : $\mu_{\text{union}(r,\, s)}(t) = \perp(\mu_r(t),\, \mu_s(t))$
- intersection : $\mu_{\text{intersection}(r,\, s)}(t) = \top(\mu_r(t),\, \mu_s(t))$
- différence : $\mu_{\text{différ}(r,\, s)}(t) = \top(\mu_r(t),\, 1 - \mu_s(t))$

Le produit cartésien de toutes relations floues r et s définies respectivement sur les domaines X et Y est obtenu par :

$$\mu_{\text{prod}(r,\, s)}(tu) = \top(\mu_r(t),\, \mu_s(u))$$

Les opérations de sélection, projection et jointure appliquées aux relations floues sont définies comme suit :

- *sélection* : $\mu_{\text{select}(r,\, cond)}(t) = \top(\mu_r(t), \mu_{cond}(t))$ où *cond* est un prédicat flou.

- *projection* : $\mu_{\text{project}(r,\, Y)}(u) = \max_v \mu_r(uv)$ où Y est un sous-ensemble de X l'ensemble d'attributs de r et u l'une de ses valeurs, alors que v prend sa valeur dans $(X - Y)$.

- *jointure* : $\mu_{\text{join}(r,\, s,\, A,\, B,\, \theta)}(tu) = \top(\mu_r(t), \mu_s(u), \mu_\theta(t.A, u.B))$ où A (resp. B) est un sous-ensemble de X (resp. Y) l'ensemble d'attributs de r (resp. s), A et B sont définis sur les mêmes domaines, θ est un opérateur relationnel binaire (qui peut être flou), $t.A$ (resp. $u.B$) représente la valeur de t pour A (resp. u pour B).

L'extension des opérations relationnelles vues ci-dessus est relativement triviale. Il n'en est pas de même pour un opérateur aussi complexe que la *division*. On pourra consulter les travaux récents de Bosc, Hadjali et Pivert qui proposent diverses approches pour le mettre en œuvre, et expliquent par ce biais l'intérêt que peut avoir l'utilisation des ensembles flous pour la modélisation des requêtes floues [Bosc *et al.* 07].

2.3.3 Exemples

Exemple 1 : Considérons une base de données avec la relation *employé*(*num*, *nom*, *salaire*, *age*, *ville*) et son extension donnée dans le *tableau 2.1*. A partir de cette relation initiale, il est possible d'obtenir la relation floue intermédiaire *emp-pjsm* montrée dans le *tableau 2.2* contenant les employés qui sont « plutôt jeunes » et dont le salaire est « moyen », les f.a.t. de ces deux prédicats étant donnés en *figures 2.4 et 2.5*. Il peut être noté qu'aucun élément n'est un membre à part entière de la relation floue *emp-pjsm*. En effet, pour qu'un n-uplet appartienne complètement à cette relation, il lui faudrait atteindre le degré maximal de 1. L'employé Ghani, par exemple, ne satisfait la requête qu'avec un degré de 0.4, qui a été obtenu en considérant le minimum (l'opérateur min ayant été choisi comme norme) des degrés des deux attributs : min(0.6, 0.4) = 0.4

Tableau 2.1 : Une extension de la relation *employé*

num	*nom*	*salaire*	*age*	*ville*
17	Dupont	13000	42	Lyon
76	Martin	12500	40	New-York
26	**Ghani**	**12000**	**37**	**Alger**
12	Smith	12000	39	Londres
55	Lucas	13000	35	Miami

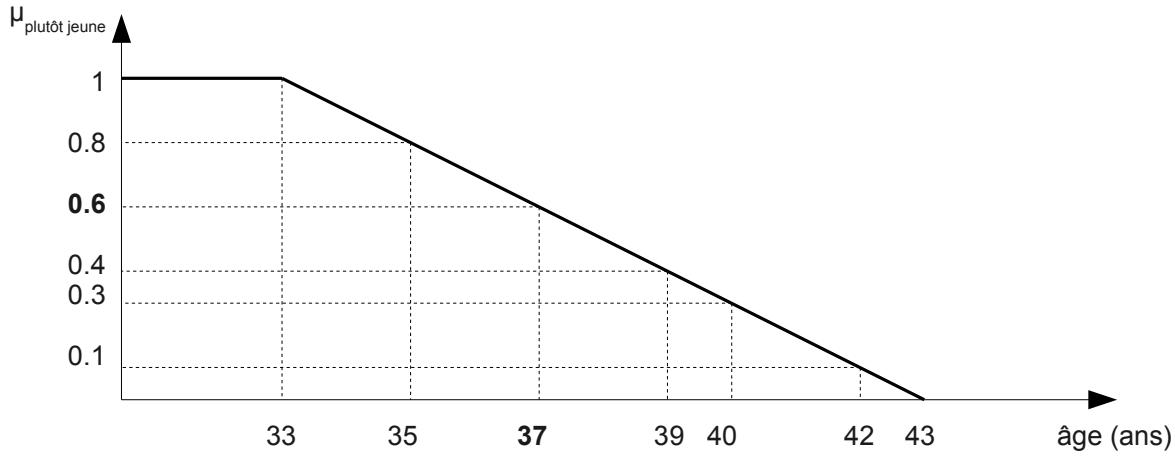


Figure 2.4 : *F.a.t.* du prédicat « plutôt jeune » = (0, 33, 0, 10)

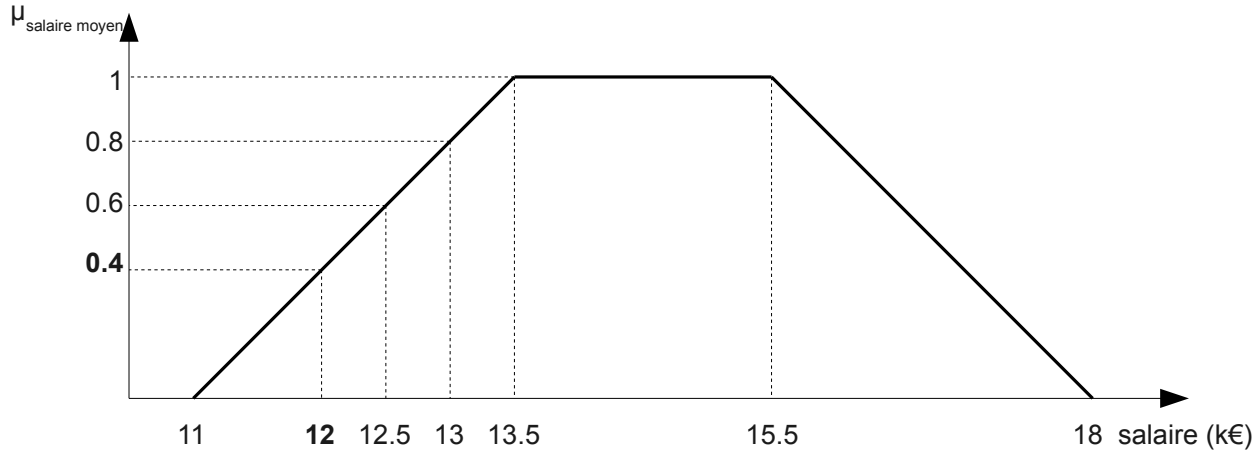


Figure 2.5 : ***F.a.t.*** **du prédicat « salaire moyen » = (13 500, 15 500, 2 500, 2 500)**

Tableau 2.2 : L'extension de la relation *emp-pjsm*

num	***nom***	***salaire***	***age***	***ville***	***degré***
17	Dupont	13000	42	Lyon	**0.1**
76	Martin	12500	40	New-York	**0.3**
26	**Ghani**	**12000**	**37**	**Alger**	**0.4**
12	Smith	12000	39	Londres	**0.4**
55	Lucas	13000	35	Miami	**0.8**

Exemple 2 : Considérons une base de données contenant les relations *employé*(*num*, *nom*, *salaire*, *âge*, *dép*) et *département*(*nd*, *budget*) décrivant les employés et le département où ils travaillent. La requête recherchant les départements ayant un budget *moyen* et aucun employé *jeune* ayant un salaire *très élevé* peut être formulée comme suit :

differ(project(select(*département*, *budget* = "moyen"), *nd*),

project(select(*employé*, *âge* = "jeune" ∧ *salaire* = "très élevé"), *dep*).

Pour ces prédicats nous utiliserons les fonctions d'appartenance trapézoïdales suivantes (*figures 2.6 à 2.8*) :

« *budget moyen* » = (190, 260, 50, 50),

« *âge jeune* » = (0, 38, 0, 5),

« *salaire très élevé* » = (14750, +∞, 2500, +∞)

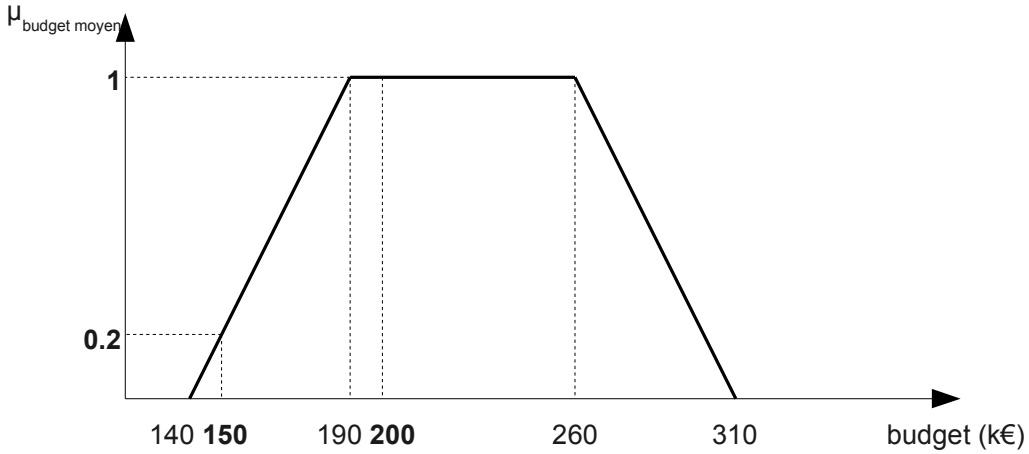


Figure 2.6 : *F.a.t.* du prédicat « *budget moyen* » = (190, 260, 50, 50)

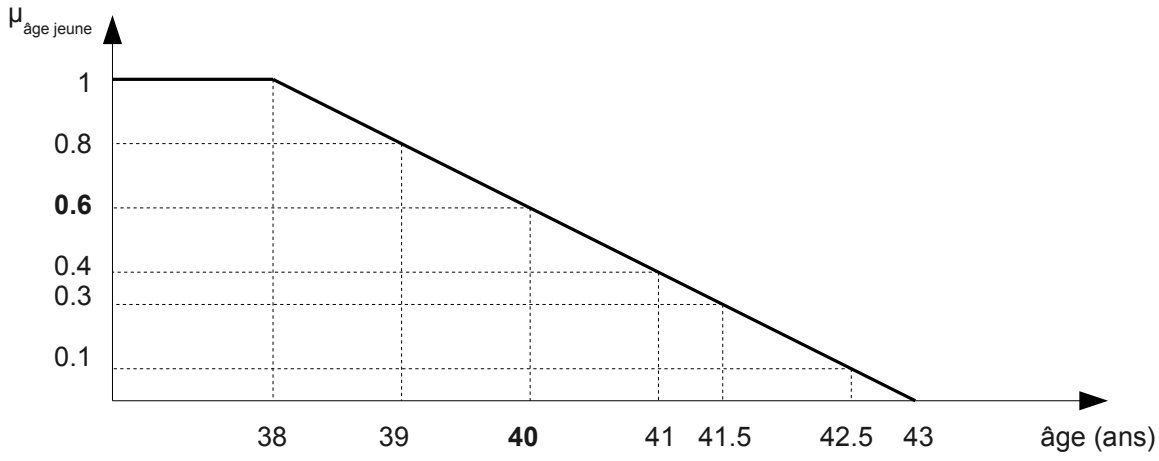


Figure 2.7 : *F.a.t.* du prédicat « *âge jeune* » = (0, 38, 0, 5)

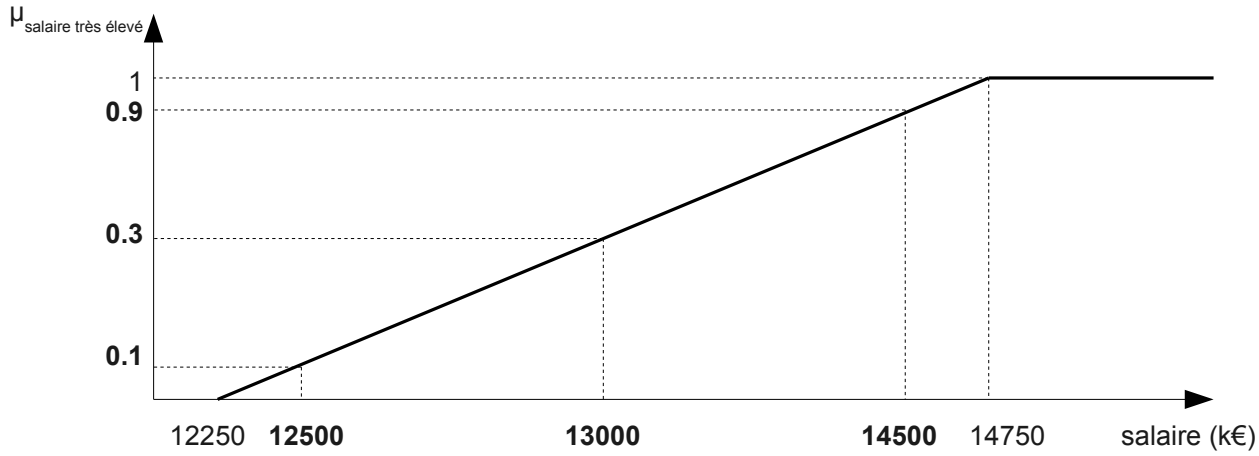


Figure 2.8 : ***F.a.t.*** **du prédicat « *salaire très élevé* » = (14750, +∞, 2500, +∞)**

Dans les extensions données dans les *tableaux 2.3 et 2.4*, le poids rattaché à l'attribut *salaire* (resp. *âge*, *budget*) exprime à quel degré le salaire (resp. l'âge, le budget) est *moyen* (resp. *jeune*, *très élevé*). En utilisant le minimum pour opérateur norme, le résultat obtenu avec ces extensions est le suivant :

{ 1/17, 0.2/23 } – { 0.1/17, 0.3/23 } = { 0.9/17, 0.2/23 }

Tableau 2.3 : Une extension de la relation *employé*

nom	***salaire***	***âge***	***dép***
Smith	12 500 (0.1)	38 (1)	23
Arnold	14 500 (0.9)	42 (0.2)	23
Willy	13 000 (0.3)	37 (1)	23
Durand	12 500 (0.1)	39 (0.8)	17

Tableau 2.4 : Une extension de la relation *département*

nd	***budget***
17	200 (1)
23	150 (0.2)

2.4 Le langage d'interrogation floue SQLf

2.4.1 Langages d'interrogation floue

La base formelle par l'extension de l'algèbre relationnelle étant établie, il est nécessaire, pour la mettre en pratique, de définir un langage de requête. Pour les bases de données actuelles, SQL est le plus souvent utilisé, c'est pourquoi les principaux travaux pour développer de tels langages ont étendu SQL pour le support des requêtes floues.

Une classification détaillée des langages de requêtes flexibles (basés ou non sur SQL) a été établie par [Rosado *et al.* 06]. Elle divise principalement ces langages en deux catégories selon qu'ils s'adressent à des bases de données floues ou à des bases de données classiques. Parmi ces langages, on retiendra les deux *plus connus* [Galindo 08] :

FSQL : Une équipe de l'université de Málaga dirigée par J. Galindo a mis au point ce langage qui est une extension de SQL permettant d'écrire des conditions flexibles dans les requêtes adressées à une base de données relationnelle floue.

SQLf : Le projet Pilgrim à l'université de Rennes 1 a parmi ses objectifs de développer le langage SQLf [Bosc, Pivert 95] présenté dans la suite de ce chapitre.

2.4.2 Le bloc de base

Les trois clauses **select**, **from** et **where** du bloc de base de SQL sont maintenus dans SQLf et la forme de la clause **from** reste inchangée. Les principales différences affectent principalement deux aspects :

1. **le calibrage du résultat ;** ce dernier étant discriminant, il est effectué selon le nombre de réponses désirées (n), le degré de satisfaction minimal (t) ou les deux ;

2. la nature des conditions autorisées, qui peuvent être booléennes et/ou floues

La syntaxe du bloc de base s'exprime donc ainsi :

select [**distinct**] [*n* | *t* | *n*, *t*] *attributs* **from** *relations* **where** *cond-floues*

Cette expression est interprétée comme :

1. la **sélection** floue du produit cartésien des relations apparaissant dans la clause **from ;**

2. une **projection** sur les attributs de la clause **select** ; les éléments dupliqués sont maintenus par défaut, sauf si **distinct** est spécifié, ce qui aura pour effet de ne garder que le degré maximal ;

3. le **calibrage** du résultat (*n* premiers éléments et/ou ceux dont le score est supérieur au seuil *t*)

Exemple 3 : Soit les relations *emp* et *dep* de schémas respectifs *E*(*emp#, nom-e, salaire, poste, âge, ville, dép*) et *D*(*dep#*, *nom-d*, *budget*, *lieu*).

Tableau 2.5 : Extension de la relation *dep*

dep	***dep#***	***nom-d***	***budget (k€)***	***lieu***
	1	réseaux	5 000 (1)	Lyon
	3	composants	1 500 (0.65)	Lyon
	5	services	400 (0.1)	Paris

Tableau 2.6 : Extension de la relation *emp*

emp	*emp#*	*nom-e*	*salaire*	*poste*	*âge*	*ville*	*dép*
	17	Dupont	3 500	ingénieur	51 (0)	Lyon	3
	76	Martin	3 000	ingénieur	40 (0.25)	Paris	5
	26	Durant	2 000	secrétaire	24 (1)	Lyon	3
	12	Dubois	2 500	technicien	39 (0.3)	Lyon	3
	55	Lorant	3 500	comptable	30 (0.75)	Lyon	1

Tableau 2.7 : Résultat de la requête de l'exemple 3

nom-e	*degré*
Durant	0.65
Lorant	0.75

La requête recherchant les noms des *jeunes* employés travaillant dans un département à *haut budget* avec un degré de satisfaction *supérieur à 0.6* s'écrira :

select distinct 0.6 *nom-e* **from** *emp, dep*

where *dep* = *dep#* **and** *âge* = "jeune" **and** *budget* = "haut".

Considérons les f.a.t. suivantes pour ces prédicats (*figures 2.9 et 2.10*) :

« *jeune* » = (0, 25, 0, 20)

« *haut* » = (200, 2200, 2000, +∞)

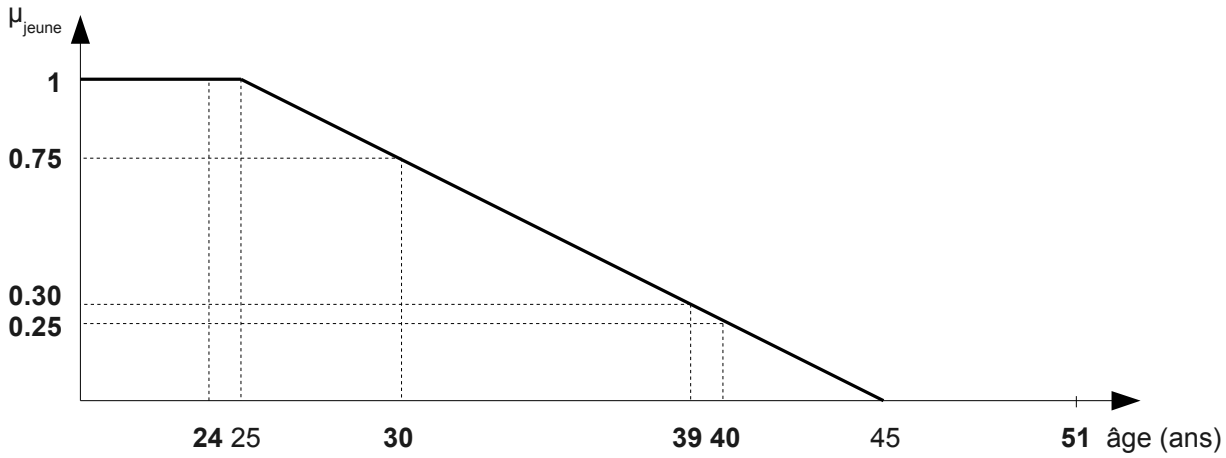


Figure 2.9 : *F.a.t.* du prédicat « *jeune* » = (0, 25, 0, 20)

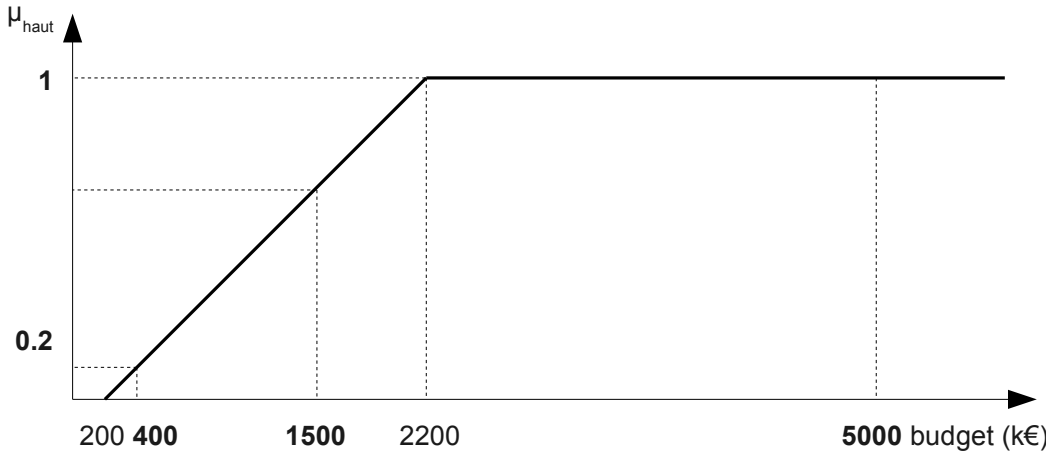


Figure 2.10 : *F.a.t.* du prédicat « *haut* » = (2200, +∞, 2000, +∞)

Avec les extensions des relations *emp* et *dep* données aux *tableaux 2.5 et 2.6* (où les degrés obtenus avec chaque condition floue apparaissent entre parenthèses), le résultat de la requête est celui du *tableau 2.7* puisque ces deux employés sont les seuls à obtenir un degré de satisfaction supérieur à 0.6.

2.4.3 Sous-requêtes

Au delà des prédicats simples précédents, SQLf permet l'utilisation de prédicats basés sur l'imbrication d'un autre prédicat qui joue un rôle de sous-requête. Ce type de construction est souvent utilisé pour simplifier l'expression d'appels de requêtes dans un bloc multi-relation.

Les opérateurs utilisés pour introduire une sous-requête sont principalement :

i) appartenance (**in**),

ii) « non-vacuité » (**exists**)

iii) quantificateurs universel et existentiel avec un opérateur de comparaison (**θ any** et **θ all**).

Ainsi, dans le contexte du prédicat « *A* **in** (**select** *B* **from** …) **»,** l'opérateur **in** retourne le degré d'appartenance de la valeur de *A* dans le n-uplet courant du bloc externe à l'ensemble (flou) de valeurs de *B* retournées par le bloc interne.

Exemple 4 : La requête de l'exemple 3 peut également être exprimée en utilisant une sous-requête ainsi :

select 0.6 *nom-e* **from** *emp* **where** *âge* = "jeune" **and** *dep* **in**

(**select** *dep#* **from** *dep* **where** *budget = "haut*")

Avec la table *dep* donnée précédemment, le bloc interne retourne l'ensemble flou :

{1 / 1, 0.65 / 3, 0.1 / 5}

et le résultat final est celui obtenu dans l'exemple 3 : 0.65 (= min (1, 0.65)) pour Durant et 0.75 (= min (0.75, 1)) pour Lorant.

Dans le même esprit, le prédicat « **exists** (**select** ...) » exprime à quel point le résultat délivré par la sous-requête n'est pas vide. Il a été prouvé dans [Bosc, Pivert 95] que la plupart des équivalences valides en SQL entre les blocs multi-relation et les sous-requêtes tiennent toujours dans SQLf.

Il est également possible de construire des prédicats en utilisant les quantificateurs existentiel et universel appliqués à des sous-requêtes retournant un ensemble flou de valeurs. Afin de maintenir les équivalences usuelles, « *A* θ **any** (**select** *B* **from** *s* **where** *cond-floue*) » doit être équivalente à

« **exists** (**select** * **from** *s* **where** *A* θ *B* **and** *cond-floue*) » ainsi qu'à « *A* **in** (**select** *B* **from** *s* **where** *A* θ *B* **and** *cond-floue*) ». C'est également vrai pour « *A* θ **all** (**select** *B* **from** *s* where *cond-floue*) » qui doit être équivalente à « **not exists** (**select** * **from** *s* **where not** (*A* θ *B*) **and** *cond-floue*) ». Ces exigences fixent la sémantique naturelle de ces quantificateurs dans le contexte des requêtes floues. Il faut noter que sous certaines conditions, il est également possible d'introduire d'autres quantificateurs (en particulier les graduels) au lieu de « **any** » et « **all** ».

2.4.4 Évaluation : principe de dérivation

SQLf étant un langage d'interrogation destiné aux bases de données relationnelles non-floues, il est nécessaire de faire subir à la requête flexible une transformation qui permettra son évaluation. Il s'agit de traduire la requête floue en une requête booléenne, cette dérivation devant tenir compte du seuil (le degré de satisfaction minimal *t*) défini dans la requête.

Le processus de transformation nécessite l'application de règles de dérivation dont le principe consiste à considérer le résultat recherché comme l'α-coupe appliquée au niveau du seuil ($\alpha = t$) sur le résultat de la requête graduelle initiale. Ce résultat devra être obtenu en n'utilisant que des opérations et expressions booléennes. La difficulté réside dans la distribution de l'α-coupe sur les différents composants de la requête qui peut être plus ou moins complexe.

Le principe de dérivation, présenté dans [Bosc, Pivert 95b] est tel que le résultat obtenu par l'évaluation de la requête dérivée est dans la majorité des cas le même que celui de l'α-coupe désirée, mais il est souvent un sur-ensemble plus grand, dont il faudra *a posteriori* éliminer les tuples indésirables. Le surcoût engendré par l'obtention puis l'élimination d'éléments superflus a motivé l'étude expérimentale de l'efficacité de cette méthode, qui s'est révélée satisfaisante pour les requêtes simples [Bosc, Pivert 95b].

L'inconvénient de cette méthode de dérivation réside dans son incapacité à traiter les requêtes floues autres que celles de type projection-sélection-jointure. Quand elle ne peut être réécrite en une telle requête, une autre approche présentée par [Liétard *et al.*, 06] peut être envisagée : la *compilation* de la requête floue. Celle-ci consiste à traduire non pas directement la requête en une requête booléenne mais en un programme faisant appel à des requêtes SQL. Le programme utilise des heuristiques pour optimiser les accès aux données et s'occupe du calcul des degrés de satisfaction. C'est pourquoi aucun post-traitement n'est nécessaire dans cette méthode. Elle est donc particulièrement utile pour le traitement des requêtes floues complexes (requêtes imbriquées notamment).

Chapitre 3 :

Traitement des requêtes à réponses vides dans un cadre flexible

3.1 Introduction

Lors d'un processus de recherche et de récupération des données désirées dans de grandes bases de données, en particulier celles accessibles via le web, les utilisateurs peuvent être confrontés au problème des *réponses vides*, c'est-à-dire que les requêtes soumises retournent un ensemble vide de réponses. Dans ces cas, les désirs des utilisateurs seraient de trouver des réponses alternatives aux requêtes infructueuses initialement posées. Une approche coopérative qui pourrait permettre de fournir de telles réponses est appelée *relaxation*. La relaxation de requêtes [Gaasterland *et al.* 92] [Gaasterland 97] vise à étendre la portée d'une requête en relaxant les contraintes qui y sont impliquées.

Dans le contexte des requêtes flexibles, le problème de réponse vide peut toujours apparaître. Cela signifie qu'aucune donnée disponible dans la base de données ne *satisfait quelque peu* la requête de l'utilisateur. Bosc *et al.* [08] font partie des rares auteurs traitant ce problème. Leurs travaux dans ce domaine visent principalement à relaxer les exigences floues incluses dans la requête infructueuse de l'utilisateur. Ceci peut être réalisé par l'application d'une transformation à certaines ou à toutes les conditions élémentaires de cette requête. Ils proposent une approche de relaxation des requêtes floues basée sur une *relation de tolérance* particulière modélisée par une *proximité relative* paramétrée par un indicateur de tolérance. Cette notion de proximité est destinée à définir un ensemble de prédicats qui sont proches, sémantiquement parlant, d'un prédicat donné *P*.

L'approche ci-dessus applique de manière itérative le mécanisme de relaxation défini sur les prédicats de la requête infructueuse en question et conduit à un treillis de requêtes modifiées. Ce processus de relaxation incrémentiel s'arrêtera quand la réponse à l'une des requêtes résultantes n'est pas vide ou quand une condition sémantique est enfreinte (dans ce cas, les requêtes modifiées obtenues sont sémantiquement éloignées de la requête infructueuse initiale).

Dans ce chapitre, on étudiera d'une manière approfondie la relaxation des requêtes flexibles basée sur l'utilisation d'une relation de tolérance. On discutera le principe de cette approche dans le cas des requêtes atomiques puis dans le cas des requêtes complexes. Le contrôle du processus de relaxation est abordé afin que les requêtes obtenues à l'issue de ce processus soient sémantiquement proches de la requête initiale.

3.2 Présentation du problème

Soit Q une requête flexible et $\sum_Q$ l'ensemble de réponses initialement retournées lors de sa soumission au système de gestion de bases de données. L'ensemble flou $\sum_Q$ contient les éléments satisfaisant quelque peu les conditions floues imposées dans Q et chacun d'eux a donc un degré de satisfaction strictement positif. Soit $\sum^*_Q$ le sous-ensemble de $\sum_Q$ constitué de ses éléments totalement satisfaisants, donc s'étant vus attribuer le degré maximal 1.

La réponse à Q est vide lorsque $\sum_Q$ ne contient aucune réponse, ce qui signifie qu'aucune donnée de la base interrogée ne satisfait un tant soit peu les conditions de Q.

Ce problème est connu dans la littérature sous le nom de Problème des Réponses Vides (PRV).

3.3 Approches proposées

La plupart des approches proposées pour traiter le PRV sont fondées sur un mécanisme de relaxation. L'idée consiste alors à relaxer la requête dans un sens à préciser, par exemple dans le cadre booléen en remplaçant une condition par une de ses généralisations qui est donc moins contraignante [Motro 86], ou en éliminant certaines conditions de la requête [Godfrey 97].

Dans le cas d'une requête flexible, le principe précédent s'applique également et il s'agit alors d'effectuer une relaxation modifiant les conditions de la

requête et permettant d'obtenir une variante moins restrictive. Cette modification est réalisée en appliquant une transformation élémentaire à un ou plusieurs des prédicats flous de la requête.

Nous avons retenu l'approche basée sur la relation de tolérance mais nous verrons par la suite que d'autres approches ont été proposées dans des travaux existants.

3.4 Relation de tolérance

Une *relation de tolérance* (ou *relation de proximité*) est une relation floue R sur un domaine scalaire U telle que pour $u, v \in U$,

$$i)\ \mu_R(u, u) = 1 \ \textit{(reflexivité)} \qquad (ii)\ \mu_R(u, v) = \mu_R(v, u) \ \textit{(symétrie)}$$

La quantité $\mu_R(u, v)$ évalue la proximité entre les éléments u et v. Il existe deux façons de mesurer la proximité entre u et v :

- **i)** soit l'on évalue à quel point la différence $u - v$ est *proche* de 0 (proximité *absolue*),
- **ii)** soit l'on évalue à quel point le rapport u/v est *proche* de 1 (proximité *relative*).

Dans le cadre de cette étude, c'est la seconde interprétation qui sera considérée. Pour plus de détails sur la première interprétation, le lecteur pourra consulter [Bosc *et al.* 07].

Proximité relative [Hadjali *et al.* 03]. L'idée de proximité relative ou de voisinage relatif *(relative closeness*) qui exprime une égalité approximative entre deux nombres réels x et y peut être capturée par la relation suivante :

$$\mu_{Cl}(x, y) = \mu_M(x/y), \tag{3.1}$$

où M, nommé *paramètre de tolérance*, est un nombre flou modélisant *« proche de 1 »*, tel que :

i) $\mu_M(1) = 1$: puisque x est proche de x ;

ii) $\mu_M(t) = 0$ si $t \leq 0$: deux nombres qui sont proches doivent avoir le même signe ;

iii) $\mu_M(t) = \mu_M(1/t)$ (*i.e.*, $M = 1/M$) : ceci garantit la propriété de symétrie de Cl, *i.e.* $\mu_{Cl}(x, y) = \mu_{Cl}(y, x)$.

De la propriété iii), nous pouvons déduire que le support $S(M)$ du paramètre M a la forme suivante : $S(M) = [1 - \varepsilon, 1/(1 - \varepsilon)]$ où ε est un nombre réel. De là, en termes de *f.a.t.*, M peut être représenté par $(1, 1, \varepsilon, \varepsilon/(1 - \varepsilon))$ comme le montre la *figure 3.1.*

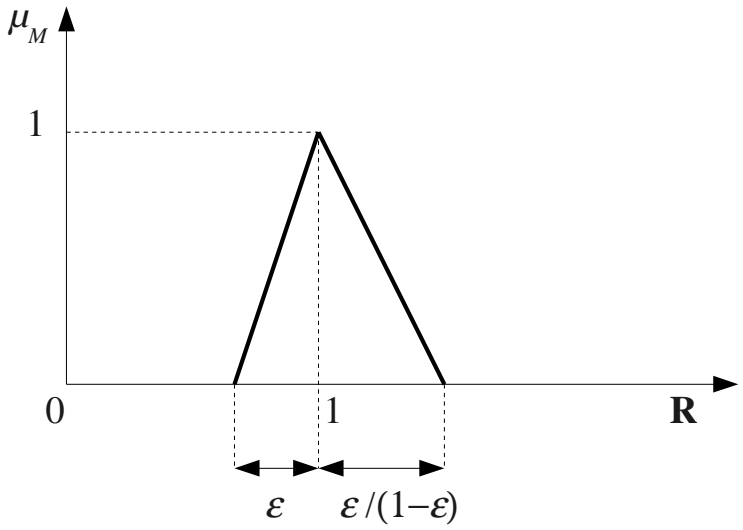


Figure 3.1 : fonction d'appartenance du paramètre de tolérance $M = (1, 1, \varepsilon, \varepsilon/(1 - \varepsilon))$

Il a été démontré dans [Hadjali *et al.* 03] que le nombre flou M qui paramètre la proximité (ainsi que la relation de négligibilité, Ne, définie par $\mu_{Ne[M]}(x, y) = \mu_{Cl[M]}(x+y, y)$) doit être choisi afin que son support $S(M)$ reste dans l'intervalle de validité[13] $V = [(\sqrt{5} - 1)/2, (\sqrt{5} + 1)/2]$. Cela signifie que si le support d'un paramètre de tolérance associé à une relation de proximité Cl n'est pas inclus

13 L'hypothèse suivante est appliquée: si x est proche de y alors ni x n'est négligeable par rapport à y, ni y n'est négligeable par rapport à x. Alors, l'intervalle V est la solution de l'inéquation $\mu_{Cl[M]}(x, y) \leq 1 - max(\mu_{Ne[M]}(x, y), \mu_{Ne[M]}(y, x))$.

dans V, alors la relation Cl n'est pas en accord avec la sémantique intuitive sous-jacente à cette notion.

De manière assez intéressante, l'intervalle de validité V jouera un rôle clé dans le processus de relaxation de requêtes proposé. Comme il sera démontré plus loin, il constitue la base pour définir un critère d'arrêt pour le contrôle de la relaxation.

3.5 Relaxation incrémentielle

Dans cette section, nous introduirons d'abord *une approche basée sur la tolérance* pour relaxer une requête à un seul prédicat (SP). Ensuite, nous discuterons la stratégie de relaxation dans le cas des requêtes conjonctives. Cette section est principalement empruntée de [Bosc *et al.* 08].

3.5.1 Requêtes SP

a. Principe de l'approche

Relaxer une requête flexible infructueuse consiste à modifier les contraintes incluses dans la requête afin d'obtenir des variantes moins restrictives. Une telle modification peut être accomplie par l'application d'une *transformation de base* sur tous ou une partie des prédicats de la requête infructueuse. Certaines propriétés désirables sont requises pour toute transformation $T^{\uparrow}$ appliquée sur un prédicat P ($T^{\uparrow}(P)$ représentant le prédicat modifié) :

i) $T^{\uparrow}$ ne diminue le degré d'appartenance d'aucun élément du domaine ;

$$\forall u \in U, \quad \mu_{T\uparrow(P)}(u) \geq \mu_P(u)$$

ii) $T^{\uparrow}$ doit étendre le support $S(P)$ du prédicat P,

$$S(P) = \{u / \mu_P(u) > 0\} \subset S(T^{\uparrow}(P)) = \{u / \mu_{T\uparrow(P)}(u) > 0\};$$

iii) $T^{\uparrow}$ préserve la spécificité de P (afin de ne pas altérer sa sémantique de façon significative), c'est-à-dire qu'elle ne modifie pas le noyau $C(P)$

du prédicat *P*.

$$C(P) = \{u \,/\, \mu_P(u) = 1\} = C(T^{\uparrow}(P)) = \{u \,/\, \mu_{T^{\uparrow}(P)}(u) = 1\}$$

Soit $Q = P$ une requête SP et P un prédicat flou. Pour relaxer Q nous remplaçons P par un prédicat flou P' défini comme suit:

$$\forall\, u \in U, \quad \mu_{P'}(u) = sup_{v \in U}\, min\, (\mu_P(v),\, \mu_{Cl[M]}(v,\, u)) = sup_{v \in U}\, min\, (\mu_P(v),\, \mu_M(v/u)).$$

En utilisant le principe d'extension, il est simple de vérifier que $P' = P \otimes M$ où $\otimes$ est l'opération de multiplication étendue aux nombres flous [Bouchon-Meunier 07]. Formellement, la transformation $T^{\uparrow}$ s'écrit :

$$P' = T^{\uparrow}(P) = P \otimes M. \tag{3.2}$$

Il est clair que le prédicat modifié P' contient tous les éléments de P et les éléments en dehors de P qui sont *assez proches* d'un élément de P. $T^{\uparrow}(P)$ résulte alors en un prédicat P' qui est *moins restrictif* que P, mais reste *sémantiquement proche* de P. Il est important de noter que $T^{\uparrow}$ n'est pas simplement un outil technique agissant sur les degrés d'appartenance mais est doté d'une sémantique claire induite par celle sous-jacente à la relation floue *Cl[M]*.

En termes de *f.a.t.*, si $P = (A,\, B,\, a,\, b)$ et $M = (1,\, 1,\, \varepsilon,\, \varepsilon/(1 - \varepsilon))$ où ε représente la *valeur de tolérance relative* appartenant à l'intervalle $[0,\, (3 - \sqrt{5})/2]$ (cet intervalle résulte de l'inclusion $S(M) \subseteq V$ précédemment évoquée, voir [Hadjali *et al.* 03]) et en utilisant la formule (3.2), le prédicat relaxé P', représenté en *figure 3.2*, est tel que :

$$P' = (A,\, B,\, a + \Delta_l(\varepsilon),\, b + \Delta_r(\varepsilon))$$

où $\Delta_l(\varepsilon) = A \cdot \varepsilon$

et $\Delta_r(\varepsilon) = B \cdot \varepsilon/(1 - \varepsilon))$

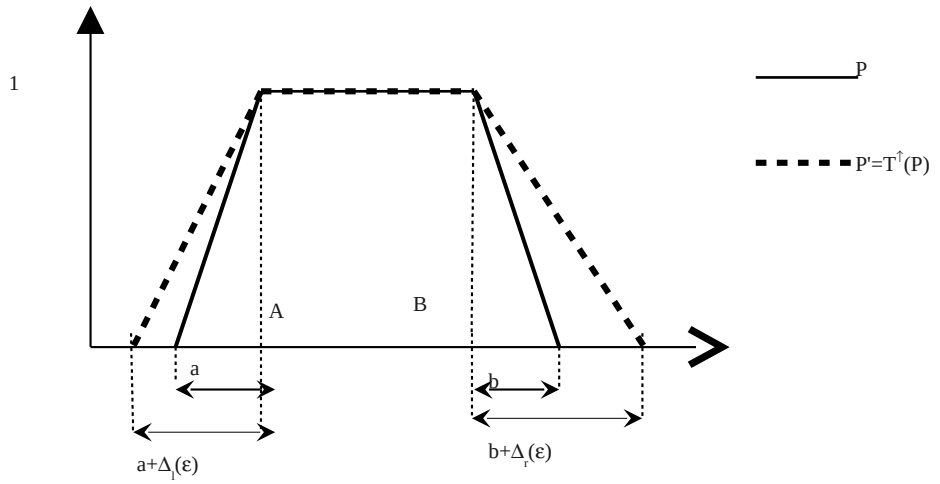


Figure 3.2 : Transformation de base utilisant une relation de proximité

De plus, il est facile de vérifier que les trois propriétés évoquées précédemment restent satisfaites :

i) $\forall\, u,\ \mu_{P'}(u) \geq \mu_P(u)$;

ii) $S(P) \subset S(P')$;

iii) $C(P) = C(P')$.

Une *relaxation maximale*, notée $T^{\uparrow max}(P)$, d'un prédicat P peut être atteinte en utilisant la valeur de tolérance $\varepsilon_{max} = (3 - \sqrt{5})/2$. Donc,

$$T^{\uparrow max}(P) = (A,\ B,\ a + \Delta_l(\varepsilon_{max}),\ b + \Delta_r(\varepsilon_{max})).$$

b. Contrôle du processus de relaxation

En pratique, si $Q = P$ est une requête SP et si l'ensemble de réponses à Q est vide, alors Q est relaxée en la transformant en $Q_1 = T^{\uparrow}(P) = P \otimes M$. Cette transformation peut être répétée n fois jusqu'à ce que la réponse à la requête modifiée $Q_n = T^{\uparrow(n)}(P) = P \otimes M^n$ ne soit pas vide.

La question qui se poserait à l'issue de l'application de cette technique concerne sa limite sémantique, c'est-à-dire le nombre maximal d'itérations de relaxations autorisé pour que la requête finale reste sémantiquement proche de la requête originale. Cette limite est complètement définie par l'intervalle de validité

$\boldsymbol{V}$ de la manière suivante : une requête modifiée Q_n reste sémantiquement proche de l'originale si le support de M^n est inclus dans $\boldsymbol{V}$.

Entrées: $Q := P$: Requête infructueuse initiale
ε : une valeur de tolérance /* $\varepsilon \in [0, (3 - \sqrt{5})/2]$ */

1. $i := 0$; $Q_i := Q$; /* i est le nombre d'étapes de relaxation */
2. calculer Σ_{Qi} ; /* Σ_{Qi} est l'ensemble de réponses à Q_i */
3. **tantque** ($\Sigma_{Qi} = \emptyset$ et $S(M^{i+1}) \subseteq \boldsymbol{V}$) **faire**
4. **début**
5. $i := i+1$;
6. $Q_i := T^{\uparrow(i)}(P)$; /* $T^{\uparrow(i)}(P) = P \otimes M^i$ */
7. calculer Σ_{Qi};
8. **fin**

Sortie: Σ_{Qi} (si $\neq \emptyset$): L'ensemble de réponses non-vides à Q

<u>Algorithme 3.1 :</u> Relaxation incrémentielle d'une requête à un seul prédicat

Remarque : La condition $S(M^{i+1}) \subseteq \boldsymbol{V}$ de l'*algorithme 3.1* est équivalente à $(i+1)\cdot\varepsilon \leq \varepsilon_{max}$ (avec $\varepsilon_{max} = (3 - \sqrt{5})/2$)

c. Propriétés de la transformation

Pour étudier les propriétés de base de la transformation $T^{\uparrow}$ dédiée à la relaxation, nous pouvons identifier trois critères [Bosc *et al.* 08d] :

i) Nature de la transformation. Elle consiste à vérifier si l'effet de la transformation est similaire sur les parties gauche et droite de la f.a.t. associée au prédicat considéré.

ii) Impact du domaine et du prédicat. Ce critère consiste à vérifier si le domaine de l'attribut et la position de la fonction d'appartenance du prédicat dans le référentiel ont un impact sur l'effet de la transformation.

iii) Applicabilité dans le cas non flou. Ce critère vérifie si la transformation est toujours valide dans le cas où les prédicats sont exprimés en termes d'intervalles traditionnels.

Le comportement de la transformation $T^{\uparrow}$ par rapport à ces critères est résumé ci-dessous :

Critère (i) : En observant la f.a.t. du prédicat modifié $T^{\uparrow}(P)$, on voit que l'effet de la relaxation à gauche et à droite du support de P n'est pas de même intensité. En effet, le support du prédicat $T^{\uparrow}(P) = (A, B, a+\Delta_l(\varepsilon), b+\Delta_r(\varepsilon))$ est étendu à gauche de $\Delta_l(\varepsilon) = A \cdot \varepsilon$ et à droite de $\Delta_r(\varepsilon) = B \cdot \varepsilon/(1-\varepsilon)$. Le mécanisme de relaxation résultant de l'application de $T^{\uparrow}$ est asymétrique : il est clair qu'il est plus important du côté droit.

Critère (ii): La position des prédicats sur le référentiel est identifiée comme facteur majeur pouvant affecter la relaxation. En effet, si $P_1 = (A_1, B_1, a_1, b_1)$ et $P_2 = (A_2, B_2, a_2, b_2)$ sont deux prédicats relatifs au même attribut, et si le noyau de P_2 est plus à droite sur le référentiel que celui de P_1 (*i.e.* $A_1 < A_2$ et $B_1 < B_2$), la relaxation entraîne une transformation plus importante sur P_2, car les valeurs de $\Delta_l(\varepsilon)$ et $\Delta_r(\varepsilon)$ sont plus grandes pour A_2 que pour A_1 : $A_2 \cdot \varepsilon > A_1 \cdot \varepsilon$ et $B_2 \cdot \varepsilon/(1-\varepsilon) > B_1 \cdot \varepsilon/(1-\varepsilon)$.

Cependant, le domaine de l'attribut n'a aucun impact sur l'effet de la relaxation (la valeur de ε est indépendante du domaine).

Critère (iii): Il est facile de vérifier que $T^{\uparrow}$ reste encore efficace pour relaxer les requêtes classiques. En effet, si $P = (A, B, 0, 0)$ représente un prédicat non flou, l'application de la transformation donne $T^{\uparrow}(P) = (A, B, A \cdot \varepsilon, B \cdot \varepsilon/(1-\varepsilon))$.

d. Prédicats particuliers

La propriété (i) ci-dessus ne s'applique pas pour la transformation de tous les prédicats. En effet, quand une borne du noyau $C(P)$ coïncide avec une borne du

domaine de l'attribut, la relaxation ne concerne que le côté *opposé* du prédicat. Cela correspond au cas du prédicat *jeune*, dont la fonction d'appartenance débute à la valeur zéro et qui ne peut pas être relaxé de ce côté, l'âge ne pouvant être négatif. C'est également le cas de *bien payé* pour lequel la borne supérieure est en théorie infinie mais correspond en pratique à la limite supérieure du domaine de l'attribut *salaire* : il est évident que la relaxation ne pourra se faire que du côté gauche.

Pour pouvoir relaxer le prédicat uniquement du côté gauche, on utilisera pour paramètre de tolérance, au lieu de $M = (1, 1, \varepsilon, \varepsilon/(1 - \varepsilon))$, le nombre flou de f.a.t. $(1, 1, \varepsilon, 0)$. De la même manière, pour relaxer le prédicat uniquement du côté droit, on utilisera le nombre flou de f.a.t. $(1, 1, 0, \varepsilon/(1 - \varepsilon))$.

e. Exemple illustratif

Soit la relation *Employés* dont l'extension est donnée au *tableau 3.1*. Soit la requête SP « *trouver les employés jeunes* » notée $Q = P$ et composée du prédicat $P =$ "*jeune*", dont la fonction d'appartenance $P = (0, 35, 0, 5)$ est représentée en *figure 3.3*.

Tableau 3.1 : Relation *Employés*

Num	Nom	Âge	Ville habitée
17	Dupont	47	Lyon
76	Martin	45	Boston
26	Tanaka	42	Chiba
12	Smith	44	London
55	Lucas	40	Miami

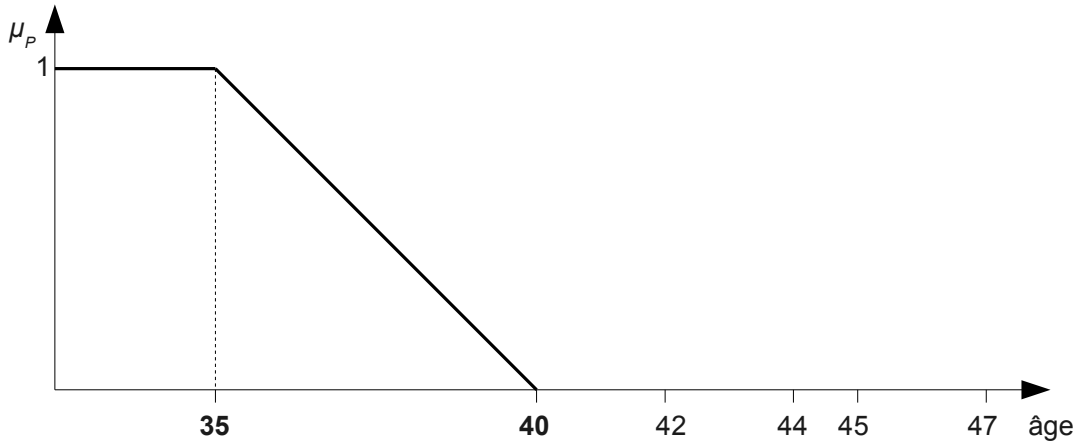


Figure 3.3 : *F.a.t.* du prédicat *P* = "jeune" = (0, 35, 0, 5)

Il est clair que l'application de *P* sur l'extension de la relation donne un résultat vide, puisqu'aucun des tuples ne satisfait la condition du prédicat, même partiellement (leur degré de satisfaction est nul).

Pour obtenir une réponse coopérative (non vide), nous appliquerons l'*algorithme 3.1*. Posons ε = 0.12. On relaxe la requête par l'application de transformations successives $T^{\uparrow(i)}$ sur le prédicat *P* : $Q_i = T^{\uparrow(i)}(P) = P \otimes M^i$

Notons que, puisque la relaxation ne doit évidemment pas être effectuée du côté gauche (il s'agit d'un prédicat particulier, comme expliqué précédemment), le paramètre de tolérance que nous utiliserons dans ce cas est le nombre flou $M = (1, 1, 0, \varepsilon/(1 - \varepsilon))$.

$Q_1 = T^{\uparrow}(P)$ avec $T^{\uparrow}(P) = P \otimes M = (0, 35, 0, 5 + \Delta_r(\varepsilon)) = (0, 35, 0, 9.77)$

car $\Delta_r(\varepsilon) = 35 \times 0.12 / (1 - 0.12)) = 4.77$

L'application sur la relation *employés* de cette requête relaxée, constituée du prédicat modifié *P'* dont la fonction d'appartenance est illustrée en *figure 3.4*, donne les degrés d'appartenance montrés dans le *tableau 3.2*.

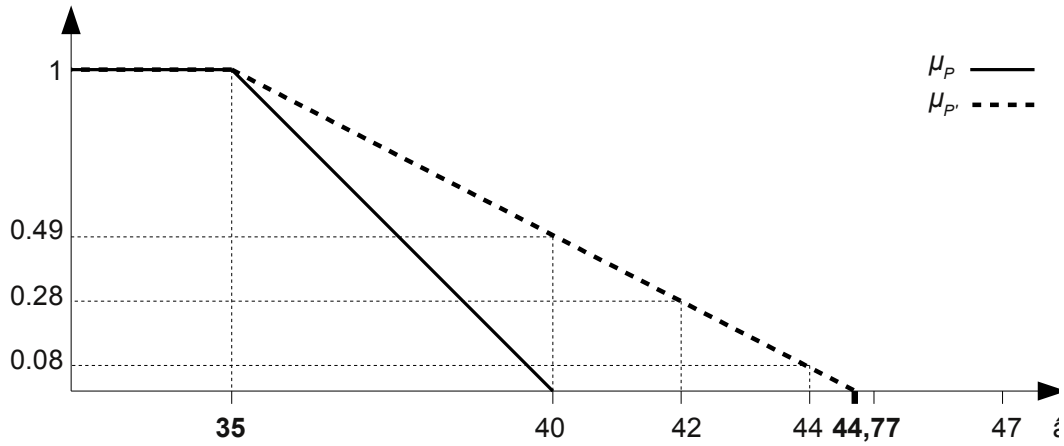


Figure 3.4 : ***F.a.t.*** **du prédicat relaxé** P' **= (0, 35, 0, 9.77)**

Tableau 3.2 : Degrés d'appartenance à la condition de la requête relaxée Q_1

Num	Nom	Âge	Ville habitée	$\mu_{Q1}(t)$
17	Dupont	47	Lyon	0
76	Martin	45	Boston	0
26	Tanaka	42	Chiba	0.28
12	Smith	44	London	0.08
55	Lucas	40	Miami	0.49

Une seule itération aura été suffisante dans ce cas, et le résultat renvoyé à l'utilisateur dans l'ordre de préférence décroissant (déterminé par le degré d'appartenance) sera : Lucas, Tanaka et Smith.

3.5.2 Requêtes conjonctives flexibles

Une requête flexible conjonctive Q est de la forme $P_1 \wedge \ldots \wedge P_N$, où le symbole '$\wedge$' correspond au connecteur *'et'* et est interprété par l'opérateur '*min*', et chaque P_i est un prédicat flou.

a. Stratégie de relaxation

Étant donné une requête $Q = P_1 \wedge \ldots \wedge P_N$, deux stratégies peuvent être envisagées pour le processus de relaxation :

i) *Une modification globale de la requête*, qui consiste à appliquer uniformément la transformation de base T sur tous les prédicats de la requête. L'ensemble des requêtes transformées résultant de l'application de T est $\left\{T^{\uparrow(i)}(P_1)\wedge\ldots\wedge T^{\uparrow(i)}(P_N)\right\}$, où $i > 0$ et $T^{\uparrow(i)}$ signifie que la transformation $T^{\uparrow}$ est appliquée i fois. Cette stratégie est simple mais ne répond pas totalement à nos besoins qui sont de trouver la requête transformée la plus proche.

ii) *Une modification locale de la requête :* ce genre de transformation affecte seulement certains prédicats (ou sous-requêtes). Elle est plus efficace que la stratégie globale, car dans la majorité des cas seule une partie des conditions de la requête infructueuse est la cause de son échec.

Nous utiliserons la deuxième stratégie dans la suite de ce travail.

Soit un ensemble de transformations $\left\{T_1^{\uparrow},\ldots,T_N^{\uparrow}\right\}$. L'ensemble des requêtes modifiées correspondant à Q résultant de l'application de ces transformations est $\left\{T_1^{\uparrow(i_1)}(P_1)\wedge\ldots\wedge T_N^{\uparrow(i_N)}(P_N)\right\}$, où $i_h \geq 0$ et $T_j^{\uparrow(i_h)}$ signifie que la transformation $T_j^{\uparrow}$ est appliquée i_h fois à P_j.

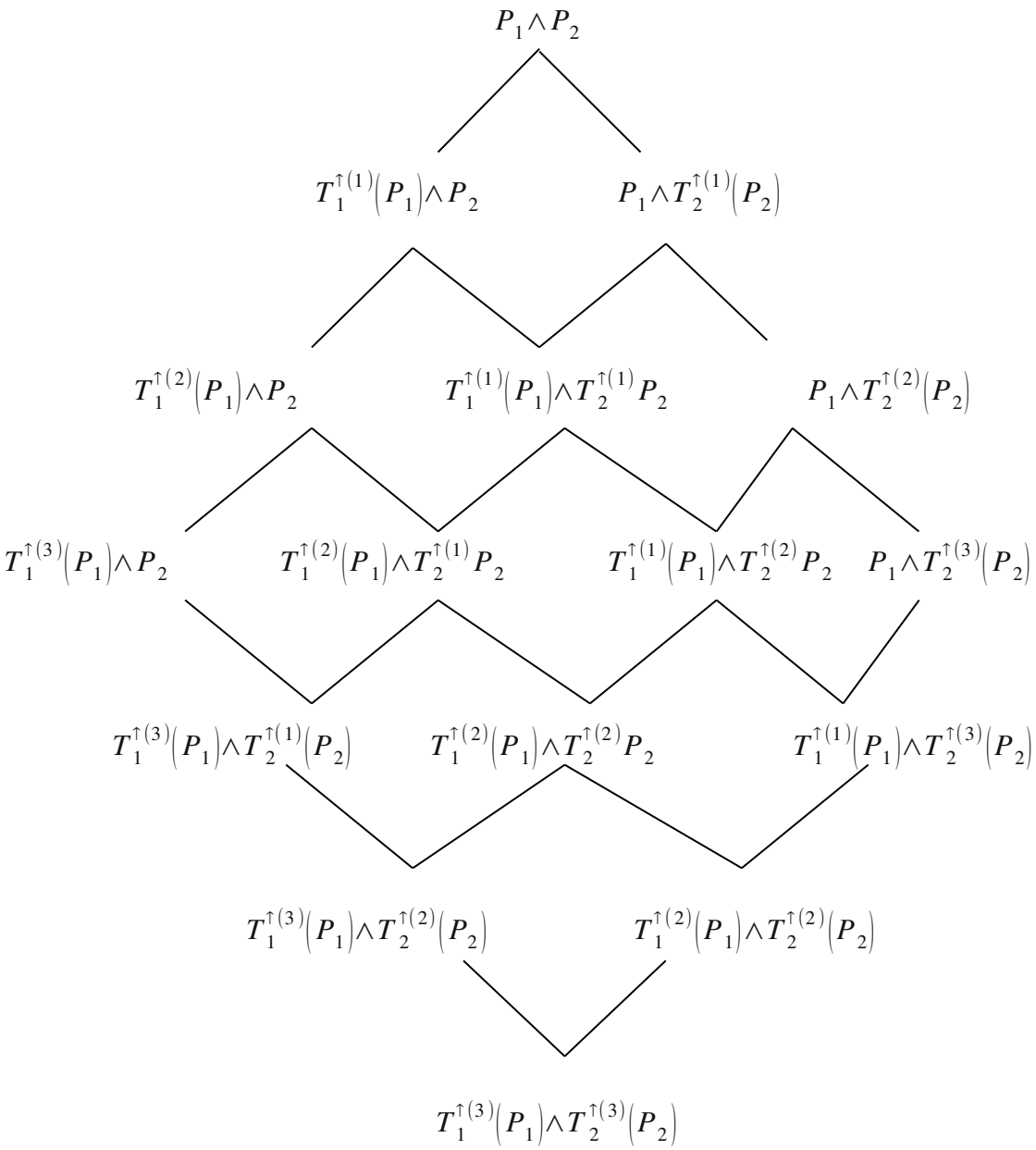
Comme souligné dans [Bosc *et al.* 08], un ordre ($\prec$) peut être défini sur l'ensemble des requêtes modifiées. Cet ordre peut être exprimé sur la base du nombre d'applications de la transformation associée à chaque prédicat. Si Q' et Q'' sont deux requêtes relaxées de Q, nous dirons que

$$Q' \prec Q'' \;\; si \;\; \sum_{i=1}^{N} \text{nombre}\left(T_i^{\uparrow} \text{ dans } Q'\right) < \sum_{i=1}^{N} \text{nombre}\left(T_i^{\uparrow} \text{ dans } Q''\right)$$

Comme précédemment avec les requêtes SP, chaque prédicat P_i peut être relaxé au maximum en $T_i^{\uparrow(\max)}(P_i)$. Donc, la requête modifiée donnée par

$$T^{\uparrow(\max)}(Q) = T_1^{\uparrow(\max)}(P_1)\wedge\ldots\wedge T_N^{\uparrow(\max)}(P_N)$$

peut être considérée comme la relaxation maximale d'une requête $Q=P_1 \wedge \ldots \wedge P_N$. Cette borne est directement inhérente aux limites sémantiques fournies par la transformation basée sur la proximité. À présent, supposons que chaque prédicat dans Q puisse être relaxé au plus ω fois. Alors, le nombre d'étapes de relaxation autorisé est $\omega \times N$. L'ensemble des requêtes modifiées correspondant à Q (i.e., $\left\{T_1^{\uparrow(i_1)}(P_1) \wedge \ldots \wedge T_N^{\uparrow(i_N)}(P_N)\right\}$) peut être organisé selon une structure de treillis. L'avantage de l'approche adoptée est le fait qu'elle conduit à un *treillis borné* de requêtes relaxées. Par exemple, le treillis associé à l'affaiblissement de la requête $P_1 \wedge P_2$ est donné en *figure 3.5*.



<u>Figure 3.5 :</u> Treillis borné de requêtes relaxées (pour ω = 3)

La mise en œuvre de cette stratégie de relaxation doit tenir compte au moins des trois aspects suivants :

1. Définir un moyen d'exploiter le treillis de requêtes relaxées de manière intelligente, son parcours pouvant se révéler fastidieux quand le nombre de prédicats/relaxations augmente,
2. Afin de rester cohérente, la relaxation des requêtes doit agir sur les différents prédicats dans des proportions équivalentes, pour éviter qu'un prédicat ne soit relaxé plus rapidement qu'un autre au fil des étapes ; c'est la propriété de l'*Égalité de l'Effet de Relaxation* (ÉER)
3. Définir le rôle de l'utilisateur dans le processus de relaxation : jusqu'à quelle limite devra-t-il être impliqué dans ce processus ?

La première question fera l'objet de la section suivante. Quant aux deux autres, elles dépendent étroitement de la définition de la propriété de l'ÉER prise en compte.

Propriété de l'Égalité de l'Effet de Relaxation (ÉER)

Bien que plusieurs façons de définir cette propriété puissent être discutées, nous ne retiendrons que celle qui utilise les rapports entre les longueurs des supports respectifs des prédicats flous relaxés et initiaux. Pour un prédicat P_i, ce rapport correspond à :

$$\Delta_i\left(P_i, T_i^{\uparrow}(P_i)\right) = \frac{L\left(T_i^{\uparrow}(P_i)\right)}{L(P_i)}$$

où $L(P_i)$ et $L(T_i^{\uparrow}(P_i))$ sont les longueurs des supports de P_i et $T_i(P_i)$ respectivement.

$$\Delta_i\left(P_i, T_i^{\uparrow}(P_i)\right) = \frac{B_i - A_i + a_i + b_i + \Delta_l^i(\varepsilon_i) + \Delta_r^i(\varepsilon_i)}{B_i - A_i + a_i + b_i} = 1 + \frac{A_i\varepsilon_i + B_i\varepsilon_i/(1-\varepsilon_i)}{B_i - A_i + a_i + b_i}$$

La propriété de l'ÉER *stipule que les rapports entre les longueurs des supports doivent être les mêmes quels que soient les prédicats, i.e.:*

$$\Delta_1\left(P_1, T_1^{\uparrow}(P_1)\right) = \ldots = \Delta_N\left(P_N, T_N^{\uparrow}(P_N)\right)$$

D'autre part, pour que le support des prédicats relaxés puisse rester dans l'intervalle de validité ***V***, il faudra que le plus grand des ε_i (et *a fortiori* les autres) vérifie les inéquations :

$$\begin{cases} \omega\Delta_l^i(\varepsilon_i) \leqslant \Delta_l^i(\varepsilon_{max}) \\ \omega\Delta_r^i(\varepsilon_i) \leqslant \Delta_r^i(\varepsilon_{max}) \end{cases}$$

Dans un but de clarté, nous ne considérerons que le côté gauche, ce qui donne

$$\omega\Delta_l^i(\varepsilon_i) \leqslant \Delta_l^i(\varepsilon_{max}) \Leftrightarrow \omega A_i\varepsilon_i \leqslant A_i\varepsilon_{max} \Leftrightarrow \omega\varepsilon_i \leqslant \varepsilon_{max} \Leftrightarrow \varepsilon_i \leqslant \varepsilon_{max}/\omega$$

Comme on ne peut savoir *a priori* quel est le prédicat qui aura le plus grand ε_i, on procède par itérations selon la procédure suivante :

1) considérer d'abord un prédicat P_s au hasard, disons le premier ($s = 1$),

2) attribuer à ε_s la valeur ε_{max}/ω ,

3) calculer pour chacun des autres prédicats P_i ($i \in [1, N] - \{s\}$) la valeur de ε_i grâce à la propriété de l'ÉER,

4) si cette valeur dépasse ε_{max}/ω, recommencer en considérant le prédicat P_i cette fois : prendre $s = i$ et retourner en **2)**.

Au final, cette procédure nous permettra de trouver les valeurs de ε_i pour tous les prédicats P_i vérifiant la propriété de l'ÉER sans qu'aucune relaxation maximale ne fasse dépasser le support en dehors de l'intervalle de validité ***V***. Elle est présentée plus en détail dans l'*algorithme 3.2*.

Entrées : $Q = P_1 \wedge \ldots \wedge P_N$: requête infructueuse initiale
ω : nombre maximal de relaxations par prédicat

1. ε_{max} := (3 − √5)/2 ;
2. s := 1 ;
3. *terminé* := **faux** ;
4. **tantque non** *terminé* **faire**
5. **début**
6. $\varepsilon_s := \varepsilon_{max}/\omega$;
7. *i* := 1 ;
8. *terminé* := **vrai** ;
9. **tantque** *i* <= *N* **et** *terminé* = **vrai faire**
10. **début**
11. **si** *i* <> *s* **alors**
12. en utilisant la propriété de l'ÉER, calculer ε_i ;
13. **si** $\omega.\varepsilon_i > \varepsilon_{max}$ **alors** *s* := *i* ; *terminé* := **faux** ;
14. **finsi** ;
15. i := i + 1 ;
16. **fin** ;
17. fin.

Sortie : Ensemble des ε_i ($i = 1 .. N$), valeurs de tolérance pour chaque P_i

<u>Algorithme 3.2 :</u> Calcul des valeurs de tolérance ε_i

Le calcul utilisant la propriété de l'ÉER est détaillé ci-dessous :

$$\Delta_s\left(P_s, T_s^{\uparrow}(P_s)\right) = \Delta_i\left(P_i, T_i^{\uparrow}(P_i)\right)$$

$$1+\frac{A_s\varepsilon_s + B_s\varepsilon_s/(1-\varepsilon_s)}{B_s - A_s + a_s + b_s} = 1+\frac{A_i\varepsilon_i + B_i\varepsilon_i/(1-\varepsilon_i)}{B_i - A_i + a_i + b_i}$$

$$\frac{(A_s\varepsilon_s + B_s\varepsilon_s/(1-\varepsilon_s))(B_i - A_i + a_i + b_i)}{B_s - A_s + a_s + b_s} = \frac{A_i\varepsilon_i(1-\varepsilon_i) + B_i\varepsilon_i}{(1-\varepsilon_i)}$$

posons $\boldsymbol{K_i} = \dfrac{(A_s\varepsilon_s + B_s\varepsilon_s/(1-\varepsilon_s))(B_i - A_i + a_i + b_i)}{B_s - A_s + a_s + b_s}$ alors :

$$K_i(1-\varepsilon_i) = A_i\varepsilon_i - A_i\varepsilon_i^2 + B_i\varepsilon_i$$

$$A_i\varepsilon_i^2 - (A_i + B_i + K_i)\varepsilon_i + K_i = 0$$

$$\varepsilon_i = \frac{A_i + B_i + K_i - \sqrt{(A_i + B_i + K_i)^2 - 4 A_i K_i}}{2 A_i} \quad \text{ou} \quad \varepsilon_i = \frac{A_i + B_i + K_i + \sqrt{(A_i + B_i + K_i)^2 - 4 A_i K_i}}{2 A_i}$$

La pratique nous a montré que seule la première valeur (la plus petite racine de l'équation) convient, l'autre donnant un résultat trop grand.

b. Parcours du treillis : approche basée sur les MFS

Dans ce qui suit, nous montrons comment parcourir le treillis de manière efficace en exploitant les sous-requêtes à réponse vide minimales (*Minimal Failing Subqueries, MFS*) de la requête à réponse vide originale *Q*. Comme nous l'avons vu en *section 1.3.2* (*page 31*), les *MFS* correspondent aux plus petites sous-requêtes de *Q* qui échouent.

Soit $Q = P_1 \wedge \ldots \wedge P_N$ une requête infructueuse, $T^{\uparrow}(Q)$ une requête relaxée de *Q*, et $SQ^{[j]}$ une sous-requête de *Q* obtenue en supprimant le prédicat P_j de *Q*. Soit également $mfs(Q) = \left\{ P_{s_1^1} \wedge \ldots \wedge P_{s_{m_1}^1}, \ldots, P_{s_1^k} \wedge \ldots \wedge P_{s_{m_k}^k} \right\}$ l'ensemble des *MFS* de *Q* avec $\left\{ s_1^h, \ldots, s_{m_h}^h \right\} \subset \{1, \ldots, N\}$ pour $1 \leq h \leq k$. Afin de définir l'ensemble des *MFS* de $T^{\uparrow}(Q)$ par rapport à celles de *Q*, nous introduirons les propositions suivantes.

Proposition 1. Si $T^{\uparrow}(Q) = SQ^{[j]} \wedge T_j^{\uparrow}(P_j)$ avec $j \in \bigcup_{h=1}^{k} \left\{ s_1^h, \ldots, s_{m_h}^h \right\}$ alors les *MFS* de $T^{\uparrow}(Q)$ doivent être recherchées dans *mfs(Q)* en substituant $T_j^{\uparrow}(P_j)$ à P_j dans chaque élément de *mfs(Q)*.

Proposition 2. Si $T^{\uparrow}(Q) = SQ^{[j]} \wedge T_j^{\uparrow}(P_j)$ avec $j \notin \bigcup_{h=1}^{k} \left\{ s_1^h, \ldots, s_{m_h}^h \right\}$ alors l'ensemble des *MFS* de $T^{\uparrow}(Q)$ est l'ensemble *mfs(Q)*.

Exemple : Supposons que $Q = P_1 \wedge P_2 \wedge P_3 \wedge P_4$ et $mfs(Q) = \{P_1 \wedge P_3, P_1 \wedge P_4\}$. Alors,

- si $T^{\uparrow}(Q)=T_1^{\uparrow}(P_1)\wedge SQ^{[1]}=T_1^{\uparrow}(P_1)\wedge P_2\wedge P_3\wedge P_4$, les *MFS* de $T^{\uparrow}(Q)$ sont recherchées dans $\{T_1^{\uparrow}(P_1) \wedge P_3, T_1^{\uparrow}(P_1)\wedge P_4\}$.

- Si $T^{\uparrow}(Q)=SQ^{[2]}\wedge T_2^{\uparrow}(P_2)=P_1\wedge T_2^{\uparrow}(P_2)\wedge P_3\wedge P_4$, Q et $T^{\uparrow}(Q)$ ont les mêmes *MFS*.

Des propositions ci-dessus, il résulte que l'ensemble des *MFS* d'une requête relaxée $T^{\uparrow}(Q)$ peut être obtenu de l'ensemble des *MFS* associés à Q. Donc, en pratique, il suffit de calculer l'ensemble des *MFS* de Q pour déduire les *MFS* de toute requête relaxée $T^{\uparrow}(Q)$. Maintenant, pour rechercher un ensemble de requêtes relaxées fructueuses à travers le treillis, nous utilisons une procédure en deux étapes comme suit :

- **Étape 1 :** *identifier k MFS de Q*

Pour ce faire, nous utilisons l'*algorithme 1.2* qui est conçu pour calculer *k MFS* en un temps acceptable (quand *k* n'est pas trop grand) [Godfrey 97]. D'après les propositions 1 et 2, nous n'exécuterons qu'une seule fois cet algorithme puisque les MFS de toute requête relaxée $T^{\uparrow}(Q)$ sont déduites de celles de Q.

- **Étape 2 :** *technique de recherche intelligente*

Les informations sur les *MFS* permettent d'éviter d'évaluer certains nœuds du treillis. En effet, un nœud du treillis qui préserve au moins une *MFS* de son nœud parent (i.e. le nœud duquel il est dérivé) n'a pas besoin d'être évalué (puisque nous sommes certains qu'il échoue). Cette technique est schématisée dans l'*algorithme 3.3*, où :

- *Niveau(i)* représente l'ensemble de requêtes relaxées au niveau *i* du treillis ;

- *parent(Q')* est l'ensemble des nœuds desquels *Q'* peut être dérivée, i.e., *Q'* est une variante relaxée immédiate de toute requête contenue dans

parent(Q'). Par exemple, en *figure 3.5*, si $Q' = T(P_1) \wedge T(P_2)$, alors *parent(Q')* $= \{T(P_1) \wedge P_2, P_1 \wedge T(P_2)\}$.

- *évaluer(Q')* est une fonction qui évalue Q' sur la base de donnée cible. Elle retourne *vrai* si Q' produit des réponses non-vides et *faux* autrement.

Entrées : $Q = P_1 \wedge \ldots \wedge P_N$: une requête infructueuse

$mfs(Q) = \left\{ P_{s_1^1} \wedge \ldots \wedge P_{s_{m_1}^1}, \ldots, P_{s_1^k} \wedge \ldots \wedge P_{s_{m_k}^k} \right\}$: ens. des MFS de Q

1. *Liste_Relaxées* = ∅; *i* := 1;
2. **tantque** ($i \leq \omega.N$) **et** (*Liste_Relaxées* = ∅) **faire**
3. **début**
4. *Niveau*(*i*) = { $Q_i^1, \cdots, Q_i^{n_i}$ };
5. **pour** *relaxée* **dans** *Niveau*(*i*) **faire**
6. **début**
7. *modif* := **vrai**;
8. **pour** *une_mfs* **dans** *mfs*(*parent*(*relaxée*)) **faire**
9. *modif* := *modif* **et** (*une_mfs* ∉ *relaxée*) ;
10. **si** *modif* **alors**
11. **si** *évaluer*(*relaxée*) **alors**
12. *Liste_Relaxées := Liste_Relaxées* ∪ {*relaxée*};
13. **finsi;**
14. **finsi;**
17. **fin**;
18. *i* := *i* + 1;
19. **fin**;
20. **si** *Liste_Relaxées* ≠ ∅ **alors retourner** *Liste_Relaxées*;
21. **finsi;**

Sortie: *Liste_Relaxées*: un ensemble de req. relaxées fructueuses associées à *Q*

<u>Algorithme 3.3 :</u> Rechercher un ensemble de requêtes relaxées fructueuses.

Malgré l'élagage du treillis effectué grâce au calcul des *MFS*, les évaluations de requêtes peuvent rester nombreuses avant d'arriver au résultat final. Afin de diminuer ce temps d'évaluation, on pourra commencer tout d'abord par

évaluer la requête qui constitue le nœud terminal du treillis, c'est-à-dire celle correspondant à une relaxation maximale $T^{\uparrow(\max)}(Q)$, puis garder localement les résultats et n'effectuer les évaluations suivantes que sur ce sous-ensemble local de la base de données distante.

Comme on peut le voir, l'*algorithme 3.3* retourne un ensemble (i.e., *Liste_Relaxées*) de requêtes relaxées fructueuses associées à la requête initiale *Q*. Selon l'ordre défini précédemment (p.71), les éléments de *Liste_Relaxées* sont incomparables. En effet, ils résultent de l'application du même nombre de transformations : $\forall Q', Q'' \in$ *Liste_Relaxées*, nous avons

$$\sum_{i=1}^{N} \text{nombre}\left(T_i^{\uparrow} \text{ dans } Q'\right) = \sum_{i=1}^{N} \text{nombre}\left(T_i^{\uparrow} \text{ dans } Q''\right).$$

Puisque la sémantique est l'un de nos points de départ (notre but final est de trouver une requête relaxée qui produit des réponses non-vides et qui est la requête la plus proche de la requête initiale *Q,* sémantiquement parlant), la question qui se pose à présent est comment sélectionner la meilleure relaxation de *Q* parmi les éléments de *Liste_Relaxées ?* Pour y répondre, nous proposons un ordre basé sur une mesure de proximité sémantique induite par la distance de Hausdorff entre les requêtes. Le principe de cet ordre sera détaillé dans le chapitre qui suit.

3.6 Exemple illustratif

Nous illustrons ici le principe de relaxation d'une requête conjonctive. L'exemple proposé, inspiré de [Bosc *et al.* 08d][14], concerne une relation d'employés décrite par deux attributs *salaire* et *âge*. Le contenu de cette relation est donné par le *tableau 3.3*.

14 Il s'agit du même exemple que celui de Bosc *et al.*, qui, contrairement à nous, y utilisent une relation de proximité *absolue* (citée en section 3.4, p.61). Le lecteur pourra s'y référer pour comparer les deux approches.

Tableau 3.3 : Relation des employés

Nom	**Salaire** (k€)	**Age**	$\mu_{P1}(u)$	$\mu_{P2}(v)$
Dupont	2	48	1	0
Martin	1.7	46	0.4	0
Durant	1.3	45	0	0
Jones	1.2	37	0	0
Smith	1	34	0	0

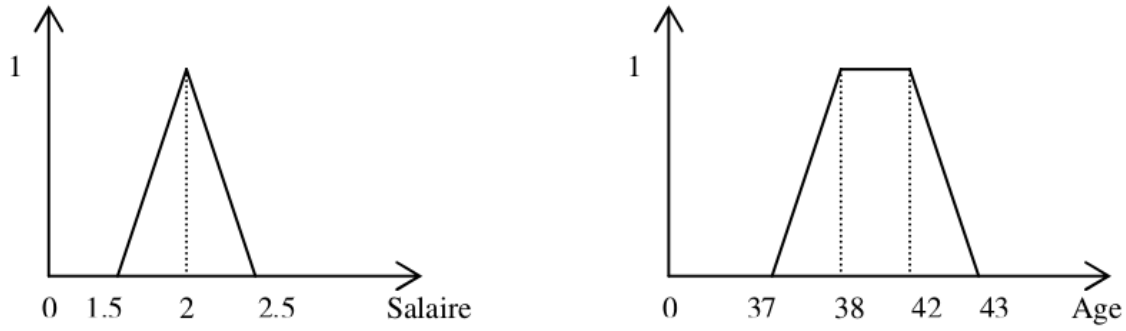
Supposons qu'une personne soit intéressée par la recherche des employés dont le salaire est *approximativement égal à 2 k€* et dont l'âge est *aux environs de 40 ans*.

La requête à formuler est de la forme $Q = P_1 \wedge P_2$ avec P_1 = « *approximativement 2* » et P_2 = « *aux environs de 40* ».

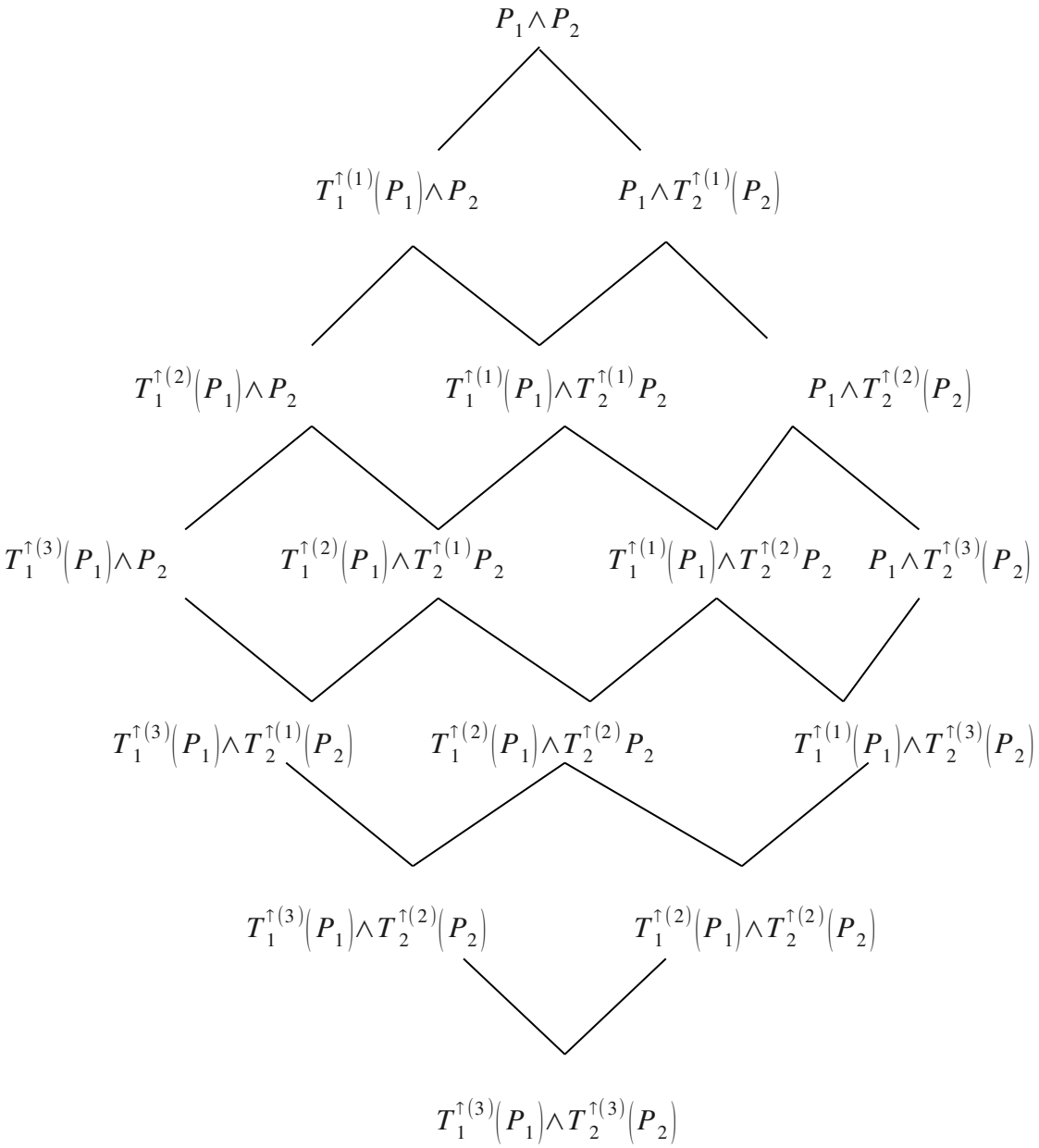
En termes de f.a.t., $P_1 = (2, 2, 0.5, 0.5)$ et $P_2 = (38, 42, 1, 1)$, voir *figure 3.6*. D'après le contenu de la table, aucun employé ne satisfait *un tant soit peu* les deux conditions de la requête Q. Pour tenter de répondre à cette requête, nous procédons à sa relaxation.

Supposons que la valeur de ω, le nombre maximal de transformations applicables à chaque prédicat, soit fixée à 3. Calculons les valeurs de tolérance ε_1 et ε_2 en appliquant l'*algorithme 3.2 :*

- Posons $\varepsilon_1 = \varepsilon_{max} / \omega = ((3 - \sqrt{5})/2) / 3 \approx 0.12$
- Grâce à la propriété de l'ÉER, on déduit $\varepsilon_2 = 0.04$
- $\varepsilon_2 \omega = 0.12 < \varepsilon_{max}$: on garde donc ces valeurs.



<u>Figure 3.6 :</u> Prédicats graduels P_1 = « *approximativement 2* »
et P_2 = « *aux environs de 40* ».



<u>Figure 3.7 :</u> Treillis borné de requêtes relaxées (pour ω = 3)

Il est facile de voir que $mfs(Q) = \{P_2\}$. L'application de l'*algorithme 3.3* dédié à la recherche d'une requête relaxée à réponse non vide dans le treillis (voir *Figure 3.7*), est détaillée ci-dessous :

- $i = 1$, niveau 1 :
  - Le nœud $Q_{1,1} = T_1^{\uparrow}(P_1) \wedge P_2$ n'est pas évalué car $mfs(Q_{1,1}) = mfs(Q) = \{P_2\}$.
  - Le nœud $Q_{1,2} = P_1 \wedge T_2^{\uparrow}(P_2)$ est évalué (car $Q_{1,2}$ ne garde aucune MFS de Q). Le nœud ne retourne aucune réponse comme indiqué dans le *tableau 3.4* (où $T_2^{\uparrow}(P_2) = (38, 42, 2.52, 2.75)$).
- $i = i+1 = 2$, niveau 2 :
  - Le nœud $Q_{2,1} = T_1^{\uparrow(2)}(P_1) \wedge P_2$ n'est pas évalué car $mfs(Q_{2,1}) = mfs(Q_{1,1}) = \{P_2\}$.
  - Le nœud $Q_{2,2} = T_1^{\uparrow}(P_1) \wedge T_2^{\uparrow}(P_2)$ est évalué car $mfs(Q_{2,2}) = \emptyset$. $Q_{2,2}$ retourne aussi une réponse vide. Voir *tableau 3.5* (où $T_1^{\uparrow}(P_1) = (2, 2, 0.74, 0.77)$).
  - Le nœud $Q_{2,3} = P_1 \wedge T_2^{\uparrow(2)}(P_2)$ est évalué car $mfs(Q_{2,3}) = \emptyset$. La requête $Q_{2,3}$ retourne comme réponse l'employé Martin avec un degré de satisfaction égal à 0.1. Voir *tableau 3.6* (où $T_2^{\uparrow(2)}(P_2) = (38, 42, 4.04, 4.50)$).

Tableau 3.4.

Nom	**Salaire** (k€)	**Age**	$\mu_{P1}(u)$	$\mu_{T2\uparrow(P2)}(v)$	$\mu_{Q1,2}(t) = \min(\mu_{P1}(u), \mu_{T2\uparrow(P2)}(v))$
Dupont	2	48	1	0	0
Martin	1.7	46	0.4	0	0
Durant	1.3	45	0	0	0
Jones	1.2	37	0	0.6	0
Smith	1	34	0	0	0

Tableau 3.5.

Nom	**Salaire** (k€)	**Age**	$\mu_{T1\uparrow(P1)}(u)$	$\mu_{T2\uparrow(P2)}(v)$	$\mu_{Q2,2}(t)$ $=\min(\mu_{T1\uparrow(P1)}(u), \mu_{T2\uparrow(P2)}(v))$
Dupont	2	48	1	0	0
Martin	1.7	46	0.6	0	0
Durant	1.3	45	0.05	0	0
Jones	1.2	37	0	0.6	0
Smith	1	34	0	0	0

Tableau 3.6.

Nom	**Salaire** (k€)	**Age**	$\mu_{P1}(u)$	$\mu_{T2\uparrow(2)(P2)}(v)$	$\mu_{Q2,3}(t)$ $= \min(\mu_{P1}(u), \mu_{T2\uparrow(2)(P2)}(v))$
Dupont	2	48	1	0	0
Martin	1.7	46	0.4	0.1	0.1
Durant	1.3	45	0	0.3	0
Jones	1.2	37	0	0.8	0
Smith	1	34	0	0.01	0

Ainsi, l'*algorithme 3.3* se termine et nous obtenons *Liste_Relaxées* = $\{Q_{2,3}\}$, $Q_{2,3} = P_1 \wedge T_2^{\uparrow(2)}(P_2)$ (où $T_2^{\uparrow(2)}(P_2)$ = (38, 42, 4.04, 4.50)) étant la seule requête fructueuse obtenue par 2 applications de transformations de prédicats (niveau 2 du treillis). L'employé Martin est donc retourné à l'utilisateur comme réponse à sa requête, avec un degré d'appartenance de 0.1.

Notons que cette fois-ci, Liste_Relaxées ne comportait qu'une requête, ce qui simplifie le résultat. Ce n'est cependant pas toujours le cas. Comme nous l'avons dit, le chapitre suivant traite le cas où il faut choisir parmi les requêtes de même niveau obtenues.

3.7 Travaux apparentés

Les travaux concernant la relaxation des requêtes floues ne sont pas très nombreux. En dehors de la solution que nous avons retenue, nous citerons trois approches existantes :

3.7.1 Approche du ν-rather

Andreasen et Pivert [Andreasen, Pivert 94] ont proposé une approche où la transformation se base sur un modificateur linguistique particulier, appelé *ν-rather*. Pour illustrer cette approche, considérons d'abord un prédicat flou P représenté par (A, B, a, b). L'élargissement de P peut être accompli en lui appliquant un modificateur linguistique expansif. Ainsi, dans le cas du modificateur *ν-rather*, nous obtenons (T_ν désignant une transformation basée sur ce modificateur) :

$$P' = T_\nu\ (P) = \nu\text{-}rather(P) = (A, B, a/\nu, b/\nu),$$

avec $\nu \in [1/2, 1[$.

Les propriétés vues précédemment dans notre approche pour les requêtes SP (section 3.5.1, page 63) sont également satisfaites avec la transformation basée sur ce modificateur :

i) $\forall u, \mu_{P'}(u) \geq \mu_P(u)$;

ii) $\mathcal{S}(P) \subset \mathcal{S}(P')$;

iii) $\mathcal{C}(P) = \mathcal{C}(P')$

Par ailleurs, en notant $\Delta_l(\nu)$ (respectivement $\Delta_r(\nu)$) l'effet de relaxation du côté gauche (respectivement du côté droit), on a :

$$\Delta_l(\nu) = \theta \cdot a, \qquad \Delta_r(\nu) = \theta \cdot b$$

avec $\theta = (1 - \nu)/\nu \in]0, 1]$.

Comme on peut le voir, les effets de relaxation résultants des côtés droit et gauche sont obtenus en utilisant le même paramètre θ. C'est pourquoi l'approche est dite *quasi-symétrique* (elle est symétrique si a = b).

Il est intéressant de noter que la transformation T_ν est une simple opération technique, dénuée de toute sémantique. Un autre inconvénient de cette approche est le fait qu'elle ne fournit aucune limite sémantique intrinsèque pour le contrôle du processus de relaxation.

De plus, cette approche échoue totalement quand elle est employée avec des requêtes non floues puisque ν-*rather(P)* = *P* si *P* est un prédicat booléen.

3.7.2 Approche SaintEtiq

Dans [Voglozin *et al.* 05], les auteurs considèrent des requêtes flexibles adressées à des résumés de données et proposent une méthode basée sur une distance spécifiée pour réparer les requêtes ayant échoué. Si aucun résumé ne correspond à une requête *Q*, des requêtes alternatives sont générées en modifiant un ou plusieurs *descripteurs* (prédicats) flous associés à *Q*. Cela nécessite un ordre préétabli sur les domaines d'attributs considérés puisqu'un descripteur est remplacé par le plus proche. Les requêtes résultantes sont ordonnées selon leur proximité avec l'originale (mesurée grâce à la distance spécifiée).

Cette mesure de proximité basée sur une distance est discutable puisqu'elle ne considère pas la proximité relative entre deux descripteurs. De plus, cette méthode ne fournit aucun critère pour arrêter le processus de modification si la réponse reste toujours vide.

3.7.3 Approche de la plateforme PRETI

Citons également le travail effectué dans la plateforme PRETI [De Calmès *et al.* 03] qui inclut un module d'interrogation flexible. La requête *Q* de l'utilisateur qui exprime la recherche d'une maison à louer implique un ensemble de profils de préférences. Chaque profil P_i est modélisé au moyen d'ensembles flous. Si aucune réponse n'est retournée à l'utilisateur, il est possible d'éviter de telles réponses vides en lui fournissant les réponses les plus proches. L'idée est

que chaque condition élémentaire se rapportant à un domaine ordonné est équipée d'un *profil de préférences par défaut* P'_i tel que P'_i coïncide avec P_i sur les valeurs où $\mu_{Pi}(t) = 1$ et est non-nul autre part sur tout le domaine de l'attribut (plus t' est proche de l'ensemble $\{t / \mu_{Pi}(t) = 1\}$, plus $\mu_{P'i}(t')$ est grand). Ainsi, en parcourant la base de données, on peut progressivement garder trace de la plus proche réponse dans le sens de P'_i, pour toutes les situations où l'un des attributs les plus importants n'est pas du tout satisfait, et calculer le degré d'acceptabilité limité aux attributs restants.

Chapitre 4 :
Recherche des meilleures relaxations

Dans l'approche que nous avons présentée au chapitre précédent, le mécanisme de relaxation défini sur la requête infructueuse est appliqué de manière itérative afin de conduire à un treillis de requêtes modifiées, ordonné par niveaux. Le treillis est parcouru niveau par niveau, jusqu'à en atteindre un comportant des requêtes à réponse non vide ou jusqu'à atteindre la borne au-delà de laquelle la sémantique de la requête initiale n'est plus préservée. Chaque niveau correspond au nombre d'applications des transformations, qui détermine un ordre des requêtes utile du point de vue sémantique.

Toutefois, quand le niveau atteint comporte plusieurs requêtes relaxées fructueuses, il n'est fourni aucun moyen de choisir parmi cette liste de requêtes de même ordre la meilleure relaxation de la requête initiale, c'est-à-dire la requête la plus proche de celle-ci, sémantiquement parlant. Nous tenterons donc de combler ce manque en proposant une mesure de proximité sémantique entre les requêtes. Cette mesure se base sur une distance entre les ensembles, appelée distance de Hausdorff. En parcourant le treillis des requêtes modifiées, la meilleure relaxation (dans le sens ci-dessus) qui produit des réponses non-vides peut être retournée en utilisant cette mesure de distance.

4.1 La distance de Hausdorff

4.1.1 Notion de distance

Une distance d est une application de $U \times U$ vers $\mathbf{R}^+$, telle que :

i) $\forall\, u, \forall\, v,\ d(u, v) = d(v, u)$

ii) $\forall\, u, \forall\, v, \forall\, w,\ d(u, w) \leq d(u, v) + d(v, w)$

iii) $\forall\, u, \forall\, v$, si $u \neq v$ alors $d(u, v) > 0$

iv) $\forall\, u,\ d(u, u) = 0$

L'une des mesures de distance les plus répandues est la distance euclidienne, définie telle que $d(u, v) = |u - v|$.

Une relation de similarité est une application de $U \times U$ vers [0, 1], telle que :

1) $\forall\, u,\ s(u, u) = 1$ (Réflexivité)

2) $\forall\, u, \forall\, v,\ s(u, v) = s(v, u)$ (Symétrie)

3) $\forall\, u, \forall\, v, \forall\, w,\ s(u, w) \geq T(s(u,v), s(v, w))$ (T-Transivité où T est une t-norme)

Le degré $s(u, v)$ évalue la *proximité* ou *similarité* entre u et v. Quand seules 1) et 2) sont satisfaites, s est appelé *relation de proximité*. Si nous posons $d = 1 - s$, alors d est une mesure de distance *normalisée* (appelée *semi-pseudométrique*).

4.1.2 Distance de Hausdorff

Nous rappelons ici le principe de ce type de distance et nous examinons une approche qui peut être suivie pour calculer une telle mesure.

a. Ensembles classiques

Considérons deux sous-ensembles A et B d'un espace U (équipé d'une métrique). L'extension scalaire de distance entre A et B la plus répandue est la *distance de Hausdorff* définie par [Dubois, Prade 83][Puri, Ralescu 83] :

$$d_H(A, B) = max\ \{H(A, B), H(B, A)\}, \qquad (4.1)$$

où $H(A, B)$ représente la distance de Hausdorff relative (ou orientée) entre A et B.

On a :

$$H(A, B) = Sup_{u \in A}\ d(u, B) \text{ et } d(u, B) = Inf_{v \in B}\ d(u, v).$$

L'expression $d(u, v)$ représente une distance standard (telle que la distance euclidienne). La formule (4.1) peut être écrite sous la forme compacte suivante :

$$d_H(A, B) = max\ \{sup_{u \in A}\ inf_{v \in B}\ d(u, v), sup_{v \in B}\ inf_{u \in A}\ d(u, v)\}. \qquad (4.1')$$

L'idée qui régit cette distance est la suivante : pour chaque élément dans A, rechercher l'élément le plus proche de B, puis vérifier l'élément de A pour lequel la distance au plus proche élément de B est maximale. On procède de même en échangeant B et A et la distance *la plus longue* des deux composantes est gardée.

Comme souligné dans [Chaudhuri, Rosenfeld 99] la distance de Hausdorff peut être vue comme une mesure estimant à quel point deux ensembles, non-vides compacts (fermés et bornés), A et B dans un espace métrique se ressemblent à l'égard de leurs positions. Notons que d_H est une métrique et l'assertion suivante est satisfaite :

$$d_H(A, B) = 0 \text{ si et seulement si } A = B.$$

Les égalités suivantes sont également supposées être vraies :

$$d_H(A, \varnothing) = d_H(\varnothing, B) = +\infty \quad \text{et} \quad d_H(\varnothing, \varnothing) = 0.$$

Exemple : Soit $A = [a_1, a_2]$ et $B = [b_1, b_2]$ deux intervalles ordinaires et soit $d(u, v) = |u - v|$. Alors, on a $d_H(A, B) = max(|a_1 - b_1|, |a_2 - b_2|)$.

b. Ensembles flous

La distance de Hausdorff entre des ensembles flous peut être soit floue soit scalaire. Ici nous nous intéressons uniquement à la version scalaire. Pour l'évaluation floue, plus de détails sont disponibles dans [Chaudhuri, Rosenfeld 99] [Dubois, Prade 83].

Des distances scalaires entre des ensembles flous qui présentent de bonnes propriétés peuvent être définies en fusionnant les valeurs $\{d_H(F_\alpha, G_\alpha), \alpha \in (0, 1]\}$[15]. Par exemple, *Puri et Ralescu* ont proposé les indices suivants [Puri, Ralescu 83]:

$$d_H^{\infty}(F, G) = sup\{d_H(F_\alpha, G_\alpha), \alpha \in (0, 1]\}; \tag{4.2}$$

$$d_H^{1}(F, G) = \int_0^1 d_H\left(F_\alpha, G_\alpha\right) d\alpha \tag{4.3}$$

15 Rappelons que F_α représente la coupe de niveau α de F, i.e., $\{u \in U / \mu_F(u) \geq \alpha\}$.

Le seul inconvénient majeur des indices (4.2) et (4.3) est le fait qu'ils sont limités aux ensembles flous ayant des valeurs d'appartenance *maximales* égales. Dans notre cas, cet inconvénient est exclu. En effet, tous les ensembles flous qui sont considérés sont normalisés (donc avec le maximum de valeur d'appartenance égal à 1).

Exemples

i) avec des ensembles flous continus :

Soit *U* un univers de discours numérique de la variable « *âge* » d'une personne. Soit également *F* = « *environ trente* » et *G* = « *entre 26 et 28* » deux sous-ensembles flous de U définis par les deux *f.a.t.* : *F* = (30, 30, 3, 3) ; *G* = (26, 28, 1, 1). *À* présent, évaluons la distance entre *F* et *G* en utilisant la formule (4.3). D'abord, précisons que F_α et G_α sont des intervalles et peuvent être exprimés comme suit : $F_\alpha = [3\alpha + 27, 33 - 3\alpha]$; $G_\alpha = [\alpha + 25, 29 - \alpha]$. Ensuite, on peut facilement vérifier que

$$d_H^1(F, G) = \int_0^1 \max\left(\left|(\alpha+25)-(3\alpha+27)\right|, \left|(29-\alpha)-(33-3\alpha)\right|\right) d\alpha = 7/2.$$

ii) avec des ensembles flous discrets :

Soit *U* = {1, 2, 3, 4, 5, 6, 7} un univers de discours. Soient également *F* et *G* deux sous-ensembles flous de *U* définis comme suit :

F = {0.7/1, 0.2/2, 0.6/4, 0.5/5, 1/6} et *G* = {0.2/1, 0.6/4, 0.8/5, 1/7}.

Regroupons les valeurs distinctes prises par les fonctions d'appartenance de *F* et *G* ensemble : $\{\alpha_1 = 0.2, \alpha_2 = 0.5, \alpha_3 = 0.6, \alpha_4 = 0.7, \alpha_5 = 0.8, \alpha_6 = 1\}$. Pour calculer la distance $d_H^\infty(F, G)$, nous devons d'abord évaluer $d_H(F_{\alpha i}, G_{\alpha i})$ pour *i* = 1, 2,..., 6.

Détaillons le calcul de $d_H(F_{\alpha_1}, G_{\alpha_1})$. En utilisant la formule (4.1), on obtient :

$$d_H(F_{\alpha_1}, G_{\alpha_1}) = max\left\{H(F_{\alpha_1}, G_{\alpha_1}), H(G_{\alpha_1}, F_{\alpha_1})\right\}$$

avec $H(A,B) = \inf_{u\in A} \sup_{v\in B} d(u,v)$,

$F_{\alpha_1} = \{u1, u2, u3, u4, u5\}$ = {1, 2, 4, 5, 6},

et $G_{\alpha_1} = \{v1, v2, v3, v4\}$ = {1, 4, 5, 7}.

D'où :

$H(F_{\alpha_1}, G_{\alpha_1}) = \sup_{u\in F_{\alpha_1}} \inf_{v\in G_{\alpha_1}} d(u,v)$ avec $d(u, v) = |u - v|$

= sup { inf(|*u1* – *v1*|, |*u1* – *v2*|, |*u1* – *v3*|, |*u1* – *v4*|),
inf(|*u2* – *v1*|, |*u2* – *v2*|, |*u2* – *v3*|, |*u2* – *v4*|),
inf(|*u3* – *v1*|, |*u3* – *v2*|, |*u3* – *v3*|, |*u3* – *v4*|),
inf(|*u4* – *v1*|, |*u4* – *v2*|, |*u4* – *v3*|, |*u4* – *v4*|),
inf(|*u5* – *v1*|, |*u5* – *v2*|, |*u5* – *v3*|, |*u5* – *v4*|) }

= sup (0, 1, 0, 0, 1) = 1.

De la même manière, nous pouvons calculer :

$$H(G_{\alpha_1}, F_{\alpha_1}) = \sup_{u\in G_{\alpha_1}} \inf_{v\in F_{\alpha_1}} d(u,v) = 1$$

et par conséquent, $d_H(F_{\alpha_1}, G_{\alpha_1}) = 1$.

On procédera d'une manière similaire pour les autres $d_H(F_{\alpha_i}, G_{\alpha_i})$. Les résultats sont rapportés dans le *tableau 4.1.*

Tableau 4.1 : Résultats intermédiaires

α_i	F_{α_i}	G_{α_i}	$H(F_{\alpha_i}, G_{\alpha_i})$	$H(G_{\alpha_i}, F_{\alpha_i})$	$d_H(F_{\alpha_i}, G_{\alpha_i})$
0.2	{1, 2, 4, 5, 6}	{1, 4, 5, 7}	1	1	1
0.5	{1, 4, 5, 6}	{4, 5, 7}	3	1	3
0.6	{1, 4, 6}	{4, 5, 7}	3	1	3
0.7	{1, 6}	{5, 7}	4	1	4
0.8	{6}	{5, 7}	1	1	1
1	{6}	{7}	1	1	1

Par conséquent :

$$d_H^{\infty}(F, G) = \sup\{1, 3, 3, 4, 1, 1\} = 4.$$

Remarque : D'autres formules que celles de Puri et Ralescu peuvent être utilisées pour le calcul de la distance de Hausdorff. On citera notamment l'approche de *Dubois et Prade* et celle de *Chaudhuri et Rosenfeld.*

4.1.3 Distance de Hausdorff entre un prédicat flou et sa version relaxée

Le calcul de la distance de Hausdorff dans notre cas peut être simplifié du fait que les ensembles flous correspondant aux prédicats comparés ont les mêmes noyaux.

Soit P' un prédicat flou résultat d'une ou plusieurs relaxations du prédicat flou P.

Écrits en termes de *f.a.t.*, on a : $P=(A, B, a, b)$ $P'=(A, B, a', b')$ avec $a' \geq a$ et $b' \geq b$

Soit $\alpha \in (0,1]$. Les α-coupes respectives de P et P' sont les intervalles :

$$P_\alpha = [A + (\alpha - 1)\, a,\ B + (1 - \alpha)\, b]$$

$$P'_\alpha = [A + (\alpha - 1)\, a',\ B + (1 - \alpha)\, b']$$

En appliquant (4.3), la distance de Hausdorff entre P et P' est :

$$d_H^1(P, P') = \int_0^1 d_H(P_\alpha, P'_\alpha)\, d\alpha$$

$$= \int_0^1 max\left(\left|(A+(\alpha-1)a)-(A+(\alpha-1)a')\right|, \left|(B+(1-\alpha)b)-(B+(1-\alpha)b')\right|\right) d\alpha$$

$$= max((a'-a),(b'-b)) \cdot \int_0^1 (1-\alpha)\, d\alpha$$

$$= \frac{1}{2} max((a'-a),(b'-b))$$

La quantité Dist(*P, P'*) est la valeur correspondant à la moitié de la quantité de l'extension du support du côté le plus relaxé du trapèze (*figure 4.1*).

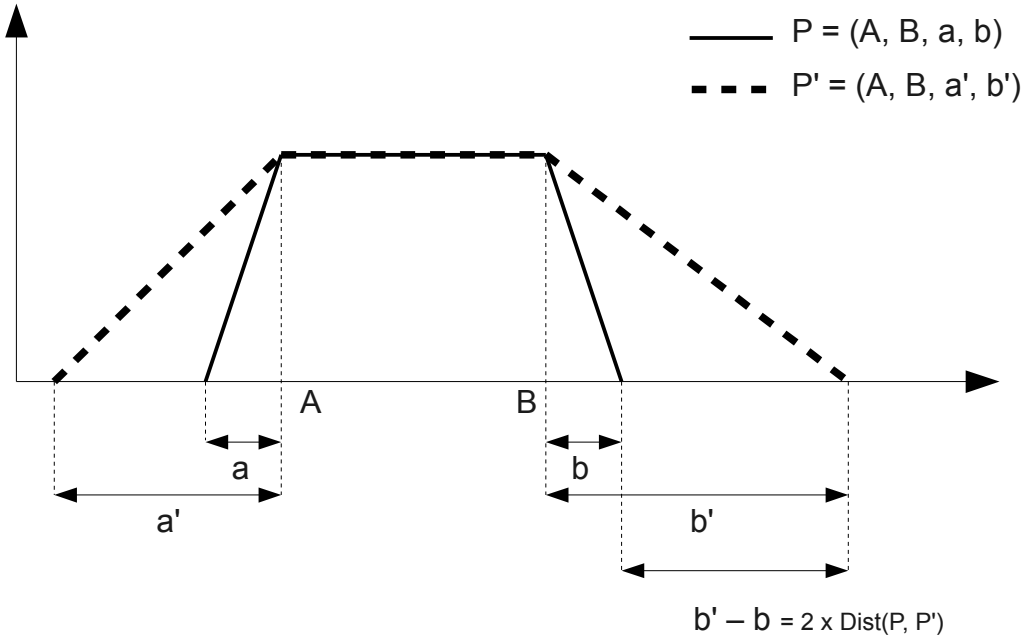


Figure 4.1 : F.a.t. d'un prédicat flou (*P*) et de sa version relaxée (*P'*)

Si *P'* correspond à la première relaxation de *P*, i.e. $P' = T^{\uparrow}(P)$:

$$Dist(P, P') = \tfrac{1}{2} \max(\Delta_l(\varepsilon), \Delta_r(\varepsilon)) = \tfrac{1}{2} \max(A \cdot \varepsilon, B \cdot \varepsilon/(1-\varepsilon))$$
$$= \tfrac{1}{2} \cdot \varepsilon \cdot \max(A, B/(1-\varepsilon))$$

S'il s'agit d'une relaxation multiple, i.e. $P' = T^{\uparrow(n)}(P)$ alors :

$$Dist(P, P') = n/2 \cdot \varepsilon \cdot \max(A, B/(1-\varepsilon))$$

Nous avons vu[16] que la relaxation est toujours plus grande du côté droit quand elle est effectuée des deux côtés. Les seuls cas où le côté gauche est le plus relaxé sont ceux correspondant aux *prédicats particuliers*[17] qui ne sont pas relaxés à droite, comme « *salaire élevé* » par exemple, (voir *figure 4.2*).

16 voir p.67

17 cités en p.67

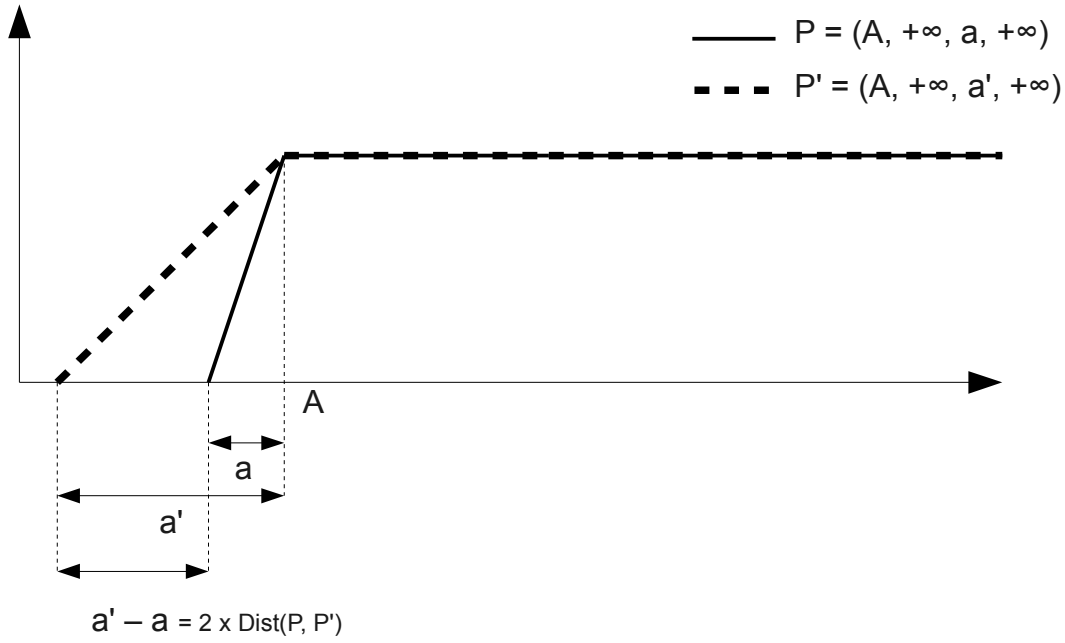


Figure 4.2 : F.a.t. d'un prédicat relaxé uniquement à gauche

Ainsi, la distance de Hausdorff entre un prédicat et sa version relaxée *n* fois est :

$$Dist(P, P') = \begin{cases} n/2 \cdot B \cdot \varepsilon/(1-\varepsilon) & \text{si le prédicat a été relaxé à droite} \\ n/2 \cdot A \cdot \varepsilon & \text{si le prédicat n'a été relaxé qu'à gauche } (\textit{prédicat particulier}) \end{cases}$$

4.2 Proximité sémantique de requêtes

En tenant compte de la relation forte et intuitive qui existe entre proximité et distance, et en utilisant la mesure de la distance de Hausdorff, nous montrons par la suite comment on peut estimer à quel point deux requêtes floues sont proches sémantiquement parlant.

4.2.1 Requêtes SP

Soit $Q = P$ et $Q' = P'$ deux requêtes atomiques (où P et P' sont des prédicats concernant le même attribut, disons A). Pour évaluer à quel point Q et Q' sont proches, sémantiquement parlant, nous faisons usage de l'indice de la distance de

Hausdorff entre les prédicats flous contenus dans ces deux requêtes. Nous avons alors :

$$Dist(Q, Q') = d_H^1 (P, P'), \tag{4.4}$$

où *Dist(Q, Q')* est la distance entre *Q* et *Q'*. Il est bien connu que l'ordre induit par une mesure de distance est l'inverse de celui induit par une mesure de proximité. Plus petit est l'indice *Dist(Q, Q')*, plus proche sont *Q* et *Q'*.

Notons par *Prox(Q, Q')* une mesure de proximité entre *Q* et *Q'*. Pour calculer sa valeur en se basant sur celle de *Dist(Q, Q')*, on peut passer par deux étapes :

i) normaliser *Dist(Q, Q')* grâce à une fonction f_{norm} permettant de la réduire à l'intervalle [0, 1] ; on pourra prendre $f_{norm}(x) = min(1, x)$ ou bien $f_{norm}(x) = x/(1+x)$;

ii) en déduire $Prox(Q, Q') = 1 - f_{norm}(Dist(Q, Q'))$.

Une autre approche permet de définir l'indice *Prox(Q, Q')* en utilisant une fonction de conversion sur la mesure de la distance. Par exemple, *Prox(Q, Q')* peut être défini par [Cross, Sudkamp 02] :

$$Prox(Q, Q') = \left(1 + \left(\frac{Dist(Q, Q')}{s}\right)^t\right)^{-1}. \tag{4.5}$$

Les constantes positives *s* et *t* ajustent la taille de la mesure de proximité. La fonction de conversion la plus simple peut être obtenue en posant $s = 1$ et $t = 1$.

Il est important de noter que pour notre étude, la quantité selon laquelle *Q* est *sémantiquement proche de Q' n'est pas cruciale* puisque nous ne nous intéressons qu'à l'ordre induit par la mesure *Prox(Q, Q')*. Cet ordre est obtenu en inversant celui induit par la mesure *Dist(Q, Q')*. Afin d'illustrer ce propos, considérons l'exemple qui suit :

Exemple 4. Soit Q, Q_1, Q_2 et Q_3 quatre requêtes atomiques. Supposons que nous ayons obtenu les mesures de distance suivantes :

$$Dist(Q, Q_1) = 6,\ Dist(Q, Q_2) = 17/3, Dist(Q, Q_3) = 8/3.$$

On peut observer que $Dist(Q, Q_1) > Dist(Q, Q_2) > Dist(Q, Q_3)$. En inversant cet ordre, nous obtenons l'ordre suivant, basé sur la mesure de proximité:

$$Prox(Q, Q_1) < Prox(Q, Q_2) < Prox(Q, Q_3).$$

Donc Q est plus proche de Q_3 que Q_2 (resp. Q_1).

4.2.2 Requêtes conjonctives

Soit $A_1, A_2,..., A_n$ n attributs avec $D(A_i)$ étant le domaine de valeurs de A_i. Soit également Q (resp. Q') une requête composée flexible de la forme $P_1 \wedge ... \wedge P_k$ (resp. $P'_1 \wedge ... \wedge P'_k$) où, pour $i=1,k$, P_i (resp. P'_i) est un prédicat flou concernant l'attribut A_i.

Pour évaluer la mesure de la distance entre Q et Q' dans l'esprit de la formule (4.4), on commencera par calculer les distances entre les prédicats respectifs des deux requêtes, c'est-à-dire les valeurs de $Dist(P_i, P'_i)$ pour tous les i entre 1 et k. Ensuite, on pourra citer au moins deux façons de faire :

1ère méthode :

En utilisant la formule suivante :

$$Dist(Q, Q') = \frac{1}{k} \sum_{i=1}^{k} Dist(P_i, P'_i) \qquad (4.6)$$

On reconnaît aisément l'expression ci-dessus, qui n'est autre que la moyenne arithmétique des proximités entre les prédicats respectifs composant les requêtes Q et Q'.

2è méthode :

On peut utiliser les valeurs de $Dist(P_i, P_i')$ pour calculer les proximités $Prox(P_i, P_i')$ (selon la formule (4.5) par exemple) puis déduire la proximité des requêtes par la formule :

$$Prox(Q, Q') = min_{i=1, k}\ Prox(P_i, P'_i)$$

Notons cependant que cela revient à considérer la proximité entre Q et Q' comme égale à la proximité entre les prédicats respectifs les moins proches.

Pour simplifier les calculs, on peut alors considérer directement la distance entre les prédicats les plus distants. Cela évitera de passer par les calculs de proximité. Ainsi, il suffira de calculer :

$$Dist(Q, Q') = max_{i=1, k}\ Dist(P_i, P'_i)$$

De la même manière que précédemment, l'ordre des requêtes qui pourrait être obtenu en comparant les proximités sera directement déduit en inversant celui obtenu par la comparaison des distances.

4.3 Principe de la méthode

Soit Q la requête infructueuse initiale et *Liste_Relaxées* l'ensemble des relaxations de Q fournies par l'*algorithme 3.3*. Pour retourner la meilleure relaxation de Q parmi les éléments de *Liste_Relaxées*, nous procédons comme suit :

- **Étape 1**: *Calcul de la mesure de Distance*

  pour chaque $relax_Q$ dans *Liste_Relaxées* :

  Calculer $Dist(Q, relax_Q)$

- **Étape 2**: *Ordonner Liste_Relaxées*

  Ordonner *Liste_Relaxées* dans le sens croissant par rapport à la mesure de la distance

- **Étape 3**: *Meilleure relaxation*

Retourner le premier élément de l'ensemble ordonné *Liste_Relaxées*

La procédure à trois étapes ci-dessus peut être formalisée dans l'*algorithme 4.1*.

On peut considérer *Dist*(*Q*, *relaxQ*) comme uniquement égale à l'indice de distance relative *H*(*relaxQ, Q*). En effet, ce sont les réponses à *relaxQ* qui sont données comme réponses alternatives à *Q* et non l'inverse.

Entrées: *Q; Liste_Relaxées;*

1. **début**
2. **pour** $relax_Q$ **dans** *Liste_Relaxées*;
3. calculer *Dist(Q,* $relax_Q$*)*;
4. Trier(*Liste_Relaxées*); /* trier dans l'ordre ascendant */
5. **retourner** Premier(*Liste_Relaxées*);

Sortie: La meilleure relaxation de *Q*;

Algorithme 4.1 : Recherche de la meilleure relaxation de *Q*.

4.4 Un exemple illustratif

Afin d'illustrer notre propos, nous présentons un exemple inspiré de [Bosc *et al.* 07]. Il concerne un utilisateur qui désire trouver les employés d'un département qui satisfont la condition complexe « *jeune* et *bien-payé* ». La relation décrivant les employés considérés est fournie par le *tableau 4.2*.

Tableau 4.2 : Relation des employés

Nom	Age	Salaire (k€)	$\mu_{P1}(u)$	$\mu_{P2}(v)$	$\mu_Q(t) = \min(\mu_{P1}(u), \mu_{P2}(v))$
Dupont	46	3	0	0	0
Martin	42	2.5	0	0	0
Durant	28	1.5	0.8	0	0
Dubois	30	1.8	0.67	0	0
Lorant	35	2	0.33	0	0

La requête en question s'écrit alors Q = *"trouver les employés qui sont jeunes et bien-payés"* où *jeune* et *bien-payé* sont des libellés d'ensembles flous représentés respectivement par les *f.a.t.* $P_1 = (0, 25, 0, 15)$ et $P_2 = (5, +\infty, 2, +\infty)$ tels que donnés en *figure 4.3*. Dans la suite, nous écrirons simplement $Q = P_1 \wedge P_2$.

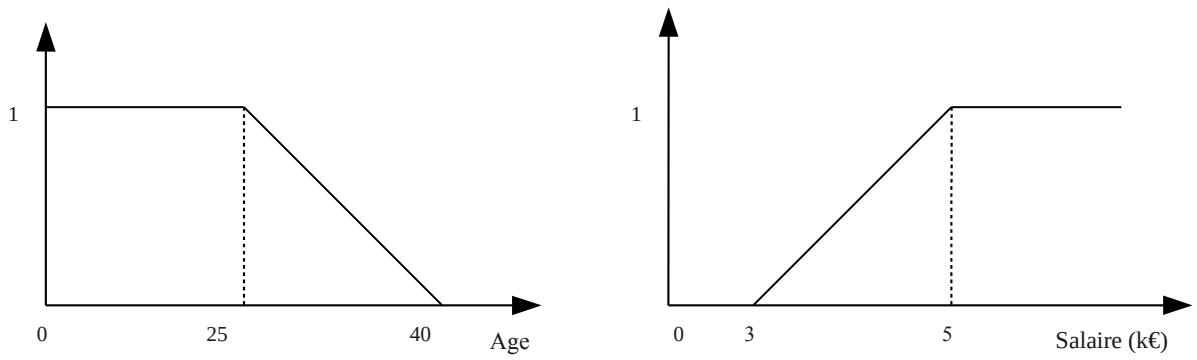
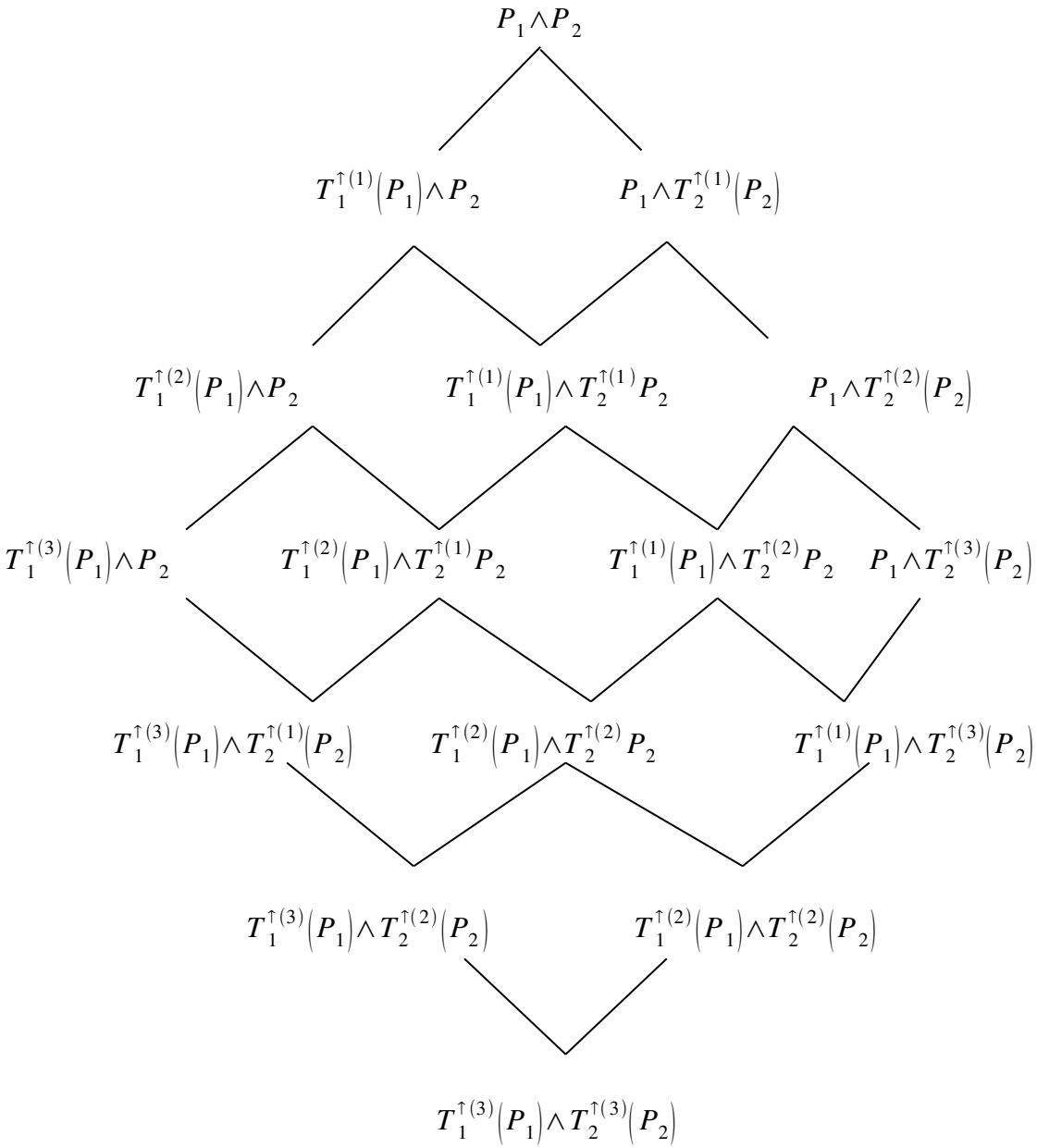


Figure 4.3 : Prédicats flous *jeune* et *bien-payé*.

Comme on peut le voir, chaque élément de la base de données obtient zéro comme degré de satisfaction pour la requête de l'utilisateur Q avec le contenu du *tableau 4.2*. À présent, afin de retourner des réponses alternatives à l'utilisateur, nous essayons de coopérer avec lui en relaxant sa question.

Supposons que nous désirions limiter le nombre de relaxations pour chaque prédicat à $\omega = 3$, et prenons pour valeurs de tolérance relative pour P_1 et P_2 respectivement : $\varepsilon_1 = 0.09$ et $\varepsilon_2 = 0.12$. Le treillis de requêtes relaxées est représenté en *figure 4.4*. Il est facile de voir que l'ensemble des MFS $mfs(Q) = \{P_2\}$.



<u>Figure 4.4 :</u> Treillis borné de requêtes relaxées (pour ω = 3)

Tableau 4.3.

Nom	Age	Salaire (k€)	$\mu_{T1\uparrow(P1)}(u)$	$\mu_{T2\uparrow(P2)}(v)$	$\mu_{Q22}(t) = \min(\mu_{T1\uparrow(P1)}(u), \mu_{T2\uparrow(P2)}(v)$
Dupont	46	3	0	0.23	0
Martin	42	2.5	0.03	0.04	0.03
Durant	28	1.5	0.83	0	0
Dubois	30	1.8	0.71	0	0
Lorant	35	2	0.43	0	0

Tableau 4.4.

Nom	Age	Salaire (k€)	$\mu_{P1}(u)$	$\mu_{T2\uparrow(2)(P2)}(v)$	$\mu_{Q23}(t) = \min(\mu_{P1}(u), \mu_{T2\uparrow(2)(P2)}(v)$
Dupont	46	3	0	0.38	0
Martin	42	2.5	0	0.22	0
Durant	28	1.5	0.8	0	0
Dubois	30	1.8	0.67	0	0
Lorant	35	2	0.33	0.06	0.06

En appliquant l'*algorithme 3.3*, nous obtenons *Liste_Relaxées* = $\{Q_{22}, Q_{23}\}$ où $Q_{i,j}$ est la $j^{ème}$ relaxation de Q (en comptant de gauche à droite) du $i^{ème}$ niveau du treillis. Nous avons :

$$Q_{22} = T_1^{\uparrow}(P_1) \wedge T_2^{\uparrow}(P_2) \quad \text{et} \quad Q_{23} = P_1 \wedge T_2^{\uparrow(2)}(P_2)$$

avec $T_1^{\uparrow}(P_1) = (0, 25, 0, 17.5)$, $T_2^{\uparrow}(P_2) = (5, 100, 2.6, 0)$ [18] et $T_2^{\uparrow(2)}(P_2) = (5, 100, 3.2, 0)$. Voir les *tableaux 4.3 et 4.4*.

Pour sélectionner l'élément de *Liste_Relaxées* qui est la meilleure relaxation pour Q, nous estimons d'abord les mesures de distance $Dist(Q, Q_{22})$ et $Dist(Q, Q_{23})$. En utilisant la formule (4.6), on obtient :

$d^1_H(P_1, T_1^{\uparrow}(P_1)) = ½ \max(0 - 0, 17.5 - 15) = 1.25$

$d^1_H(P_2, T_2^{\uparrow}(P_2)) = ½ \max(2.6 - 2, 0 - 0) = 0.3$

$Dist(Q, Q_{22}) = (1.25 + 0.3)/2 = \mathbf{0.775}$

18 Afin de simplifier le calcul, nous avons fixé la borne supérieure du support de P_2, i.e., $P_2 = (5, 100, 2, 0)$. Du fait que cette borne n'est jamais atteinte par les salariés, la transformation de relaxation affectera uniquement le côté gauche de P_2.

$d^1_H(P_2,\ T_2^{\uparrow(2)}(P_2)) = ½ \max\ (3.2 - 2,\ 0 - 0) = 0.6$

$Dist(Q,\ Q_{23}) = (0 + 0.6)/2 = 0.3$

Nous déduisons alors que $Dist(Q,\ Q_{22}) > Dist(Q,\ Q_{23})$. Ceci signifie que Q_{23} est la meilleure relaxation (la moins éloignée) de Q. Donc l'employé Lorant est retourné à l'utilisateur comme réponse à sa requête.

Chapitre 5 :
Implémentation et mise en œuvre

5.1 Introduction

La problématique abordée dans cette étude constitue une petite partie d'un projet mené au sein de l'équipe PILGRIM de l'ENSSAT, école d'ingénieur à Lannion (France) faisant partie de l'Université de Rennes I. Un des thèmes majeurs de l'équipe est l'interrogation flexible de bases de données par les techniques issues de la théorie des ensembles flous et des possibilités.

Bien qu'elle soit plutôt orientée vers la recherche fondamentale, l'équipe a développé un prototype logiciel, baptisé iSQLf capable de traiter des requêtes floues écrites en langage SQLf. Son développement suit un cycle itératif durant lequel les avancées dans le domaine théorique donnent lieu à des implémentations sous forme de modules logiciels.

Dans le cadre de la mise en place d'un module de traitement coopératif de requêtes à réponse vide ou pléthorique, dans le prototype iSQLf, une implémentation de l'algorithme de Godfrey [Godfrey 97] pour la recherche de MFS a été réalisée dans un projet en 2008 [Joseph 08] en Java (en utilisant le SGBD MySQL).

Afin de mettre en œuvre les algorithmes que nous avons développés au cours de ce travail, nous nous sommes appuyés sur la réalisation de [Joseph 08] dont nous avons directement utilisé la fonction de recherche de MFS, et qui nous a servi de base pour construire la partie logicielle apportant la fonctionnalité de relaxation des requêtes. L'interface a été améliorée et modifiée afin d'inclure cette nouvelle fonctionnalité.

5.2 Algorithme de relaxation détaillé

La mise en œuvre des *algorithmes 3.3* et *4.1* a été réalisée en manipulant une structure de données différente de celle utilisée par [Joseph 08]. Ce dernier a créé la structure ***RequeteConjonctive*** qui est essentiellement un tableau

d'enregistrements de type ***PredicatSimple***, composés de deux chaînes de caractères (l'*attribut* concerné dans la base de données et l'*étiquette* nommant le prédicat) et quatre nombres réels (les valeurs de A, B, a et b dans la f.a.t. du prédicat).

Pour notre algorithme, nous avons minimisé la représentation des requêtes. En effet, notre type ***Requête***, qui est à la fois celui des variables correspondant aux nœuds du treillis et aux MFS calculées, n'est constitué que de simples tableaux de N nombres entiers où chaque élément (d'indice i) est le nombre de relaxations du prédicat correspondant (P_i).

Exemple : Le nœud du treillis correspondant à la requête modifiée :

$$P_1 \wedge T_2^{\uparrow(3)}(P_2) \wedge T_3^{\uparrow(2)}(P_3)$$

est représenté par le tableau d'entiers [0, 3, 2].

Dans le cas des MFS, la valeur –1 est attribuée aux éléments dont l'indice correspond à celui d'un prédicat absent.

Exemple : La requête $T_2^{\uparrow(2)}(P_2) \wedge P_3$ est représentée par le tableau [-1, 2, 0].

L'*algorithme 5.1*, permettant la relaxation d'une requête par la méthode décrite précédemment, nécessite l'apport en entrée de la requête conjonctive Q à relaxer et de la liste des MFS préalablement calculées par l'algorithme de [Godfrey 97] (ces MFS auront été converties au format que nous emploierons, i.e. un tableau d'entiers). Notre algorithme n'effectuant qu'un parcours du treillis, il n'utilisera pas directement la requête Q qui n'est nécessaire qu'au moment d'évaluer les requêtes relaxées. En effet, ce n'est qu'à ce moment que les informations contenues dans le type de données RequeteConjonctive seront transmises à la fonction d'évaluation pour y être utilisées.

Le parcours du treillis se basant sur les MFS et les propositions vues au *chapitre 3* (p.77), l'algorithme utilise la fonction ***mfs(Q')*** qui déduit une liste de MFS d'une requête relaxée Q' du treillis à partir de celles de la requête initiale Q.

Cette fonction opère en remplaçant les prédicats des MFS de Q par les prédicats relaxés correspondant de Q'.

Le résultat final du déroulement de l'algorithme sera constitué de la *meilleure* requête relaxée, telle que déterminée par la mesure de sa distance de Hausdorff à la requête initiale, comme expliqué dans le chapitre précédent.

Entrées : Q : **tableau d'enregistrements** :attribut: **chaîne**;
étiquette: **chaîne**;
A, B, a, b: **réels**; *(* Q = $P_1 \wedge \ldots \wedge P_N$ *)*
mfs_Q : **liste de tableaux d'entiers** *(* les MFS de Q préalablement calculées avec l'algorithme de [Godfrey 97] *)*

1. **Type** Requête : **tableau d'entiers** *(* nbre de relaxations pour chaque prédicat. Utilisé aussi pour les MFS, avec -1 pour les prédicats manquants *)*

2. **Fonction** mfs(req : Requête) : **liste de** Requête
3. **var** nouvelle_mfs : Requête;
4. liste_mfs : **liste de** Requête;
5. **début**
6. **pour** une_mfs **dans** mfs_Q **faire**
7. **début**
8. nouvelle_mfs := ∅;
9. **pour** Pi **dans** une_mfs **faire**
10. **si** Pi = -1 **alors** nouvelle_mfs.ajouter(-1)
11. **sinon** nouvelle_mfs.ajouter(req[i]);
12. liste_mfs.ajouter(nouvelle_mfs);
13. **fin;**
14. **retourner** liste_mfs;
15. **fin**;

16. **Fonction** contient(req : Requete, une_mfs : Requete) : **booléen**;
17. **var** existe : **booléen**;
18. **début**
19. existe := vrai;
20. **pour** Pj **dans** une_mfs **faire**
21. **si** Pj >= 0 *(* ignorer les prédicats manquants *)*
22. existe := existe **et** (req[j]=Pj);
23. **retourner** existe;
24. **fin**;

25. **Variables**
26. relaxée, parent : Requête;

27. **DÉBUT**
28. liste_relaxées = ∅; i := 1;

```
29.  tantque (i ≤ ω.N ) et (liste_relaxées = ∅) faire
30.  début
31.    pour relaxée dans niveau(i) faire
32.    début
33.      modif := vrai;
34.      pour Pj dans relaxée faire
35.      début
36.        si Pj > 0 alors
37.        début
38.          parent := relaxée;      (* Les parents ont les mêmes prédicats... *)
39.          parent[j] := relaxée[j] – 1;  (* ...avec une relaxation en moins. *)
40.          pour une_mfs dans mfs(parent) faire
41.            modif := modif et non(contient(relaxée, une_mfs));
42.        fin;
43.      fin;
44.      si modif alors
45.        si évaluer(Q, relaxée) alors
46.          liste_relaxée.ajouter(relaxée);
47.    fin;
48.    i:=i+1;
49.  fin;
50.  si liste_relaxées ≠ ∅ alors
51.  début
52.    hausdorff := ∞ ;
53.    pour req dans liste_relaxées faire
54.    début
55.      dist_req := distance(req);
56.      si dist_req < hausdorff alors
57.      début
58.        hausdorff := dist_req;
59.        meilleure_req := req;
60.      fin;
61.    fin;
62.    retourner(meilleure_req);
63.  fin;
64. FIN.
```

Sortie : meilleure_req *(* la meilleure des requêtes relaxées qui n'échouent pas *)*

Algorithme 5.1 : Algorithme de relaxation détaillé

Les fonctions suivantes sont utilisées dans l'*algorithme 5.1* sans que leur mise en œuvre n'y soit développée :

- ***niveau(i)*** fournit l'ensemble des requêtes relaxées (sous forme de tableaux d'entiers) au niveau *i* du treillis pour un nombre *N* de prédicats. *Exemple* : si *N* = 2, *niveau(2)* retournera l'ensemble de tableaux {[2, 0], [1, 1], [0, 2]}.
- ***évaluer(Q, Q')*** est une fonction qui évalue *Q'* sur la base de données cible. Elle retourne *vrai* si *Q'* produit des réponses non-vides et *faux* autrement. Notons que si *Q'* reste au format de tableau d'entiers, il est cependant nécessaire que *Q* comporte toutes les informations permettant l'évaluation de sa requête relaxée *Q'* et soit donc de type ***RequeteConjonctive*** (tableau de *N* ***PredicatSimple****s*).
- ***distance(Q')*** est une fonction qui retourne la distance de Hausdorff de la requête initiale à *Q'*.

5.3 Classes Java

Les classes que nous avons implémentées utilisent celles déjà existantes du projet de [Joseph 08] pour la recherche des *MFS*, auquel le lecteur pourra se référer pour une description détaillée. Nous ne décrirons que la partie réalisée dans notre travail, qui a été mise à part dans un paquetage nommé ***relax***. On trouvera en *annexe* le code source complet. Nous avons également modifié la classe ***InterfaceGraphique*** pour y ajouter l'affichage des résultats de la relaxation.

5.3.1 Classe Requete

Comme expliqué précédemment, cette classe modélise les requêtes correspondant aux nœuds du treillis de requêtes relaxées d'une part et leurs MFS d'autre part.

Elle hérite de ***ArrayList<Integer>*** et se présente donc comme un tableau dynamique d'entiers.

Ses variables de classe sont :

- ***Qinitiale*** : objet de type ***RequeteConjonctive*** (*i.e.* liste de ***PredicatSimple***s) destiné à recevoir la requête initiale *Q*.
- ***N*** : nombre de prédicats de la requête *Q*.
- ***epsilon*[]** : tableau qui comportera les *valeurs de tolérance relatives* utilisées pour relaxer les prédicats respectifs de la requête *Q*
- ***omega*** : nombre maximal de relaxations par prédicat (il sera défini par l'utilisateur)
- ***mfs_Q*** : liste chaînée de requêtes (***LinkedList<Requete>***) qui contiendra les MFS de *Q*.

Les méthodes de la classe ***Requete*** sont les suivantes :

- ***setMfs_Q***(*listeMfs*) : méthode statique appelée avec en paramètre la liste des *MFS* calculées par l'algorithme de Godfrey (dans le paquetage ***application***). Elle construit la liste chaînée de ***Requete***s correspondant aux éléments de *listeMfs*. Dans chaque requête (tableau d'entiers), si un prédicat se trouve dans la *MFS*, il (l'entier correspondant) reçoit la valeur 0 (c'est un prédicat non relaxé), s'il n'en fait pas partie, il reçoit –1 (prédicat « absent » de la *MFS*). Une fois terminée, la liste chaînée est affectée au champ statique ***mfs_Q***.
- ***getBorneA(i)***, ***getBorneB(i)***, ***getBorne_a(i)*** et ***getBorne_b(i)*** : retournent les « bornes » de la f.a.t. du prédicat d'indice *i*. Les valeurs de *A* et *B* sont directement récupérées de ***Qinitiale*** (le noyau ne change pas) et celles de *a* et *b* sont calculées à partir des bornes de ***Qinitiale***, de la valeur de tolérance ***epsilon***[*i*] et de l'entier correspondant au nombre *n* de relaxations. Rappelons que $a_i' = a_i + n\ \Delta_l(\varepsilon_i)$ avec $\Delta_l(\varepsilon_i) = A \cdot \varepsilon_i$ et $b_i' = b_i + n\ \Delta_r(\varepsilon_i)$ avec $\Delta_r(\varepsilon_i) = B_i \cdot \varepsilon_i/(1-\varepsilon_i)$.

- ***convertirEnRequeteConjonctive()*** : convertit la requête en un objet du type ***RequeteConjonctive***. Le tableau d'entiers ne comportant que le nombre de relaxations par prédicats, cette méthode doit récupérer les autres informations concernant les prédicats de ***Qinitiale***, directement et à travers les méthodes ***getBorne_a(i)*** et ***getBorne_b(i)***. Dans le cas des *MFS,* les prédicats « absents » (-1) sont ignorés.

- ***clone()*** : retourne une nouvelle ***Requete*** construite en copiant un à un les éléments de celle-ci. Cette méthode redéfinit la méthode homonyme de ***ArrayList,*** qui ne fait qu'une copie *superficielle*, sans copier le contenu.

- ***mfs()*** : déduit l'ensemble des *MFS* de la requête à partir de celles de la requête initiale *Q* précédemment calculées. Une nouvelle liste chaînée de ***Requete***s est générée et retournée, où pour chaque requête les prédicats existants (de valeur entière 0) des *MFS* de *Q* sont remplacés par les prédicats de la requête relaxée (valeur entière correspondant au nombre de relaxations). Les prédicats absents (de valeur –1) des *MFS* restent absents (prennent également la valeur –1).

- ***evaluer()*** : évalue la requête sur la base de données et retourne **vrai** si la réponse est non vide. Pour ce faire, elle commence par la convertir en ***RequeteConjonctive*** afin d'appeler la méthode ***Outils.test()*** (paquetage ***outils***)

- ***contient(****une_mfs)* : retourne **vrai** si tous les prédicats de la *MFS une_mfs* (de type ***Requete*****)** sont des prédicats de cette requête (c'est-à-dire si tous les entiers différents de -1 du tableau *une_mfs* se retrouvent à la même place dans ce tableau).

- ***toString()*** : redéfinit la méthode de même nom de la classe ***AbstractCollection*** (qui retourne une chaîne de caractères dans le style "[élément1, élément2..., élémentN]") et renvoie une chaîne de caractères

représentant la requête avec pour chaque prédicat présent (≠ -1) autant de signes *apostrophe* (') qu'il a subi de relaxations puis sa f.a.t. *Exemple :* "P1(5, 10, 1, 1) ^ P2''(100, 150, 14.191, 17.322) ^ P3'(20, 40, 3.212, 4.154)".

- ***distance()*** : calcule et renvoie la distance de Hausdorff de la requête ***Qinitiale*** à cette requête. Rappelons qu'il s'agit de la moyenne des distances entre les prédicats, calculées par la formule :

$$d_H^1(P_i, P_i') = \frac{1}{2} max((a_i' - a_i), (b_i' - b_i))$$

5.3.2 Classe Treillis

La seule méthode dont nous avons eu besoin pour le parcours du treillis est :

- ***niveau(****nbPredicats, niveau)* : donne la liste de toutes les requêtes relaxées du treillis correspondant au niveau donné en paramètre. En effet, nous avons vu en *section 3.5.2* (p.70) que les requêtes sont ordonnées selon le niveau du treillis où elles sont situées qui n'est autre que la somme des nombres de relaxations par prédicat :

$$niveau = \sum_{i=1}^{N} \text{nombre}\left(T_i^{\uparrow} \text{ dans } Q'\right)$$

L'algorithme de cette méthode consiste simplement à trouver les combinaisons de *nbPredicats* nombres entiers dont la somme est *niveau*.

5.3.3 Classe Relaxation

Cette classe contient les méthodes permettant la relaxation de la requête proprement dite :

- Le **constructeur** ***Relaxation(****Qinitiale, omega, epsilon[])* prépare les champs ***listeRelaxees***, qui recevra une liste de ***Requete***s, et ***treillis,*** instance de la classe ***Treillis.*** Il utilise par ailleurs les informations qui lui sont transmises en paramètre pour initialiser les champs statiques de ***Requete*** :

Qinitiale, ***N***, ***omega*** et ***epsilon[]***. Dans le cas où ***epsilon[]*** est reçu avec des valeurs nulles, un appel à ***valeursDeTolerance()***, permet d'abord de les calculer en utilisant la propriété de l'ÉER.

- ***valeursDeTolerance(****Qinitiale, omega)* : cette méthode est l'implémentation de l'*algorithme 3.2* (voir son explication p.76) qui calcule les valeurs de tolérance ε_i pour tous les prédicats P_i de la requête *Qinitiale* qu'elle reçoit en paramètre, en tenant compte de la propriété de l'Égalité de l'Effet de Relaxation (ÉER). Elle utilise la valeur de ω (*omega*) qui lui est transmise en paramètre et fait appel à la méthode ***calculEER()*** pour le calcul proprement dit. Elle retourne les valeurs calculées dans un tableau (*double[]*).

- ***calculEER(****Pi, Ps, epsilon_s)* : en utilisant la propriété de l'ÉER et les valeurs des bornes des prédicats $P_i(A_i, B_i, a_i, b_i)$ et $P_s(A_s, B_s, a_s, b_s)$, cette méthode calcule la valeur de tolérance (ε_i) de P_i à partir de celle de P_s reçue en paramètre (*epsilon_s*). Nous avons vu (p.76) la formule appliquée, qui est :

$$\varepsilon_i = \frac{A_i + B_i + K_i - \sqrt{(A_i + B_i + K_i)^2 - 4 A_i K_i}}{2 A_i}$$

$$\text{avec } K_i = \frac{(A_s\varepsilon_s + B_s\varepsilon_s/(1-\varepsilon_s))(B_i - A_i + a_i + b_i)}{B_s - A_s + a_s + b_s}$$

- ***relaxer()*** : lance le parcours du treillis de requêtes relaxées et stocke les résultats dans les champs ***listeRelaxees*** (liste de requêtes relaxées non échouantes) et ***meilleure*** (requête relaxée la plus proche de la requête initiale). ***Requete.mfs_Q*** doit être initialisé avec ***Requete.setMfs_Q()*** avant l'appel à cette méthode. Cette méthode correspond à l'*algorithme 5.1* dont nous avons discuté précédemment dans ce chapitre. Elle fait appel entre

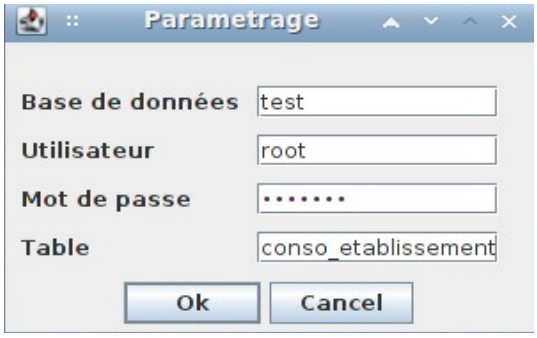
autres aux méthodes ***Treillis.niveau(), Requete.clone(), Requete.mfs(), Requete.contient(), Requete.distance()***.

- ***getMeilleure()*** : est un accesseur pour le champ ***meilleure*** qui contient, après l'exécution de ***relaxer()***, le résultat de la recherche de la meilleure requête relaxée.

5.4 Utilisation du programme

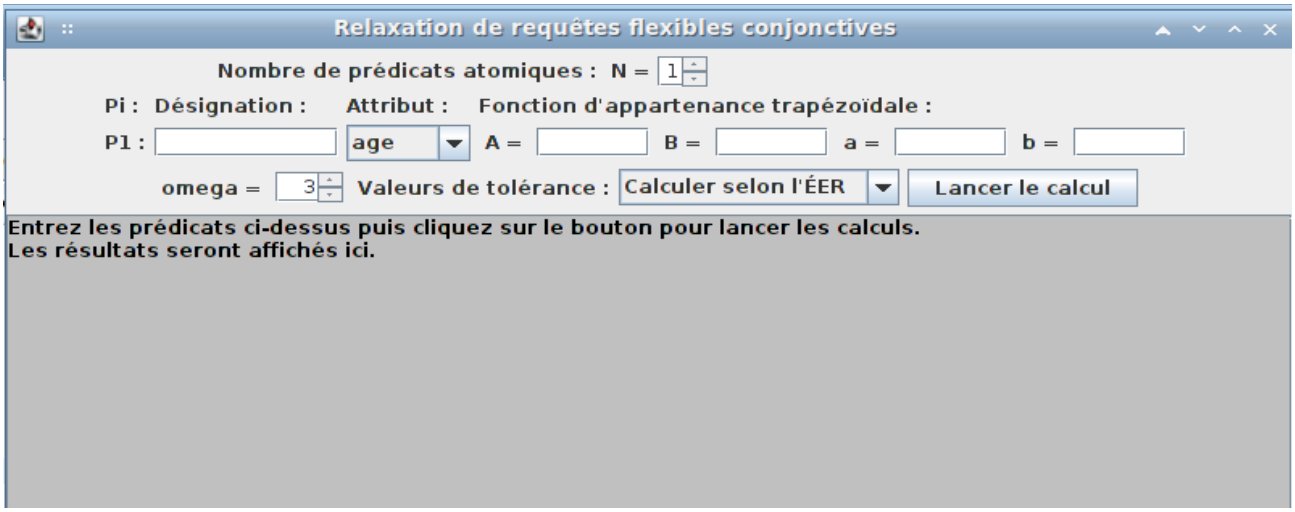
5.4.1 Connexion à la base de données

L'exécution du programme commence par la boîte de dialogue « Paramétrage », qui permet d'introduire le nom de la base de données, l'utilisateur, son mot de passe, ainsi que le nom de la table de cette base de données sur laquelle se feront les requêtes.



5.4.2 Définir les prédicats

Dans la fenêtre principale (« Relaxation de requêtes flexibles conjonctives ») l'utilisateur commence par choisir le nombre de prédicats.



Pour chaque prédicat, une ligne s'affiche pour :

- remplir la désignation du critère flou,
- sélectionner l'attribut concerné dans le *ComboBox* déroulant comportant la liste des colonnes de la table choisie dans la base de données au départ,
- remplir les quatre valeurs (A, B, a, b) définissant la fonction d'appartenance trapézoidale.

L'utilisateur devra ensuite choisir le nombre de relaxations maximal ω (omega).

Pour les valeurs de tolérance ε_i l'utilisateur doit choisir dans la liste déroulante entre :

- les faire calculer selon la propriété de l'Égalité de l'Effet de Relaxation (ÉER) (*section 3.5.2, p.74*),
- les fixer toutes à la valeur $\varepsilon_{max} / \omega$,
- leur donner des valeurs personnalisées. Ce troisième choix lancera l'affichage d'une boîte de dialogue permettant d'introduire ces valeurs.

5.4.3 Lancement du calcul

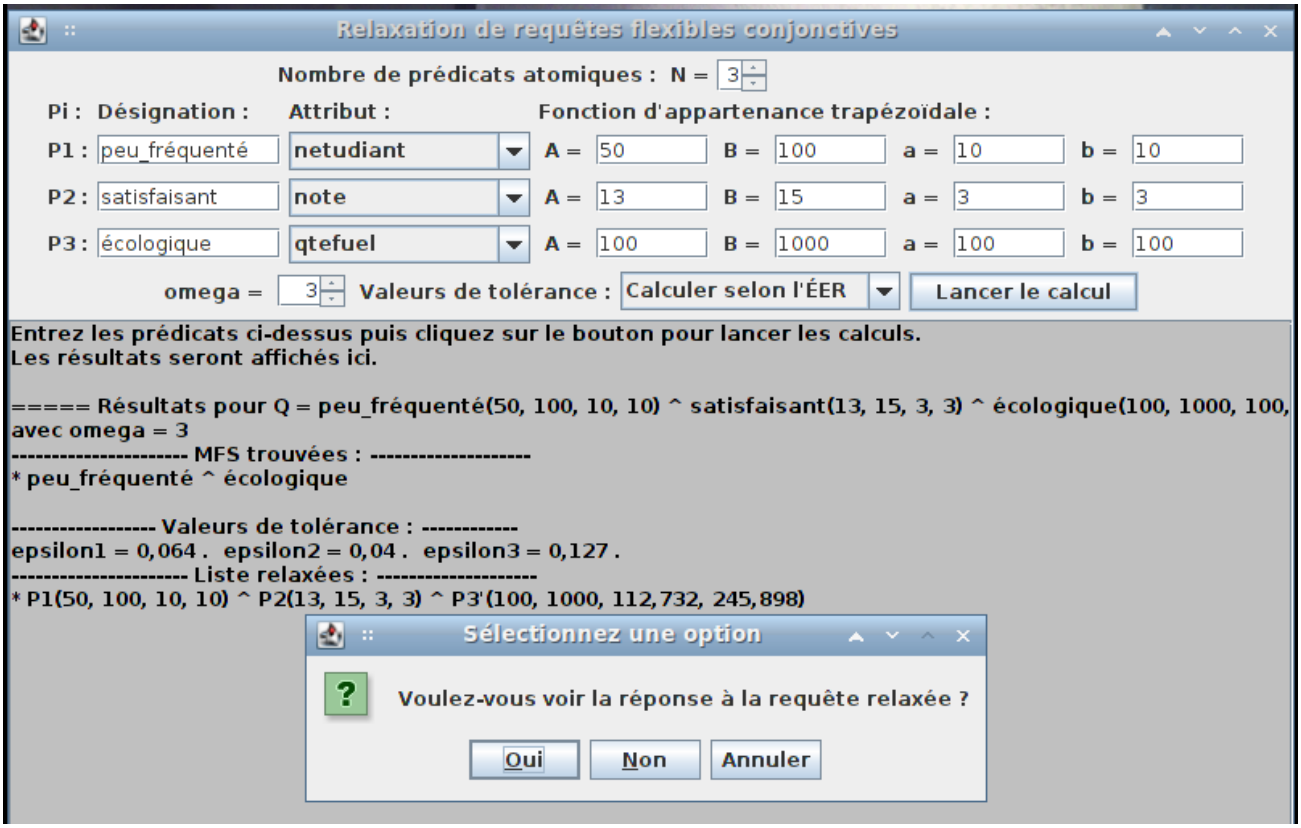
Les résultats du calcul s'affichent dans le panneau inférieur de la fenêtre principale. Après leur obtention, il est possible de modifier les prédicats et

relancer le calcul avec les nouveaux paramètres. Les nouveaux résultats seront affichés à la suite des premiers pour permettre la comparaison.

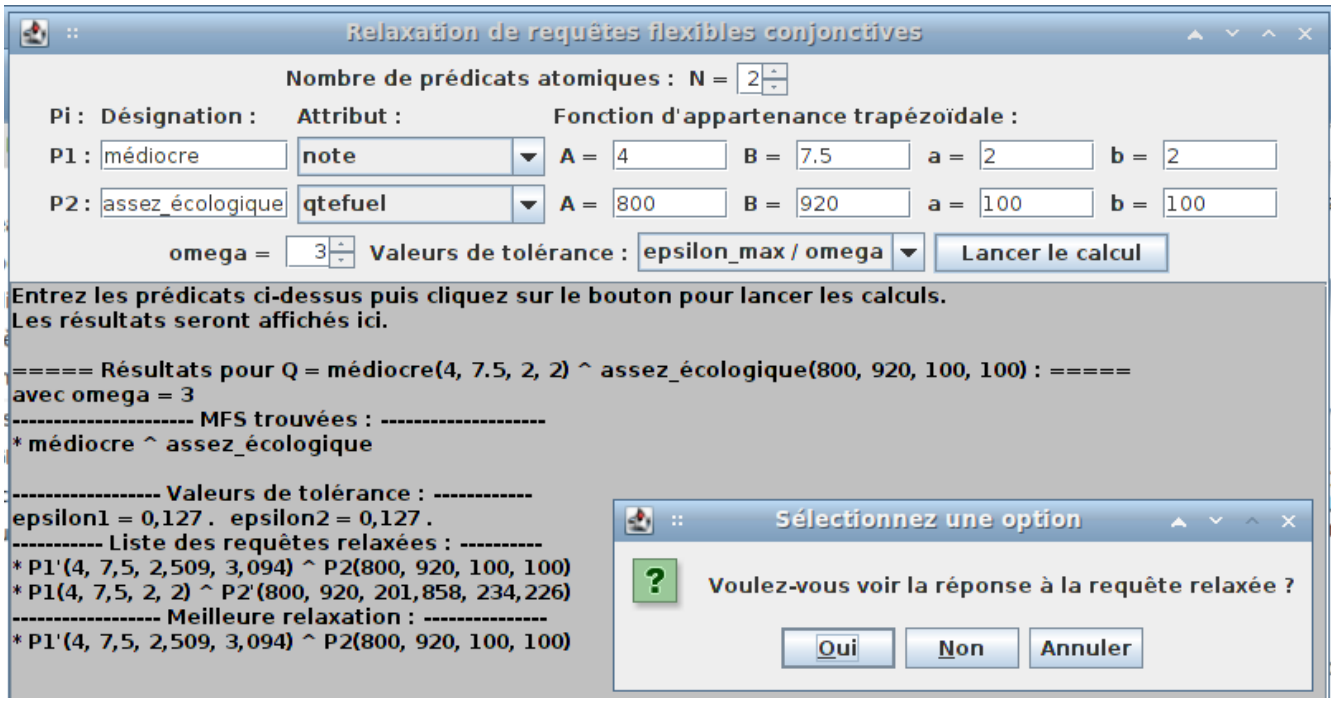
1. **Recherche de MFS :** L'algorithme de [Godfrey 97], implémenté par [Joseph 08] permet d'obtenir une liste de MFS qui sont ensuite utilisés par l'algorithme de relaxation pour élaguer le treillis des requêtes transformées. Cette liste est affichée dans la première partie des résultats.

2. **Valeurs de tolérance :** Si l'option « Calculer selon l'ÉER » a été choisie, ces valeurs de ε_i sont calculées en utilisant notre implémentation de l'*algorithme 3.2* (p.76).

3. **Relaxation :** La mise en œuvre de l'*algorithme 3.3* (p.79) de recherche d'un ensemble de relaxations fructueuses permet d'obtenir en résultat la liste des requêtes relaxées à réponse non vide. La notation abrégée utilisée indique le nombre de relaxations d'un prédicat par des caractères « apostrophe » (par exemple P2''' pour $T_2^{\uparrow(3)}(P_2)$) et la *f.a.t* entre parenthèses.



4. **Recherche de la meilleure relaxation :** Dans les cas où la liste des requêtes relaxées obtenues comporte plus d'une requête, notre implémentation de l'*algorithme 4.1* (p.101) se charge de mesurer les distances de Hausdorff pour déterminer la requête la plus proche sémantiquement parlant de la requête initiale, et qui sera finalement affichée en résultat.



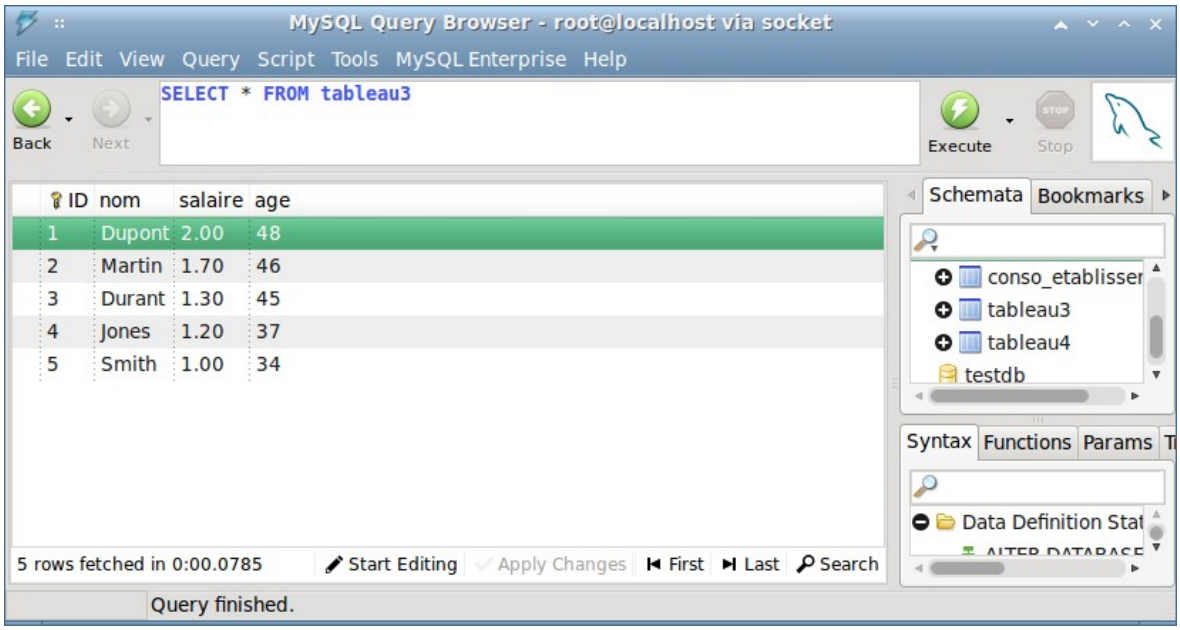
5.4.4 Affichage de la réponse

Si les calculs aboutissent à une requête relaxée satisfaisante, c'est-à-dire dont la réponse n'est pas vide et qui ne soit pas trop éloignée de la requête initiale (son support reste dans l'intervalle de validité **V**), alors la réponse à cette requête obtenue est affichée (après confirmation de l'utilisateur). Le résultat de cette requête est constitué non seulement des tuples qui satisfont *un tant soit peu* (et non *totalement*, sinon la requête initiale aurait donné une réponse non vide) ses conditions mais également du degré de satisfaction (degré d'appartenance) $\mu(t)$ de chaque tuple, qui peut aider l'utilisateur dans son choix puisqu'il reflète ses préférences et permet d'ordonner les résultats.

Notons que dans le cas où la requête initiale introduite par l'utilisateur n'est pas une requête à réponse vide, seule la réponse à la requête est donnée en résultat puisqu'il n'y a pas besoin de la relaxer.

5.4.5 Exemple

L'exemple donné en *section 3.6* (p.80) est repris ci-dessous afin de tester notre implémentation. La relation du *tableau 3.3* a été créée sous la forme d'une table dans la base de données MySQL à laquelle se connectera le programme.

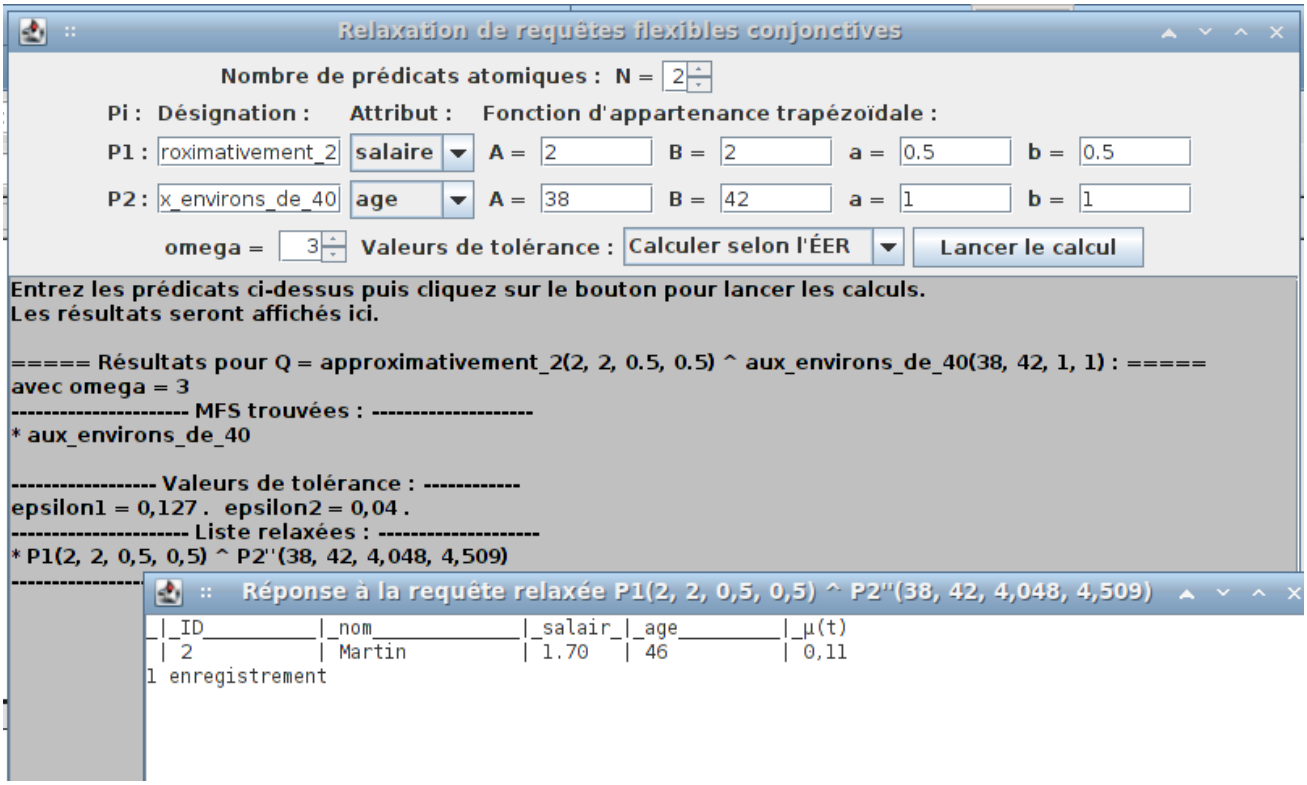


Les deux prédicats sont introduits :

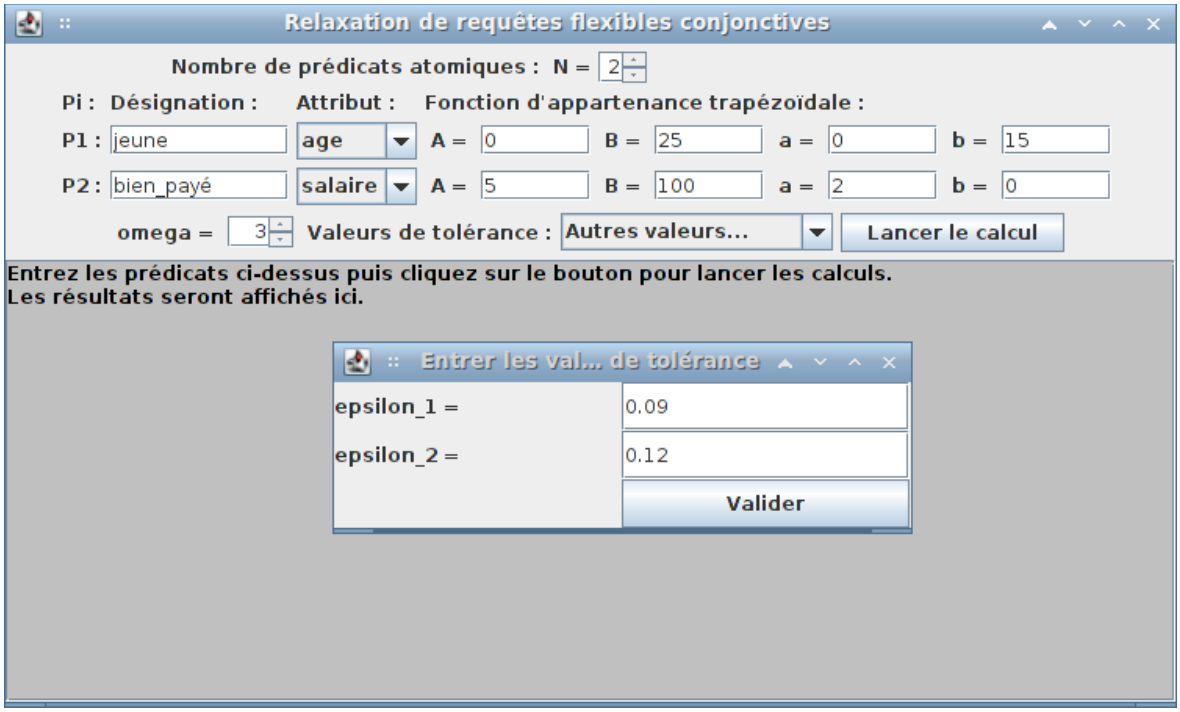
- P_1 : *approximativement_2*, sur l'attribut *salaire*, de *f.a.t.* (2, 2, 0.5, 0.5)
- P_2 : *aux_environs_de_40*, sur l'attribut *age*, de *f.a.t.* (38, 42, 1, 1)

Afin de confronter les résultats de notre programme à ceux que nous avons calculés manuellement, nous fixons à 3 la valeur de ω et nous demandons au programme de calculer les valeurs de ε_i en utilisant la propriété de l'ÉER.

Après avoir calculé que $mfs(Q) = \{P_2\}$. Le programme retourne les mêmes valeurs pour ε_1 (0.127) et ε_2 (0.04). Ces valeurs sont utilisées pour relaxer Q et le parcours du treillis retrouve la requête $P_1 \wedge T_2^{\uparrow(2)}(P_2)$ avec $T_2^{\uparrow(2)}(P_2)$ = (38, 42, 4.048, 4.509). Ceci suggère que notre programme applique correctement les *algorithmes 3.2* et *3.3*.



La réponse à la requête relaxée est l'employé Martin, avec un degré d'appartenance de 0.11. Nous constatons que l'on retrouve bien le même résultat que celui obtenu manuellement.


Relaxation de requêtes flexibles conjonctives
Nombre de prédicats atomiques : N = 2
Pi : Désignation : Attribut : Fonction d'appartenance trapézoïdale :
P1 : jeune age A = 0 B = 25 a = 0 b = 15
P2 : bien_payé salaire A = 5 B = 100 a = 2 b = 0
omega = 3 Valeurs de tolérance : Autres valeurs... Lancer le calcul
Entrez les prédicats ci-dessus puis cliquez sur le bouton pour lancer les calculs.
Les résultats seront affichés ici.
Entrer les val... de tolérance
epsilon_1 = 0.09
epsilon_2 = 0.12
Valider

Conclusion et perspectives

Actuellement, le developpement de nouvelles méthodes visant à améliorer la pertinence de l'interrogation des bases de données, qui n'arrête de croître, constitue un axe de recherche très important et prometteur en permettant de proposer rapidement des solutions coopératives qui évitent les réponses vides.

Au cours de ce travail, nous avons essayé d'aborder le problème des requêtes à réponse vide et d'apporter une solution à ces problèmes. Nous avons ainsi proposé une approche visant à trouver la meilleure relaxation de requêtes infructueuses dans un contexte flexible. Le concept clé de cette approche est la proximité sémantique des requêtes définie en utilisant la mesure de la distance de Hausdorff. L'approche proposée peut aussi bien être appliquée à la fois pour les requêtes ponctuelles et à intervalle.

En perspective, il serait intéressant de :

- étendre l'approche à des attributs avec des domaines non métriques (tels que l'attribut de *couleur*). En effet, dans ce travail, seuls les attributs avec des domaines dotés d'une métrique ont été considérés.
- réaliser d'autres études expérimentales, nécessaires pour démontrer l'efficience et l'efficacité de l'approche,
- adapter l'approche aux systèmes de recommandation.

Annexe : Code source des classes Java

L'implémentation des algorithmes étudiés a été effectuée sous la forme d'un paquetage de classes qui complètent le prototype de [Joseph 08]. Les classes de ce prototype qui effectuent le calcul des *MFS* ont ainsi pu être utilisées directement.

En plus des classes dont nous donnons le code source ici, nous avons également effectué de nombreux ajouts, corrections et améliorations du code restant, en particulier celui concernant l'interface graphique.

Classe Requete

```
package relax;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.LinkedList;

import outils.Outils;
import entites.PredicatSimple;
import entites.RequeteConjonctive;

/**
 * Type Requete utilisé pour representer non seulement les nœuds du treillis
 * mais aussi pour les mfs.
 * Tableau d'entiers qui correspondent aux nombres de relaxations des prédicats
respectifs.
 * ex: P1 ^ P2''' ^ P3' ^ P4 ^ P5'' => [0, 3, 1, 0, 2]
 * dans le cas des mfs, des -1 sont mis à la place des prédicats manquants
 * ex: P2''' ^ P4 ^ P5'' => [-1, 3, -1, 0, 2]
 * @author nh2
 *
 */
public class Requete extends ArrayList<Integer> {


  public static int omega; // nbre max de relaxations par prédicat
  public static RequeteConjonctive Qinitiale;
  public static int N; // nb de prédicats

  public static LinkedList<Requete> mfs_Q; //liste de mfs de la requete initiale
  public static double epsilon[]; // valeurs de tolérance pour chaque prédicat
```

```
private static final long serialVersionUID = 1L;


/**
 * Constructeur
 *
 */
public Requete() {
  super();
}

/**
 * Retourne une copie de cette instance de Requete.
 * Contrairement à ArrayList.clone(), cette méthode copie les éléments du tableau.
 * @return un clone de cette instance de Requete
 * @see java.util.ArrayList#clone()
 */
public Requete clone(){
  Requete copie = new Requete();
  for (Integer predicat : this) {
    copie.add(predicat);
  }
  return copie;
}

/**
 * Déduit l'ensemble des MFS de cette requête à partir
 * de celles de la requête Q initiale:
 * Les prédicats des MFS de Q sont remplacés par les prédicats relaxés
 * de cette requête correspondants.
 * @return liste de MFS
 */
public LinkedList<Requete> mfs() {
  LinkedList<Requete> listeMfs = new LinkedList<Requete>();
  for (Requete une_mfs : mfs_Q) {
    Requete nouvelle_mfs = new Requete();
    for (int i = 0; i < une_mfs.size(); i++) {
      if (une_mfs.get(i) == -1) {
        nouvelle_mfs.add(-1);
      } else {
        nouvelle_mfs.add(this.get(i));
      }
    }
```

```
      listeMfs.add(nouvelle_mfs);
    }
    return listeMfs;
  }

  /**
   *
   * @param une_mfs
   * @return vrai si une_mfs est une sous-requete de cette requete
   */
  public boolean contient(Requete une_mfs){
    boolean existe=true;
    for (int i = 0; i < une_mfs.size(); i++) {
      if (une_mfs.get(i)>=0) {
        existe = existe && (this.get(i)==une_mfs.get(i));
      }
    }
    return existe;
  }

  /**
   * borne a du prédicat Pi calculée en fonction du nombre de relaxations du
prédicat initial
   *
   * @param i indice du prédicat
   * @return borne a du prédicat désigné par indicePredicat t.m.f.=(A, B, a, b)
   */
  public double getBorne_a(int i){
    PredicatSimple Pinitiale = Qinitiale.get(i);
    double a_initiale = Pinitiale.getBorne_a();
    double delta_l_epsilon = Pinitiale.getBorneA() * epsilon[i];
    double a = a_initiale + this.get(i) * delta_l_epsilon;
    return a;
  }

  /**
   * borne b du prédicat Pi calculée en fonction du nombre de relaxations du
prédicat initial
   *
   * @param i indice du prédicat
   * @return borne b du prédicat désigné par indicePredicat t.m.f.=(A, B, a, b)
   */
  public double getBorne_b(int i){
```

```
  PredicatSimple Pinitiale = Qinitiale.get(i);
  double b_initiale = Pinitiale.getBorne_b();
  double delta_r_epsilon = Pinitiale.getBorneB() * epsilon[i]/(1-epsilon[i]);
  double b = b_initiale + this.get(i) * delta_r_epsilon;
  return b;
}

/**
 * borne A du prédicat Pi rapportée simplement du prédicat initial
 *
 * @param i indice du prédicat
 * @return borne A du prédicat désigné par indicePredicat t.m.f.=(A, B, a, b)
 */
public double getBorneA(int i){
  PredicatSimple Pinitiale = Qinitiale.get(i);
  double A_initiale = Pinitiale.getBorneA();
  return A_initiale;
}


/**
 * borne B du prédicat Pi rapportée simplement du prédicat initial
 *
 * @param i indice du prédicat
 * @return borne B du prédicat désigné par indicePredicat t.m.f.=(A, B, a, b)
 */
public double getBorneB(int i){
  PredicatSimple Pinitiale = Qinitiale.get(i);
  double B_initiale = Pinitiale.getBorneB();
  return B_initiale;
}


/**
 * Convertit cette requête au format RequeteConjonctive (package entites)
 * @return la requête au format RequeteConjonctive
 */
public RequeteConjonctive convertirEnRequeteConjonctive(){
  RequeteConjonctive req = new RequeteConjonctive();
  for (int i = 0; i < this.size(); i++) {
    if (this.get(i)>=0){     // -1 = prédicat absent
      PredicatSimple p = Qinitiale.get(i);
      req.add(new PredicatSimple(p.getValeurPredicat(), p.getBorneA(),
```

```
p.getBorneB(),
                    this.getBorne_a(i), this.getBorne_b(i),
                    p.getNomColonne()));
      }
    }
    return req;
  }

  /**
   * Initialise le champ mfs_Q avec la liste des MFS de la requête initiale Q
   * (calculée grâce au package application) en les convertissant au préalable
   * au format convenu (type Requete avec des -1 pour les prédicats manquants)
   * @param listeMfs
   */
  static public void setMfs_Q(LinkedList<RequeteConjonctive> listeMfs){
    mfs_Q = new LinkedList<Requete>();
    for (RequeteConjonctive mfs : listeMfs) {
      Requete uneMfs=new Requete();
      for (int i = 0; i < N; i++) {
        PredicatSimple p = Qinitiale.get(i);
        int predicat = -1; // -1 si le prédicat ne fait pas partie de la Mfs
        for (PredicatSimple pMfs : mfs) {
          if (p.getValeurPredicat().equals(pMfs.getValeurPredicat())
              && p.getNomColonne().equals(pMfs.getNomColonne())){
            predicat = 0; // 0 si le prédicat fait partie de la Mfs (pas de
relaxations ici donc pas de valeurs > 0)
          }
        }
        uneMfs.add(predicat);
      }
      mfs_Q.add(uneMfs);
    }
  }


  /**
   * Evalue la requête sur la base de données.
   * utilise Outils.test() (package outils)
   * @return vrai si le résultat est non-vide
   */
  public boolean evaluer(){
    RequeteConjonctive req = this.convertirEnRequeteConjonctive();
    return !Outils.test(req);
```

```
  }

  /**
   *
   * @return distance de Hausdorff de la requête Q initiale à cette requête
   */
  public double distance(){
    double moy = 0;
    /* C'est la moyenne des distances entre les prédicats :
     * dH(P,P') = max(a'-a, b'-b) / 2
     */
    for (int i = 0; i < this.size(); i++) {
      double aPrimeMoinsa = this.getBorne_a(i)-Qinitiale.get(i).getBorne_a();
      double bPrimeMoinsb = this.getBorne_b(i)-Qinitiale.get(i).getBorne_b();
      double max;
      if (aPrimeMoinsa > bPrimeMoinsb)
        max = aPrimeMoinsa;
      else
        max = bPrimeMoinsb;
      moy = moy + 0.5 * max;
    }
    return moy / this.size();
  }

  /** Retourne une chaîne de caractères représentant cette requête avec les
   *  f.a.t. des prédicats (et autant de signes ' qu'il y a eu de relaxations)
   *
   * exemple: "P1(5, 10, 1, 1) ^ P2''(100, 150, 14.191, 17.322) ^ P3'(20, 40, 3.212,
4.154)"
   *
   */
  @Override
  public String toString() {
    String chaine="";
    for (int i = 0; i < size(); i++) {
      if (get(i)>=0){
        chaine += (chaine.length()>0?" ^ ":"")+"P"+(i+1);
        for (int k = 0; k < get(i); k++) {
          chaine += "'";
        }
        DecimalFormat df = new DecimalFormat("0.###");
        chaine += "("+df.format(getBorneA(i))+", "+df.format(getBorneB(i))+", "
            + df.format(getBorne_a(i))+", "+df.format(getBorne_b(i))+")";
```

```
        }
      }
      return chaine;
    }

}
```

Classe Treillis

```
package relax;

import java.util.ArrayList;

/**
 * @author nh2
 *
 */
public class Treillis {

  /**
   * Donne toutes les requêtes relaxées correspondant à un niveau du treillis
   * @param nbPredicats nombre de prédicats dans chaque requête
   * @param niveau niveau du treillis (c'est aussi la somme des nombres de
relaxations des prédicats de chaque requête)
   * @return liste de requêtes
   */
  public ArrayList<Requete> niveau(int nbPredicats, int niveau){
    int omega=Requete.omega; // nbre max de relaxations par prédicat
    ArrayList<Requete> niv = new ArrayList<Requete>();
    /* l'algorithme correspond en fait à la recherche des nombres écrits
     * en base (omega+1) à (nbPredicats) chiffres dont la somme des chiffres
     * est (niveau)
     */
    int reste, quotient;
    for (int k = 0; k < Math.pow(omega+1, nbPredicats); k++) {
      quotient=k; // le nombre décimal k est converti en base (omega+1) par
divisions successives
      int somme=0;
      for (int i = 0; i < nbPredicats; i++) {
        reste = quotient % (omega+1);
        quotient = quotient / (omega+1);
        somme+=reste;
      }
      if (somme==niveau) {  // ne prendre que les nombres dont la somme des chiffres
est (niveau)
        Requete req = new Requete();
        quotient=k;
        for (int i = 0; i < nbPredicats; i++) {
          reste = quotient % (omega+1);
          quotient = quotient / (omega+1);
```

```
            req.add(reste);
          }
          niv.add(req);
        }
      }
      return niv;
    }
}
```

Classe Relaxation

```
package relax;

import java.util.LinkedList;

import entites.PredicatSimple;
import entites.RequeteConjonctive;

public class Relaxation {
  private LinkedList<Requete> listeRelaxees;

  private Treillis treillis;

  private Requete meilleure;

  /**
   * Constructeur
   * @param Qinitiale
   * @param omega
   * @param epsilon
   */
  public Relaxation(RequeteConjonctive Qinitiale, int omega, double[] epsilon) {
    listeRelaxees = new LinkedList<Requete>();
    Requete.Qinitiale = Qinitiale;
    Requete.N = Requete.Qinitiale.size();
    Requete.omega = omega;
    if (epsilon[0] == 0)
      epsilon = valeursDeTolerance(Qinitiale, omega);
    Requete.epsilon = epsilon;
    treillis = new Treillis();
  }

  /**
   * Calcule les valeurs de tolérance epsilon_i pour tous les prédicats Pi, en
   * prenant en compte la propriété de l'Égalité de l'Effet de Relaxation
   *
   * @param Qinitiale
   *            la requête infructueuse initiale Q = P1 ^ ... ^ PN
   * @param omega
   *            nombre maximal de relaxations par prédicat
   * @return les valeurs de tolérance epsilon_i calculées pour tous les
   *         prédicats
```

```
   */
  private double[] valeursDeTolerance(RequeteConjonctive Qinitiale, int omega) {
    int N = Qinitiale.size();
    double[] epsilon = new double[N];
    double epsilonMax = (3 - Math.sqrt(5)) / 2;
    int s = 1;
    boolean termine = false;
    while (!termine) {
      epsilon[s - 1] = epsilonMax / omega;
      termine = true;
      for (int i = 1; (i <= N) && termine; i++) {
        if (i != s) {
          PredicatSimple Pi = Qinitiale.get(i - 1);
          PredicatSimple Ps = Qinitiale.get(s - 1);
          epsilon[i - 1] = calculEER(Pi, Ps, epsilon[s - 1]); // calcule epsilon_i
selon ÉER
          if (omega * epsilon[i - 1] > epsilonMax) {
            s = i;
            termine = false;
          }
        }
      }
    }
    return epsilon;
  }

  /**
   * Calcul de la valeur de tolérance epsilon_i pour le prédicat Pi donné en
   * paramètre, à partir du prédicat Ps et sa valeur de tolérance epsilon_s,
   * en se basant sur la propriété de l'Égalité de l'Effet de Relaxation (ÉER)
   *
   * @param Pi
   *            prédicat dont on veut calculer la valeur de tolérance
   *            epsilon_i
   * @param Ps
   *            prédicat (différent de Pi) à partir duquel le calcul est
   *            effectué
   * @param epsilon_s
   *            valeur de tolérance de Ps
   * @return
   */
  private double calculEER(PredicatSimple Pi, PredicatSimple Ps,
      double epsilon_s) {
```

```
  double Ai = Pi.getBorneA();
  double Bi = Pi.getBorneB();
  double ai = Pi.getBorne_a();
  double bi = Pi.getBorne_b();
  double As = Ps.getBorneA();
  double Bs = Ps.getBorneB();
  double as = Ps.getBorne_a();
  double bs = Ps.getBorne_b();

  double Ki = (As * epsilon_s + Bs * epsilon_s / (1 - epsilon_s))
      * (Bi - Ai + ai + bi) / (Bs - As + as + bs);

  double epsilon_i = (Ai + Bi + Ki - Math.sqrt((Ai + Bi + Ki)
      * (Ai + Bi + Ki) - 4 * Ai * Ki))
      / (2 * Ai);

  return epsilon_i;
}

/**
 *
 * @return liste de requêtes relaxées à réponse non vide
 */
public LinkedList<Requete> getListeRelaxees() {
  return listeRelaxees;
}

/**
 * Lance le parcours du treillis de requêtes relaxées les résultats sont stockés
 * dans les champs listeRelaxees (liste de requêtes relaxées non échouantes)
 * et meilleure (requête relaxée la plus proche de la requête initiale).
 * Attention: Requete.mfs_Q doit être initialisé avec Requete.setMfs_Q() !
 */
public void relaxer() {
  int i = 1;
  while ((i <= (Requete.omega * Requete.N)) && listeRelaxees.isEmpty()) {
    for (Requete relaxee : treillis.niveau(Requete.N, i)) {
      boolean modif = true;
      /*
       * on obtient les parents en otant une relaxation a chaque
       * predicat respectif
       */
      for (int j = 0; j < relaxee.size(); j++) {
```

```
          if (relaxee.get(j) > 0) {
            Requete parent = relaxee.clone();
            parent.set(j, relaxee.get(j) - 1);
            for (Requete une_mfs : parent.mfs()) {
              modif = modif && !(relaxee.contient(une_mfs));
              if ((relaxee.contient(une_mfs))) System.out.println(relaxee+" contient
"+une_mfs);
            }
          }
        }
        System.out.println(relaxee+" : modif="+modif);
        if (modif) {
          if (relaxee.evaluer()) {
            listeRelaxees.add(relaxee);
          }
        }
      }
      i++;
    }
    if (!listeRelaxees.isEmpty()) {
      double hausdorff = 1e300d; // pas l'infini mais presque :)
      for (Requete req : listeRelaxees) {
        double distReq = req.distance();
        if (distReq < hausdorff) {
          hausdorff = distReq;
          meilleure = req;
        }
      }
    }
  }

  /**
   * Retourne la requête relaxée la plus proche de la requête initiale,
   * telle que calculée par la méthode relaxer()
   * Cette dernière doit avoir été exécutée au préalable.
   * (Accesseur pour le champ privé meilleure)
   * @return meilleure requête
   */
  public Requete getMeilleure() {
    return meilleure;
  }
}
```

Références bibliographiques

[Andreasen, Pivert 94] T. Andreasen, O. Pivert, (1994). *On the weakening of fuzzy relational queries.* Proc. of the 8th Int. Symp. on Methodologies for intelligent Systems, pp. 144-151, Charlotte, USA.

[Börzsönyi *et al.* 01] S. Börzsönyi, D. Kossmann, K. Stocker, (2001). *The Skyline operator*. Proc. 17th IEEE Inter. Conf. on Data Engineering, 421-430.

[Bosc *et al.* 04] P. Bosc, A. Hadjali, O. Pivert, (2004). *Fuzzy closeness relation as a basis for weakening fuzzy relational queries.* 6th International Conference on Flexible Query Answering Systems (FQAS'04), Lyon, France, June 24-26, pp. 41-53,

[Bosc *et al.* 04] P. Bosc, L. Liétard, O. Pivert, D. Rocacher, *Gradualité et imprécision dans les bases de données.* Paris, Editions Ellipses.

[Bosc *et al.* 05] P. Bosc, A. Hadjali, O. Pivert, (2005) *Towards a tolerance-based technique for cooperative answering of fuzzy queries against regular databases*, in the 7th Int. Conf. CoopIS, Cyprus, LNCS 3760, pp. 256-273.

[Bosc *et al.* 06] P. Bosc, A. Hadjali, O. Pivert, (2006) *Relaxation paradigm in a flexible querying context,* in the *7th Int. Conference on Flexible Query Answering Systems* (FQAS'06), Milano, LNCS 4027, Springer-Verlag,, pp. 39-50.

[Bosc *et al.* 06] P. Bosc, A. Hadjali, O. Pivert, (2006). *Weakening of fuzzy relational queries: An absolute proximity relation-based approach,* Journal of Mathware & Soft Computing, Vol. 14(1), 2007.

[Bosc *et al.* 07] P. Bosc, A. Hadjali, O. Pivert, (2007) *On the versatility of fuzzy sets for modeling flexible queries*. In Galindo, J. (Ed.), *Handbook of Research on Fuzzy Information Processing in Databases,* Vol. I, pp. 143-166. Hershey, PA, USA

[Bosc *et al.* 07] P. Bosc, A. Hadjali, O. Pivert, (2007) *Weakening of fuzzy relational queries: An absolute proximity relation-based approach,* Journal of Mathware & Soft Computing, Vol. 14(1), pp. 35-55.

[Bosc *et al.* 08a] P. Bosc, A. Hadjali, O. Pivert (2008), *Incremental controlled relaxation of failing flexible queries*, Journal of Intelligent Information Systems, doi:10.1007/s10844-008-0071-6.

[Bosc *et al.* 08b] P. Bosc, A. Hadjali, O. Pivert, (2008). *Empty versus*

overabundant answers to flexible relational queries, Fuzzy sets and Systems Journal, Vol 159 , 12.

[Bosc *et al.* 08c] P. Bosc, A. Hadjali, O. Pivert (2008), *Cooperative Answering to Flexible Queries Via a Tolerance Relation*, A. An *et al.* (Eds.): ISMIS 2008, LNAI 4994, pp. 288–297.

[Bosc *et al.* 08d] P. Bosc, A. Hadjali, O. Pivert (2008), *Une Approche Coopérative pour le Traitement des Requêtes Flexibles à Réponse Vide ou Pléthorique*

[Bosc, Pivert 92] P. Bosc, O. Pivert, (1992).*Some approaches for relational databases flexible querying.* Journal of Intelligent Information Systems, 1, 323-354.

[Bosc, Pivert 95a] P. Bosc, O. Pivert (1995). *SQLf: A relational database language for fuzzy querying*. IEEE Transactions on Fuzzy Systems, 3(1).

[Bosc, Pivert 95b] P. Bosc, O. Pivert (1995). *On the efficiency of the alpha-cut distribution method to evaluate simple fuzzy relational queries*. In B. Bouchon-Meunier, R. R. Yager, & L. A. Zadeh (Eds.), *Advances in fuzzy systems: Applications and theory* (vol. 4: *Fuzzy logic and soft computing*, pp. 251-260). World Scientific.

[Bouchon-Meunier 07] B. Bouchon-Meunier (2007) *La logique floue*. Collection « Que sais-je ? » N° 2702. 4e éd. Presses Universitaires de France.

[Chaudhuri, Rosenfeld 99] B. Chaudhuri and A. Rosenfeld (1999), *A modified Hausdorff distance between fuzzy sets*, Information Sciences, Vol. 118, pp. 159-171.

[Chomicki 03] J. Chomicki, (2003). *Preference formulas in relational queries*. ACM Transactions on Database Systems, 28, 427-466.

[Christiansen *et al.* 97] H. Christiansen, H. Larsen, T. Andreasen (1997) *Flexible Query Answering Systems*, Kluwer Academic Publishers.

[Codd 70] E.F. Codd (1970), *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 13, N°6, pp. 377-387.

[Cross, Sudkamp 02] V. Cross and T. Sudkamp (2002), *Similarity and*

Compatibility in Fuzzy Set Theory: Assessment and Applications, Studies in Fuzziness and Soft Computing, No93, Physica-Verlag.

[Cuppens, Demolombe 88] E Cuppens and R. Demolombe (1988), *Cooperative Answering: A Methodology to Provide Intelligent Access to Databases*, Proceedings of the Second International Conference on Expert Database Systems, Tysons Comer, VA, April 25-27,1988, George Mason University, Fairfax, VA, pp. 333-353.

[D'Atri, Tarantino 89] A. D'Atri and L. Tarantino (1989), *From Browsing to Querying*. Data Eng., 12(2), 46-5.

[De Calmès *et al.* 03] M. de Calmès, D. Dubois, E. Hullermeier, H. Prade, F. Sedes (2003) *Flexibility and fuzzy case-based evaluation in querying: an illustration in a n experimental setting.* Int. J. of Uncertainty, Fuzziness and Knowledge-based Systems, 11(1), 43-66.

[Dix, Patrick 94] A. J. Dix and A. Patrick (1994), *Query by Browsing,* in Interfaces to Database Systems, Proceedings of the Second International Workshop, Lancaster University, July 13-15 1994, P. Sawyer, ed. Workshops in Computing, Springer-Verlag, 1995, pp. 236-248.

[Dubois, Prade 83] D. Dubois and H. Prade (1983), *On distances between fuzzy points and their use for plausible reasoning*, In Proc. Int. Conf. on Systems, Man and Cybernetics, pp. 300-303.

[Dubois, Prade 88] D. Dubois, Prade H. (1988), *Possibility Theory,* Plenum Press,.

[Gaasterland 97] T. Gaasterland (1997), *Cooperative answering through controlled query relaxation*, *IEEE Expert*, 12(5), pp. 48-59.

[Gaasterland *et al.* 92] T. Gaasterland, P. Godfrey, J. Minker (1992) *An overview of cooperative answering.* Journal of Intelligent Information Systems, 1(2), pp. 123-157.

[Gaasterland *et al.* 92] T., Gaasterland, P. Godfrey, and J. Minker (1992), *Relaxation as a platform for cooperative answering*. Journal of Intelligent Information Systems, 1(3-4), pp. 293-321.

[Galindo 08] J. Galindo (2008), *Introduction and trends to fuzzy logic and fuzzy databases.* In Galindo, J. (Ed.), *Handbook of Research on Fuzzy Information Processing in Databases,* Vol. I, pp. 1-33. Hershey, PA, USA

[Gardarin 00] G. Gardarin (2000), *Bases de données. Objet et relationnel*. Ed. Eyrolles. Deuxième tirage.

[Godfrey 97] P. Godfrey (1997), *Minimization in cooperative response to failing database queries*, Int. Journal of Cooperative Information Systems, 6(2), pp. 95-149.

[Grice 75] H.P. Grice (1975). *Logic and conversation.* In P. Cole et J.L. Morgan, editors, *Syntax and semantics: Speech Acts.* Vol.3. Academic Press, New York, pp. 41-58.

[Hadjali *et al.* 03] A. Hadjali, D. Dubois and H. Prade (2003), *Qualitative reasoning based on fuzzy relative orders of magnitude*. IEEE Transactions on Fuzzy Systems, Vol. 11, No 1, pp. 9-23.

[Hadjali *et al.* 08] A. Hadjali, S. Kaci, H. Prade (2008) *Database preferences queries – A possibilistic logic approach with synmbolic priorities*. In Proc. of the 5th Inter. Symposium on Foundations of Information and Knowledge Systems (FoIKS'08), Feb. 11-15, Pisa (Italy), LNCS 4932, 291-310.

[Joseph 08] B. Joseph (2008), *Réponses Coopératives et Interrogation Flexible de Bases de Données.* Rapport de Projet d'Ingénieur Logiciels et Systèmes Informatiques, ENSSAT-Lannion. Université de Rennes 1.

[Kaplan 82] S.J. Kaplan (1982), *Cooperative responses from a portable natural language query system*, Artificial Intelligence 19, 165–187.

[Kiessling, Köstler 01] W. Kiessling, G. Köstler, (2001). *Preference SQL – Design, Implementation, Experiences.* Report 2001-7 Universität Augsburg, Institut für Informatik.

[Larsen *et al.* 01] H. Larsen, J. Kacpryk, S. Zadrozny, T. Andreasen, and H. Christiansen (Eds.) (2001), *Flexible Query Answering Systems, Recent Advances,* Physica Verlag.

[Liétard *et al.* 06] L. Liétard, O. Pivert, D. Rocacher (2006) *Réécriture et*

évaluation de requêtes flexibles rapport APMD, mars 2006.

[Motro 86] A. Motro (1986). *SEAVE: A mechanism for verifying user presuppositions in query systems.* ACM Trans. on Off. Inf. Syst., 4(4), 312-330.

[Motro 90] A. Motro (1990). *FLEX: A tolerant and cooperative user interface databases.* IEEE Transactions on Knowledge and Data Engineering, 2(2), 231-246.

[Motro 96] A. Motro (1996) *Panorama: A Database System that Annotates its Answers to Queries with their Properties*. Journal of Intelligent Information Systems, Vol. 7, No. 1, pp. 51–73.

[Motro 00] A. Motro (2000) *Cooperative database systems*. In *Encyclopedia of library and information science*, vol. 66, suppl. 29. Marcel Dekker Inc., pp. 79-97

[Puri, Ralescu 83] M.L. Puri and D.A. Ralescu (1983), *Differentials of fuzzy functions, Journal of Mathematical Analysis and Applications*, Vol. 91, pp. 552-558.

[Ras, Dardzinska 06] Z.W. Ras, A. Dardzinska (2006), *Solving failing queries through cooperation and collaboration, world wide web*: internet and web information systems, 9, 173-186

[Rosado *et al.* 06] A. Rosado, R. A. Ribeiro, S. Zadrozny, J. Kacprzyk (2006), *Flexible query languages for relational databases: An overview.* In G. Bordogna & G. Psaila (Eds.), *Flexible databases supporting imprecision and uncertainty.* Springer-Verlag.

[Voglozin *et al.* 05] W.A. Voglozin, G. Rashia, L. Ughetto, N. Mouaddib (2005), *Querying the SaintEtiq summaries: Dealing with null answers.* Proc. IEEE Inter. Conf. on Fuzzy Systems, pp. 585-590, USA.

[Zadeh 65] L.A. Zadeh, *Fuzzy sets*, Inform. Cont. 8 (3) (1965) 338-353.