# Une approche pour la construction d'évaluateurs adaptables de requêtes

Tuyet-Trinh Vu

# *INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE*

***N° attribué par la bibliothèque***
|__|__|__|__|__|__|__|__|__|__|

***T H E S E***

pour obtenir le grade de

**DOCTEUR DE L'INPG**

***Spécialité : « Informatique : Systèmes et Communications »***

préparée au laboratoire Logiciels, Systèmes, Réseaux (LSR)
dans le cadre de **l'Ecole Doctorale *« Mathématiques, Sciences et Technologie de l'Information »***

présentée et soutenue publiquement

par

**Tuyet-Trinh VU**

Le 10 Février 2005

***Titre :***

**Une approche pour la construction d'évaluateurs adaptables de requêtes**

***Directeur de thèse :***

Pr. Christine COLLET

**JURY**

| | |
|---|---|
| Mme. Brigitte PLATEAU | , Président |
| M. Georges GARDARIN | , Rapporteur |
| M. Patrick VALDURIEZ | , Rapporteur |
| Mme. Christine COLLET | , Directeur de thèse |
| Mme. Béatrice FINANCE | , Examinateur |
| M. Alexandre LEFEBVRE | , Examinateur |

Tuyet-Trinh Vu

***Une approche pour la construction d'évaluateurs adaptables de requêtes***

190 pages.

# Remerciements

Je tiens à remercier sincèrement :

Brigitte Plateau, Professeur de l'Institut National Polytechnique de Grenoble (INPG) qui m'a fait l'honneur de présider mon jury de thèse.
Georges Gardarin, Professeur de l'Université de Versailles Saint-Quentin-en-Yvelines (UVSQ) et Patrick Valduriez, Directeur de recherche de l'Institut National de recherche en Informatique et en Automatique (INRIA), pour avoir accepté de juger mon travail et pour leurs remarques qui m'ont permis d'améliorer ce manuscrit.
Béatrice Finance, Maître de Conférences de UVSQ, Alexandre Lefebvre, Ingénieur-Chercheur à France Télécom R&D pour avoir accepté d'examiner mon travail et pour leur intérêt apporté sur mon travail de thèse.
Christine Collet, Professeur de l'INPG et responsable de l'équipe de base de données du laboratoire LSR-IMAG, qui est à l'origine de cette thèse. Je remercie la directrice de thèse qui m'a accueillie dans son équipe. Je la remercie pour tous ses conseils, ses remarques, et ses corrections pendant la réalisation de cette thèse. Son optimisme et ses encouragements m'ont permis de surmonter les moments de doute et d'angoisse de la thèse.

Les membres anciens et actuels ainsi que ceux de passage de l'équipe base de données, du projet NODS pour leur aide, leur encouragement tout au long de la réalisation de cette thèse. Un merci tout particulier à Christophe Bobineau pour sa participation active à la préparation des démonstrations de ce travail. Je le remercie aussi pour les discussions qui m'ont permis d'améliorer ce travail ainsi que pour son temps consacré à la lecture de ce manuscrit. Une pensée chaleureuse à : MariadelPilar Villamil avec qui j'ai eu le plaisir de partager de nombreuses discussions scientifiques et autres. Merci *Pili* pour tes encouragements dans les moments difficiles de la thèse, pour tes leçons d'espagnol et pour ton amitié ; Genoveva VargasSolar pour son aide dans la préparation de la soutenance et de m'avoir fait découvrir la culture mexicaine ; Claudia Rocancio pour ses encouragements tout au long de la préparation de cette thèse ; mes collègues de bureau Edgard Benitez, Khalid Belhajjame, Denis Biennier, Giang Vu, Kien Cao avec qui je partage, entre autres, les goûters, la musique et aussi les rigolades. Certains sont déjà partis, d'autres sont restés pour maintenir l'ambiance de ce D304 (*Bon courage les "petits"!*). A tous les autres : Quynh, Gennaro, Patricia, Luciano, Stéphane, Tanguy, Mourad, Hanh, Laurent, Fabrice, Naga, Cyril, Trini, Pierre, ... pour leur soutien et leur amitié.

Les membres du projet ACI MediaGRID pour les discusssions qui ont contribué à la validation de mon

travail. Un merci particulier à Béatrice Finance pour son aide lors de mes premiers pas dans mon travail de thèse.

Les membres du laboratoire LSR pour l'ambiance agréable au sein de cet établissement. Un merci particulier à Farid Ouabdesselam pour ses encouragements pendant les moments difficiles de la rédaction de cette thèse.

Mes professeurs à l'Institut Polytechnique de Hanoï et à l'Institut Francophonie de l'Informatique pour leur cours d'initiation à l'informatique. Je les remercie surtout pour leurs encouragements et leur confiance.

Mes amis vietnamiens à Grenoble pour les moments agéables partagés ensemble. Je pense en particulier à Quynh, Hoang, Huân, Giang, Cuong, Bac, Tuyêt, Binh, Kien, Minh et quelques autres.
Les amis que j'ai eu la chance de rencontrer lors de ce séjour : Florence, François, *les* Hervé, Xiaohui, David, *les* Nicolas, Cristina, Nathalie, *les* Olivier, Antonella, Antoine, la *petite* Julie et tous les autres.
Mes amis de longue date au Vietnam, en France ou ailleurs : *les* Hà, Chi, *les* Mai, Câm, *les* Anh, *les* Thang, *les* Tuân, Tùng et quelques autres pour leur soutien dépassant les frontières.

Ma famille : mon père, ma mère, mon frère, ma *belle* sœur et mes deux adorables neveux pour l'amour qu'ils m'apportent depuis toujours. Je les remercie de m'avoir enseignée (et de m'enseigner encore), de m'avoir soutenue tout au long de mes études (et me soutenir encore). Merci aussi pour la question stressante du dimanche (*Quand rentres tu?*) qui a été un des facteurs décisifs pour finir la rédaction de cette thèse. *Que cette thèse soit l'accomplissement de leur soutien et leur confiance!* Ma famille est tellement grande qu'il m'est impossible de faire figurer ici les noms de tous ses membres à qui je voudrais exprimer toutes mes reconnaissances. Je pense en particulier à ma tante partie durant cette thèse, à ma grand-mère, à mes oncles, à mes tantes, à mes cousins et cousines.
Jean-Louis, Annie et toute la famille pour leur aide et leur soutien tout au long de ces années loin de mon pays. Je les remercie de m'avoir accueillie chaleureusement dans leur famille qui depuis quelques années fait partie de ma grande famille. Un merci tout particulier à Jean-Louis pour toutes les choses qu'il a faites pour moi depuis ... avant mon arrivée en France.
Un grand merci à Son ... *Il sait pourquoi.*

Merci à toutes et à tous pour la présence et les messages de soutien, à l'occasion de la soutenance, qui me touchent beaucoup.

Grenoble, 10 Février 2004

Tuyet-Trinh Vu

# Table des matières

# Liste des tableaux

# Liste des figures

# CHAPITRE 1

# Introduction

## 1.1 Motivation

L'utilisation croissante de systèmes informatiques dans différents domaines d'application et les récents progrès en infrastructure de communication conduisent à la réalisation de systèmes d'information répartis de grande envergure. De tels systèmes disposent souvent d'une fonction d'interrogation, plus ou moins complexe, permettant l'accès et le partage des données. Bien que l'intérêt de cette fonction ne soit plus à démontrer, sa construction reste une tâche difficile et laborieuse pour les raisons suivantes :

*La diversité des besoins en interrogation*

Construite pour faciliter l'accès aux données préexistantes, la fonction d'interrogation doit tenir compte de la nature des données interrogées afin d'assurer un accès suffisamment précis selon leur modèle de représentation. Par exemple, les données relationnelles peuvent être interrogées par l'intermédiaire de requêtes SQL sensées traduire des manipulations des tables, les objets sont interrogés par un langage approprié, comme OQL, capable d'exprimer les manipulations sur les graphes, ou encore les données XML sont interrogées par des requêtes XQuery permettant la navigation sur les structures plus ou moins rigides. Par ailleurs, les applications peuvent exiger des traitements spécifiques sur les données. Certaines ont besoin juste de récupérer les données tandis que d'autres peuvent vouloir réaliser des calculs complexes (e.g. fonctions définies par l'utilisateur) voire imposer des contraintes spécifiques sur les résultats récupérés. Outre ces besoins, la structure de la fonction d'interrogation pourrait aussi être complexe et variée. Elle pourrait être centralisée, distribuée ou composée d'unités capables de "collaborer" entre elles selon les conditions du déploiement.

De ce fait, il est difficile de fournir des fonctions d'interrogation génériques répondant à *tous* les besoins d'application *dans n'importe quelle circonstance*.

*La complexité et la dynamicité de l'environnement d'interrogation*

L'interrogation des données provenant des sources distantes dépend, d'une part, de la "collaboration" de ces sources, et d'autre part, de l'infrastructure de communication. En effet, les connaissances sur les sources, nécessaires pour la décision de la meilleure stratégie d'exécution des requêtes, peuvent ne pas être disponibles. De plus, les conditions d'accès aux sources peuvent varier en raison du trafic sur le réseau et/ou de la disponibilité des données, et les besoins d'interrogation de l'utilisateur eux-mêmes, peuvent évoluer. L'environnement d'exécution des requêtes est donc très complexe et souvent imprévisible.

Dans un tel environnement, l'interrogation basée sur un mode de traitement par lot (*batch*), en appliquant un processus de traitement statique, n'est plus adaptée. Faute de connaissances complètes sur les données interrogées ainsi que sur l'environnement d'exécution, il est difficile de prendre, a priori, une décision efficace sur l'exécution des requêtes. De plus, rien ne peut garantir l'exactitude des connaissances présentes dans le système. En outre, le volume important des données interrogées et la communication à longue distance font que l'exécution de requête peut être longue, voire interminable. L'utilisateur soumet la requête et attend longtemps sans connaître le déroulement de sa requête ni recevoir aucune information du système. Par ailleurs, l'utilisateur peut se contenter de quelques résultats même si ces derniers ne sont pas complets.

De ce fait, il est souhaitable que le système d'évaluation de requêtes soit capable de s'adapter à des changements ayant lieu pendant une évaluation longue de requête : il fournit des retours (*feebacks*), autorise les interactions avec l'utilisateur, et modifie la stratégie d'exécution des requêtes.

Pour toutes ces raisons, la construction de systèmes ayant une capacité d'interrogation adaptée à différents contextes est complexe. Cela motive notre travail en vue d'assister les programmeurs dans la conception et la construction de systèmes d'interrogation de données, appelés également évaluateurs de requêtes.

## 1.2 Objectif de la thèse

Cette thèse a pour but de proposer **une approche pour la construction d'évaluateurs adaptables de requêtes**. L'adaptabilité de l'évaluateur correspond à sa capacité d'être adapté et/ou de s'adapter au mieux aux besoins de l'application qui va l'utiliser et au contexte dans lequel il va être utilisé. Nous tentons d'aborder l'adaptation dans sa globalité en considérant *tout ou partie* de l'évaluateur de requêtes.

Nous distinguons trois niveaux d'adaptabilité que sont : (i) l'**adaptabilité statique** obtenue avant le démarrage de l'évaluateur ; (ii) la **personnalisation** obtenue au début de l'exécution d'une requête ayant des contraintes spécifiques ; et (iii) l'**adaptabilité dynamique** obtenue pendant l'exécution d'une requête en cours.

Ainsi, ce travail de thèse concerne aussi bien les aspects architecturaux du système[1] que les techniques d'évaluation de requête. Il s'intéresse à un support (ou un outil) pour la construction d'évaluateurs de requêtes, permettant d'assurer les trois niveaux d'adaptabilité ci-dessus. La solution proposée consiste, d'une part, à re-architecturer l'évaluateur de requêtes afin de définir un cadre d'utilisation des mécanismes d'évaluation existants de manière à favoriser leur adaptation pour différents contextes, et d'autre part, à proposer les mécanismes d'évaluation nécessaires pour assurer un ou plusieurs niveau(x) d'adaptabilité souhaité(s).

## 1.3 Démarche

Afin d'accomplir l'objectif présenté ci-dessus, nous nous sommes posés les trois questions suivantes :

1. *Quelles sont les charactéristiques d'un évaluateur de requêtes ?*

   L'évaluation de requêtes est une des fonctions principales et souvent indispensables dans les systèmes de gestion de données tels que INGRES [SWKH76], System-R* [ABC+76], O2 [BDK91], IRO-DB [SFF94], Information Manifold [LRO96], TSIMMIS [GMHI+95], Lore [MAG+97], AMOS II [JKR99], Tukwila [ILW+00], Telegraph [KCC+03], etc. Le traitement de requêtes se différencie dans les systèmes par les aspects suivants :

   - **Modèle de données** : Le modèle de données est le *"plan"* pour construire une base de données[2]. Il se compose de trois éléments : (i) la collection de *structures de données*, (ii) la collection d'*opérateurs* qui définissent les manipulations possibles sur ces structures pour extraire et pour calculer les données, et (iii) la collection de *règles d'intégrité* [Cod80]. Depuis l'apparition du premier modèle de données défini par E. F Codd, plusieurs modèles de données ont été proposés. A ce sujet, nous pouvons citer, entre autres, le modèle relationnel [Cod70], le modèle objet [ZW86, CBB+97], le modèle OEM (*Object Exchange Model*) pour représenter les données semi-structurées [MAG+97] ou encore le modèle associé à XML en cours de définition par le W3C [XML].

     Compte tenu des différents besoins en matière de modélisation et de traitement des données dans les applications, il est nécessaire de pouvoir construire des évaluateurs capables d'assurer les traitements appropriés à différents modèles. Malgré les différences dans les structures et dans la manière de manipuler les données, la plupart des données à interroger sont représentées sous forme de *collections* [GM93, Gra04]. L'abstraction de ces modèles pourrait être un moyen d'obtenir l'adaptabilité vis-à-vis du modèle de données.

   - **Optimisation** : L'optimisation a pour but de choisir une meilleure solution pour répondre à une requête donnée. C'est un élément complexe et important qui détermine les performances de l'évaluateur. L'optimisation se compose de trois dimensions principales que sont l'*espace*

[1] D'ores et déjà, sauf mention explicite contraire, le terme "système" signifie "système d'évaluation de requêtes" ou "système d'interrogation" qui sont utilisés d'une manière interchangeable avec le terme "évaluateur".

[2] "*A data model is a plan for building a database*" (http://www.computerworld.com/databasetopics/data/story/)

*de recherche*, le *modèle de coût* et la *stratégie de recherche*. L'espace de recherche est constitué par un ensemble de ses solutions candidates pour une requête donnée. Le choix de la "meilleure" solution peut se baser sur différents critères, appelé *objectif d'optimisation*, et s'appuyer sur un modèle de coût permettant de comparer quantitativement les solutions proposées. Afin d'améliorer la performance globale de l'évaluateur, l'optimisation peut appliquer une stratégie de recherche appropriée sur l'espace de recherche afin d'éviter des parcours exhaustifs [GMUW00].

En considérant le volume important de données interrogées et les différents besoins des applications, il est souhaitable que le système d'évaluation de requêtes puisse répondre à différents objectifs d'optimisation (e.g. le temps d'obtention des premiers résultats, les ressources utilisées, etc.). Par ailleurs, l'optimisation de requêtes utilise souvent une stratégie qui convient à certaines classes de requêtes et à certains objectifs d'optimisation mais pas à tous [SAC$^+$79, IW87, Swa89, UF00, BFMV00b, AH00]. Pour mieux répondre à un plus grand nombre de besoins, il serait bénéfique de pouvoir "supporter" plusieurs objectifs et stratégies d'optimisation. Ceux-ci vont être adaptés aux besoins spécifiques de l'application, voire aux requêtes individuelles. Nous remarquons que l'abstraction de la tâche d'optimisation peut être un moyen d'obtenir l'adaptabilité vis-à-vis de l'optimisation. Une telle abstraction peut être obtenue par la séparation des différents éléments de l'optimisation, à savoir le *modèle de coût* (ou l'objectif d'optimisation), l'*espace de recherche* et la *stratégie de recherche*.

- **Résultat** : La plupart des évaluateurs de requêtes calculent exactement ce que demandent les applications clientes. Pour faire face à une évaluation longue de requête, il serait préférable que le client soit capable d'exprimer ses besoins (en terme de données récupérées) d'une manière plus précise ainsi que les résultats soient retournés progressivement. Les réponses peuvent être incomplètes, approximatives, appelées communément *résultats partiels*, et sont complétés au fur et à mesure. Ces problématiques ont été abordées dans les travaux récents, présentés dans [BT98, RRH99, RH02], mais les solutions proposées concernent souvent des systèmes spécifiques et sont difficilement utilisables et exploitables dans d'autres systèmes ou d'autres contextes.

  Il serait intéressant de revoir ces propositions, voire de les re-concevoir et/ou de proposer de nouveaux mécanismes afin de rendre la construction de résultats aussi *flexible* que possible. Cela peut être un moyen d'obtenir l'adaptabilité vis-à-vis de la construction des résultats des requêtes.

- **Adaptation** : Elle signifie la capacité de l'évaluateur de requêtes à s'adapter à des changements intervenant pendant l'exécution de la requête. Cela a fait l'objet de nombreux travaux récents [GW89, UFA98, KD98, ILW$^+$00, AH00, BFMV00b, UF01] visant à procéder à l'optimisation de requêtes en plusieurs phases et à adapter la stratégie d'exécution de requête dynamiquement selon des connaissances acquises. Nous parlons d'*évaluation dynamique*. Alors que la plupart des travaux existants considèrent une adaptation à des changements des propriétés des sources et de leur accès, l'adaptation aux besoins de l'utilisateur est encore

très peu abordé [RRH99, RH02]. L'idée est d'autoriser l'intervention de l'utilisateur pendant l'exécution d'une requête afin de contrôler cette dernière et de l'affiner si l'utilisateur le souhaite. Que ce soit l'adaptation aux changements des propriétés des sources ou des besoins de l'utilisateur, il est toujours nécessaire d'observer l'exécution de requête et son environnement afin de détecter une situation qui va *déclencher* l'adaptation, de *décider* des modifications nécessaires et de les *réaliser*.

Malgré le foisonnement des travaux récents sur l'évaluation dynamique, leur usage reste très restreint. En effet, les techniques proposées sont souvent conçues pour répondre à un ou quelques problèmes de l'évaluation dynamique, mais pas à tous. Il n'est pas évident de les utiliser dans d'autres contextes et/ou ensembles [GPSF04]. Nous pensons qu'il est nécessaire de séparer les différents éléments concernant l'évaluation dynamique de requêtes, de les situer dans un cadre uniforme et compréhensible. Une telle séparation faciliterait la (ré-)utilisation, la composition des techniques existantes ainsi que la définition de nouvelles techniques tout en profitant de celles qui existent.

- **Distribution** : Les évaluateurs classiques se basent souvent sur une architecture plus ou moins centralisée, où les données sont stockées dans une base locale, et/ou sont gérées par un système de gestion d'une manière centralisée. Ils sont considérés comme une unité "robuste", "indivisible" et "fermée". Son fonctionnement est donc plus ou moins "gérable". Les récents progrès en matière de communication ainsi que le besoin de partage des données font naître de nouveaux systèmes dont l'architecture est de plus en plus dispersées. A titre d'exemple, nous pouvons citer des systèmes de médiation (centralisé ou hiérarchique) [SFF94, LRO96, GMHI$^+$95, JKR99], des systèmes pair à pair [leS, LIST03], etc.

  Dans un tel contexte, il serait préférable que les évaluateurs soient considérés comme des unités ouvertes et collaboratives permettant de déployer aisément différentes techniques d'évaluation de requêtes dans diverses architectures.

2. *Quelles sont les mécanismes pour réaliser l'adaptation dans les aspects décrits ci-dessus ?*

   Ces mécanismes dépendent fortement du moment de l'adaptation ainsi les aspects la concernant.

   Lorsqu'il s'agit de l'adaptation statique (i.e. avant le démarrage d'un évaluateur), tous les aspects peuvent être concernées. Cette adaptation consiste principalement à spécifier le modèle de données ainsi que les traitements nécessaires (e.g. l'optimisation, l'adaptation, la construction des résultats) en utilisant les techniques de paramétrage, de composition, d'héritage, de polymorphisme, de délégation, de généricité, etc.

   Quand il s'agit de la personnalisation (i.e. adaptation au début de l'évaluation d'une requête ayant des contraintes spécifiques), il est nécessaire d'employer les techniques d'évaluation de requêtes appropriées. Ces techniques consistent à choisir une stratégie d'optimisation ainsi qu'une stratégie de construction de résultats appropriée si nécessaire. Dans ce cas, nous parlons de *requête personnalisée* et d'**évaluation personnalisée**.

Pour l'adaptation dynamique, il est nécessaire d'employer les mécanismes d'observation et de réaction afin de détecter les situations qui déclenchent une adaptation et les modifications à réaliser. Par ailleurs, il faut garantir que les adaptations n'entraînent aucune erreur dans les résultats produits. Dans ce cas, nous parlons d'**évaluation adaptative** et **interactive**, les techniques essentielles pour réaliser l'adaptation dynamique.

Il est important de noter que les trois niveaux d'adaptabilité correspondant à trois formes d'adaptations ci-dessus *ne sont pas exclusifs*. En effet, pour assurer les niveaux de personnalisation et d'adaptabilité dynamique, il faut que l'évaluateur de requêtes construit dispose les capacités d'évaluation appropriées. Cela devra être prévu avant le démarrage de l'évaluateur (i.e. à l'adaptation statique) par la mise en place des mécanismes appropriés.

3. *Comment peut-on construire les évaluateurs adaptables souhaités ?*

   Etant donné l'objectif de fournir un support adaptable de l'évaluation des requêtes, nous ne cherchons pas à construire un ou quelques évaluateurs de requêtes pour une (classe d') application particulière mais à proposer une approche suffisamment générale pour un plus grand nombre d'applications (ou de classes d'applications). Après avoir étudié les mécanismes et les systèmes d'évaluation de requêtes selon les aspects décrits ci-dessus, nous pouvons constater qu'il est difficile, voire impossible, de construire un évaluateur de requêtes générique et adaptable. Cela est expliqué, d'une part, par la variété des besoins des applications et de l'environnement dans lequel les requêtes sont évaluées, et d'autre part, par la complexité des mécanismes d'évaluation de requêtes dans ces différentes conditions.

   Par conséquent, nous avons choisi de construire un *canevas (framework) d'évaluation de requêtes* qui couvre aussi bien les aspects d'évaluation de requêtes que les aspects architecturaux des évaluateurs de requêtes nécessaires pour garantir l'adaptabilité statique. **Un canevas logiciel signifie une conception réutilisable, définie par un ensemble d'interfaces et/ou de classes abstraites, et par la façon par laquelle ces interfaces interagissent** [Joh97]. Un canevas d'évaluation de requêtes permet de construire des évaluateurs de requêtes, appelés les *instances du canevas*, à la demande des utilisateurs (qui sont programmeurs).

## 1.4 Contributions

Notre approche pour la construction d'évaluateurs adaptables de requêtes s'appuie sur la proposition d'un canevas logiciel et considère les trois types d'adaptation, à savoir l'adaptation statique, la personnalisation et l'adaptation dynamique. Les principales contributions de cette thèse peuvent être résumées comme suit :

**Une analyse des aspects d'un évaluateur de requêtes**
Afin de considérer l'adaptation des évaluateurs de requêtes dans sa globalité, nous avons débuté ce travail par une analyse des différents aspects d'un évaluateur de requêtes. Cette analyse nous permet

d'identifier les possibilités de l'adaptation (i.e. moments et sujets de l'adaptation) ainsi que les mécanismes nécessaires pour les réaliser et enfin, de définir la signification du mot "adaptable" dans les évaluateurs de requêtes. Cela constitue une grille de "lecture" qui permet de comparer les nombreux travaux sur l'évaluation de requêtes et d'identifier les fonctionnalités d'un évaluateur à séparer.

**QBF, un canevas pour la construction d'évaluateurs adaptables de requêtes**

Partant de la constatation de la complexité de la conception et de la réalisation des évaluateurs de requêtes dans divers contextes, nous proposons QBF (*Query Broker Framework*), un canevas d'évaluation de requêtes. L'approche de QBF s'appuie sur la séparation et l'abstraction des fonctionnalités de base des évaluateurs de requêtes, dans le but de favoriser leur réutilisation et leur adaptation. Chacune de ces fonctionnalités est conçue comme un composant. Les composants de QBF recouvrent aussi bien les aspects d'optimisation classique que les aspects d'évaluation avancée tels que les mécanismes d'évaluation personnalisée et dynamique. Le canevas défini offre un nouveau cadre pour l'utilisation et l'intégration des mécanismes existants ainsi que pour leur comparaison.

Nous fournissons également une implémentation (partielle) de ces interfaces, pouvant être (ré-)utilisée par les programmeurs lors de la construction de leur évaluateur. Cela permet de réduire les efforts de conception et de réalisation des évaluateurs de requêtes. L'implémentation de QBF constitue donc le point de départ de la construction des évaluateurs basés sur QBF. Nous proposons également une méthodologie de construction d'un évaluateur de requête (appelé instanciation).

**Évaluation personnalisée de requêtes**

Il s'agit des mécanismes d'évaluation nécessaires pour réaliser le deuxième niveau d'adaptabilité qu'est la *personnalisation*. A la différence des travaux existants concernant l'évaluation personnalisée, nous cherchons à intégrer les traitements de la personnalisation dans chacune des phases du processus du traitement de requêtes afin de pouvoir mieux répondre à une requête personnalisée, tout en évitant les calculs inutiles.

Plus précisément, nous considérons une représentation des contraintes spécifiques de requête comme des éléments de son contexte, et revisitons le processus de traitement de requêtes, phase par phase, au regard de ce contexte. Nous proposons des mécanismes, à intégrer dans chacune des phases, dans le but de *sélectionner des sources pertinentes* pour répondre à une requête personnalisée, de *limiter l'espace de recherche* effectif de cette requête, de *calculer les coûts d'une manière flexible* selon le contexte de requête et d'*éliminer les solutions non-candidates* pendant l'optimisation de ces requêtes.

**Évaluation interactive de requêtes**

Il s'agit de mécanismes permettant l'échange d'informations et de contrôle entre le système d'évaluation de requête et son utilisateur. Concrètement, pendant l'exécution d'une requête (longue), le système re-

tourne à son utilisateur les résultats (incomplets, partiels) afin que ce dernier soit au courant du déroulement de l'évaluation de sa requête et intervienne, s'il le souhaite, pour contrôler, voire affiner, sa requête en cours d'exécution. Ces mécanismes contribuent à la réalisation de l'adaptabilité dynamique dans notre approche.

Pour cela, nous proposons de considérer un ensemble d'interactions, appelés *opérations d'interaction*, et présentons quelques mécanismes pour en tenir compte. Il s'agit de mécanismes pour la construction de résultats partiels et la transformation de requête suite à des interactions utilisateur.

**Validation et expérimentation**

Une implémentation de QBF, ainsi que de quelques instances (i.e. évaluateurs), montrent la faisabilité de notre approche. Nous étudions également les critères d'évaluation à considérer dans notre approche. Nous avons réalisé quelques mesures de coût de l'adaptation. L'objectif de ces mesures est double : (i) elles permettent d'évaluer les mécanismes implémentés ; et (ii) elles aident les programmeurs dans les choix des éléments (composants) appropriés pour leur évaluateur tout en considérant les surcoûts introduits par l'adaptation. Les expériences dans le développement des évaluateurs de requêtes basés sur QBF nous permettent d'évaluer la réutilisation de QBF ou plus précisément, le gain de l'utilisation de QBF dans la construction d'évaluateurs de requêtes.

## 1.5 Organisation du document

Ce document se compose de neuf chapitres (celui-ci inclus) comme le montre la figure 1.1. Il reprend, étape par étape, la démarche de nos travaux et aborde les trois niveaux d'adaptabilité de la solution proposée.

**Le chapitre 2** rappelle le principe du traitement de requêtes. Il se focalise sur les deux tâches les plus importantes et complexes que sont l'optimisation et l'exécution. L'extension de ces tâches dans le cas des systèmes multi-sources est également présentée [VM03].

**Le chapitre 3** analyse un nombre important de travaux autour du traitement de requêtes. Tout d'abord, ce chapitre introduit une définition générale de l'adaptation et présente trois niveaux de l'adaptabilité des évaluateurs de requêtes. En s'appuyant sur ces définitions, les travaux existants sont analysés comme étant les mécanismes pour réaliser l'adaptabilité. L'analyse met en évidence les *éléments* plus ou moins orthogonaux des mécanismes existants et la nécessité de les *séparer* et les *placer* dans un *cadre uniforme* [VM03, Vu04b].

**Le chapitre 4** introduit notre proposition nommée QBF (*Query Broker Framework*), un canevas pour la construction d'évaluateurs adaptables de requêtes, en commençant par une description globale de l'approche. Elle est suivie par une présentation détaillée de la spécification de QBF [VCB03, VC04b].

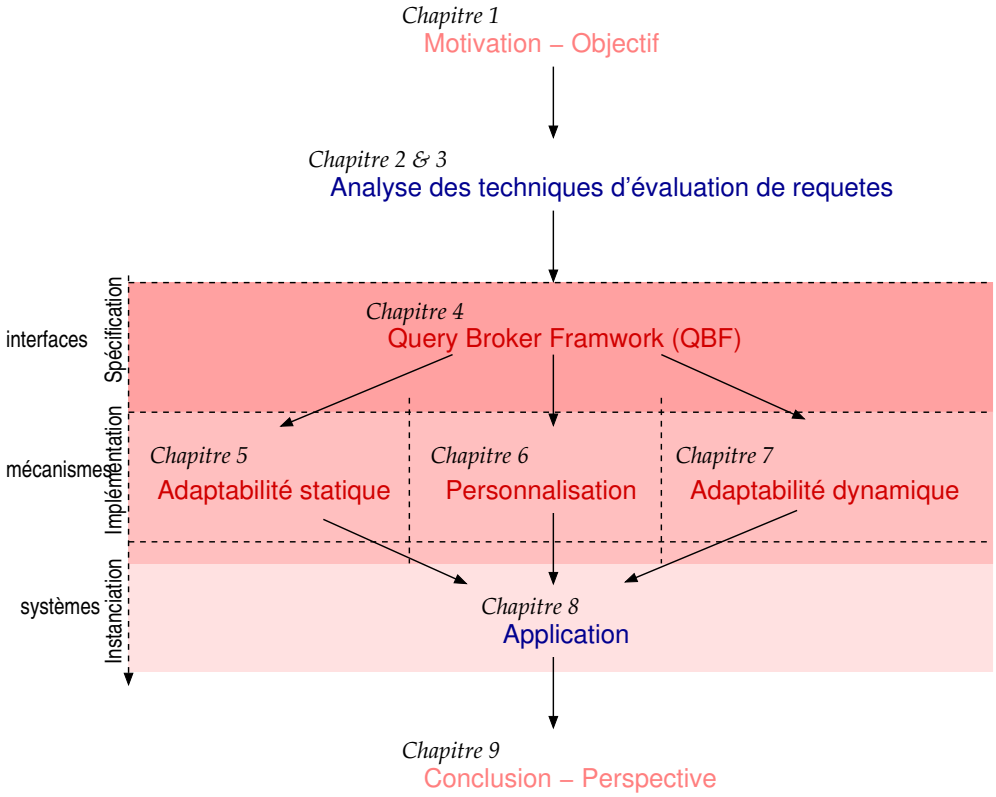



Figure 1.1: Organisation du document

**Le chapitre 5** évoque les points importants de la mise en œuvre de QBF et donne également des indications pour l'instanciation (ou utilisation) de QBF [VC03, VC04a]. Il concerne essentiellement l'*adaptabilité statique* obtenue lors de la construction des évaluateurs, sans oublier le choix des mécanismes pour l'adaptation ultérieure (i.e. la personnalisation et l'adaptation dynamique), si cette dernière est souhaitée.

**Le chapitre 6** présente une approche systématique pour l'*évaluation personnalisée* de requêtes. Le chapitre propose une classification des paramètres de personnalisation, puis présente les mécanismes à intégrer dans le processus de traitement de requêtes. Une mise en œuvre de ces mécanismes basée sur QBF est brièvement présentée. Ces mécanismes concernent essentiellement la réalisation du deuxième niveau de l'adaptabilité (i.e. la *personnalisation*).

**Le chapitre 7** présente, d'une façon similaire au chapitre précédent, les aspects liés à l'*adaptabilité dynamique*. Il concerne essentiellement les mécanismes d'évaluation interactive de requêtes en supposant la réutilisation des mécanismes d'évaluation adaptative. Les mécanismes présentés ont pour but de fournir d'une manière incrémentale

les résultats partiels, qui sont des retours (*feedbacks*) potentiels de l'évaluateur de requêtes, et de transformer la requête selon les interactions utilisateur pendant l'évaluation [Vu04a].

**Le chapitre 8** illustre l'instanciation de QBF et les mécanismes proposés à travers nos expériences de construction d'évaluateurs de requêtes [MED03, CBB+04]. Ce chapitre donne également la méthodologie d'utilisation de QBF.

**Le chapitre 9** conclut le document en résumant les travaux réalisés et en donnant quelques perspectives de cette thèse.

## 1.6 Convention typographique

Nous utilisons des termes anglais dans la plupart des figures, car ils se rapportent souvent à nos spécifications, définitions et implémentations.

Nous utilisons les conventions typographiques suivantes :

- Les noms de composants et de classes (en anglais) sont en police SansSerif.
- Les noms de méthodes sont en police `truetype`.
- Les noms d'attributs sont en police *`truetype penché`*.
- Les valeurs sont en police *Time penché*.
- Le texte en *italique* désigne soit les termes en anglais (donnés pour faciliter la compréhension de la traduction en français), soit les remarques, les constats, les idées que nous voulons souligner.
- La définition des termes est donnée dans le texte en **gras**.

# CHAPITRE 2

# Traitement de requêtes : principe

Le traitement de requêtes est un des éléments majeurs à considérer dans l'implémentations des systèmes de gestion de données [GMUW00]. Ce sujet a fait l'objet de nombreux travaux depuis plusieurs années. Quels que soient le domaine d'application et le contexte de déploiement, le traitement de requêtes (base de données) exploite davantage l'aspect ensembliste des requêtes en utilisant les optimiseurs et les moteurs d'exécution traitant de très gros volumes de données. Ce chapitre a pour objectif de rappeler les principes du traitement de requêtes. Nous commençons par le processus de traitement (la section 2.1) suivi une analyse de deux fonctions principales que sont l'optimisation (la section 2.2) et l'exécution (la section 2.3). Ensuite, nous présentons l'extension de ces fonctions dans le cas de systèmes multi sources (la section 2.4). Enfin, nous concluons ce chapitre dans la section 2.5.

## 2.1 Processus de traitement de requêtes

La figure 2.1 présente le processus classique de traitement de requêtes dans les SGBD [GMUW00]. Il comprend les phases d'*analyse*, d'*optimisation*, de *génération de code*, et d'*exécution*. En général, le processus s'exécute en deux étapes : une étape de **compilation** qui recouvre souvent les phases de l'analyse jusqu'à la génération de code, et une étape d'**exécution**. Les fonctions principales du traitement de requêtes sont :

- l'**analyse** : Une requête exprimée dans un langage déclaratif tel que SQL[DD93], OQL[BCD89] ou XQuery[XQu], est analysée syntaxiquement et sémantiquement. Le résultat de cette phase est une *représentation interne* de la requête, souvent, sous forme d'un *arbre algébrique* dont les nœuds sont les opérateurs comme la sélection, la projection, la jointure, etc., et les arcs représentent la dépendance de données entre les nœuds.

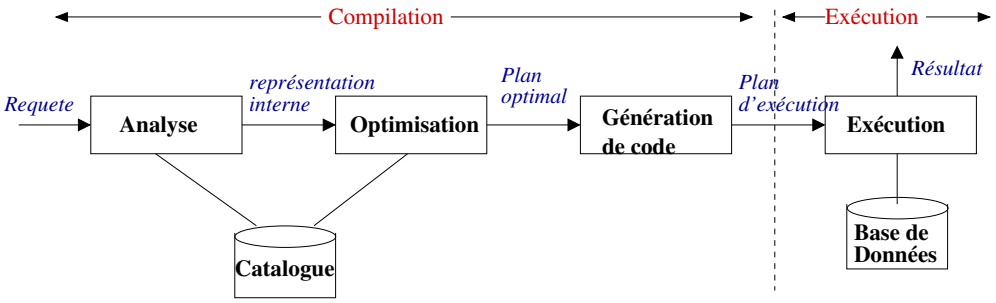



Figure 2.1: Processus de traitement de requêtes

- l'**optimisation** : Étant donné une requête, il existe plusieurs expressions de calcul équivalentes. Ces expressions sont examinées afin de choisir la "meilleure", appelée *plan optimal*[1]. Ce dernier sera évalué par la suite afin de répondre à la requête d'origine.

- la **génération de code** : L'objectif de cette phase est de générer les appels des méthodes, les codes pour pouvoir évaluer le plan optimal. Le résultat de cette phase est le code exécutable, appelé *plan d'exécution*.

- l'**exécution** : Les codes générés par la phase précédente sont exécutés pour produire les résultats de la requête d'origine.

La phase d'analyse est considérée comme l'interface entre le système et l'utilisateur (ou l'application cliente[2]). La fonction d'analyse dépend du langage déclaratif utilisé. En revanche, la phase de génération de code dépend de l'implémentation du système, des opérateurs, des fonctions, etc. Quant à l'optimisation et à l'exécution, elles exploitent l'aspect ensembliste des données en utilisant les techniques plus ou moins complexes pour garantir l'efficacité du traitement des requêtes. Ces deux dernières sont les fonctions les plus importantes dans les évaluateurs de requêtes sur lesquelles la suite de ce chapitre se focalise.

## 2.2 Optimisation

L'objectif de l'optimisation est de trouver le plan optimal des requêtes en entrée (ou requêtes utilisateurs[3]). Il s'agit de réécrire la requête, si nécessaire, de choisir l'ordre d'exécution des opérateurs de la requête et les algorithmes implémentant ces opérateurs. Le choix du plan optimal s'appuie souvent sur une estimation de coût en exploitant les connaissances sur les données interrogées et sur l'environnement

[1]Nous utilisons le terme "plan optimal" pour désigner une (meilleure) solution retenue sans forcement être *la solution la plus optimisée*.

[2]Dorénavant, ces deux termes sont interchangeables. Ils signifient "l'expéditeur" des requêtes quel qu'il soit l'utilisateur humain ou un *autre* système (e.g. application).

[3]Ces deux termes, interchangeables, font allusion à la requête d'origine soumise au système par l'utilisateur.

d'exécution. La figure 2.2 montre la vue globale du processus d'optimisation qui se compose de deux étapes : une pour la *réécriture* et l'autre pour la *planification* [Ioa00].

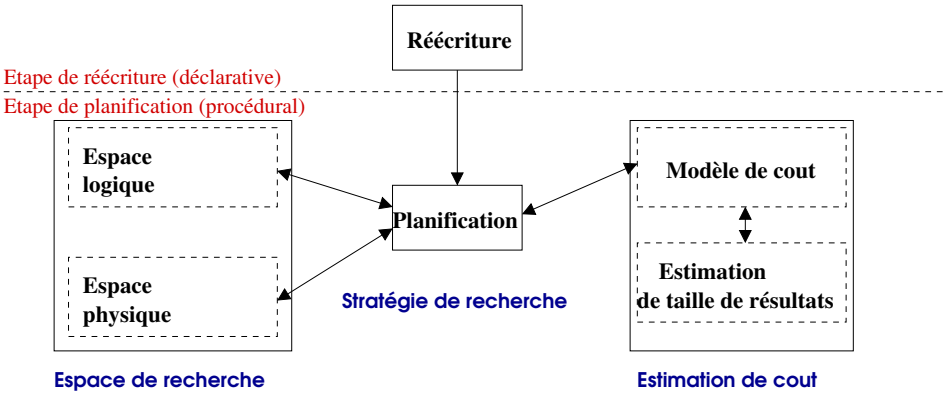



Figure 2.2: Architecture de l'optimiseur

L'étape de **réécriture** vise à transformer la requête en entrée en une ou plusieurs requête(s) équivalente(s) pour une évaluation plus efficace. Elle s'appuie sur les déclarations statiques (e.g. la définition des vues) et sur les caractéristiques de la requête (e.g. l'imbrication des (sous-)requêtes), et ne considère pas les caractéristiques spécifiques de l'implémentation du SGBD. Cette étape est donc *déclarative*. En revanche, la **planification** est une étape *procédurale*. Elle est susceptible d'examiner les plans équivalents et de choisir le "meilleur" plan qui sera exécuté par la suite. La planification peut être décomposée en trois éléments principaux qui sont : (i) l'*espace de recherche*, (ii) l'*estimation du coût* et (iii) la *stratégie de recherche*.

La suite de cette section détaille les trois éléments de la planification en laissant la fonction de réécriture qui sera abordée dans la section 2.4.2 pour le cas de systèmes multi sources.

### 2.2.1 Espace de recherche

Etant donné une requête, il peut exister plusieurs plans équivalents, chacun représentant une façon d'évaluer la requête que le système peut considérer. Ces plans constituent l'**espace de recherche** de la requête composé de l'*espace logique* et de l'*espace physique*. La génération de cet espace, étant une des tâches de l'optimiseur, se base sur les propriétés de l'algèbre (e.g. la commutativité, l'associativité) pour l'espace logique et utilise les propriétés physiques de l'implémentation (e.g. les index, les algorithmes) pour l'espace physique. Elle consiste donc à appliquer successivement différentes *opérations* sur le(s) plan(s) initial et intermédiaires.

### 2.2.2 Estimation de coût

Le choix du meilleur plan d'une requête se base souvent sur l'estimation de coût des plans dans son espace de recherche. Le coût peut s'exprimer en terme de temps d'exécution (du lancement de l'exécution de la requête jusqu'à l'obtention de son résultat), de consommation des ressources (e.g. communication, mémoire, CPU, etc.) ou encore le coût économique. La mesure du coût reflète effectivement l'objectif d'optimisation dont les plus courants sont de minimiser, soit la consommation de ressources, soit le temps de réponse.

D'une manière générale, l'estimation de coût s'appuie sur un *modèle de coût* dont les principaux facteurs à prendre en considération sont (i) le coût de communication, et (ii) le coût du traitement de données (à distance et locale). Ces coûts dépendent aussi de la taille et de la cardinalité des données traitées (i.e. données en entrée et intermédiaires) qui sont estimées par les formules mathématiques en se basant sur les informations nécessaires[4] stockées dans le catalogue (i.e. méta-données) du système [GMUW00].

### 2.2.3 Stratégie de recherche

La stratégie de recherche spécifie la manière dont l'optimiseur examine l'espace de recherche. Plusieurs stratégies ont été proposées sous forme d'algorithmes d'optimisation, dont la plupart regroupent aussi les aspects liés à la génération de l'espace de recherche et à l'estimation de coût. A titre d'exemples, nous pouvons citer les algorithmes basés sur la programmation dynamique [SAC$^+$79, IK90], les algorithmes basés sur le parcours aléatoire [NSS86, IW87, SG88, Swa89], et ceux basés sur les règles [HFLP89, PHH92, GD87, GM93].

L'objectif de ces algorithmes est de rechercher le plan optimal de la requête utilisateur en réduisant le coût de la phase d'optimisation[5] dans le but d'améliorer la performance globale de l'évaluation de requête. Ces algorithmes se différencient par leur complexité (polynomial vs. combinatorial) et/ou par la nature du parcours de l'espace de recherche (déterministe vs. aléatoire, heuristique vs. systématique, constructive vs. transformative). Du point de vue du fonctionnement, nous nous intéressons à la "forme" du parcours de la stratégie de recherche. Ainsi donc, ces algorithmes peuvent être regroupés en deux grandes approches, une basée sur la construction de plan (*bottom-up*) et l'autre appuyée sur la transformation de plan (*top-down*) comme l'illustre la figure 2.3. Nous présentons ci-après le principe de ces deux approches en introduisant quelques heuristiques pouvant être utilisées pour améliorer la recherche du plan optimal.

**Approche par construction (*bottom-up*)** Suivant cette approche, l'énumération de plans d'une requête débute par les plans les plus simples (i.e. les plans d'accès ou les méthodes d'accès de chacune des entrées de la requête). A partir de ces premiers plans, l'algorithme construit les plans de plus en plus complets en combinant ceux construits précédemment (cf. la figure 2.3(a)).

[4] e.g. la taille et la cardinalité des données en entrée, la sélectivité, etc.

[5] Ce coût peut être mesuré par le temps de l'optimisation et/ou par les ressources nécessaires pour l'optimisation, etc.

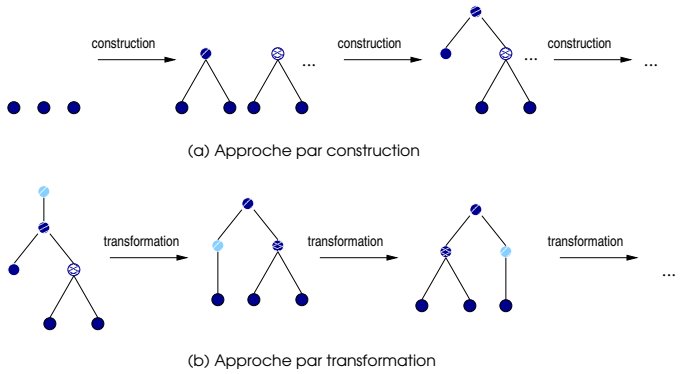



Figure 2.3: Illustration des stratégies de recherche

L'algorithme le plus représentatif de cette approche est proposé par P. Selinger et al. [SAC+79] dans le contexte du *System R* [ABC+76]. Le principe de cet algorithme est de *réduire dynamiquement l'espace de recherche* par l'itération sur le nombre des opérateurs de jointure du plan de requête. Par conséquent, l'optimisation s'effectue localement et elle ne peut pas garantir que le plan optimal retenu à la fin du processus soit le plan le plus optimisé. Cependant, elle permet de réduire le coût de l'optimisation en limitant le nombre des plans générés à chaque étape. L'algorithme s'exécute en trois phases comme suit :

1. examiner les plans d'accès de chacune des entrées de la requête : tous les plans d'accès possibles sont énumérés et seul celui le *moins coûteux* de chaque entrée est conservé pour l'étape suivante.

2. (*Itération*) énumérer les plans en considérant toutes les jointures possibles entre deux entrées, puis les jointures de trois entrées, etc., jusqu'à l'obtention des jointures de $n$ entrées de la requête à optimiser. A chaque étape, seulement les plans les *moins coûteux* sont conservés.

3 compléter les plans obtenus de la phase précédente avec les opérateurs de sélection, de projection, etc.

**Approche par transformation** A la différence de l'approche précédente, l'optimisation par transformation consiste à appliquer successivement les transformations possibles sur les plans complets de la requête afin d'énumérer tous les plans équivalents avant d'en choisir le meilleur pour son exécution. La figure 2.3(b) illustre le principe de cette stratégie.

Les algorithmes de cette approche ont été proposés initialement dans le contexte de SGBD extensibles [GD87, GM93, PGLK97]. Dans cette approche, les heuristiques consistent en l'*ordonnancement* des transformations à appliquer. Quelques heuristiques les plus utilisées sont "sélection d'abord", "évitement les produits cartésiens", "diminution des constituants" (i.e. réaliser les projections aussi tôt que possible), etc. Ces heuristiques visent à réduire la taille et la cardinalité des résultats intermédiaires.

D'autres algorithmes utilisent les heuristiques pour explorer l'espace de recherche par un parcours aléatoire (*random walks*). On trouve différentes variantes de cet algorithme tels que *Simulated Annealing* (SA)[IW87], *Iterative Improvement* (II)[SG88, Swa89], *Two-Phase Optimization*(2PO)[IK90, IK91].

**Remarques globales sur l'optimisation** Nous trouvons que les algorithmes d'optimisation recouvrent souvent les trois éléments de la planification, à savoir l'espace de recherche, l'estimation de coût et la stratégie de recherche. Cela fait qu'un changement dans l'un de ces trois éléments nécessite des modifications dans l'ensemble de l'algorithme. De ce fait, il est difficile de rendre l'optimisation adaptable en la "paramétrant" par exemple et/ou de l'étendre.

## 2.3 Exécution

La plupart des moteurs d'exécution de requêtes adoptent le modèle itérateur [Gra93]. Suivant ce modèle, les données sont considérées comme une collection et traitées par l'itération. D'une manière générale, les opérateurs sont implémentés comme des itérateurs ayant la même interface. Cette interface fournit principalement les opérations : (i) *open* qui prépare les ressources pour produire les données, (ii) *next* qui produit un élément de la collection, et (iii) *close* qui libère les ressources d'exécution allouées.

Remarquons que la composition d'un plan de requête à partir des opérateurs ayant la même interface itérateur est simple. L'exécution est commencée par l'appel des méthodes du nœud racine du plan. Cet appel est propagé dans l'arbre de requête jusqu'aux feuilles. Les opérateurs produisent un élément à chaque appel de la méthode `next`. Il n'est donc pas nécessaire de stocker les données temporairement pour attendre leur traitement. Par ailleurs, les opérateurs d'un plan peuvent être ordonnés et exécutés par un *seul processus* sans l'assistance du système d'exploitation sous-jacent ou son interaction.

Le principe du traitement de requêtes que nous venons de présenter peut aussi être appliqué à des contextes autres que les SGBD, notamment les systèmes multi sources. Toutefois, il est nécessaire d'y apporter quelques extensions que nous détaillons dans la suite de ce chapitre.

## 2.4 Le cas particulier de systèmes multi sources

De manière générale, un système multi sources (SMS) est un logiciel permettant un accès "homogène" à plusieurs sources de données préexistantes souvent *distribuées*, *autonomes* et potentiellement *hétérogènes*. Par un accès homogène, nous entendons la possibilité d'utiliser un *seul* langage de requête pour l'interrogation de l'ensemble des sources quelque soit langage utilisé par chacune des sources. Dans cette optique, la plupart des SMS choisissent un *modèle de données commun*, appelé également *modèle pivot* [DD99]. L'interrogation se fait alors par l'intermédiaire du langage de requête associé au modèle pivot et les résultats sont présentés dans ce modèle.

La figure 2.4 présente l'architecture générale de système de médiation. Le site de coordination, appelé **médiateur**, offre une *vue intégrée* (définie comme schéma global) des systèmes sous-jacents

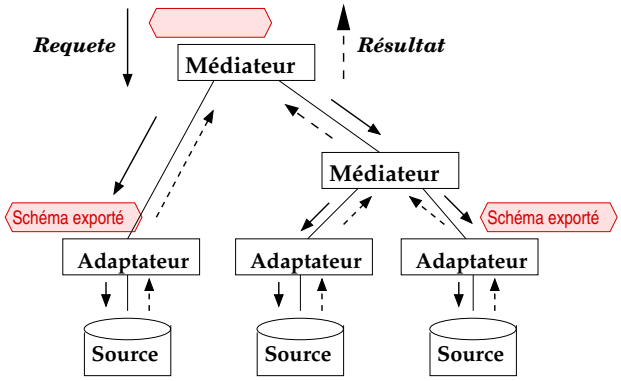



Figure 2.4: Architecture de système médiation

qui, quant à eux, sont d'autres médiateurs ou des **adaptateurs** des sources offrant une vue exportée (le *schéma exporté*). L'utilisateur soumet une requête exprimée sur le schéma global, appelée **requête globale**, à un médiateur susceptible de communiquer les systèmes sous-jacents pour accéder aux données et de calculer les résultats de la requête. Le médiateur *réécrit* la requête globale en terme de schémas exportés des sources, appelée **requête locale** (la *réécriture*). Ensuite, le système décompose la requête réécrite en sous-requêtes évaluables par les systèmes sous-jacents (la *décomposition*). Les résultats de ces sous-requêtes sont retournés au médiateur qui les combine pour construire le résultat final.

L'étape de **réécriture** de requête vise essentiellement à résoudre le problème de l'hétérogénéité structurelle et sémantique des données provenant des sources multiples. La complexité de la réécriture dépend de la méthode d'intégration. En revanche, la **décomposition** tient compte de l'hétérogénéité de capacité de calcul des systèmes sous-jacents. Son objectif est d'identifier les sous-requêtes pouvant être résolues par les sources afin de pourvoir y "déléguer" certains traitements dans le but d'améliorer la performance globale de l'évaluation. Le médiateur est responsable du traitement des tâches non-résolues par les systèmes sous-jacents. Ce traitement est similaire à ceux que nous avons présentés dans les deux sections précédentes. Ainsi, seules les fonctions de réécriture et de décomposition de requête sont considérées dans la suite de cette section.

### 2.4.1 Réécriture

Dans les systèmes médiation, la réécriture des requêtes globales en requêtes locales dépend des méthodes d'intégration de sources dont les deux les plus utilisés sont *Global-as-View* (GaV) et *Local-as-View*

(LaV). Cette section décrit la technique de réécriture dans ces deux cas.

#### 2.4.1.1 Approche *Global-as-View*

Selon l'approche *Global-as-View*, le schéma global est défini comme une vue (i.e. une requête) sur les schémas exportés des sources [GMPQ$^+$97, Gar]. Le processus de réécriture est donc assez simple et direct. Il est connu en terme de processus de dépliage (*unfolding*) qui s'exécute en deux phases comme suit :

1. *remplacer les vues dans la requête par leur définition.*

   Il s'agit de remplacer les termes du schéma global apparaissant dans la requête par leur définition. Le résultat est donc une requête exprimée en terme de schémas exportés des sources.

2. *nettoyer la requête locale.*

   Cette phase a pour but d'éliminer toutes les redondances dans la requête locale obtenue de la phase précédente.

**Exemple** Pour illustrer l'utilisation de cet algorithme, nous considérons un exemple d'intégration des sources `V1`, `V2` et `V3` comme le montre la figure 2.5.

**Schémas exportés des sources**

```
V1(title, director, year)
V2(title, director, year)
V3(title, review)
```

**Schéma global**

```
MovieYear(title, year) :- V1(title,_,year)                    (R1)
MovieYear(title,year) :- V2(title,_,year)                     (R2)
MovieReview(title, director, review) :- V1(title,director,_)  (R3)
                                        & V3(title, review)
MovieReview(title, director, review) :- V2(title,director,_)  (R4)
                                        & V3(title, review)
```

Figure 2.5: Exemple de schémas (locaux et global) selon l'approche *Global-as-View*

Ces sources contiennent des informations concernant les films (`title`, `director`, `year`) et les critiques des films (`review`). Le schéma global (`MovieYear`,`MovieReview`) est défini par les règles en DataLog (de `R1` à `R4`) comme montré dans la partie droite de la figure 2.5. Les paramètres manquants sont représentés par "_" (souligné) dans les règles. Par exemple, `MovieYear` est défini à partir de `V1` (règle `R1`) et `V2` (règle `R2`), `MovieReview` est défini à partir d'une jointure de `V1` et `V3` (règle `R3`), et d'une jointure de `V2` et `V3` (règle `R4`). Notons qu'en Datalog, l'union est représentée comme deux

règles séparées ayant la même entête. Par exemple, les deux premières règles définissent `MovieYear` et les deux dernières définissent `MovieReview`.

**Requête utilisateur**

```
Q(title,review):- MovieYear(title,1997) &
       MovieReview(title,director,review)
```

**Requêtes réécrites**

```
Q'(title,review):- V1(title,_,1997)        (q1)
                 & V3(title,review)
Q'(title,review):- V2(title,_,1997)        (q2)
                 & V3(title,review)
```

Figure 2.6: Exemple de la réécriture de requête dans l'approche *Global-as-View*

Nous considérons l'exemple de la requête suivante : "Chercher les critiques des films en 1997". Cette requête `Q` est formulée en Datalog sur le schéma global comme montré dans la partie gauche de la figure 2.6. Par application de la première phase du processus de réécriture, nous remplaçons `MovieYear` et `MovieReview` par leurs définitions. Le résultat de cette phase contient les quatre règles (= 2 x 2) qui sont les combinaisons de `R1`, `R2` et `R3`, `R4`. Après avoir éliminé toutes les redondances (la phase de nettoyage), le résultat final de la réécriture est montré dans la partie droite de la figure 2.6 sous la forme des règles `q1` et `q2`. Elles correspondent en fait à une requête conjonctive dénotée `Q'` composée de deux prédicats (sous-requêtes).

#### 2.4.1.2 Approche *Local-as-View*

Selon l'approche *Local-as-View*, les schémas exportés des sources sont définis comme des vues sur le schéma global antérieurement défini. Il n'existe donc pas de correspondances directes entre les termes globaux et les termes locaux. La réécriture est reconnue comme étant difficile [Lev01]. En effet, à partir de telles correspondances, il n'est pas toujours possible de trouver la requête équivalente exprimée en terme de schémas des sources. Les solutions existantes reviennent donc à trouver la requête locale la plus "proche" de la requête donnée, c'est-à-dire la requête dont le résultat est *presque* équivalent au résultat de la requête d'origine. D'une manière générale, l'algorithme de réécriture dans l'approche *LaV* se déroule en trois phases comme suit :

1. *réécrire les règles de correspondance en règles inverses.*

   L'objectif est de trouver les *règles inverses* à partir des règles de correspondance données. Les règles inverses d'une règle sont de même format que celle-ci mais écrites pour chacun des prédicats de la clause en partie droite. Cela permet de calculer *approximativement* la définition des termes du schéma global en termes des schémas locaux afin de rendre possible la réécriture.

2. *réécrire la requête en utilisant les règles inverses*.

   Cette phase est similaire à celle de la réécriture dans l'approche *GaV*.

3. *nettoyer la requête obtenue*.

   Similaire à la réécriture dans l'approche *GaV*, cette phase a pour but d'éliminer des redondances. Elle est pourtant plus compliquée car les règles inverses résultant de la phase (1) ne sont pas toujours précises et complètes. Par conséquent, la requête locale résultant de la phase (2) contient non seulement les prédicats redondants mais aussi ceux invalides, i.e. ceux qui ne peuvent pas être évalués : il faut donc les détecter.

**Exemple** Nous considérons le même scénario que pour *GaV*. La figure 2.7 présente le schéma global et les schémas exportés des sources définis selon l'approche *LaV*. Le schéma global est défini par `MovieYear` et `MovieReview` dans la partie gauche de la figure et les schémas exportés des sources, dénotés `V1`, `V2`, `V3` sont définis comme des vues sur le schéma global (dans la partie droite de la figure). Par exemple, la vue `V1` (de la source 1) est définie comme une requête sur `MovieYear` et `MovieReview`.

**Schéma global**

```
MovieYear(title, year)
MovieReview(title, director, review)
```

**Schémas exportés des sources**

```
V1(title, director, year) :- MovieYear(title, year) &
                     MovieReview(title, director, _)
V2(title, director, year) :- MovieYear(title, year) &
                     MovieReview(title, director, _)
V3(title, review) :- MovieReview(title, _, review)
```

Figure 2.7: Exemple de l'approche *Local-as-View*

La figure 2.8 montre le processus de la réécriture phase par phase. Le résultat de la première phase contient les règles inverses (dénotées `RIx.x`) obtenues à partir des règles données (dénotées `Rx`). Dans les règles inverses, des variables non-importantes sont remplacées par "_" et les variables absentes sont remplacées par "*?*". Notons que les variables non-importantes se trouvent toujours dans le prédicat en partie droite et les absents sont toujours dans les prédicats en partie gauches. Pour revenir à notre exemple dans la figure 2.8, à partir de la règle `R1`, on obtient deux règles inverses dénotées `RI1.1` et `RI1.2`. Dans `RI1.1`, l'information sur `director` (partie droite de la règle) n'est pas nécessaire pour calculer `MovieYear(title,year)`. Par contre, l'information sur `review` est absente dans la règle `RI1.2`, i.e. cette information n'est pas calculable à partir de `V1`. Étant donné les trois règles de

**Phase 1 : Règles inverses**

```
V1(title, year, director) :- MovieYear(title, year) &
                             MovieReview(title, _, director)   (R1)
-->
MovieYear(title, year) :- V1(title, year, _)                   (RI1.1)
MovieReview(title, director,?) :-V1(title, director, _)        (RI1.2)

V2(title, director, year) :- MovieYear(title, year) &
                             MovieReview(title, director, _)   (R2)
-->
MovieYear(title, year) :- V2(title, _ , year)                  (RI2.1)
MovieReview(title, director, ?) :- V2(title, director, _)      (RI2.2)

V3(title, review) :- MovieReview(title, _, review)             (R3)
-->
MovieReview(title, ?, review):-V3(title, review)               (RI3.1)
```

**Phase 2 : Dépliage (*Unfolding*)**

```
Q'(title, ?):- V1(title, 1997, _) &  V1(title, _, director)    (q1-RI1.1&RI1.2)
Q'(title,?):- V1(title,1997,_) & V2(title,director,_)          (q2-RI1.1&RI2.2)
Q'(title, review):- V1(title, 1997, _) & V3(title,review)      (q3-RI1.1&RI3.1)
Q'(title,?):- V2(title,_,1997) & V1(title,1997,_)              (q4-RI2.1&RI1.2)
Q'(title, ?):- V2(title,_,1997) & V2(title,director,_)         (q5-RI2.1&RI2.2)
Q'(title, review):- V2(title,_,1997) & V3(title,review)        (q6-RI2.1&RI3.1)
```

**Phase 3 : Vérifier les requêtes obtenues après la phase 2**

Eliminer `q1`, `q2`, `q4`, `q5`.

Figure 2.8: Exemple de la réécriture de l'approche *Local-as-View*

correspondance `R1`, `R2`, `R3`, nous obtenons les cinq règles inverses `RI1.1`, `RI1.2`, `RI2.1`, `RI2.2`, `RI3.1` ( cf. figure 2.7).

La deuxième phase applique simplement le processus de dépliage sur l'ensemble des règles inverses. Avec cinq règles inverses dont deux définissant `MovieYear` et les trois autres `MovieReview`, nous obtenons six requêtes (=2x3) de `q1` à `q6` comme montré dans la figure 2.8. Parmi ces requêtes, il existe des requêtes invalides car les prédicats de la partie droite ne permettent pas de calculer le prédicat de la partie gauche. Revenons à notre exemple : `q1`, `q2`, `q4`, `q5` sont invalides car on ne peut pas obtenir l'information sur `review` par ces règles.

La troisième phase a donc pour but d'examiner toutes les requêtes générées par la phase précédente et d'éliminer toutes les requêtes redondantes ou non-valides.

L'algorithme de réécriture pour l'approche *LaV* a été présenté initialement dans [LRO96, DG97]. Une variante de cet algorithme, appelé *Minicon*, est proposée par R. Pottinger et al. dans [PL00]. Cet algorithme vise à limiter le nombre de requêtes générées par la phase 2. Précisément, une fois que

l'algorithme trouve les prédicats similaires dans la requête donnée et celles de vues, il regarde directement les variables de la requête afin de vérifier si une vue est utilisable, c'est-à-dire si une vue peut être utilisée pour répondre à la requête donnée. Nous reprenons notre exemple de requête `Q`. Après avoir trouvé les règles inverses, on examine les variables de ces règles. Les règles `RI1.2` et `RI2.2` sont éliminées car elles ne permettent pas de calculer `review` de requête `Q`. Par conséquent, le nombre de requêtes générées par l'algorithme est deux (=2x1) au lieu de six (=2x3) générées par l'algorithme classique (cf. la figure 2.8). Ceci montre que le nombre de sous-requêtes générées par l'algorithme est réduit de manière significative.

### 2.4.2 Décomposition

Rappelons que l'objectif majeur de la décomposition est de tenir compte de l'hétérogénéité de capacité de calcul des sources. Ces capacités peuvent être classifiées en deux catégories : la limitation d'accès et la limitation de traitement. La **limitation d'accès** signifie la façon par laquelle (et *seulement* par laquelle) l'accès aux données de la source est possible alors que la **limitation de traitement** signifie la capacité de traiter des données fournies par la source. Pour une décomposition efficace, il faut que les SMS disposent les connaissances des capacités de sources sous-jacentes. De telles connaissances peuvent être communiquées aux SMS à différents moment, soit à l'enregistrement des sources au SMS, soit pendant l'évaluation d'une requête. La suite de cette section présente brièvement les techniques de décomposition de requête dans ces différents cas de figure.

#### 2.4.2.1 Restriction d'accès

La restriction d'accès aux sources est souvent modélisée sous forme de règles spécifiant les attributs "liés" (*bound* et dénoté *b*) et les attributs "libres" (*free* et dénoté *f*). Cela signifie qu'il faut fournir les valeurs des attributs *liés* pour pouvoir accéder aux attributs *libres*.

La figure 2.9 présente l'exemple de trois relations `Experiment`, `Circulation`, `Location` et leurs règles de restriction d'accès. Par exemple, la règle (1) spécifie que l'accès aux attributs `key` et `depth` de la relation `Experiment` n'est que possible si la valeur de l'attribut `date` est connue à l'avance.

| | | |
|---|---|---|
| `Experiment(key,date,depth)` | `Experiment`(**`key`**$^f$,**`date`**$^b$,**`depth`**$^f$) | (1) |
| `Circulation(key,circulation)` | `Circulation`(**`key`**$^b$,**`circulation`**$^f$) | (2) |
| `Location(key,location)` | `Location`(**`key`**$^b$,**`location`**$^f$) | (3) |

Figure 2.9: Exemple de règles de restriction d'accès

Dans ce cas, l'objectif de la décomposition est de trouver les **plans valides**. Un plan est dit valide si et seulement si l'exécution de chacun des opérateurs du plan est possible pour les règles de restriction

```
Select [publications { bind Author (=)              (1)
                       bind Keyword (=)
                     }]
Project[publications { bind combine Author ()       (2)
                       bind combine Title ()
                     }]
```

Figure 2.10: Exemple de la capacité de traitement de requêtes

données. La plupart des algorithmes de décomposition proposés (e.g. [GMPQ+97, FLMS99]) consistent à étendre les algorithmes d'optimisation de l'approche par construction (cf. la section 2.2.3). Les extensions concernent les deux points suivants : (i) les plans sont "décorés". Un **plan décoré** est un plan dont les nœuds sont annotés par la règle de restriction utilisée; (ii) avant d'estimer le coût des plans pour choisir le plan le moins coûteux, les plans sont vérifiés et ceux invalides sont éliminés. Puisque les nœuds de plan sont annotés par la règle de restriction, la vérification de la validité est faite simplement par l'examen de règle associée au nœud racine. Si les variables liées au nœud racine sont fournies dans la requête, le plan correspondant est valide. Pour plus détail sur ces algorithmes, les lecteurs sont invités à consulter [GMPQ+97, FLMS99, Man01].

#### 2.4.2.2 Capacité de traitement

Les capacités de traitement de la source sont souvent décrites par les opérateurs logiques que la source (ou l'adaptateur) peut assurer. Pour chaque opérateur, la description contient la liste des paramètres acceptés (e.g. des attributs, des prédicats, etc.) [CHS+95, NKT+97].

La figure 2.10 montre un exemple de présentation de capacité de source `publications` : cette source peut exécuter l'opérateur *Select* sur l'ensemble `publications` avec la condition de sélection d'égalité sur les attributs `Author` et `Keyword` (cf. la figure 2.10(1)), et peut aussi exécuter l'opérateur *Project* sur les attributs `Author` et `Title` de l'ensemble `publications` (cf. la figure 2.10(2)).

Dans ce cas, l'objectif de la décomposition est d'identifier les sous-plans pouvant être résolus par les sources (ou adaptateurs). Elle nécessite un parcours de l'arbre de requête de manière ascendante. Le traitement démarre par les feuilles et lorsqu'un opérateur logique *op* ne fait pas partie des opérateurs qu'accepte la source, le sous-arbre est divisé en deux arbres ayant *op* comme point de séparation. L'arbre immédiatement au-dessous de *op* est préparé en vue d'être envoyé à la source en question pour son évaluation et l'arbre immédiatement au-dessus de *op* sera évalué par le système lui-même (i.e. SMS). Ce processus revient donc à annoter les nœuds du plan de requête selon leur localisation d'exécution.

#### 2.4.2.3 Négociation

Dans les approches présentées ci-dessus, il est supposé que la capacité des sources est connue à l'avance, permettant aux SMS de décider les tâches que doivent assumer les sources sans avoir besoin de contacter ces dernières. A la différence de ces approches, M. Carey et al. supposent qu'il est impossible de connaître, a priori, les capacités de traitement de requêtes des sources [CHS+95]. Lors du traitement d'une requête, le SMS "négocie" avec les sources (ou plus précisément les adaptateurs) le plan qu'elles peuvent traiter.

La décomposition se base sur l'approche d'optimisation par construction (cf. la section 2.2.3) en ajoutant une phase de "négociation". Concrètement, à chaque étape, le médiateur envoie à chaque adaptateur une description du travail le concernant. L'adaptateur retourne un ou plusieurs plans d'exécution qu'il peut traiter alors que ce(s) plan(s) ne couvre(nt) pas forcément tout le travail proposé à l'adaptateur. Ce plan contient une liste de paramètres indiquant ce que la source peut faire et leur coût. En fonction des propriétés des plans qui lui sont retournés, le médiateur compense les capacités de certaines sources en assumant le travail refusé par ces sources. Ainsi, il choisit la meilleure solution.

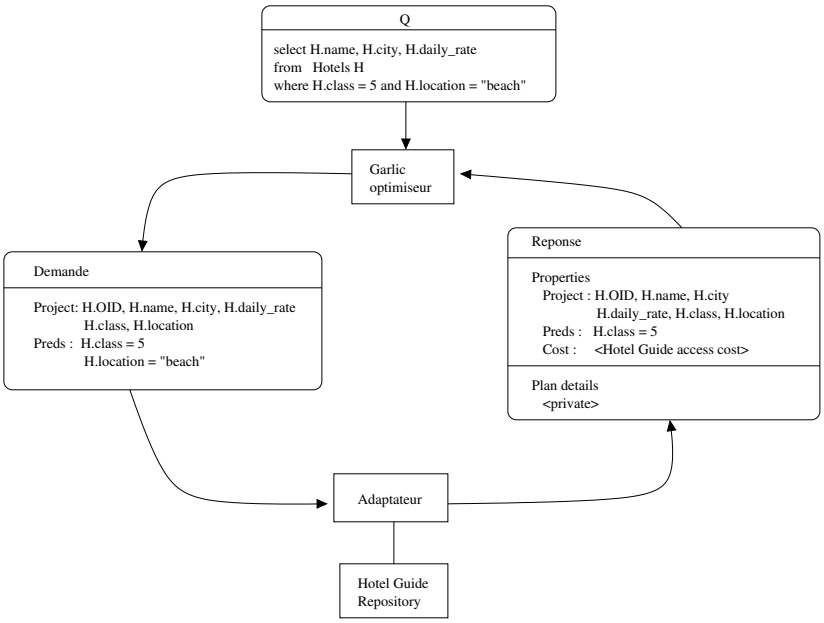



Figure 2.11: Exemple de négociation de plan d'exécution dans Garlic

La figure 2.11 montre l'exemple d'une étape du processus de construction et/ou de décomposition de requête dans Garlic. Lors du traitement d'une requête $\mathcal{Q}$, le médiateur envoie à l'adaptateur une `Demande` de l'exécution d'une sous-requête (qui est la projection dans cet exemple). L'adaptateur traite cette demande et retourne au médiateur sa `Reponse` qui est la requête qu'il peut exécuter. Dans notre exemple, la sélection sur l'attribut `location` est rejetée par l'adaptateur.

Notons que, la capacité de “négociation” des adaptateurs est le facteur clé de la réussite (ou tout simplement de la faisabilité) de cette approche.

## 2.5 Conclusion du chapitre

Dans ce chapitre, nous avons rappelé les principes de base de l'évaluation de requêtes en commençant par le processus de traitement de requêtes. Nous nous sommes focalisés par la suite à deux fonctions les plus importantes que sont l'optimisation et l'exécution. Et puis, nous considérons les techniques particulières dans le contexte des systèmes de médiation.

Généralement, l'optimisation exploite les connaissances des données interrogées et de l'environnement d'exécution afin de choisir un plan optimal pour l'évaluation de requête utilisateur. L'optimisation précède l'exécution de requête en supposant que le choix du plan optimal basé sur l'estimation de coût de ses opérateurs est fiable et “indiscutable”. Par ailleurs, il est à noter que les techniques proposées sont souvent appropriées à certains contextes spécifiques mais pas à tous. Les programmeurs de l'évaluateur doivent donc choisir l'ensemble de techniques adaptées au contexte de leur évaluateur. Ce choix n'est pas simple car il nécessite des connaissances approfondies sur les techniques d'évaluation de requêtes tout en considérant le contexte d'application. En plus, que se passera-t-il si ce contexte change ?

De toutes ces raisons, il est souhaitable de fournir un support *adaptable* pour aider les programmeurs à la construction de leur évaluateur ainsi pour rendre ces derniers *adaptables*. Que signifie le mot “adaptable” ? Le chapitre prochain tente de donner une réponse précise et détaillée à cette question.

# CHAPITRE 3

# Adaptabilité dans les évaluateurs de requêtes

## 3.1 Introduction à l'adaptabilité

### 3.1.1 Définitions

*"Adapter (v.tr.) : rendre (dispositif, des mesures, etc.) apte à assurer ses fonctions dans des conditions particulières ou nouvelles.*
*Adaptation (n.f.) : action d'adapter ou de s'adapter*
*Adaptabilité (n.f.) : capacité d'être adapté ou de s'adapter"* [1]

Qu'entend-on par adaptation dans les évaluateurs de requêtes ? En reprenant la définition sémantique du terme, une adaptation correspond au processus de modification nécessaire pour permettre l'évaluation de requête adéquate dans un contexte donné[2]. Ce *contexte* consiste en (i) la représentation de données (interrogées et résultats), (ii) la capacité de manipulation de données (i.e. opérateurs de requête), (iii) les besoins de l'application et/ou de l'utilisateur pour l'évaluation de sa (ses) requête(s), (iv) de l'environnement d'exécution constitué des systèmes participants, (v) de l'infrastructure de communication, etc. Le terme *adéquate* signifie que l'évaluation de requête correspond parfaitement à ce que l'on attend dans ce contexte précis.

D'une manière générale, l'adaptation se décompose en trois étapes : (i) le **déclenchement** qui consiste à détecter et à notifier un changement ; (ii) la **décision** qui est susceptible de déterminer les modifications devant être effectuées pour réagir aux changements détectés dans la première étape ; et (iii)

[1] cf. Dictionnaire, *Le nouveau petit Robert*.

[2] Nous adaptons ici la définition de l'adaptabilité dans les systèmes logiciels présentée par T. Ledoux et al. dans [Led01] au cas de l'évaluateur de requêtes.

la **réalisation** qui concerne tous les moyens devant être mis en œuvre pour appliquer la décision prise à l'étape précédente. Plusieurs éléments interviennent dans l'adaptation : le *moment* de l'adaptation, l'*acteur* de l'adaptation, le *sujet* de l'adaptation, et le *mécanisme* pour l'adaptation. Ces éléments sont souvent dépendants les uns des autres. Dans le cadre d'évaluation de requêtes, il existe trois temps importants pour l'adaptation : (i) la construction d'un évaluateur, (ii) le lancement de l'évaluation d'une requête, et (iii) pendant l'exécution d'une requête en cours[3]. Respectivement, nous parlons d'adaptation statique, de personnalisation et d'adaptation dynamique comme les trois formes d'adaptation, mais elles ne sont pas exclusives.

En effet, pour autoriser des adaptations ultérieures, il est nécessaire, en *phase de conception et développement*, de prévoir les possibilités d'adaptabilité de l'évaluateur et de faire des choix en conséquence. Ces possibilités se trouvent dans la représentation des données interrogées, dans la capacité d'interrogation (i.e. les opérateurs de requête), dans la capacité d'optimisation (i.e. les fonctionnalités de l'optimiseur telles que la génération de l'espace de recherche, l'estimation de coût et la stratégie de recherche), dans la capacité d'exécution de requête (i.e. les algorithmes). D'ailleurs, si on s'intéresse à l'adaptation de l'évaluation de requêtes au début ou pendant l'exécution, il faut choisir un ou plusieurs mécanismes appropriés et les mettre en œuvre. Toutefois, ces choix se font *une seule fois au moment du codage de l'évaluateur de requêtes par les programmeurs*. Nous parlons alors d'**adaptabilité statique**. La compilation, la technologie objet sont, entre autres, des candidats pour réaliser ce type d'adaptation.

Lorsqu'un évaluateur est construit et mis en place, il peut être utilisé pour répondre à des requêtes de plusieurs utilisateurs ayant des besoins différents sur l'évaluation de leurs requêtes. Selon ces besoins, les critères du choix de plan optimal peuvent être différent d'un utilisateur à un autre, voire d'une requête à une autre. Un évaluateur de requêtes capable d'adapter le choix de la stratégie d'exécution d'une requête aux besoins de l'utilisateur offre un niveau d'adaptabilité, appelé **personnalisation**. Effectivement, cette adaptation permet de rendre un évaluateur existant adapté à des besoins spécifiques de chaque utilisateur, voire de chaque requête utilisateur. La paramétrage est une des techniques permettant de réaliser ce type d'adaptation.

Notons que la "puissance" du système d'évaluation de requête se trouve dans son optimiseur susceptible de construire un plan optimal pour l'exécution de requête. Comme nous l'avons vu dans le chapitre précédent, l'optimisation utilise les connaissances sur les données et sur l'environnement d'exécution afin de décider du plan optimal de requête qui doit être exécuté. Pourtant, au moment de l'exécution, les conditions peuvent déjà être différentes de celles estimées, du fait des changements dans les données interrogées et de ceux de l'environnement d'exécution. Pour faire face à ces changements, il est souhaitable que l'exécution de requête puisse s'adapter à de nouvelles conditions. Il s'agit de l'adaptation de la stratégie d'exécution de la requête en cours, i.e. de l'algorithme de requête, de l'ordonnancement d'exécution de requête, du plan de requête. Nous la qualifions d'**adaptabilité dynamique**. Les techniques de paramétrage et de réflexion permettent de réaliser ce type d'adaptation.

[3]Nous considérons ici le moment pour déclencher l'adaptation mais non pas celui pour prendre de décision ou de réaliser l'adaptation.

### 3.1.2 Adaptabilité dans les travaux existants

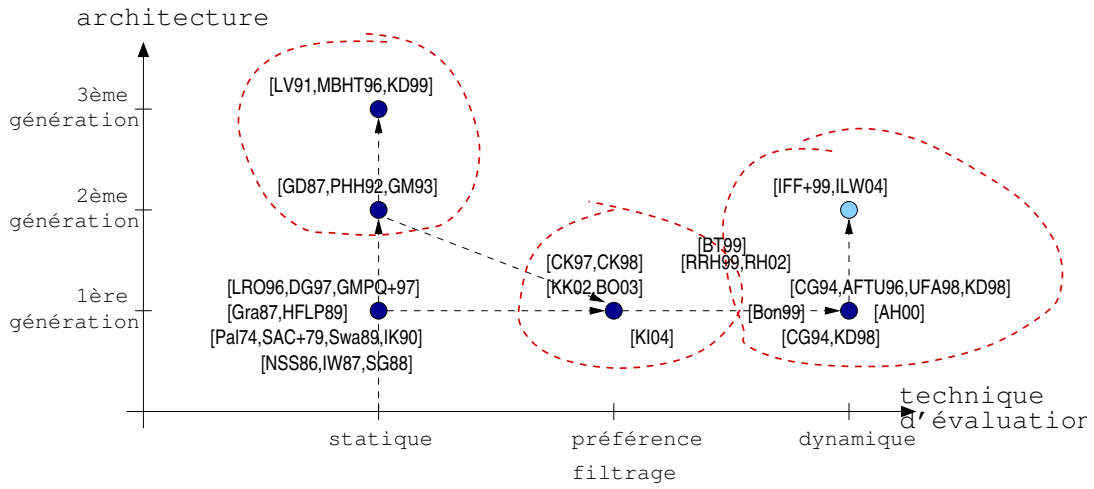



Figure 3.1: Vue d'ensemble des travaux existants

Au vu des éléments de l'adaptation (i.e. sujet, acteur, moment, mécanisme), quelques travaux existants sur l'évaluation de requêtes considèrent *déjà* une ou plusieurs niveau(x) d'adaptabilité bien que cette notion ne soit pas toujours mentionnée. Ces travaux peuvent s'axer sur l'architecture de l'optimiseur ou sur la technique d'évaluation, comme le montre la figure 3.1.

En ce qui concerne l'architecture, on peut considérer trois générations d'optimiseurs de requêtes dont la capacité d'adaptation est connue sous le nom d'*extensibilité*. La première génération est marquée par de premiers SGBD commerciaux basés sur les variations de l'optimisation à la System R [SAC$^+$79]. Cette génération dispose une extensibilité très limitée. Par exemple, l'ajout d'un nouvel opérateur requiert beaucoup de changements dans l'optimiseur. Au début des années 90s, l'arrivée de nombreux modèles de données a été la motivation des travaux connus sous le terme de *générateur d'optimiseur* et/ou *SGBD extensible* [GD87, LFL88, PHH92, GM93]. Ces travaux, formant la deuxième génération d'optimiseurs, permettent d'offrir la capacité d'adaptation à différentes classes d'applications. Ils proposent d'utiliser des règles pour générer les plans pendant l'optimisation. L'extensibilité se trouve dans l'ajout de nouveaux opérateurs de requête et donc de nouvelles règles d'optimisation. Par conséquent, ce que l'on peut étendre ici n'est que l'espace de recherche alors que la stratégie de recherche est fixée à l'avance. Récemment, une nouvelle génération d'optimiseurs utilise la *technologie objet* pour faciliter la construction et l'extension des optimiseur en rendant flexible les stratégies de recherche [LV91, MBHT96, KD99].

L'extensibilité qu'offrent les systèmes des deux dernières générations correspondent à l'*adaptabilité statique*. En effet, ils permettent aux programmeurs d'adapter les systèmes existants à différentes classes d'application ayant différents besoins de représentation de données et donc ceux d'interrogation de données. L'adaptabilité est obtenue par l'utilisation de la technologie de compilation (la deuxième généra-

tion) ou par l'utilisation de la technologie objet (la troisième génération).

En ce qui concerne les techniques d'évaluation de requêtes, nous pouvons les classer en trois catégories selon leur complexité : (i) l'*évaluation statique de requête*, (ii) l'*évaluation personnalisée de requête*, et (iii) l'*évaluation dynamique de requête*. Par évaluation statique, nous entendons l'approche dans laquelle les requêtes en entrée sont optimisées, d'une façon identique, une seule fois, en utilisant les connaissances pré-acquises et puis exécutées comme il était décidé [SAC$^+$79, SG88, Swa89, IK90, IK91]. C'est le cas de tous les optimiseurs de la première génération ainsi que ceux résultant d'une adaptation des systèmes de deux dernières générations que nous avons évoquées précédemment. Dans le deuxième groupe de techniques d'évaluation de requête [CK97, CK98, RRH99, Bon99, BO03, KI04], il s'agit d'introduire un ordre de préférence parmi les critères de sélection, de jointure dans la requête et/ou sur les résultats de requête dans le but de rendre l'évaluation adaptée aux besoins spécifiques de chaque utilisateur, voire de chaque requête utilisateur. L'expression de telles requêtes se fait souvent grâce à l'extension d'un langage de requêtes sensée traduire les préférences de l'utilisateur sur l'évaluation de requête. A la réception de la requête, le système interprète les préférences de l'utilisateur et construit un plan optimal pour l'exécution de requête dans le but d'assurer le retour des données satisfaisant les préférences de l'utilisateur. La dernière catégorie de techniques d'évaluation de requête, connue sous le terme d'*évaluation adaptative et/ou interactive*, consiste dans la capacité d'adapter l'exécution des requêtes en cours d'exécution suite à un ou plusieurs changements dans l'environnement d'exécution quels que soit provenant de sources ou de l'utilisateur [GW89, WA91, INSS92, CG94, UFA98, KD98, Bon99, HH99, IFF$^+$99, BFMV00b, AH00, UF01, RH02, ILW04]. L'idée est de ne plus considérer la stratégie d'exécution de requête (i.e. plan optimal) généré par la phase d'optimisation précédant l'exécution comme étant toujours correct et invariable. Par l'utilisation de différentes technologies telles que la *paramétrage*, la *réflexion*, l'évaluateur est capable d'adapter la stratégie d'exécution de requête en cours selon les connaissances acquises pendant l'exécution.
En considérant les trois niveaux d'adaptabilité définis antérieurement, les techniques d'évaluation concernent surtout la personnalisation et l'adaptabilité dynamique. La première classe des techniques d'évaluation n'autorise aucune adaptation. La deuxième classe est un support, plus ou moins complexe, pour la personnalisation car elles sont capables d'adapter l'évaluation d'une requête donnée à quelques besoins spécifiques de l'utilisateur. Les techniques de troisième catégorie permettent d'assurer l'adaptabilité dynamique définie comme l'adaptation en cours d'exécution.

Remarquons que tous les travaux axés sur l'architecture autorisent, d'une manière ou d'une autre, l'adaptabilité statique mais ils considèrent seulement les techniques d'évaluation statique. Dans ces travaux, aucun support de la personnalisation et de l'adaptabilité dynamique n'est prévu. Par ailleurs, la plupart des travaux sur les techniques d'évaluation complexes, permettant une ou quelques adaptation(s) dans les évaluateurs construits, sont orientés système, c'est-à-dire ils offrent quelques adaptations dans des systèmes spécifiques ayant des caractéristiques spécifiques. L'utilisation de ces techniques dans d'autres systèmes, d'autres contextes n'est pas chose aisée. Notre propos est justement de faire s'approcher ces deux directions de travail dans le but d'offrir un outil permettant la construction des évaluateurs adaptables de requêtes. Comprendre comment les travaux existants supportent l'adaptation

est le premier pas *indispensable* vers la solution attendue.

**Plan du chapitre** La suite de ce chapitre présente une analyse des travaux existants comme étant les mécanismes pour obtenir les trois niveaux d'adaptabilité que nous avons définis, à savoir l'adaptabilité statique (la section 3.2), la personnalisation (la section 3.3) et l'adaptabilité dynamique (la section 3.4.2). Nous concluons le chapitre dans la section 3.5 par une synthèse de l'ensemble des travaux existants du point de vue de l'adaptation et de l'adaptabilité.

## 3.2 Adaptabilité statique : les systèmes extensibles

L'adaptabilité statique est considérée principalement dans les travaux de générateurs d'optimiseurs ou de SGBD extensibles. Il s'agit des outils d'aide à la construction de nouveaux systèmes de gestion de données (ou plus concrètement leur module d'optimisation). Ces travaux, formant les deux dernières générations d'optimiseurs de requêtes, se différencient par leur niveau d'extensibilité. La première permet une extension de l'espace de recherche en fixant la stratégie de recherche. Concrètement, il est possible de spécifier de nouveaux modèles de données, de nouveaux opérateurs sachant que tous les optimiseurs générés adoptent la même stratégie de recherche. La seconde offre, de plus, l'extensibilité des stratégies de recherche.

Cette section présente une analyse de ces travaux selon ces niveaux : l'extensibilité de l'espace de recherche (la sous-section 3.2.1) et l'extensibilité de la stratégie de recherche (la sous-section 3.2.2).

### 3.2.1 Extensibilité de l'espace de recherche

Ce sont principalement les travaux de l'optimisation basée sur les règles. L'idée est de re-concevoir les fonctionnalités d'optimisation d'une manière suffisamment générale pour qu'elles soient indépendantes du modèle de données et soient réutilisables.

La figure 3.2 montre la vue générale de cette approche où l'utilisation se fait en deux étapes : la *génération* et l'*exécution*. Au moment de la génération, l'utilisateur, qui est en fait le concepteur du SGBD, fournit (i) l'ensemble des opérateurs logiques, (ii) les règles de transformation algébrique, (iii) l'ensemble des algorithmes implémentant les opérateurs logiques et les *enforcers* (appelés opérateurs physiques)[4], (iv) les règles d'implémentation, (v) la liste des propriétés logiques et physiques, (vi) la fonction de coût pour chaque opérateur physique. Etant donné ces informations, le générateur d'optimiseur génère le code de l'optimisation approprié et l'associe aux modules déjà implémentés dans le générateur, notamment le moteur de recherche (i.e. l'implémentation de la stratégie de recherche). Les optimiseurs résultant de la génération fonctionnent d'une manière traditionnelle au moment de l'exécution.

[4]Les *enforcers* sont les opérateurs physiques qui ne correspondent à aucun opérateur logique. L'objectif de ces opérateurs est "d'imposer" aux données les propriétés physiques nécessaires pour les traitements ultérieurs. Ces opérateurs sont aussi appelés *glue* dans quelques systèmes.

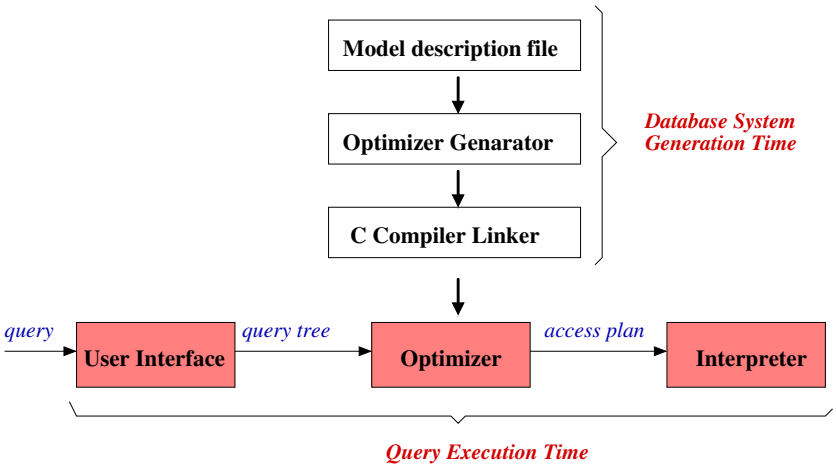



Figure 3.2: Approche de générateur d'optimiseur

Quelques travaux représentatifs de cette approche sont EXODUS [GD87], Starburst [LFL88, HFLP89] et Volcano [GM93]. Bien que les informations d'entrée de ces générateurs soient relativement similaires, ils se différencient par la complexité de langage de spécification ainsi que la stratégie de recherche fournie.

EXODUS [GD87] et son successeur Volcano [GM93] utilisent une stratégie de recherche par transformation (*top-down*). Dans cette stratégie, il s'agit d'appliquer successivement les règles d'optimisation sur un plan complet selon un ordre de priorité : les règles de transformation de plans logiques, celles d'implémentation permettant de générer les plans physiques à partir des plans logiques, puis l'insertion des opérateurs *enforcer*s. L'efficacité de Volcano (au moins, par comparaison à EXODUS) se trouve dans sa stratégie de recherche basée sur la programmation dynamique dirigée (*directed dynamic programming*). L'idée principale est d'explorer seulement les sous-requêtes ou plans qui participent effectivement à une expression plus large. Pour une plus ample présentation de l'algorithme, les lecteurs sont invités à consulter [GM93]. L'utilisateur de Volcano fournit la description contenant les informations présentées précédemment dans un fichier de description de modèle contenant deux parties : une pour les déclarations des opérateurs logiques et des algorithmes (*declaration part*), et l'autre contient les règles de transformation et d'implémentation (*rule part*) comme illustré dans le suivant :

```
1: declaration part:
2:    %operator 2 join
3:    %method 2 hash_join  loops_join cartesian_product
   ...
4: rule part:
5:   join(1,2) ->! join(2,1),
6:   join(1,2) by hash_join(1,2),
```

Dans cet exemple, l'opérateur `join` (ligne 2) et les trois méthodes `hash_join`, `loops_join`,

`cartesian_product` (ligne 3) ont `2` entrées pour chacun ; la ligne 5 représente la commutativité de jointures (règle de transformation) et la ligne 6 représente que `hash_join` est un algorithme implémentant l'opérateur `join` (règle d'implémentation). A part cela, le programmeur doit fournir également le code des fonctions de coût et d'autres traitements, si nécessaire, sous forme des procédures en langage C.

A la différence de EXODUS et de Volcano, Starburst [LFL88, HFLP89] adopte la stratégie de recherche à la System-R, c'est-à-dire l'approche par construction (*bottom-up*). Le processus d'optimisation se compose de deux phases : une pour la transformation de requête (réécriture) et l'autre pour le choix de la stratégie d'exécution. Chacune de ces deux phases s'appuie sur les règles qui sont écrites dans deux langages différents. La particularité de Starburst se trouve dans la complexité de sa représentation interne de requête dans le modèle QGM (*Query Graph Model*). Dans ce modèle, chaque opérateur est représenté comme une table composée d'un en-tête qui décrit les données en sortie de la table (i.e. nom des colonnes et type de données) et d'un corps qui est la représentation graphique de l'opérateur. En se basant sur cette structure commune, les auteurs de Starburst proposent d'un langage de règles de réécriture. Une règle composée d'une partie de *condition* et d'une partie de *action* permet de transformer une représentation QGM cohérent en une autre. Etant donné une représentation QGM de requête, l'optimiseur estime le coût des plans alternatifs et choisit le moins coûteux. Starburst génère les plans alternatifs selon l'approche par construction en s'appuyant sur les règles de production STAR (*STrategy Alternative Rule*). Une règle STAR se compose d'un nom, de zéro ou plusieurs paramètre(s) et d'un ou plusieurs plan(s) de bas niveau (i.e. plan composé des algorithmes implémentant des opérateurs) ou d'autre STARs. Pour plus détails sur le modèle QGM et la grammaire de deux langages de règles de Starburst, les lecteurs sont invités à consulter [HFLP89].

Quelques d'autres systèmes [Gra95, FG91] permettent d'ajouter des contrôles plus ou moins complexes guidant la stratégie de recherche afin d'améliorer la tâche d'optimisation mais elle s'appuie souvent sur l'approche par transformation. En résumé, dans tous ces travaux, l'optimisation se base sur des règles représentant les manipulations possibles sur les plans de requête que ce soit la transformation ou la construction. La spécification de l'optimiseur revient donc à écrire cet ensemble de règles tandis que la stratégie de recherche est fixée à l'avance.

### 3.2.2 Extensibilité de la stratégie de recherche

Les travaux autours de l'extensibilité de la stratégie de recherche constituent la troisième génération d'optimiseurs que nous avons évoquée précédemment. La plupart de ces optimiseurs [LV91, MBHT96, KD99] bénéficient les techniques de la composition, de l'héritage, de la délégation, de la généricité de la technologie objet comme étant un moyen d'obtenir l'extensibilité.

L'un des premiers travaux dans cette approche proposé par R. Lanzelotte et al. [LV91] consiste en la séparation conceptuelle de l'espace de recherche et de la stratégie de recherche. Pour cela, les auteurs proposent de les modéliser par trois classes State, SeachSpace et SearchStrategy (voir la figure 3.3). Dans ce travail, le problème d'optimisation de requête est considéré comme le problème de recherche général. Concrètement, chaque solution candidate d'une requête (i.e. un plan d'exécution) est modélisée

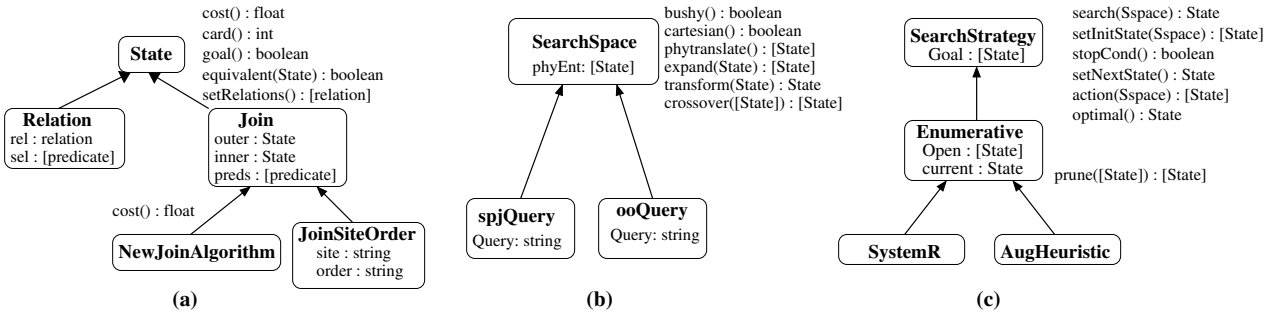



Figure 3.3: Modélisation de l'optimisation dans [LV91]

comme un arbre de traitement (*processing tree*) composé des nœuds opérateurs représentés par la classe State. Les méthodes de cette classe permet d'effectuer le calcul des propriétés de chacun des nœuds (e.g. le coût estimé, la cardinalité estimé de résultats, etc.). L'espace de recherche représenté par la classe SearchSpace contient alors les arbres de traitement. Les méthodes de la classe SearchSpace telles que `bushy`, `cartesian`, etc. (voir la figure 3.3(b)) définissent la forme de l'espace de recherche (e.g. l'arbre à gauche, à droit ou *bushy*) ainsi que les opérations de manipulation possibles sur l'arbre de requête (e.g. `phytranslate`, `expand`, `transform`, etc.). La stratégie de recherche modélisée par la classe SearchStrategy (voir la figure 3.3(c)) correspond au parcours de l'espace de recherche ou plus précisément aux actions à appliquer sur les états pour produire un ensemble de nouveaux états. Ces derniers correspondent en effet à des plans partiels ou complets de la requête. Deux états spéciaux sont *l'état initial* (défini par la méthode `setInitState`) et *l'état final* (vérifié par la méthode `stopCondition`).

L'implémentation d'un optimiseur revient donc à spécialiser les classes State, SearchSpace, SearchStrategy afin de définir les opérateurs/algorithmes (e.g. Join, JoinSiteOrder, etc.), l'espace de recherche approprié pour les différents types de requête (e.g. requête sur les relation spjQuery, requête sur les objets ooQuery, etc.), et la stratégie de recherche (e.g. stratégie à la SystemR, par heuristique AugHeuristic, etc.).

En adoptant la même approche que le travail décrit ci-dessus, quelques travaux récents comme EROC [MBHT96], OPT++ [KD99] proposent une conception à grain plus fin afin de faciliter l'extension de l'optimiseur. La figure 3.4 montre la vue d'ensemble de OPT++.

A la différence de [LV91], les auteurs de OPT++ proposent de séparer la représentation de plan logique et de plan physique. Ces deux derniers sont représentés respectivement par les classes Operator et Algorithm. Par ailleurs, les opérations de manipulation de plan de requête sont représentées dans OPT++ sous forme des classes mais non pas par des méthodes comme dans le travail antérieur. Les trois différents types d'opération sont la transformation de plan logique (la classe TreeToTreeGenerator), la translation d'un plan logique à un plan physique (la classe TreeToPlanGenerator) et l'insertion des *enforcers* dans le plan (la classe PlanToPlanGenerator). Toutes ces classes définies

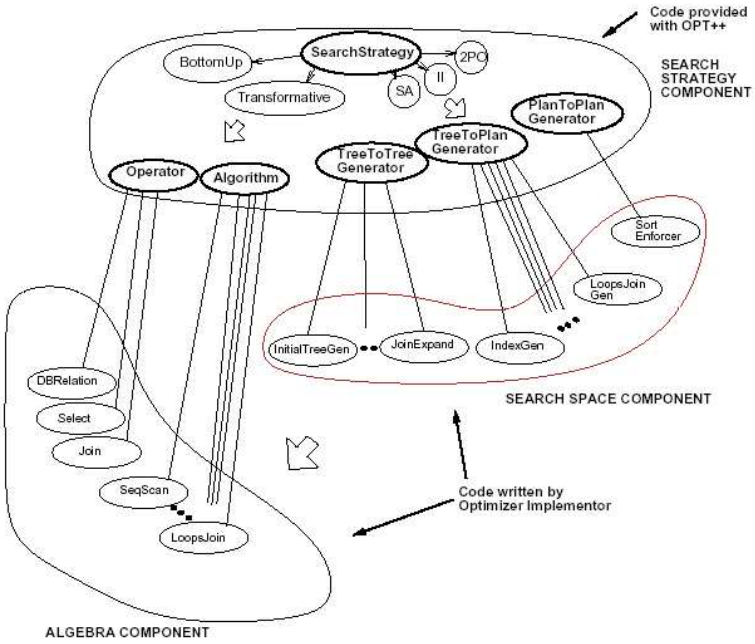



Figure 3.4: Vue générale de OPT++ [KD99]

comme étant les classes abstraites dans OPT++ seront spécialisées lors de l'implémentation d'un optimiseur. Dès lors, l'implémentation des différentes stratégies de recherche se fait sur les classes abstraites d'une manière plus ou moins indépendante de l'implémentation des opérateurs, des algorithmes et des opérations d'optimisation d'un optimiseur concret.

### 3.2.3 Discussion

Dans cette section, nous avons présenté les travaux ayant pour but d'offrir les outils pour la construction des systèmes de traitement de requêtes ou plus concrètement leur module de l'optimisation. Ces travaux sont classifiés en deux catégories : l'une se base sur l'utilisation des règles pour offrir l'extensibilité de l'espace de recherche [GD87, PHH92, GM93] et l'autre s'appuie sur la technologie objet pour assurer l'extensibilité de la stratégie de recherche aussi bien que celle de l'espace de recherche [LV91, KD99]. La table 3.1 résume les propriétés de ces travaux.

Il est à noter que chacun de ces travaux fournissent un certain nombre de stratégies de recherche « prêtes-à-utiliser » mais le changement d'une stratégie à une autre ne semble pas facile. Cela est aussi vrai si on considère l'ensemble des fonctionnalités de l'optimisation (i.e. la stratégie de recherche, la génération de l'espace de recherche et l'estimation de coût). En effet, la composition de ces fonctionnalités ne se fait qu'une seule fois lors de la construction de l'évaluateur[5]. En outre, tous les évaluateurs

[5]Sans surprise, tous ces travaux ont pour *seul* objectif d'offrir un support pour la construction d'évaluateur (!)

| Références | Espace de recherche | Stratégie de recherche | Estimation de coût |
|---|---|---|---|
| EXODUS [GD87] | **extensible** - à définir : les opérateurs, les règles d'optimisation | **fix** - *topdown* | **extensible** - à définir : la fonction de coût |
| Starburst [PHH92] | **extensible** - *idem.* | **fix** - *bottomup* | **extensible** - *idem.* |
| Volcano [GM93] | **extensible** - *idem.* | **fix** - *topdown* | **extensible** - *idem.* |
| [LV91] | **extensible** - à définir : les classes spécialisées de State et SearchSpace | **extensible** - à définir : la classe spécialisée de SearchStrategy | **extensible** - à définir : la méthode `cost` de la classe spécialisée de State |
| OPT++ [KD99] | **extensible** - à définir : les classes spécialisées de Operator, Algorithm, TreeToTreeGenerator, TreeToPlanGenerator, PlanToPlanGenerator | **extensible** - à définir : la classe spécialisée de SearchStrategy | **extensible** - à définir : description associée à des opérateurs |

Table 3.1: Propriétés des propositions concernant les architectures extensibles et les générateurs d'optimiseur

résultant ne permettent que l'évaluation statique de requêtes, i.e. l'évaluation de requête se fait en deux étapes d'optimisation et d'exécution, l'une après l'autre.

Il serait intéressant d'étudier la possibilité de séparer clairement toutes les fonctionnalités de l'optimisation ainsi qu'une possibilité de les composer de façon plus dynamique, i.e. non seulement à la construction d'évaluateur mais aussi à l'évaluation de certaines requêtes particulières. En outre, pourrait-on étendre une telle approche afin de supporter d'autres mécanismes d'évaluation tels que ceux de l'évaluation personnalisée, adaptative et interactive présentés dans les sections suivantes ?

## 3.3 Personnalisation : quelques techniques

Le titre de cette section montre, peut-être déjà, le manque de travaux fournissant systématiquement la capacité de personnalisation dans les SGBD. En effet, la personnalisation dans les SGBD apparaît principalement au niveau du langage de manipulation de données. La définition de vues, l'utilisation de critères de filtrages approximatifs, l'introduction d'un ordre de préférence parmi les critères ou d'un ordre de préférence sur les résultats des requêtes constituent les principales contributions à la personnalisation [Bou04]. Par ailleurs, la plupart des travaux existants proposent la *construction de systèmes de personnalisation au dessus des SGBD existants*.

Ainsi donc, il est difficile de fournir ici une vue globale des techniques concernant la personnalisation. Cette section présente une analyse de *quelques* travaux répondant, d'une manière ou d'une autre, à certains besoins de la personnalisation. Il s'agit des techniques qui prennent en compte les besoins spé-

cifiques de l'utilisateur *explicitement* définis à l'entrée de l'évaluateur et qui sont intégrées à l'*intérieur* du processus d'évaluation de requête du SGBD mais non pas en dehors du SGBD.

Les techniques présentées dans cette section sont : (i) les requêtes *Top/Bottom-N* qui permettent le retour d'un nombre limité des résultats donné par l'utilisateur (la sous-section 3.3.1) ; (ii) les requêtes parachutes qui permettent le retour des résultats incomplets selon les préférences d'utilisateur dans le cas de sources indisponibles (la sous-section 3.3.2) ; (iii) les requêtes ayant des prédicats flous qui permettent de le retour des "meilleurs" résultats selon des conditions floues (la sous-section 3.3.3).

### 3.3.1 Requête *Top/Bottom-N*

Partant de l'observation que le nombre de résultats d'une requête peut être très grand pendant que l'utilisateur peut ne pas avoir besoin d'autant, M. Carey et al. proposent un type de requête nommé *Top/Bottom-N* [CK97]. L'idée est d'offrir à l'utilisateur un moyen de spécifier le nombre maximal de résultats qu'il souhaite recevoir. Pour cela, les auteurs proposent d'étendre le langage de requête SQL par la clause `STOP AFTER` sémantiquement équivalente à l'arrête de l'exécution de requête après l'obtention de *N* n-uplets. La grammaire de la requête SQL étendue est comme suit :

```
SELECT   ...   FROM   ...   WHERE  ...
GROUP BY  ... HAVING   ...
ORDER BY   <sort specification list>
STOP AFTER <value expression>
```

*N* résultats de la requête peuvent être ordonnés ou non selon la spécification de la clause `ORDER BY`. Dans le langage algébrique, la clause `STOP AFTER` est traduite par un nouvel opérateur logique nommé *Stop(n)* qui produit au maximal *N* n-uplets. Cet opérateur a trois paramètres dont la valeur est fournie au moment d'initier de la requête : (i) *N*, le nombre de n-uplets souhaité ; (ii) *SortDirective* ayant une des trois valeurs *desc*, *asc*, ou *non* correspondant à l'ordre de tri des n-uplets ; et (iii) *SortExpression* correspond à la clause `ORDER BY`. L'optimisation de requête *Top-N* revient donc à déterminer l'endroit où l'opérateur *Stop(n)* serait inséré dans le plan de requête. Elle suit une des deux stratégies nommés *Conservative* et *Aggressive*.

La première stratégie d'optimisation (*Conservative policy*) assure *d'une manière certaine* qu'il y aura *N* n-uplets produits (excepté, bien sûr, le cas où le nombre total de résultats ne dépasse pas *N*). Pour cela, l'opérateur *Stop(n)* est inséré entre des nœuds de plan si et seulement si chaque élément du flux entre ces nœuds produit au moins un élément de résultat. La figure 3.5(a) montre le plan de requête optimisé en appliquant cette stratégie d'optimisation.

La deuxième stratégie d'optimisation (*Aggressive policy*) tente de bénéficier de la réduction de cardinalité de résultats (intermédiaires). L'opérateur *Stop(n)* est propagé dans l'arbre de requête à partir de la racine vers les feuilles. A chaque étape, le paramètre *N* est recalculé par la fonction suivante : $N_{Stop} = \frac{ALL_{subplan}}{ALL_{query}} * N$, où $All_{subplan}$ est la cardinalité estimée du sous-plan de l'opérateur *Stop(n)* en question. La figure 3.5(b) montre un plan généré selon cette stratégie. Selon cette approche, l'insertion des opérateurs *Stop(n)* s'appuie sur l'estimation de la cardinalité qui peut être erronée. Ainsi donc, dans le

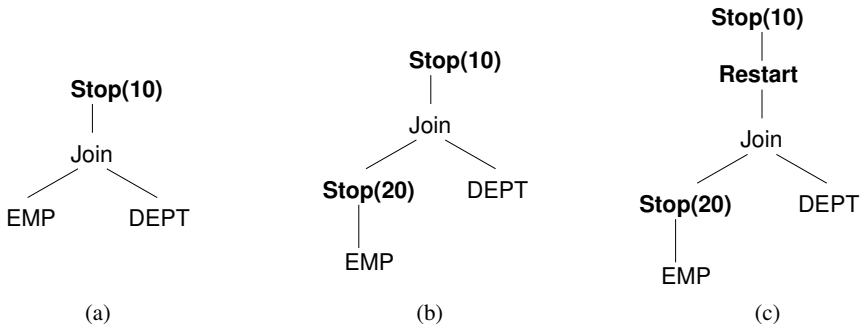



Figure 3.5: Exemple de plan de requête *Top-N* avec les différentes stratégies d'optimisation

cas où le nombre de résultats obtenu n'est pas suffisant, le système relance l'exécution de requête. Afin d'éviter de ré-exécuter la requête entière, les auteurs proposent l'insertion d'un opérateur *Restart* dans le plan au point où la re-exécution de sous-plan est bénéfique. La figure 3.5(c) montre l'exemple de plan de requête avec l'opérateur *Restart*.

### 3.3.2 Requête parachute

La notion de *requête parachute* proposée par P. Bonnet et al. fait allusion au traitement de requête en présence de sources de données indisponibles [BT98]. Partant de la constatation que le traitement d'une requête dans les systèmes de médiation tente probablement d'accéder à une source qui n'est pas disponible et que l'utilisateur peut parfois se contenter des données partielles, les auteurs proposent de fournir à l'utilisateur une *réponse partielle* dans le cas où la *réponse complète* ne peut pas être construite.

Etant donné une requête sur les sources multiples, le système procède à une phase de détection des données indisponibles. S'il existe au moins une source indisponible, une phase de matérialisation est réalisée afin d'obtenir les données des autres sources (i.e. celles disponibles) de manière de les utiliser ultérieurement. Ainsi, le système génère une *requête incrémentale*, étant le résultat de la réécriture de la requête initiale en utilisant les relations matérialisées. L'évaluation de la requête incrémentale fournit, en principe, le même résultat que la requête initiale[6].

L'objectif de la requête parachute est effectivement d'extraire les données utiles pour l'utilisateur à partir des données partielles matérialisées. L'extraction des données partielles dépend de plusieurs facteurs tels que les informations disponibles, les informations intéressantes pour l'utilisateur, etc. Par ailleurs, plusieurs scénarios peuvent être imaginés, e.g. l'utilisateur peut intervenir après que les données partielles sont calculées (et matérialisées) pour formuler la requête parachute extrayant les informations qui l'intéressent ou lorsqu'il formule une requête, il indique les données qui l'intéressent [Bon99]. Nous nous intéressons ici seulement au deuxième scénario où l'utilisateur *indique* à priori ce qu'il peut gérer

[6]Les auteurs supposent qu'aucune mise à jour effectuée sur les sources n'impacte la validité des données matérialisées. Si tel n'est pas le cas, le système laisse à l'application le soin de décider si elle peut tolérer les différences éventuelles entre la réponse à la requête incrémentale et la réponse à la requête initiale.

comme résultats partiels lors de la formulation de requête et c'est à la charge de système de trouver la requête parachute appropriée pour pouvoir lui retourner les résultats partiels. En effet, c'est le cas le plus complexe pour le système et ainsi ce scénario est conforme à ce que nous définissons comme la capacité de personnalisation d'un système d'évaluation de requêtes (cf. la section 3.1).

#### 3.3.2.1 Indication fournie par l'utilisateur

L'utilisateur a plusieurs moyens pour indiquer les réponses partielles qui l'intéressent.

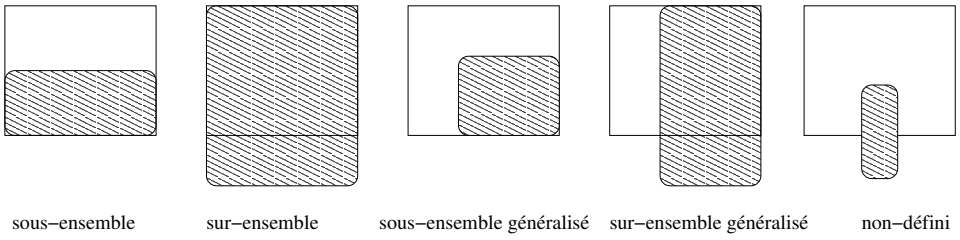



Figure 3.6: Relation entre réponse complète et réponse partielle (en grisé)

Une première possibilité est d'indiquer la relation entre la réponse partielle et celle complète en laissant le système de décider de la réponse partielle qu'il pourra générer. Cette relation peut être un sous-ensemble, un sur-ensemble, un sous-ensemble généralisé, un sur-ensemble généralisé ou non-défini comme illustré dans la figure 3.6.

Une deuxième possibilité est que l'utilisateur formule explicitement les requêtes parachutes en utilisant un langage de requête, comme SQL, ou un langage visuel tel que QBE. L'avantage de cette approche est que l'utilisateur indique très précisément les informations partielles qui l'intéressent et que le système peut en tirer profit pour retourner la réponse partielle convenant le mieux à l'utilisateur.

Les deux sections suivantes présentent respectivement les solutions proposées pour ces deux scénarios.

#### 3.3.2.2 Génération de requêtes parachutes

L'idée de l'algorithme de génération de requêtes parachutes est de réécrire la requête initiale en utilisant les vues matérialisées. Seules les requêtes conjonctives sont considérées et la réécriture est effectuée pour chacun des blocs de la requête initiale. Le principe de l'algorithme de génération de requêtes parachutes est le suivant :

**Entrée** la requête d'utilisateur $\mathcal{Q} = \mathcal{Q}_1 \cup ... \ \mathcal{Q}_i \ ... \cup \mathcal{Q}_n$
l'ensemble de vues matérialisées $\mathcal{V}_j$
la relation $\mathcal{R}$ entre la réponse complète et la réponse partielle acceptée par l'utilisateur.

**Sortie** un ensemble de requêtes parachutes

(i) déterminer pour chaque $\mathcal{Q}_i$, l'ensemble des vues $\mathcal{V}_j$ associées, i.e. les vues sont ses sous-requêtes.

(ii) déterminer pour chaque $\mathcal{Q}_i$, les sous-ensembles maximum de vues $\mathcal{V}_j$ qui peuvent être jointes en respectant les conditions exprimées dans $\mathcal{Q}_i$. L'objectif est d'obtenir des requêtes parachutes qui rapprochent le plus possible des blocs $\mathcal{Q}_i$ de la requête initiale.

(iii) déterminer les unions qui peuvent être effectuées entre les requêtes parachutes générées par l'étape précédente.

(iv) déterminer la relation entre la requête parachute et requête initiale et retourner seulement les requêtes parachutes dont la relation à la requête initiale est conforme à $\mathcal{R}$

#### 3.3.2.3 Optimisation sous contrainte

Dans le cas où l'utilisateur spécifie les données qui l'intéressent dans des requêtes parachutes associées à la requête initiale, le système peut tirer profit de la connaissance des requêtes parachutes pour s'assurer que les informations nécessaires à l'évaluation des requêtes parachutes sont obtenues et conservées. C'est le propos de l'algorithme d'optimisation sous contrainte qui a pour le principe d'identifier les sous-requêtes partagées (SRP) entre la requête initiale et les requêtes parachutes ainsi qu'une stratégie de matérialisation appropriée. La racine de chaque SRP est annotée. Ces annotations sont utilisées par l'algorithme de matérialisation pour stocker le résultat des sous-requêtes partagées. Un plan d'exécution est également produit pour chaque requête parachute en utilisant les sous-requêtes partagées [Bon99]. Le principe de l'algorithme d'optimisation sous-contrainte est le suivant :

**Entrée** la requête d'utilisateur $\mathcal{Q}$ = $\mathcal{Q}_1 \cup ... \ \mathcal{Q}_i \ ... \cup \mathcal{Q}_n$
l'ensemble de requêtes parachutes $\rho_j$

**Sortie** le plan d'exécution de $\mathcal{Q}$ où la racine de chaque sous-requête partagée entre la requête initiale et les requêtes parachutes est identifiée et annotée.

(i) identifier l'ensemble des SRP. Intuitivement, cet ensemble correspond à l'ensemble des intersections possibles entre la requête initiale et les requêtes parachutes

(ii) considérer deux stratégies de matérialisation : (a) chaque SRP identifiée doit être matérialisée en un seul bloc (stratégie maximale) ; (b) chaque littéral apparaissant dans une SRP doit être matérialisée séparément (stratégie minimale).

(iii) réécrire $\mathcal{Q}$ pour chaque groupe de SRP matérialisables (la requête réécrite dénotée $\mathcal{Q}'$)

(iv) pour les deux stratégies, l'optimiseur est appelé pour générer le plan d'exécution des SRP matérialisables ainsi que de la requête $\mathcal{Q}'$ qui luis associée.

### 3.3.3 Requête avec les prédicats flous

Partant de l'observation que le nombre des résultats d'une requête peut être très grand dans certains cas et vide dans d'autres cas pendant que l'utilisateur peut se contenter de quelques résultats même si ces derniers ne sont pas complets et/ou s'ils ne satisfont pas toutes les conditions posées par l'utilisateur (appelées conditions rigides), les auteurs de [BKS01, KK02] proposent de considérer les **contraintes floues** (*softs constraints*) définissant les préférences/souhaits de l'utilisateur sur les données de requête. L'idée principale est de retourner seulement les résultats les plus *"appropriés"* à l'utilisateur selon ses préférences qui sont les contraintes floues. Les propositions reposent souvent sur une extension de langage de requête SQL.

Dans *Preference SQL* [KK02], W. Kießling et al. proposent d'étendre SQL par la clause `PREFERRING` définissant des ordres partiels stricts (*strict partial orders*). La syntaxe de *Preference SQL* est comme suit :

```
SELECT  ... FROM ... WHERE ....
PREFERRING ... AROUND | BETWEEN[low,up] | LOWEST | HIGHEST | IN | <> | ...
GROUPING <attribute_list>
BUT ONLY <but_only_condition>
ORDER BY <attribute_list>
```

La sémantique de la requête *Preference SQL* est décrite comme suit : (i) chercher tous les n-uplets satisfaits les préférences définies dans la clause `PREFERRING` ; (ii) si aucun n-uplet n'est trouvé, considérer alors toutes les n-uplets satisfaisant les conditions définies dans la clause `BUT ONLY` et retourner seulement ceux non-dominées. L'exécution de requête *Preference SQL* utilise une "fenêtre" (*window*) qui contient des n-uplets en cours de traitement qui vont *probablement* participer aux résultats finaux de la requête. Lors de la lecture d'un n-uplet, ce n-uplet est vérifié d'après les contraintes floues en regardant, si nécessaire, les n-uplets lus (i.e. ceux dans la fenêtre de traitement). Si ce n-uplet correspond aux contraintes floues[7], il est inséré dans la fenêtre de traitement. Le traitement de requête Preference SQL se fait selon d'une des deux approches suivantes : (i) la première vise à construire au dessus d'un SGBD existant un système de réécriture permettant de traduire la requête Preference SQL (i.e. la requête ayant la clause `PREFERRING`) en une requête SQL-92 qui est envoyé par la suite au SGBD sous-jacent pour son évaluation ; (ii) dans la deuxième approche, les traitements de préférence sont intégrés dans le processus de traitement de requête de SGBD. Pour cela, les auteurs proposent deux nouveaux opérateurs que sont la sélection basée sur préférence (*preference selection*) et le groupement basé sur préférence (*grouped preference selection*). Ces deux opérateurs sont semblables respectivement à l'opérateur de sélection et à l'opérateur de groupement dans l'algèbre relationnelle à l'exception des prédicats qui définissent les contraintes floues. L'optimisation de requête se compose de deux phases comme le montre la figure 3.7. Le module Pref. Laws contient les règles de transformation concernant les deux opérateurs ci-dessus pour son optimisation. Pour plus de détails, les lecteurs sont invités à consulter [KK02].

Ayant un objectif proche de *Preference SQL*, S. Borzsonyi et al. proposent les requêtes *Skyline*

[7] Notons que, tous les n-uplets lus ont déjà satisfaits aux conditions rigides.

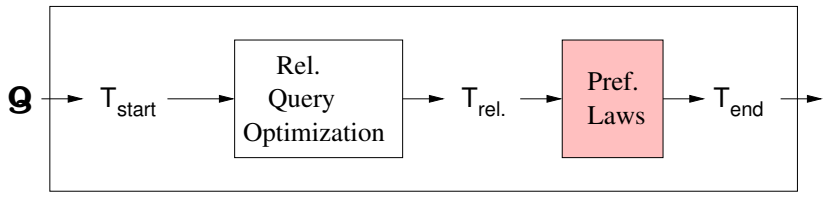



Figure 3.7: Optimisation en deux phases

permettant de filtrer les résultats les plus intéressants parmi un grand ensemble des résultats [BKS01]. Les requêtes Skyline sont écrites dans un langage SQL étendu dont la syntaxe est comme suit :

```
SELECT   ...  FROM   ...   WHERE   ...
GROUP BY   ...  HAVING   ...
SKYLINE OF [DISTINCT] d_1 [MIN | MAX | DIFF],..., d_m [MIN | MAX | DIFF]
ORDER BY   ...
```

où $d_1$, ..., $d_m$ dénotent les dimensions de *Skyline*, e.g. price, distance, rating, etc. Les termes `MIN`, `MAX`, `DIFF` indiquent si la valeur de la dimension correspondante devrait être maximisé, minimisé ou simplement être différente. La clause `SKYLINE` est traduite en langage algébrique par un nouvel opérateur logique nommé *Skyline*. La sémantique de cet opérateur est de sélectionner tous les n-uplets *intéressants*, i.e. les n-uplets qui ne sont pas dominés par d'autres n-uplets. Intuitivement, cet opérateur est exécuté après les opérateurs *Scan*, *Join*, et *Group-By* et avant le dernier opérateur *Sort*, s'il existe. Les auteurs proposent également différentes implémentations de l'opérateur *Skyline* [BKS01].

### 3.3.4 Discussion

La table 3.2 résume les propriétés des techniques d'évaluation de requête que nous avons présentées dans cette section.

Nous trouvons que la plupart de ces techniques consistent en l'ajout d'un ordre de préférence parmi les critères [BKS01, KK02] ou sur les résultats des requêtes [CK97, KK02]. Elles peuvent donc produire les meilleurs résultats pour l'utilisateur *à condition que* toutes les données interrogées par la requête soient disponibles (!). A contrario, les techniques dans DISCO [BT98] permettent de retourner les résultats incomplets dans le cas d'indisponibilité des sources. Le processus de traitement de requête dans DISCO se compose des phases spécifiques pour la détection des sources indisponibles, pour la matérialisation des données temporaires, et pour la réécriture de requête initiale en fonction des données matérialisées. Notons que la détection des sources indisponibles se fait au début de l'évaluation d'une requête et peut se répéter pour l'évaluation de chaque requête incrémentielle. Si une source disparaît pendant l'exécution d'une requête (incrémentielle), le processus sera naturellement bloqué (!).

En résumé, les techniques d'évaluation de requête présentées dans cette section ne répondent qu'à un ou quelques problèmes de la personnalisation dans les systèmes spécifiques. Certains entre eux reposent

| Référence | Spectre de la personnalisation | Mécanisme d'évaluation |
|---|---|---|
| TopK [CK97] | nombre de résultats | extension de SQL, opérateur *Stop* et règles d'optimisation |
| Disco [BT98] | préférence sur les résultats partiels | détection des sources indisponibles, génération de requêtes parachutes, optimisation sous contraintes |
| Skyline [BKS01] | contraintes floues pour la sélection des résultats | extension de SQL, opérateur *Skyline* et heuristiques pour l'optimisation |
| Preference SQL [KK02] | contraintes floues pour le tri de résultats et pour la sélection des résultats | règles d'optimisation |

Table 3.2: Techniques d'évaluation de requête pour la personnalisation

sur les (processus de) traitements spécifiques dont l'intégration dans d'autres systèmes pour d'autres contextes n'est pas chose aisée. Par ailleurs, plusieurs problèmes de l'évaluation personnalisée de requête n'ont pas été traités ou ont été traité à l'extérieur du processus de traitement de requête en construisant un système de personnalisation au dessus d'un SGBD existant[8].

Il serait intéressant de revisiter le processus de traitement de requêtes à l'égard des besoins de la personnalisation afin de fournir une approche systématique pour la personnalisation d'évaluateurs de requêtes. Une telle approche permettrait de répondre à un plus grand nombre de besoins de personnalisation en facilitant leur extension et leur adaptation dans les contextes spécifiques.

## 3.4 Adaptabilité dynamique : les techniques d'évaluation adaptative et interactive

Ces deux catégories des techniques, qu'elles soient pour l'évaluation adaptative ou interactive, visent à adapter *dynamiquement* l'exécution des requêtes à des changements provenant de l'environnement d'exécution ou de l'utilisateur. Elles sont considérées comme des mécanismes pour obtenir l'adaptabilité dynamique.

L'idée principale de l'évaluation adaptative est de ne plus considérer la stratégie d'exécution d'une requête générée par la phase d'optimisation comme étant toujours efficace et invariable. Les techniques d'évaluation adaptative ont pour but de rectifier les estimations erronées faites au moment de la compilation dues au manque de connaissances sur l'environnement d'exécution ou aux changements de ce

[8] [KK02, KI04, KI05] sont, entre autres, les systèmes de personnalisation construits au dessus des SGBD commerciaux. Etant donné que notre objectif est de personnaliser l'évaluateur de requêtes lui-même (cf. le chapitre 1), nous ne les avons pas présentés en détail dans cette section.

dernier durant une exécution longue de requête. Un système d'évaluation de requêtes est dit adaptatif s'il offre les trois caractéristiques suivantes : (i) il reçoit les informations de son environnement, (ii) il utilise ces informations pour déterminer son comportement, et (iii) ce processus est itératif pour générer une boucle de *feedbacks* entre l'environnement et le système [HFC+00]. L'évaluation adaptative de requêtes a fait l'objet de nombreux travaux que nous pouvons citer comme l'*optimisation paramétrée* [GW89, INSS92, CG94], l'*optimisation dynamique* [UFA98, IFF+99, BFMV00b, ILW04], les opérateurs adaptatifs [HH99, IFF+99, WA91, UF00], ou encore *Eddy*, une technique d'évaluation *dynamiquement* adaptative [AH00]. Malgré les différences de leurs mécanismes pour l'adaptation, tous ces travaux visent à améliorer la performance globale de l'exécution de requêtes sans aucune interaction avec l'utilisateur.

En ajoutant à ces systèmes les capacités (i) de construction de résultats partiels, (ii) d'être contrôlé par l'utilisateur et (iii) de transformation interactive par l'utilisateur, nous obtenons un système d'évaluation interactive. En effet, ces capacités ajoutées permettent de prendre en compte l'interaction avec l'utilisateur pendant l'évaluation de requête. Les résultats partiels sont considérés comme des retours (*feedback*) du système à l'utilisateur. Vu les premiers résultats (incomplets, partiels) l'utilisateur peut décider d'arrêter sa requête, de continuer à attendre d'autres résultats, ou encore d'affiner sa requête. A ce jour, cet aspect est encore très peu abordé dans le domaine de bases de données vu le nombre des travaux le concernant [Bon99, RRH99, RH02].

La suite de cette section présente une synthèse de ces techniques : l'optimisation paramétrée (la sous-section 3.4.1), l'optimisation dynamique (la sous-section 3.4.2), les opérateurs adaptatifs (la sous-section 3.4.3), les mécanismes de *Eddy* (la sous-section 3.4.4) et les interactions entre l'utilisateur et le système pendant l'évaluation de requêtes (la sous-section 3.4.5). Enfin, nous donnons une synthèse de l'ensemble des travaux existants sur les techniques pour l'adaptabilité dynamique (la sous-section 3.4.5).

### 3.4.1 Optimisation paramétrée

La paramétrage est un cas "simple" de l'évaluation adaptative. L'idée est de ne plus considérer l'*unicité* du plan optimal de la requête généré par l'optimiseur mais ses alternatives (i.e. des plans d'exécution) [GW89, INSS92]. Ces dernières correspondent aux plans optimaux dans différentes situations caractérisées par les valeurs possibles des paramètres non-déterminées au moment de la compilation. Lors de l'exécution, le système choisit, parmi les alternatives, celle la plus adaptée. Selon cette approche, l'optimisation pendant la phase d'exécution est plutôt aisée.

Pour cela, G. Graefe et al. ont proposé d'organiser les alternatives d'une requête sous forme de **plan dynamique de requête** (*Dynamic Query Plan*) [GW89]. Ce dernier se compose des opérateurs algébriques et d'un ou plusieurs opérateur(s) **choose-plan** qui est (sont) effectivement une "jonction" des alternatives. A l'exécution, cet opérateur choisit une alternative adaptée à la situation produite (i.e. les variables de requête, les ressources disponibles). La figure 3.8 montre un exemple du plan dynamique de la requête de jointure entre deux relations $\mathcal{R}$ et $\mathcal{S}$. L'utilisation de deux opérateurs *choose-plan* forme quatre alternatives qui se différencient par la méthode d'accès à la relation $\mathcal{R}$ (l'opérateur *choose-plan* dénoté `1`) et le rôle de ces deux relations dans la jointure (l'opérateur *choose-plan* dénoté `2`).

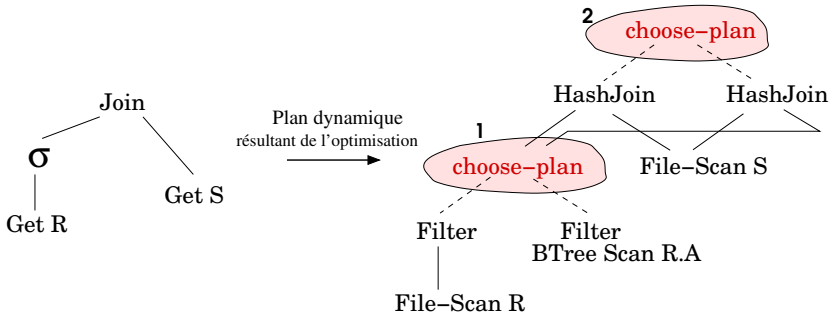



Figure 3.8: Exemple de plan d'exécution dynamique

L'efficacité de plans dynamiques dépendent de plusieurs facteurs : (i) le temps de leur construction, (ii) leur taille (i.e. le nombre des opérateurs *choose-plan* utilisés), (iii) le temps de prise de décision de l'opérateur *choose-plan*, et (iv) la robustesse de ces plans durant l'évaluation (i.e. la compilation et l'exécution). Pour la construction efficace des plans dynamiques, les auteurs proposent de considérer *l'incomparabilité de coût* à la compilation (*cost incomparability at compilation-time*) [CG94]. Plus précisément, les plans dans l'espace de recherche sont comparés entre eux par une fonction de coût et ceux qui sont incomparables (dus à la non connaissanca de valeurs de paramètres) sont reliés par l'opérateur *choose-plan*. En conséquence, un plan dynamique peut faire partie d'un plan plus grand. Le coût d'un plan dynamique est estimé à la somme du coût de l'opérateur *choose-plan* (i.e. le coût de la prise de décision) et le coût minimal des choix au sein d'une alternative.

A la différence de plans dynamiques où toutes les alternatives se relient entre elles dans un plan complexe, l'**optimisation paramétrique** (*Parametric Query Optimization*) considère un ensemble de plans générés lors de la compilation dont chacun est associé à une combinaison possible des valeurs des paramètres inconnus à la compilation [INSS92].

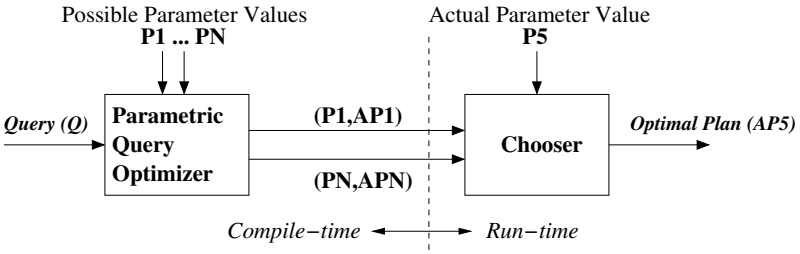



Figure 3.9: Optimisation paramétrique de requêtes

La figure 3.9 montre la vue globale de l'optimisation paramétrique. Le choix d'un plan optimal se fait en deux étapes : (i) l'optimisation paramétrique (*Parametric Query Optimizer*) susceptible de produire

un ensemble de plans ($AP_i$) associés à des valeurs possibles des paramètres ($P_i$) ; et (ii) la sélection (*Chooser*) chargée de choisir le plan à exécuter conformément à la valeur des paramètres déterminée lors de l'exécution.

Nous remarquons que selon cette approche, le résultat de l'optimisation (phase de compilation) est assez complexe. Dans [GW89, CG94], cette complexité dépend du nombre des opérateurs *choose-plan* du plan dynamique généré, lequel pourrait conduire à la construction d'un plan dynamique couvrant entièrement l'espace de recherche de la requête (!). L'évaluation de requête devient alors inefficace. Dans [INSS92, INSS97], le nombre de plans générés est le produit des $|P_i|$ où $|P_i|$ dénote la taille du domaine de $P_i$. Par ailleurs, dans tous les cas, l'adaptation n'est réalisée qu'au début de l'exécution de requête. Elles peuvent donc résoudre le problème de valeur inconnue de paramètres de requête, ou d'absence de certaines connaissances lors de la compilation, à condition que ces dernières soient visibles dès le début de l'exécution de requête. *Aucune* décision d'adaptation n'est prise en cours d'exécution.

### 3.4.2 Optimisation dynamique

Dans cette approche, l'optimisation peut se répéter afin de modifier la stratégie d'exécution en tout ou en partie selon les changements détectés pendant l'exécution de requêtes. Concrètement, ceux que l'on peut modifier sont l'allocation de ressources, les algorithmes des opérateurs, l'ordre d'exécution des opérateurs, et/ou les opérateurs eux-mêmes (i.e. la structure du plan de la requête). Dans les deux premiers cas, les modifications à réaliser sont plutôt locales, c'est-à-dire elles concernent des opérateurs individuels (e.g. le changement de la taille de tampon). Dans les deux derniers cas, ils nécessitent certains contrôles, ou synchronisations, sur l'ensemble de plan de requête afin d'assurer une cohérence dans son exécution. Dans cette section, nous nous intéressons aux deux derniers cas et présentons quelques travaux représentatifs de cette approche.

#### 3.4.2.1 Re-optimisation dynamique [KD98]

L'idée est de *collectionner les statistiques sur les données* pendant l'exécution d'une requête et de les utiliser pour ré-optimiser les parties de la requête qui ne sont pas encore exécutées. Pour cela, N. Kabra et al. proposent un nouvel opérateur, appelé **StatisticsCollector**, susceptible de "calculer" les informations concernant la cardinalité, la taille moyenne des n-uplet, la distribution des valeurs d'un attribut, etc. Cet opérateur est inséré dans le plan de requête comme d'autres opérateurs algébriques comme le montre la figure 3.10(a).

De façon générale, l'opérateur *StatisticsCollector* ne doit pas interrompte l'exécution d'autres opérateurs du plan de requête. Cela semble être avantageux mais elle rend, parfois, la collection des statistiques vaine car la totalité des opérateurs ont déjà commencés et aucune ré-optimisation ne aurait pu être réalisée. Dans le cas contraire, l'utilisation des opérateurs *StatisticsCollector*s brise les chaînes non-bloquantes. Le plan de requête est découpé en quelques sous-plans et leurs résultats, appelés résultats intermédiaires, sont matérialisés (cf. la figure 3.10(b)). Les statistiques calculées par les opérateurs *StatisticsCollector*s sont utilisées par la suite pour optimiser la partie non-évaluée de la requête en con-

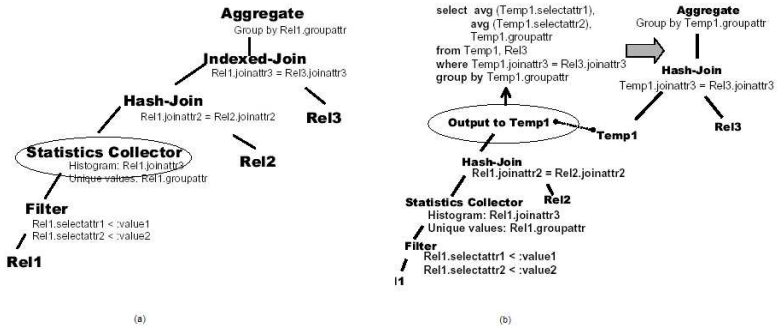



Figure 3.10: Plan de requête avec l'opérateur *StatisticsCollector*

sidérant les données matérialisées.

Notons que les opérateurs *StatisticsCollector*s sont insérés dans le plan d'une manière statique par une phase de post-optimisation lors de la compilation de la requête et que cette technique ne permet que de minimiser le temps total de l'évaluation de la requête.

#### 3.4.2.2 *Query Scrambling* [AFTU96, UFA98]

Rappelons que l'exécution d'une requête dans le modèle itérateur peut être assurée par un *seul* processus qui est susceptible d'ordonner *tous* les opérateurs du plan (cf. la section 2.3). En conséquence, l'exécution de la requête se bloque s'il y a un délai d'accès à une seule source. Cependant, plusieurs fragments du plan, notamment ceux concernant des différentes sources distribuées, peuvent être exécutés indépendamment les uns les autres. Partant de cette observation, L. Amsaleg et al. proposent d'en tirer profit afin de "masquer" les délais d'accès aux sources à distance [AFTU96]. Concrètement, les mécanismes proposés, appelés *Query Scrambling*, visent à choisir dynamiquement les fragments à exécuter : l'évaluation d'une requête est démarré d'une manière classique (i.e. l'optimisation puis l'exécution) mais lors que l'exécution est bloquée (dû au manque de données), le processus de re-ordonnancement est déclenché. Les opérateurs du plan sont divisés en deux files disjointes, l'une contient les opérateurs bloqués (*blocked operators*), i.e. les opérateurs ancêtres des relations (sources) indisponibles, et l'autre contient les opérateurs non-bloqués (*runnable operators*). L'ordre des opérateurs dans chacune de ces deux files se conforme à l'ordre dans lequel leur évaluation est initiée selon le modèle itérateur. Le système analyse la file des opérateurs non-bloqués pour trouver la **sous-requête non-bloquée maximale** (*maximal runnable subtree*). Cette dernière contient les opérateurs formant un sous-arbre dont la racine est le premier opérateur non-bloqué descendant d'un opérateur bloqué. Le choix de sous-plans à exécuter se fait en traversant le plan de la requête de gauche à droite. Lors que tous les fragments non-bloqués sont calculés, l'algorithme procède à une restructuration du plan en ajoutant de nouveaux opérateurs, supprimant d'autres et/ou les re-arrangeant. L'algorithme est résumé en deux phases comme suit :

**Phase 1 (Ré-ordonnancement)** : l'objectif de cette phase est de réordonner dynamiquement l'exécution de la requête lors de la détection de délais d'accès aux données. Dans cette phase, la structure du plan n'est pas changée, excepté quelques opérateurs de matérialisation ajoutés pour stocker les résultats de l'exécution de certains fragments du plan de la requête.

**Phase 2 (Synthèse d'opérateurs)** : pendant cette phase, de nouveaux opérateurs, notamment les jointures inexistantes dans le plan d'origine, peuvent être ajoutés. Dans cette phase, l'ajout de nouveaux opérateurs, la suppression des opérateurs du plan d'origine ou le réarrangement des opérateurs du plan font que la structure du plan est modifié "radicalement".

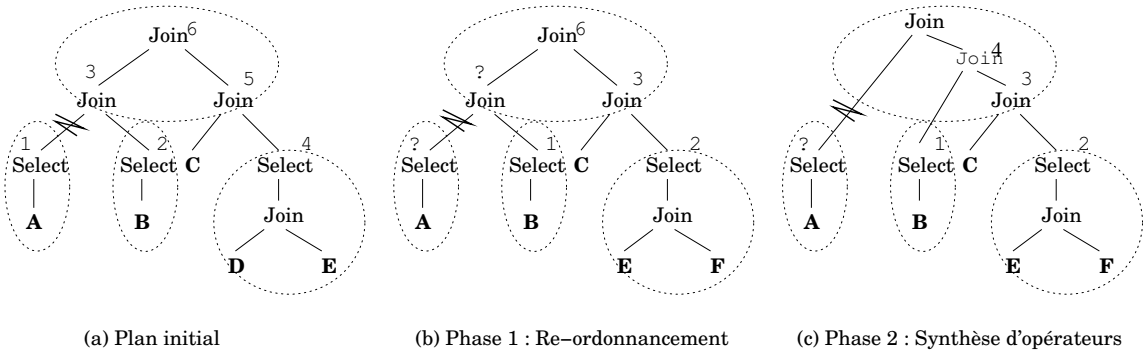



Figure 3.11: Optimisation en présence de délais de réseaux

La figure 3.11 montre l'exemple de la requête de jointure entre les relations $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$. Commençant par le plan optimisé initial (a), les étiquettes (de `1` à `6`) représentent l'ordre des opérateurs à exécuter selon le modèle itérateur. Au début de l'exécution, l'accès à la source $\mathcal{A}$ est bloqué. Par conséquent, tous les opérateurs ascendants de $\mathcal{A}$ (i.e. le *Select*, le *Join*(3) et le *Join*(6)) sont également bloqués. Le processus de ré-ordonnancement est déclenché. Les deux files d'attente, l'une pour les opérateurs bloqués (`1`, `3`) et l'autre les opérateurs non-bloqués (`2`, `4`, `5`) sont créées[9]. L'analyse de la file contenant les opérateurs non-bloqués permet de déterminer un nouvel ordre d'exécution des opérateurs comme le montre la figure 3.11(b). Les sous-requêtes, dont les racines sont respectivement les opérateurs étiquetés (`1`) et (`3`), sont exécutées et leur résultat est matérialisé. Supposons qu'après avoir effectué tous les ré-ordonnancements sur le plan d'exécution initial, il est toujours impossible d'accéder à la relation $\mathcal{A}$. La deuxième phase est déclenchée et le nouveau plan d'exécution est généré (voir la figure 3.11(c)). L'exécution de la requête est continue. Lorsque l'accès à $\mathcal{A}$ se rétablit, il ne reste qu'à réaliser la jointure de $\mathcal{A}$ et du résultat obtenu antérieurement, dans le but de retourner le résultat final et complet à l'utilisateur.

[9]Notons que les éléments de ces deux listes sont déjà ordonnés selon leur priorité. Par exemple, l'opérateur dénoté (`1`) doit être exécuté avant l'opérateur (`3`).

Notons que la matérialisation de résultats et l'exploitation de ces résultats matérialisés sont souvent très coûteuses. Cela s'explique par le coût d'écriture et de lecture des résultats matérialisés sur le disque. Afin d'améliorer l'efficacité de ces mécanismes, les auteurs ont introduit dans leur travaux ultérieurs une stratégie d'ordonnancement basée sur le coût [UFA98]. L'idée est de choisir les fragments dont le résultat est matérialisé selon le *gain* qu'apporte leur matérialisation. Etant donné que le coût d'exécution d'un sous-plan est *P*, que le coût de matérialisation du sous-plan est *MW* et que le coût de lecture de données sur disque est *MR*, le gain est calculé par la formule $\frac{P-MW}{P+MR}$.

Que ce soit l'ordonnancement basé sur les heuristiques ou sur le coût, nous remarquons que cette technique considère principalement le délai initial de l'accès aux données (i.e. le délai apparu au début de l'exécution de requête). Si l'accès à $\mathcal{A}$ est bloqué après le commencement de l'exécution de *Select*(1) et *Join*(3), aucun changement de plan ne peut être effectué. L'exécution peut passer de la phase (a) à (b) pendant le blocage de $\mathcal{A}$ mais il est impossible de passer à la phase (c) car la jointure des relations $\mathcal{A}$ et $\mathcal{B}$ est déjà commencée. En outre, aucun résultat ne peut être retourné à l'utilisateur pendant le blocage de l'accès à $\mathcal{A}$

#### 3.4.2.3 Ordonnancement dynamique [BFMV00b]

A la différence des mécanismes présentés ci-dessus où les tâches de ré-ordonnancement et de synthèse d'opérateurs se déclenchent pour réagir à des délais détectés lors de l'exécution, L. Bouganim et al. supposent que *les délais vont avoir lieu* et proposent d'étendre l'optimiseur pour en tenir compte [BFMV00b]. Leur proposition consiste principalement à ré-ordonner l'exécution des fragments du plan de la requête. Etant donné un plan de requête, les arcs représentant les flux de données entre les nœuds d'opérateurs peuvent être *bloquants* ou *non-bloquants*. **Un arc est bloquant si les données doivent être produites en totalité avant d'être consommées**. Plus précisément, un opérateur ayant une entrée bloquante doit attendre que les données de cette entrée soient entièrement matérialisées avant de commencer les calculs. En revanche, **les arcs non-bloquants signifient que les données peuvent être consommées élément par élément**. Par conséquent, l'opérateur peut commencer les calculs lors d'arrivée d'un élément. La suite des arcs non-bloquants forme un **fragment non-bloquant** dont l'exécution est indépendante de l'exécution d'autres fragments. Le plan d'une requête se compose donc des fragments bloquants et non-bloquants comme illustré dans la figure 3.12(a).

L'extension de l'optimiseur a pour but de déterminer les fragments bloquants, non-bloquants ainsi que leur priorité. L'ordre d'exécution de ces fragments respecte cette priorité. Dans un premier temps, le moteur d'exécution exécute le fragment dont la priorité est la plus élevée. Une fois l'accès aux données bloqué, un autre fragment ayant la priorité moins élevée sera choisi pour son exécution. Il est à noter qu'un fragment est exécuté *si et seulement si* aucun autre fragment ayant la priorité plus élevée ne peut être exécuté et la priorité des fragments est déterminé à l'avance.

Les techniques proposées sont intégrées dans une architecture générale comme le montre dans la figure 3.12(b). Le ré-ordonnancement présenté précédemment concerne essentiellement le module DQS. Les techniques de ré-optimisation [KD98, UFA98] peuvent être intégrées dans le module DQO. Dans ce travail, les trois événements produits pendant l'exécution de requête sont *TimeOut*, *EndOfQF* et *Rat-*

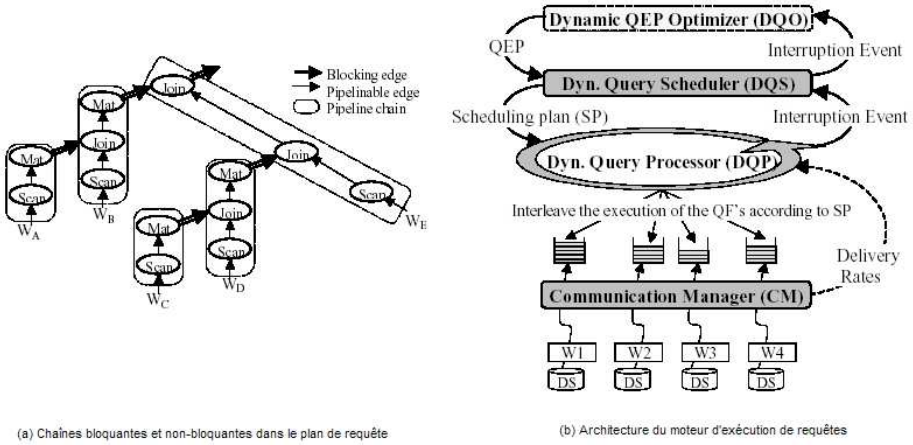



Figure 3.12: Ordonnancement dynamique [BFMV00b]

*eChange* [BFMV00b] et ce sont les deux modules DQS et DQO qui sont chargés de procéder à des adaptations nécessaires. L'architecture proposée semble être générale et les auteurs déclarent que la spécification de DQS et DQO requiert des descriptions de (i) leur stratégies, (ii) les événements auxquels ils réagissent et (iii)les réactions correspondantes [BFMV00a]. Mais comment les décrit-on ? L'ajout des nouveaux événements, nouvelles réactions, est-il facile ? Pour cela, faut-il spécifier et implémenter ces modules en partant de zéro ?

#### 3.4.2.4 Tukwila [IFF+99]

Proposé par Z. Ives et al. [IFF+99, ILW+00], Tukwila peut être considéré comme la première proposition qui aborde l'aspect architectural du système d'évaluation adaptative de requêtes. Il considère un couplage "étroit" entre l'optimisation et l'exécution. La figure 3.13 montre l'architecture générale de Tukwila où le module Event Handler est susceptible de répondre à des événements produits pendant l'exécution de requête et de (ré-)invoquer le module d'optimisation pour réaliser l'adaptation.

Une idée importante à retenir dans Tukwila est l'utilisation de règles comme étant le moyen pour définir le comportement du système. Une règle est de la forme "**Quand** *Evénement* **Si** *Condition* **Alors** *Action*" (E-C-A). Effectivement, une règle dans Tukwila est représenté par un quintuple `(event, condition, actions, owner, is_active)` où `owner` est l'opérateur que *la règle contrôle* ou *surveille*. Seules les règles actives (i.e. les règles dont l'état est actif) peuvent être déclenchées suite à un événement [Ive02]. Les règles sont associées aux opérateurs du plan de requête. Ici, nous trouvons que la surveillance et l'adaptation dans Tukwila sont relativement liées (!). Plus précisément, les mécanismes de surveillance et de prise de décision d'adaptation semblent d' "être groupés" en ce que les auteurs définissent comme règles.

L'architecture proposée permet de réaliser les différents mécanismes d'évaluation adaptative de re-

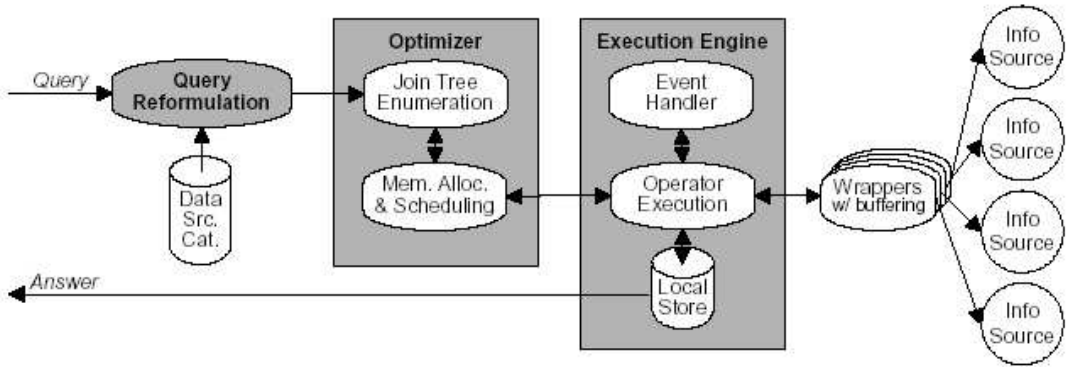



Figure 3.13: Architecture de Tukwila [Ive02]

quêtes, tels que ceux présentés dans les sections précédentes. Dans Tukwila, la surveillance de l'exécution d'une requête concerne principalement la *cardinalité* et l'*utilisation de CPU* de chacun des opérateurs du plan par l'extension des implémentations des opérateurs. Les événements considérés sont `state_becomes(open | close | error)`,`end_of_fragment`,`wrapper_timeout(n)`, `out_of_memory`,`every_n_tuple(n)`,`every_n_ms(n)`. Les actions concernent la ré-optimisation (pour le cas [KD98]), le ré-ordonnancement (pour le cas [UFA98]), le choix de la prochaine étape de l'exécution (pour le cas [CG94]), etc. Ces actions sont définies pour répondre à des problèmes spécifiques tels que la limite de mémoire, le délai d'accès aux sources, etc. Il serait donc difficile de les réutiliser (en tout ou en partie) pour répondre à d'autres situations produites pendant l'exécution d'une requête (!).

Par ailleurs, nous remarquons que la plupart des techniques d'optimisation dynamique présentées dans cette section ont pour but de "masquer" les délais d'accès aux données, le manque de mémoire, etc. en choisissant dynamiquement les sous-requêtes à exécuter. Elles limitent le temps d'attente inutile et permettent donc de réduire le temps de production des résultats en totalité mais ne garantissent pas le retour des premiers résultats dans un meilleur délai. Ce dernier objectif (i.e. minimiser le temps pour obtenir les premiers résultats) est le propos des opérateurs adaptatifs que nous présentons dans la section suivante.

### 3.4.3 Opérateurs adaptatifs

Ayant le même objectif que les techniques présentées précédemment, les opérateurs adaptatifs, appelés également opérateurs non-bloquants, adaptent leur fonctionnement aux délais d'accès aux données et/ou l'insuffisance de mémoire dans le but d'accélérer les calculs des données et la production des résultats. Toutefois, les adaptations se réalisent au niveau de l'algorithme de chacun des opérateurs sans aucune considération d'autres opérateurs du plan de requête. Dans cette section, nous nous focalisons seulement sur l'opérateur de jointure en présentant quelques algorithmes permettant une ou plusieurs adaptation(s).

Le choix de cet opérateur s'explique par la complexité de la jointure dont l'exécution se fait souvent en plusieurs étapes en lisant les données de plusieurs entrées. En effet, l'opérateur de jointure a été le sujet de plusieurs travaux que nous pouvons citer *RippleJoin* (RJ) [HH99], *Symetric Hash Join* (SHJ) [WA91], *Double Pipeline Hash Join* (DPHJ) [IFF+99], *XJoin* [UF00].

Partant de l'observation que l'efficacité de la jointure par boucle imbriquée (*Nested-Loop-Join*) dépend fortement du choix de la relation externe et interne, J. Hellersteins et al. proposent d'interchanger le rôle de deux relations d'entrée [HH99]. L'algorithme proposé, nommé **RippleJoin**, repose sur la définition du **point symétrique**. Un point symétrique est le point où le rôle des relations interne et externe peut être interchangeable. Intuitivement, la fin de chaque boucle de la relation interne peut être un point symétrique. En arrivant sur ce point, l'algorithme analyse la "situation" (e.g. le temps d'accès à la relation interne et externe, la cardinalité de la relation interne par rapport à la cardinalité estimée de la relation externe) afin de décider si ces deux relations changent leur rôle pour que la production des résultats soit plus avantageuse.

D'autres opérateurs se basent sur l'algorithme de jointure par hachage. Rappelons que la jointure par hachage classique est effectuée en deux phases séquentielles : (i) la première phase susceptible de construire entièrement la table de hachage de la relation interne et (ii) la deuxième phase de test (*probe*) qui applique la même fonction de hachage à chaque tuple lu de la relation externe et le teste avec le paquet correspondant afin de produire le résultat (cf. la section 2.3). En conséquence, si l'accès à la relation interne est bloqué ou s'il n'y a pas assez de mémoire pour stocker sa table de hachage, aucun résultat ne peut être produit. Plusieurs variations de cet algorithme ont été proposées pour améliorer la production des résultats.

A. Wilschut et al. proposent une extension de la jointure par hachage, nommé **jointure par hachage symétrique**, par l'utilisation de la deuxième table de hachage pour la deuxième relation [WA91]. La jointure fonctionne de façon symétrique pour la relation interne et externe, et deux tables de hachage sont construites au fur et à mesure de l'exécution. Chaque n-uplet d'une relation, $\mathcal{R}$ par exemple, est haché et testé avec la table de hachage de S existante à un instant donné. Il est en suite inséré, quelque soit le résultat du test, dans la table de hachage de $\mathcal{R}$. De la même manière, un n-uplet de $\mathcal{S}$ est testé avec la table de hachage de $\mathcal{R}$ puis inséré dans la table de hachage de $\mathcal{S}$. Cet algorithme permet de produire les premiers n-uplets résultat intermédiairement. Toutefois, il nécessite une quantité de mémoire suffisamment large pour stocker les tables de hachage de toutes les relations. Malheureusement, cela n'est pas toujours le cas. Afin de résoudre le problème de limitation de mémoire, les auteurs de *DPHJ* (**Double pipeline hash join**) [IFF+99] et **XJoin** [UF00] proposent d'utiliser la mémoire secondaire (i.e. le disque) pour stocker une partie des tables de hachage. La jointure se fait en plusieurs étapes pour garantir la complétude de son résultat.

Remarquons que tous les opérateurs adaptatifs ont pour but de produire les résultats dans un meilleur délais dans le cas d'accès lent aux données et/ou l'insuffisance de mémoire. S'il manque une ou plusieurs entrée(s), l'opérateur est bien sûr bloqué (!).

### 3.4.4 *Eddy*, un nouveau paradigme d'optimisation

A la différence de toutes les techniques décrites précédemment, R. Avnur et al. proposent *Eddy* [AH00] une architecture innovante pour l'évaluation adaptative de requêtes. *Eddy* peut aussi être considéré comme un opérateur qui "achemine" (*route*) des n-uplets entre des opérateurs algébriques appelés *modules*. L'ordre des opérateurs à exécuter est défini *à la volée* et pour chaque n-uplet. Plus précisément, après l'arrivée d'un n-uplet, *Eddy* décide le module qui va le traiter. Chaque n-uplet est associé un vecteur appelé *Ready* et un vecteur appelé *Done*. Le vecteur *Ready* indique les *modules* pouvant traiter ce n-uplet et le vecteur *Done* indique les modules qui l'ont traité. En utilisant la valeur de ces deux vecteurs, *Eddy* décide d'envoyer tel n-uplet à tel module.

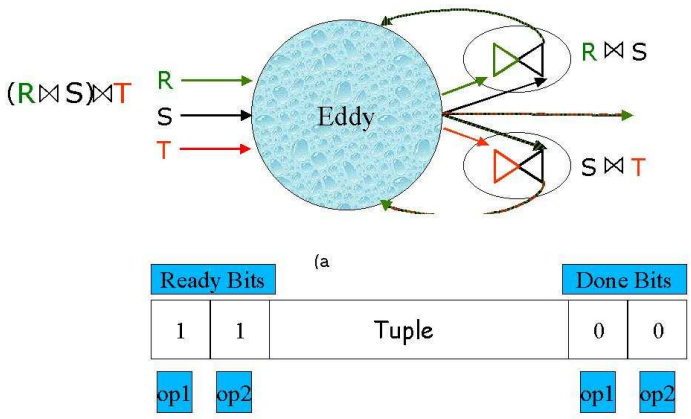



Figure 3.14: Architecture de *Eddy*

La figure 3.14 illustre *Eddy* pour la jointure entre les trois relations $\mathcal{R}$, $\mathcal{S}$ et $\mathcal{T}$. Les n-uplets traités par *Eddy* ont la structure présentée dans la figure 3.14(b) : *op1* et *op2* représentent respectivement la jointure de $\mathcal{R}$ et $\mathcal{S}$, et la jointure de $\mathcal{S}$ et $\mathcal{T}$. La figure 3.14 montre la valeur d'un n-uplet provenant de $\mathcal{S}$. Ce n-uplet est donc prêt à être traité par les deux opérateurs de jointure.

Nous trouvons que l'évaluation des requêtes à la *Eddy* est très dynamique. L'ordre des opérateurs à exécuter dépend du flux de données arrivées et peut être déterminé pour chaque n-uplet. Pourtant dans un environnement peu dynamique, i.e. la bande passante, les statistiques sont assez stables, les auteurs de *Eddy* eux-mêmes ont constaté que cette approche est moins efficace que les approches classiques. Cela est expliqué par le fait que la tâche d'ordonnancement doit se répéter à chaque arrivée de n-uplet.

### 3.4.5 Interaction pendant l'évaluation de requête

Les derniers mécanismes pour réaliser l'adaptabilité dynamique que nous examinons concernent l'interaction entre l'application (ou l'utilisateur) et le système pour l'évaluation interactive de requêtes. L'objectif est d'impliquer l'utilisateur dans le processus d'évaluation afin que ce dernier puisse contrôler l'exécution

de sa requête et l'affiner s'il le souhaite. Pendant une évaluation interactive, le système retourne les résultats incomplets (partiels) pour que l'utilisateur soit au courant de l'avancement de l'évaluation sa requête.

#### 3.4.5.1 Construction des résultats partiels avec *Eddy*

Pour cet objectif, V. Raman et al. proposent d'étendre les mécanismes de *Eddy* pour acheminer les résultats incomplets vers l'utilisateur (i.e. application cliente) lorsque c'est possible [RH02]. Les auteurs supposent une application cliente capable de composer les résultats incomplets d'une manière continue par l'emploi d'une *table de hachage*. L'application est alors appelé **Hash Client**. Par ailleurs, on distingue les *résultats partiels* et les *résultats intermédiaires*. Comme illustré dans la figure 3.15, les résultats partiels sont ceux que retournent le serveur pendant que les résultats intermédiaires sont les résultats incomplets affichés du côté client.

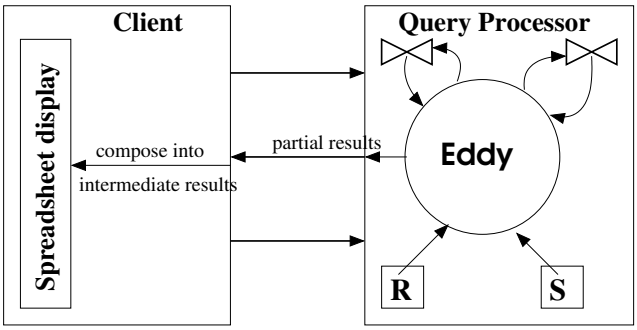



Figure 3.15: Architecture pour la génération des résultats partiels

Selon leur définition, **les n-uplets du résultat partiel doivent contenir tous les attributs clés**. Les attributs clés sont identifiés à partir des schémas des relations et des attributs des jointures. Dans le cas d'un `GROUP-BY`, les attributs du `GROUP-BY` appartiennent eux-mêmes aux résultats. Cette caractéristique permet de produire le maximum de n-uplets du résultat partiel et de garantir la simplicité de l'application cliente. En effet, ces attributs clés sont aussi utilisés par la table de hachage de l'application cliente, permettant de composer continuellement les résultats du côté client. **Les résultats intermédiaires**, affichés par l'application cliente, **sont les n-uplets résultats incomplets mais il existe la correspondance 1-1 avec n-uplets du résultat final**. Cette contrainte a pour but de garantir que chaque n-uplet des résultats intermédiaires affiché par l'application cliente participera à *un et un seul* n-uplet du résultat complet final. Les n-uplets de résultats partiels peuvent arriver du processus de traitement de requêtes d'une façon aléatoire, selon la vitesse d'accès aux sources de données et l'ordre de l'exécution des opérateurs.

L'application cliente fonctionne comme suit : les attributs clés sont identifiés. Le `Hash client` maintient le résultat intermédiaire dans une table de hachage contenant les attributs clés en mémoire

principale. La fonction de hachage est appliquée pour chaque n-uplet partiel de résultat afin de déterminer quel n-uplet partiel de résultat est ajouté à la table de hachage comme nouveau n-uplet dans le résultat intermédiaire. Si une correspondance (*match*) existe, chacune des valeurs des attributs dans le n-uplet partiel de résultat est employée pour compléter (ou mettre à jour, dans le cas des attributs d'agrégation) les attributs correspondants dans le n-uplet intermédiaire du résultat. De côté du serveur, *Eddy* retourne les données au client dès que ces dernières satisfont à la condition des résultats partiels, i.e. contenant tous les attributs clés.

#### 3.4.5.2 Interaction d'utilisateur

Dans le système que nous venons de décrire, l'utilisateur peut intervenir pour raffiner sa requête au vu des premiers résultats incomplets. L'interaction se fait par l'utilisation d'une interface simple permettant exprimer trois types de préférences : (i) la priorité sur quelques attributs particuliers du résultat (dénoté $\mathcal{CP}$s, *Column Priorities*), (ii) la priorité sur les n-uplet du résultat satisfaits les prédicats particuliers (dénoté $\mathcal{RP}$s, *Row Priorities*), et (iii) leur combinaison, i.e. quelques attributs de quelques n-uplet particuliers (dénoté $\mathcal{RCP}$s, *Row Column Priorities*). En basant sur ces priorités, le système détermine le degré de bénéfice de chaque "cellule" (calculé par le produit de $\mathcal{CP}$,$\mathcal{RP}$ et $\mathcal{RCP}$). Le système décide de l'ordre des opérateurs à exécuter ainsi que de l'ordre des n-uplets à traiter afin que les résultats intermédiaires produits soient le plus bénéfiques pour l'utilisateur [RRH99, RH02].

Notons que l'affinement de requête se limite à modifier les priorités des attributs et n-uplet du résultat, i.e. la préférence sur les données.

**Interaction dans DISCO [Bon99]** L'interaction de l'utilisateur est considérée d'une manière simple dans les travaux de P. Bonnet et al. que nous avons présentés dans la section 3.3.2. A la réception de la réponse partielle, l'utilisateur peut intervenir pour arrêter l'exécution de requête, pour formuler les requêtes parachutes dans le but de récupérer certaines données disponibles, ou tout simplement pour accepter de continuer attendre l'exécution de requête incrémentielle pour récupérer le résultat final complet. Ces "interactions" peuvent se répéter tant que la requête initiale n'est pas entièrement résolue mais c'est la seule interaction utilisateur autorisée dans DISCO.

### 3.4.6 Discussion

Dans cette section, nous avons présenté les différentes techniques d'évaluation adaptative et interactive comme étant les mécanismes pour réaliser l'adaptabilité dynamique dans les systèmes d'évaluation de requêtes. Bien que toutes ces techniques aient pour but d'adapter l'évaluation de requêtes à des changements ayant lieu pendant l'exécution, elles se différencient en plusieurs points : (i) le sujet de l'adaptation, (ii) l'origine de l'adaptation, (iii) la fréquence de l'adaptation, (iv) la réalisation de l'adaptation et (v) le gain de l'adaptation. Elles permettent de répondre à un ou quelques besoins de l'adaptation, mais pas à tous. La table 3.3 résume les propriétés des techniques présentées tout au long de cette section.

| Référence | Sujet | Origine | Fréquence | Réalisation | Gain sur |
|---|---|---|---|---|---|
| Plan dynamique [GW89, CG94] | remplacement des algos., ordre des opers., | paramètre de requête, propriété physique inconnu | par lot (*batch*)[a] | algorithme d'optimisation (considérant *Choose-Plan*) | temps total |
| Optimisation paramétrique [INSS97] | plan (complet) | paramètre de requête, ressources disponibles | par lot (*batch*)[a] | algorithme d'optimisation (générant l'ensemble de plans) | temps total |
| *Query Scrambling* [UFA98] | ordonnancement, ordre des opers. | délais | inter-oper. | algorithme d'adaptation | temps total |
| *Mid-query re-optimization* [KD98] | (re-optimisation d') une partie de requête | statistiques | inter-oper. | algorithme d'optimisation (considérant *StatisticsCollector*) | temps total |
| Ordonnancement dynamique [BFMV00b] | ordonnancement | délais et mémoire | inter-oper. | algorithme d'optimisation (considérant le parallélisme) | temps total |
| Tukwila [ILW+00] | ordre des opérateurs, algo. | tous | inter- & intra-opers | système | temps[b] |
| RippleJoin [HH99] | algo. | délais, taille d'entrées | intra-oper. | opérateur | temps de 1ère réponse |
| XJoin [UF00] | algo. | délais, mémoire | intra-oper. | opérateur | temps[b] |
| Eddy [AH00] | ordre des opers. | statistiques | par n-uplet | algorithme | temps de 1ère réponse |
| Juggle [RRH99] | n-uplet | préférence | intra-oper. | oper. *Juggle* | temps de 1ère réponse |
| JuggleEddy [RH02] | ordre des opers. et n-uplets | préférence, statistiques | par n-uplet | algorithme | temps de 1ère réponse |

Table 3.3: Propriétés des techniques d'évaluation adaptative et interactive de requêtes

[a] L'adaptation est faite une seule fois au début de l'exécution.
[b] Gain sur temps de 1ère réponse ainsi que temps total.

D'une manière générale, l'adaptation dynamique se compose en trois étapes, à savoir le déclenchement, la décision et la réalisation (cf. la section 3.1). Pourtant, ces étapes ne sont rarement séparées et abordées indépendamment les unes des autres dans les travaux existants. En effet, ces différentes étapes sont souvent inclues dans les algorithmes (d'optimisation) spécifiques conçus pour répondre à des besoins spécifiques. Par conséquent, l'utilisation et/ou l'adaptation de ces techniques en tous ou une partie pour les systèmes différents, pour les contextes différents n'est pas facile. Le besoin de fournir un cadre uniforme pour faciliter l'exploitation de ces mécanismes est abordé dans quelques travaux récents [BFMV00b, ILW$^+$00]. Bien que les auteurs de ces travaux aient montré comment les mécanismes proposés antérieurement sont intégrés dans leur architecture, la définition des nouvelles techniques d'adaptation ainsi que la réutilisation des techniques proposées en tout ou en partie restent relativement flous.

Pour le développement des systèmes répondant à différents besoins, il est nécessaire de séparer ces trois étapes de l'adaptation [GPSF04]. Séparer, afin de rassembler d'une manière flexible en vue des besoins des applications, est un moyen d'offrir l'adaptation dans l'évaluation adaptative et interactive de requêtes. Une telle séparation faciliterait la définition des nouvelles techniques d'adaptation en mieux réutilisant les mécanismes déjà définis et implémentés.

## 3.5 Conclusion du chapitre

Dans ce chapitre, nous avons présenté l'adaptabilité dans les travaux sur l'évaluation de requêtes. Nous avons commencé par la définition des trois niveaux de l'adaptabilité que sont l'adaptabilité statique, la personnalisation et l'adaptabilité dynamique. Ensuite, nous avons analysé les travaux existants comme étant les mécanismes pour réaliser ces différentes formes de l'adaptabilité. L'analyse confirme notre première observation de l'absence de la "continuité" de trois niveaux de l'adaptabilité (cf. la section 3.1).

L'approche des systèmes extensibles permet, d'une manière ou d'une autre, l'adaptabilité statique mais n'offre aucun support pour la personnalisation et l'adaptabilité dynamique. A contrario, la plupart des techniques d'évaluation avancées répondant à un ou quelques besoins de la personnalisation et/ou de l'adaptabilité dynamique sont conçues et implémentées pour les systèmes spécifiques ayant des caractéristiques spécifiques. L'utilisation de ces techniques dans d'autres systèmes, d'autres contextes n'est pas chose aisée.

L'augmentation des besoins de développement des applications ayant la capacité d'interrogation de données dans l'environnement distribué et autonome implique qu'il est nécessaire de disposer des outils capables de fournir une ou plusieurs fonctionnalités décrites ci-dessus qui de plus être capable d'adapter et de s'adapter aux besoins de l'application et à son environnement d'exécution. Ces outils facilitent le développement de nouvelles applications ayant de nouveaux besoins de traitement de requêtes.

La construction d'un tel outil pourrait s'inspirer des travaux de génération de l'optimisation et/ou de SGBD extensibles (i.e. ceux orientent l'architecture), lesquels permettent certains niveaux d'extension de différents aspects de l'optimisation statique, à savoir l'espace de recherche et la stratégie de recherche. D'une part, il faut y ajouter les aspects d'évaluation personnalisée, adaptative et interactive de requêtes

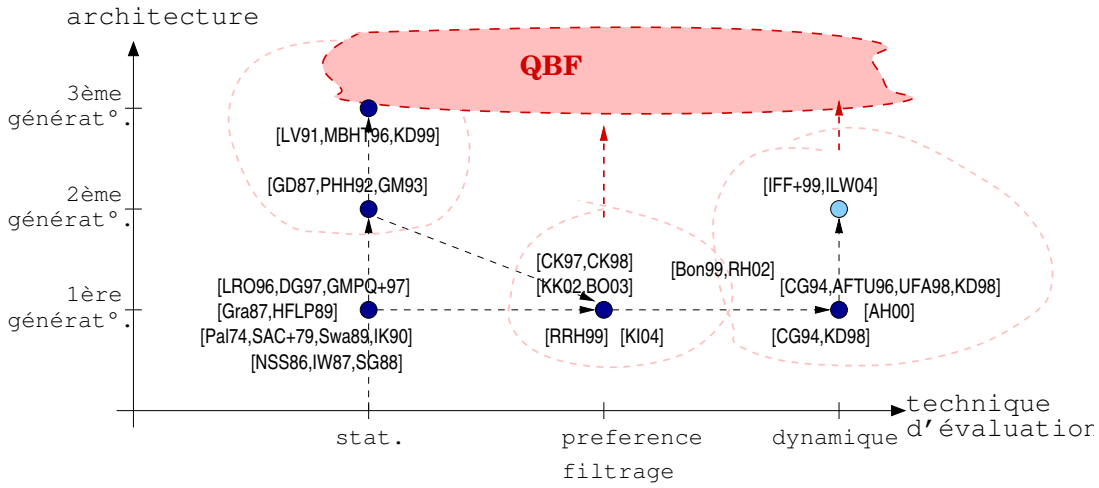



Figure 3.16: Vers l'approche QBF

(proposées dans les travaux axés techniques d'évaluation de requêtes), d'autre part, nous pensons qu'il est nécessaire de fournir une conception de ces aspects de manière fine afin de faciliter leur extension et d'augmenter la réutilisation de code. Le travail de cette thèse poursuit cette piste en essayant de faire approcher ces deux directions de travail comme illustré dans la figure 3.16.

# CHAPITRE 4

# QBF, un canevas d'évaluation de requêtes

Ce chapitre présente notre proposition nommée QBF (**Q***uery* **B***roker* **F***ramework*), un canevas pour la construction d'évaluateurs adaptables de requêtes.

## 4.1 Principe de conception

QBF consiste en *la séparation et l'abstraction des fonctionnalités* d'un évaluateur de requêtes. Il s'agit d'un ensemble d'interfaces dont chacune représente une fonctionnalité conçue comme un composant[1]. Grâce à cette abstraction, l'interdépendance entre ces fonctionnalités peut être exprimée *complètement* en terme d'opérations fournies par les composants. En conséquence, les changements dans l'implémentation d'un composant ne requièrent aucune modification d'autres composants. Ainsi, la réutilisation de ces composants est davantage possible. Par ailleurs, la séparation de ces fonctionnalités permet de les composer lors de la construction d'un évaluateur, voire pendant son fonctionnement, selon les besoins de l'application et/ou de l'utilisateur.

La figure 4.1 présente l'architecture globale de QBF qui se compose d'un gestionnaire de requêtes (QueryManager), d'un gestionnaire de plans de requête (PlanManager), d'un gestionnaire de contextes de requête (ContextManager), d'un gestionnaire de tampons (BufferManager), d'un surveillant (Monitor) et d'un gestionnaire de règles (RuleManager)[2].

Le QueryManager gère une liste de requêtes et coordonne d'autres composants pour les évaluer. Lors de la réception d'une requête, le QueryManager la décompose en deux parties : une partie correspond au contexte et l'autre partie correspond au plan de requête, ces deux parties étant envoyées

[1]Un composant est un objet (ou un ensemble d'objets) qui exporte des attributs, des propriétés ou méthodes (appelés également opérations), qui est capable d'être auto-description, qui peut être configuré et assemblé avec d'autres composants, qui peut réagir à des conditions de son environnement d'exécution[Riv04].

[2]Dorénavant, nous utilisons le nom en anglais des composants afin de faciliter la référence à leur spécification.

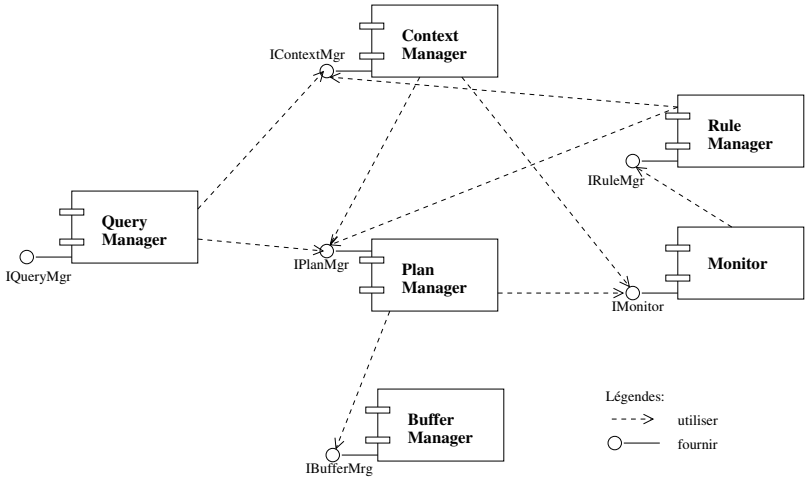



Figure 4.1: Architecture de QBF

respectivement au ContextManager et au PlanManager. Le ContextManager gère la partie de contexte correspondant à l'ensemble des contraintes spécifiques de l'évaluation. Il fournit également les outils permettant la prise en compte du contexte pendant l'évaluation de requête. Le PlanManager gère la liste des plans et fournit les mécanismes d'évaluation, notamment ceux de l'optimisation. Il couvre les trois aspects principaux de l'optimisation, à savoir la génération de l'espace de recherche, l'estimation de coût et la stratégie de recherche. Le BufferManager fournit les outils pour le stockage temporaire des données en cours de traitement. Il permet de créer et gérer des tampons (*buffers*) ayant différentes capacités de stockage et/ou différentes stratégies de remplacement. Les deux composants Monitor et RuleManager permettent d'assurer une évaluation dynamique de requête où la stratégie d'évaluation peut s'adapter, en cours d'exécution, à des changements dans l'environnement d'exécution, e.g. propriétés des sources participantes, réseaux ou clients. Le Monitor est susceptible de surveiller l'exécution des requêtes et l'environnement d'exécution afin de détecter les changements (i.e. conditions *non-souhaitées*) qui sont à l'origine du déclenchement des adaptations étant définies comme des règles gérées par le RuleManager.

Par son principe d'abstraction des fonctionnalités, QBF n'est pas conçu pour être utilisé directement par les utilisateurs finals. Il s'agit d'une approche de développement et de déploiement d'évaluateur en plusieurs étapes comme le montre la figure 4.2. La **spécification de QBF** consiste en un ensemble d'interfaces représentant les fonctionnalités de base de l'évaluateur de requête. Ces fonctionnalités sont implémentées complètement (par les classes concrètes) ou partiellement (par les classes abstraites) par les *programmeurs de QBF* lors de son **implémentation**. Un évaluateur (ou application) basé sur QBF est appelé une **instance de QBF** (ou *Query Broker*). Pour le construire, les *programmeurs d'évaluateur* (appelés également *utilisateurs de QBF*) peuvent utiliser les classes fournies dans l'implémentation de QBF et/ou ajouter d'autres implémentations afin de répondre à des besoins de leur évaluateur.

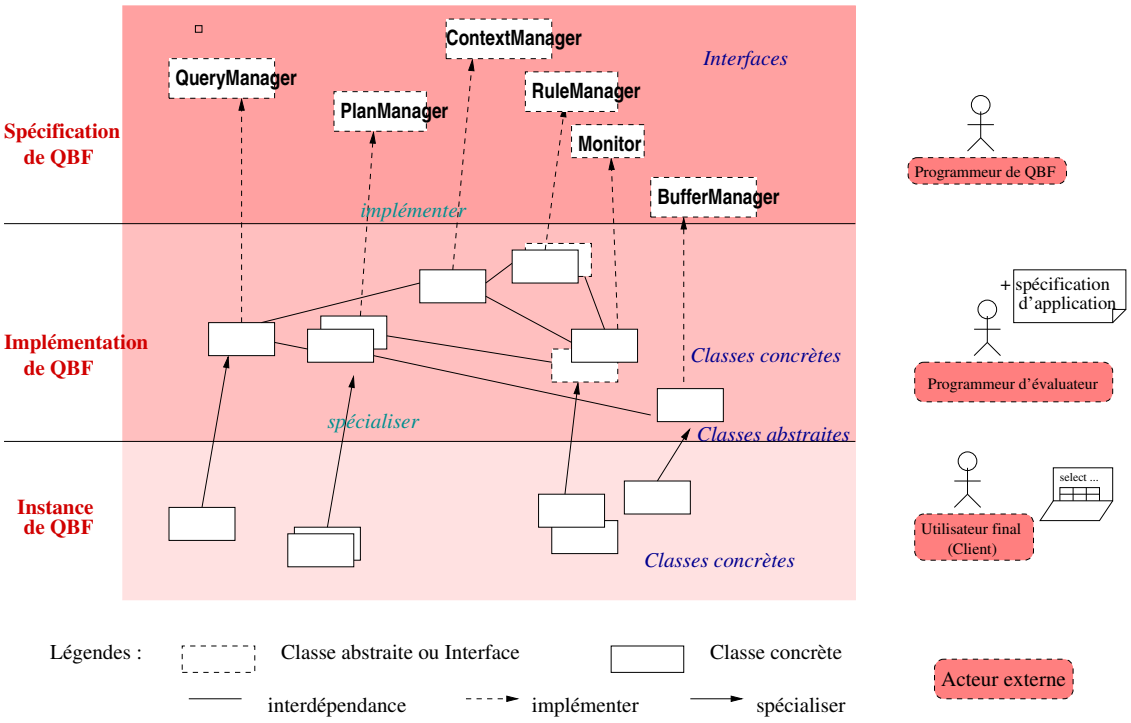



Figure 4.2: Approche QBF

Notons qu'à partir de la spécification de QBF, il est possible d'avoir différentes implémentations. Ainsi, plusieurs instances de QBF, peuvent être construites en utilisant une implémentation de QBF. Ce chapitre est consacrée à la spécification de QBF, i.e. les interfaces de ses composants. La mise en œuvre et l'instanciation de QBF seront présentées dans les chapitres suivants.

**Plan du chapitre** La suite de ce chapitre est organisée de la manière suivante : premièrement, nous présentons quelques hypothèses de la spécification de QBF (la section 4.2) ; la section 4.3 présente le noyau minimal de QBF pour l'évaluation de requêtes ; d'autres composants de QBF conçus dans le but d'assurer une évaluation personnalisée, dynamique et distribuée de requêtes sont décrits respectivement dans les sections 4.4, 4.5 et 4.6. Pour chacun d'eux, nous analysons leur responsabilité puis décrivons leur interface ; enfin, nous comparons notre proposition avec les travaux existants dans la section 4.7 et concluons le chapitre dans la section 4.8.

## 4.2 Préliminaires

Lorsque l'on évoque un évaluateur de requêtes, les deux premières questions posées sont : (i) comment formule-t-on les requêtes ? et (ii) comment récupère-t-on les résultats, dans quel format ? D'une manière générale, selon le modèle de données interrogées, l'utilisateur peut formuler sa requête dans un langage

déclaratif approprié tel que SQL [DD93] pour les données relationnelles, OQL [CBB+97] pour les données objets, XQuery [XQu] pour les données XML, etc. Le système traduit la requête d'utilisateur en une représentation interne souvent sous forme d'arbre d'opérateurs pour son évaluation et retourne les résultats conformes au modèle de données interrogées qui est toujours représenté sous forme d'ensemble de données (*dataset*).

Etant donné que notre objectif est de proposer un outil d'aide à la construction des évaluateurs de requêtes et que nous avons choisi l'approche par canevas, la conception de ceci doit se baser sur l'abstraction de la représentation de données et celle de requêtes dans les évaluateurs. Nous les détaillons ci-dessous.

### 4.2.1 Représentation de données

Nous considérons une représentation minimale de données sous forme d'*ensemble d'items*, idée proposée initialement par G. Graefe dans le système Volcano [GM93].

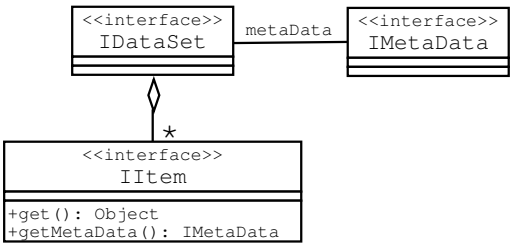



Figure 4.3: Représentation de données

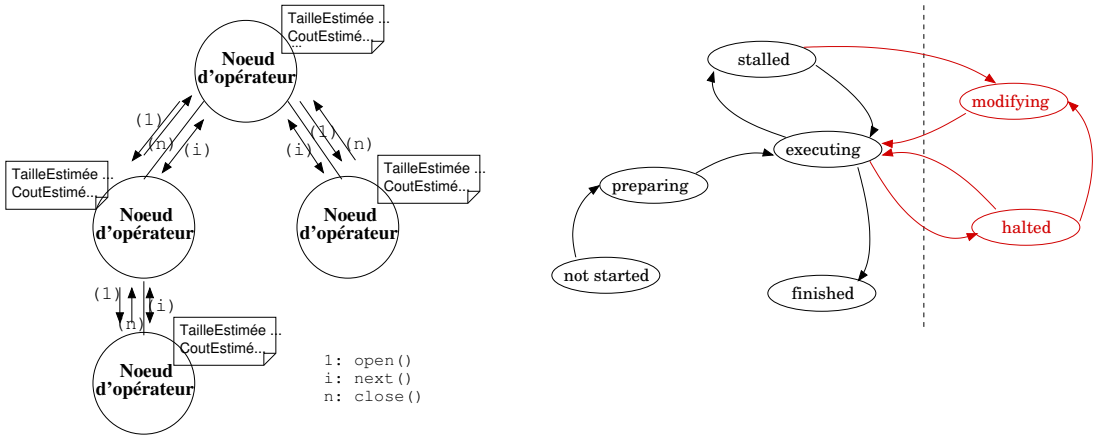
L'ensemble d'items est représenté par l'interface IDataSet regroupant des items représentés par l'interface IItem comme le montre la figure 4.3. Les informations sur les données et leur structure, si elles existent, sont représentées par interface IMetaData qui peut éventuellement être vide.

A l'instanciation de QBF, le programmeur peut implémenter l'interface IMetaData en définissant la structure des données à traiter (e.g. structure de n-uplet, structure d'arbre), ainsi que les informations supplémentaires nécessaires pour les traiter (e.g. taille d'un item).

### 4.2.2 Représentation de requêtes

Nous considérons *une représentation de requête à laquelle est associé un contexte*. Alors que l'expression de requête (ou plan de requête) précise ce que souhaitent recevoir les utilisateurs, le contexte de requête décrit la(les) condition(s) spécifique(s) dans laquelle(lesquelles) ils souhaitent que leur requête soit évaluée.

Généralement, le plan de requête est représenté sous forme de graphe d'opérateurs dont les nœuds correspondent à des opérateurs tels que la sélection, la projection, la jointure, etc., et les arcs représentent les flux de données (i.e. relation production - consommation) entre les opérateurs (voir la figure 4.4(a)).




(a) Plan de requete

(b) Diagramme de transition d'état des noeud

Figure 4.4: Représentation de plan de requête

Par ailleurs, chaque nœud est annoté par ses propriétés pendant l'évaluation de la requête afin de faciliter leurs traitements. Quelques exemples des propriétés sont la taille de résultats produit par le nœud, la sélectivité, le site responsable de l'exécution de cet opérateur dans le cas de requête distribuée, etc. Durant leur cycle de vie, les nœuds sont dans l'un des états correspondants à des processus de traitement comme le montre la figure 4.4(b). Par exemple, au commencement, l'état d'un nœud est *not started* ; il est préparé par la suite pour son exécution (*preparing*) ; pendant l'exécution (*executing*) il retourne les résultats. Pendant l'exécution d'une requête, un nœud est dans l'état *stalled* lors de l'absence de données ou dans l'état *halted* lorsqu'il est suspendu *volontairement* dans l'attente d'un autre traitement tel que l'adaptation de plan[3]. Pendant la phase d'adaptation de requête, il est dans l'état *modifying*.

Les algorithmes des opérateurs sont implémentés dans le modèle itérateur [Gra93]. Selon ce modèle, l'implémentation des opérateurs fournit principalement les trois opérations : (i) `open`, qui prépare l'opérateur pour son exécution (e.g. allocation des ressources, initiation des structures internes de données) ; (ii) `next`, qui déclenche le calcul et le renvoi d'un item par l'opérateur ; et (iii) `close`, qui libère les ressources allouées. L'exécution d'une requête commence par l'appel de la méthode `open` de l'opérateur racine qui est propagée récursivement à tous les opérateurs de l'arbre. Ensuite, la méthode `next` de l'opérateur racine est appelée et, par récursivité, entraîne le calcul d'un item du résultat de la requête. Lorsque toutes les données sont traitées, la méthode `close` du nœud racine est appelée et propagée, de la même manière, à tous les nœuds de l'arbre pour libérer les ressources et finir l'exécution.

Les structures de données ci-dessus sont définies dans la spécification de QBF sous forme d'interfaces. Chaque nœud (représenté par l'interface IOperNode, voir la figure 4.5) est caractérisé par le nom d'opérateur qu'il représente et un état. Il est associé à un Annotation regroupant les propriétés de ce

[3]L'adaptation de requête signifie les changements de plan de requête en cours d'exécution pour adapter le plan d'exécution aux changements de l'environnement d'exécution et/ou à de nouveaux besoins du client.

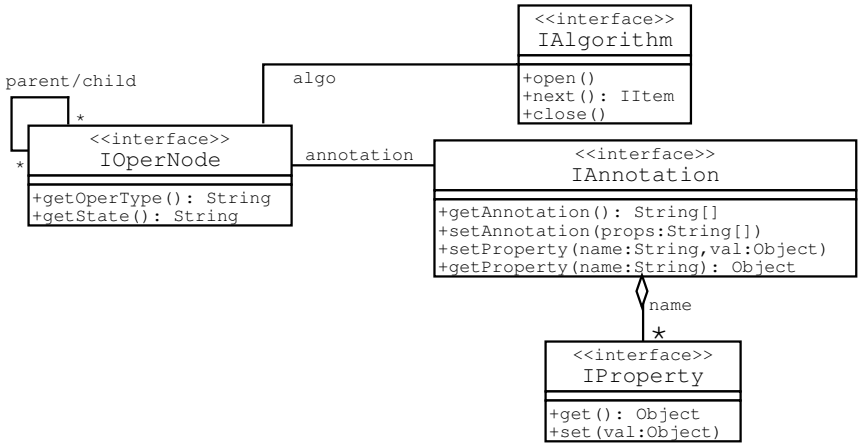



Figure 4.5: Représentation d'un nœud opérateur

nœud et à un Algorithm implémentant l'opérateur dans le modèle itérateur.

Notons que ce modèle représente les requêtes à tout moment de leur cycle de vie, i.e. depuis la création de requête jusqu'à la fin de son évaluation. Cela permet de maintenir explicitement et uniformément la structure de plan de requête, i.e. relation parent - enfant des nœuds, afin de pouvoir les modifier à tout moment.

A l'implémentation et/ou à l'instanciation de QBF, les nouvelles classes implémentées ou spécialisées de l'interface IOperNode sont définies en fournissant les traitements particuliers aux données ayant les caractéristiques spécifiques, i.e. les opérateurs définis pour les structures particulières de données. Ces classes représentent les opérateurs d'une algèbre choisie, tels que la sélection ayant une condition de restriction, ou la projection ayant une liste des propriétés, etc.

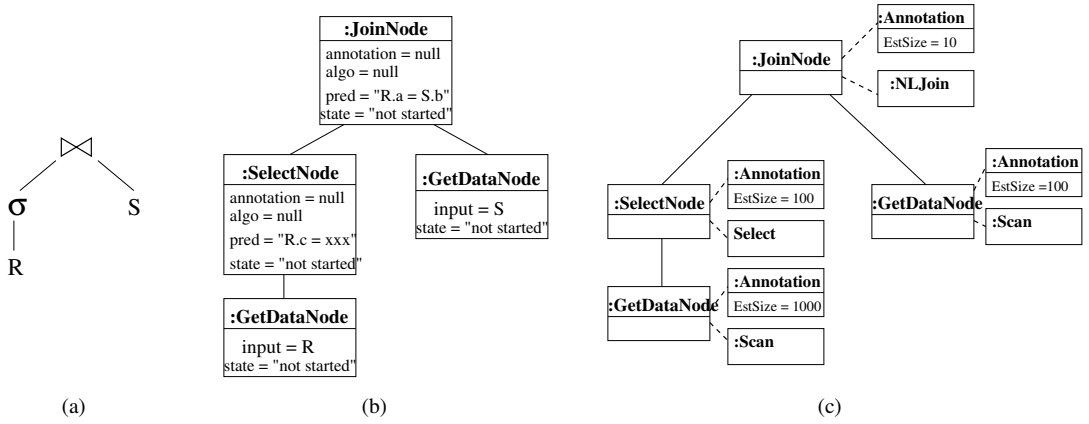



Figure 4.6: Exemple de plan de requête.

Pour illustrer l'utilisation de ces interfaces, nous considérons un exemple de plan de requête sur les deux relations $\mathcal{R}$ et $\mathcal{S}$. La requête est exprimée dans l'algèbre relationnelle sous forme d'arbre algébrique comme le montre la figure 4.6(a). La figure 4.6(b) représente ce plan de requête en terme des classes dont JoinNode, SelectNode et GetDataNode sont les classes implémentant l'interface IOperNode présentée ci-dessus. Les opérateurs *GetData* permettent de lire les données en entrée des relations $\mathcal{R}$ et $\mathcal{S}$. D'autres opérateurs comme la sélection et la jointure ont des prédicats associés (`pred`) représentant les conditions de sélection et de jointure. Durant le traitement de requête, les attributs (e.g. annotation, algorithme, etc.) sont initialisés. La figure 4.6(c) présente le plan de requête après avoir choisi les algorithmes implémentant chaque opérateur : par exemple, l'algorithme de jointure de boucle imbriquée est choisi pour réaliser l'opérateur de jointure ; les relations $\mathcal{R}$ et $\mathcal{S}$ sont lues séquentiellement. Les nœuds opérateur sont annotés par la taille de résultat estimée (dénoté `EstSize` dans Annotation).

**Contexte de requête** Un contexte de requête est représenté par un ensemble de paramètres, i.e. couples de nom et de valeur.

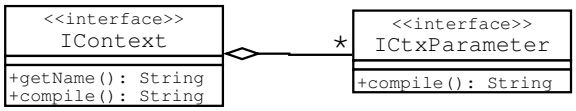



Figure 4.7: Représentation de contexte de requête

Le contexte de requête est représenté par l'interface IContext regroupant un ensemble de paramètres représentés par l'interface ICtxParameter (voir la figure 4.7). L'interface IContext fournit, entre autres, la méthode permettant d'interpréter la sémantique des paramètres de contexte à la compilation. La table 4.1 montre quelques exemples de paramètre de contexte avec leur définition.

| **Paramètre** | **Domaine** | **Définition** |
|---|---|---|
| *ResultNumber* | *int* | Le nombre maximum de résultats obtenus |
| *Timeout* | *int* | La durée limitée de l'évaluation de requête |
| *ResultType* | *complete, vertical, horizontal, generalized* | Type de résultat accepté |
| *Preference* | *String* | Les données préférées parmi celles constituant le résultat |

Table 4.1: Exemple de paramètres de contexte

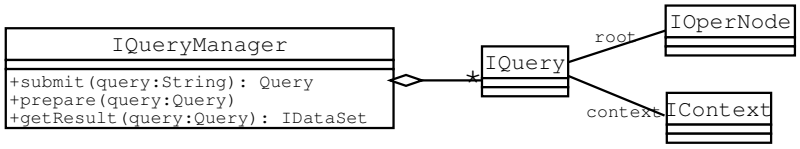



Figure 4.8: Interface du QueryManager

### 4.2.3 Gestionnaire de requêtes

Les requêtes utilisateur composées d'un plan et d'un contexte, dont la structure est représentée précédemment, sont gérées par le QueryManager. En effet, ce composant joue le rôle d'interface de l'évaluateur. Il offre, *au moins*, trois méthodes permettant la soumission d'une requête (`submit`), la préparation de la requête pour son exécution (`prepare`) et l'accès au résultat (`getResult`) (voir la figure 4.8). Cette interface peut être utilisée par l'utilisateur ou l'application cliente pour soumettre des requêtes à évaluer mais elle peut aussi être utilisée par d'autres évaluateurs afin que, tous ensemble, ces évaluateurs résolvent une requête notamment distribuée. Le QueryManager assure l'évaluation des requêtes en coordonnant d'autres composants que nous décrivons dans les sections suivantes.

## 4.3 QBF pour une évaluation statique de requête

### 4.3.1 Analyse

Une des tâches les plus complexes de l'évaluation de requête est l'optimisation (statique). Son objectif est d'explorer l'*espace de recherche* afin de choisir un plan optimal qui devra être exécuté. Comme présenté dans la section 2.2, la génération de l'espace de recherche d'une requête consiste à appliquer des opérations, appelées opérations d'optimisation, sur le(s) plan(s) de requête. Ces opérations consistent en (i) les *transformations logique-logique* permettant la génération des plans qui se différencient par leur structure (e.g. l'ordre d'exécution des opérateurs, le nombre d'opérateurs), et (ii) les *translation logique-physique* permettant de générer des plans qui sont différents dans le choix d'algorithme implémentant chaque opérateur. L'exploration de l'espace de recherche peut s'appuyer sur de différentes *stratégies de recherche* se différenciant par les opérations d'optimisation à appliquer et/ou l'ordre d'application de ces opérations. Le choix du plan optimal peut se baser sur des critères différents tels que les *heuristiques*, les *propriétés* exigées dans le résultat ou encore plus souvent le coût d'exécution calculé par un *modèle de coût*.

La figure 4.9 rappelle l'architecture de l'optimiseur de requête présentée dans la section 2.2. Elle montre également les modules principaux de l'optimisation à concevoir selon l'approche de QBF, i.e. séparation des fonctionnalités de l'optimiseur. Nous notons que l'objet de tous ces traitements est le plan de requête composé des nœuds d'opérateur. Par conséquent, ces modules sont regroupés dans le composant PlanManager décrit ci-après.

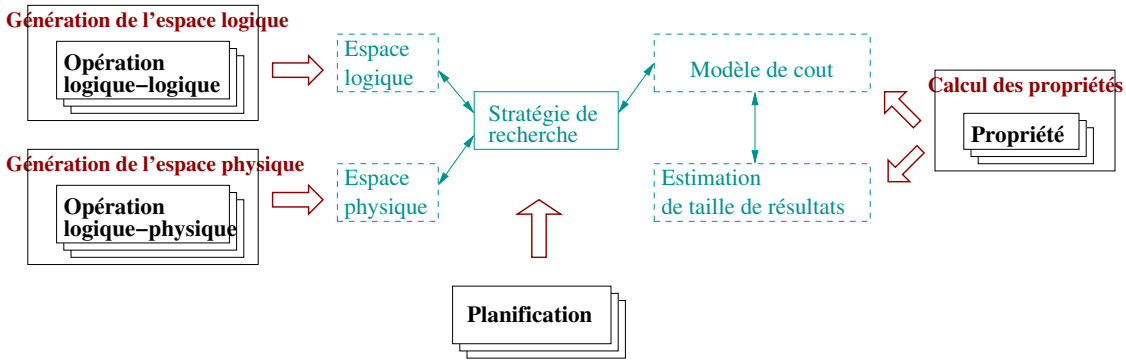



Figure 4.9: Analyse des fonctionnalités de l'optimiseur de requêtes

### 4.3.2 Gestionnaire de plans de requête

Le composant PlanManager est susceptible de gérer une liste de plans de requête représentés par les nœuds racines et de fournir les opérations nécessaires pour effectuer les manipulations sur chaque plan. La figure 4.10 présente l'interface IPlanManager du composant ainsi que sa structure statique, i.e. ses sous-composants[4]. De nouveaux plans sont enregistrés par appel de la méthode `add`. L'évaluation d'un plan de requête consiste, d'une part, à l'optimiser (par appel de la méthode `optimize`), et d'autre part, à exécuter la requête et à retourner les résultats (par appel la méthode `getResult`). L'optimisation est réalisée par coordination des sous-composants, dont chacun représente un des aspects de l'optimisation présentés précédemment. Les sous-composants de PlanManager sont :

- Le composant Transformer : Il contient une liste des transformations possibles des plans de requête au niveau logique (représentée par ITransformingPattern). Ce composant est susceptible de générer les plans équivalents d'une requête formant l'espace "logique" de recherche. L'interface de ce composant (ITransformer) fournit la définition des opérations permettant l'application d'une ou plusieurs transformation(s) (par appel de méthode `apply`).

- Le composant Translator : Il contient la liste des correspondances entre des opérateurs logiques et ceux physiques représentées par IMappingOperator permettant de générer les plans physiques de la requête.

  Les deux sous-composants Transformer et Translator sont susceptible de générer l'espace de recherche.

- Le composant Annotator : Il gère une liste des fonctions (représentées par IPropFunction) permettant calculer les propriétés des nœuds. L'interface IAnnotator contient la définition des opérations pour le calcul (`init`, `initProperty`).

[4]Pour une raison de lisibilité, les sous-composants se représentent d'une manière simplifiée par les rectangles pointillés.

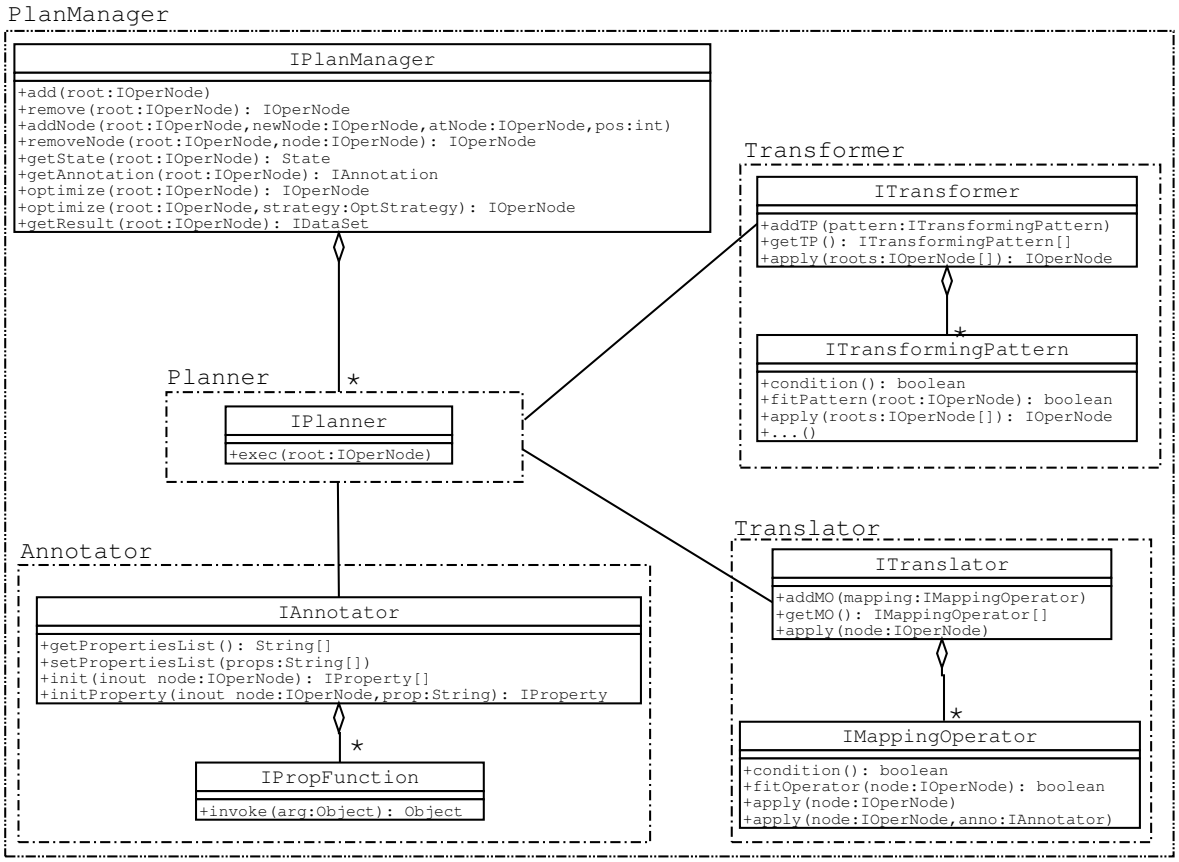



Figure 4.10: Structure du PlanManager

- Le composant Planner : Il applique un ensemble de transformations (logiques et physiques) suivant un ordre approprié afin d'explorer l'espace de recherche dans le but de choisir un plan optimal à exécuter. Il fournit effectivement la stratégie de recherche. L'application d'une stratégie de recherche est réalisée par appel de la méthode `exec` sur le Planner.

### 4.3.3 Optimisation de requêtes

Après avoir séparé les aspects de l'optimisation comme nous venons de le décrire, une stratégie d'optimisation spécifique est obtenue par une "simple" coordination, i.e. interaction, de ces composants.

La figure 4.11 présente les interactions entre les sous-composants du PlanManager. Le PlanManager appelle la méthode `exec` sur un Planner. Selon la stratégie employée par le Planner, les méthodes `apply` sont appelées une ou plusieurs fois sur Transformer et Translator afin d'appliquer les manipulations sur le plan d'une requête. Les méthodes `init`*xx* de l'interface IAnnotator peuvent être appelées pour calculer l'annotation de chacun des nœuds du plan d'une requête si cela est nécessaire pour le choix du plan optimal. Il est à noter que l'ordre et le nombre des appels des méthodes entre Planner

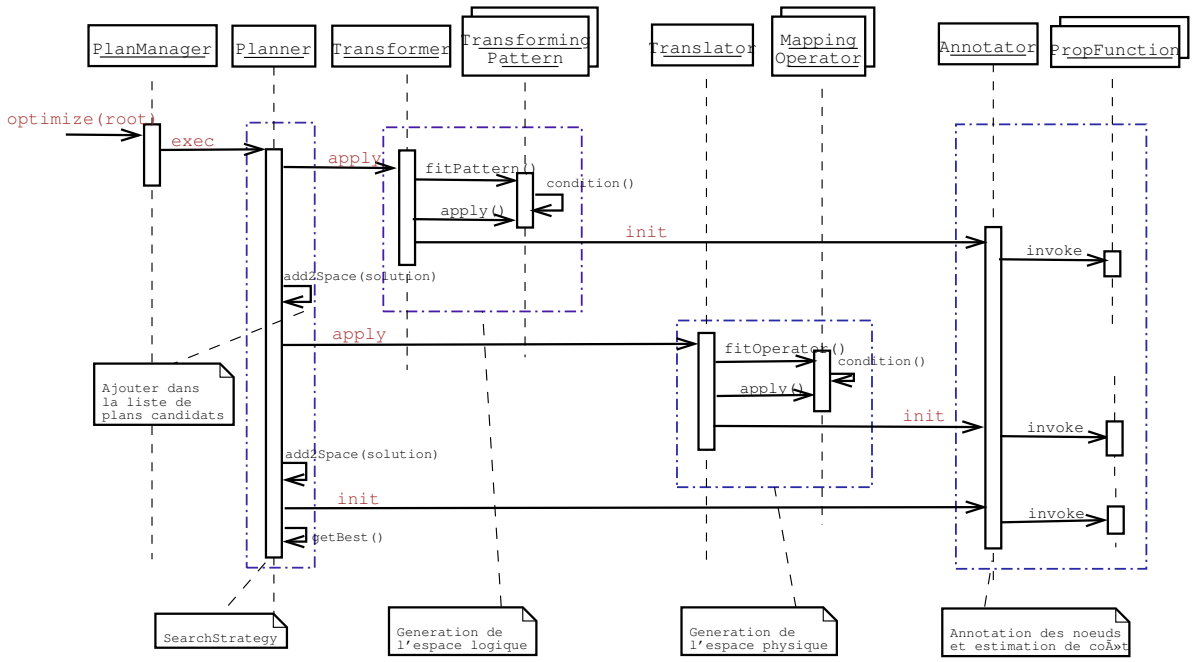



Figure 4.11: Interaction des sous-composants de PlanManager pour l'optimisation de requête

et Transformer, Translator, Annotator dépendent de la stratégie fournie par Planner. Par ailleurs, la composition de Planner, Transformer, Translator, Annotator est dynamique.

A l'implémentation de QBF, le programmeur fournit les implémentations complètes ou partielles de ces interfaces et définit les structures de données nécessaires et leur traitement, facilitant l'instanciation de QBF ultérieurement.

## 4.4 QBF pour une évaluation personnalisée de requête

### 4.4.1 Analyse

L'évaluation personnalisée de requête signifie la capacité du système d'adapter le processus de traitement de requêtes à des contraintes spécifiques définies comme contexte de requête d'utilisateur. Afin de tenir compte de la partie de contexte pendant l'évaluation de requête, il faut que le système soit capable de la gérer de même qu'il gère le plan de requête. Hormis les outils pour accéder au contexte en lecture et en écriture, il est nécessaire de disposer des mécanismes permettant d'interpréter des paramètres de contexte et de les utiliser pour guider l'optimisation. Dans cet objectif, plusieurs aspects de l'évaluation de requête doivent être revus.

Dans certains cas, il faut créer des *éléments de contrôle* tels que "le compteur de résultats" permettant de connaître le nombre de résultats calculés (pour le paramètre *ResultNumber*), "le compteur de temps" pour mesurer la durée de l'évaluation de requête (pour le paramètre *Timeout*), etc. Les éléments de

contrôle créés en fonction du contexte de requête peuvent être des opérateurs de contrôle[5] insérés dans le plan de requête. Par exemple, afin de limiter le nombre des résultats, le système peut ajouter à la racine du plan de requête un opérateur *MaxSize* qui renvoie les items en entrée inchangés jusqu'à l'obtention du n-ième item. Dès lors, la méthode `close` de cet opérateur est appelée "automatiquement" et propagée aux autres nœuds du plan dans le but d'arrêter l'exécution de requête.

Dans d'autres cas, le contexte de requête peut "guider" le système dans le choix de la stratégie d'optimisation, i.e. l'espace de recherche à considérer (l'ensemble des opérations d'optimisation logique et physique à appliquer), le critère ou l'objectif d'optimisation (annotation), ainsi que dans le choix de la stratégie pour explorer cet espace et pour éliminer les plans non-candidats. Cela permettrait d'améliorer le fonctionnement du système en satisfaisant les besoins de l'utilisateur. Pour cela, il faut être capable d'identifier l'influence du contexte de requête sur les aspects de l'optimisation. Par exemple, si l'utilisateur souhaite obtenir les premiers résultats le plus tôt que possible, l'optimiseur peut considérer seulement l'espace de plans non-bloquants, i.e. ceux qui utilisent seulement (ou au maximum) les opérateurs non-bloquants ; lorsque l'utilisateur a des préférences sur certaines sources, les calculs sur celles-ci devront être favorisés ; ou encore, lorsque l'utilisateur spécifie la durée de l'évaluation d'une requête, il sous-entend que l'optimisation de requête vise à maximiser le débit de production de résultat. Dans cette optique, il faut que le système soit capable de *configurer* les éléments de l'optimiseur et aussi de les *composer* dynamiquement.

Ces besoins sont *aisément* pris en compte dans notre approche, où les éléments de l'optimisation sont séparés pour être recomposés "à la demande", permettant également de fournir différentes stratégies d'optimisation au sein d'un même système. Un canevas pour la construction des évaluateurs de requêtes ayant la capacité de personnalisation devra fournir les facilités de prise en compte du contexte de requête telle qu'elle est présentée précédemment. Les deux sections suivantes décrivent respectivement la spécification de composant de gestion de contextes, et les interactions entre ce composant et les autres décrits antérieurement dans le cas d'évaluation personnalisée. Les mécanismes d'évaluation personnalisée de requêtes seront présentés en détail dans le chapitre 6.

### 4.4.2 Gestionnaire de contextes

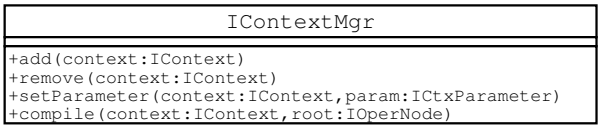



Figure 4.12: Interface de ContextManager

La figure 4.12 présente l'interface du composant ContextManager. Les contextes de requête sont

[5]Les opérateurs de contrôle ne sont pas définis directement sur la structure de données, i.e. modèle de données, mais permettent d'ajouter, de changer des propriétés aux données traitées et/ou de contrôler l'exécution de requête.

enregistrés par appel de la méthode `add`. La prise en compte de contexte est assurée par la méthode `compile` lors de la phase de compilation. L'objectif de cette méthode est de générer les opérateurs de contrôle, de les insérer dans le plan de requête et/ou d'assister le système dans le choix d'une stratégie d'optimisation appropriée (par introduction le paramètre *`OptStrategy`* en appelant la méthode `optimize` sur le PlanManager).

### 4.4.3 Evaluation personnalisée de requêtes

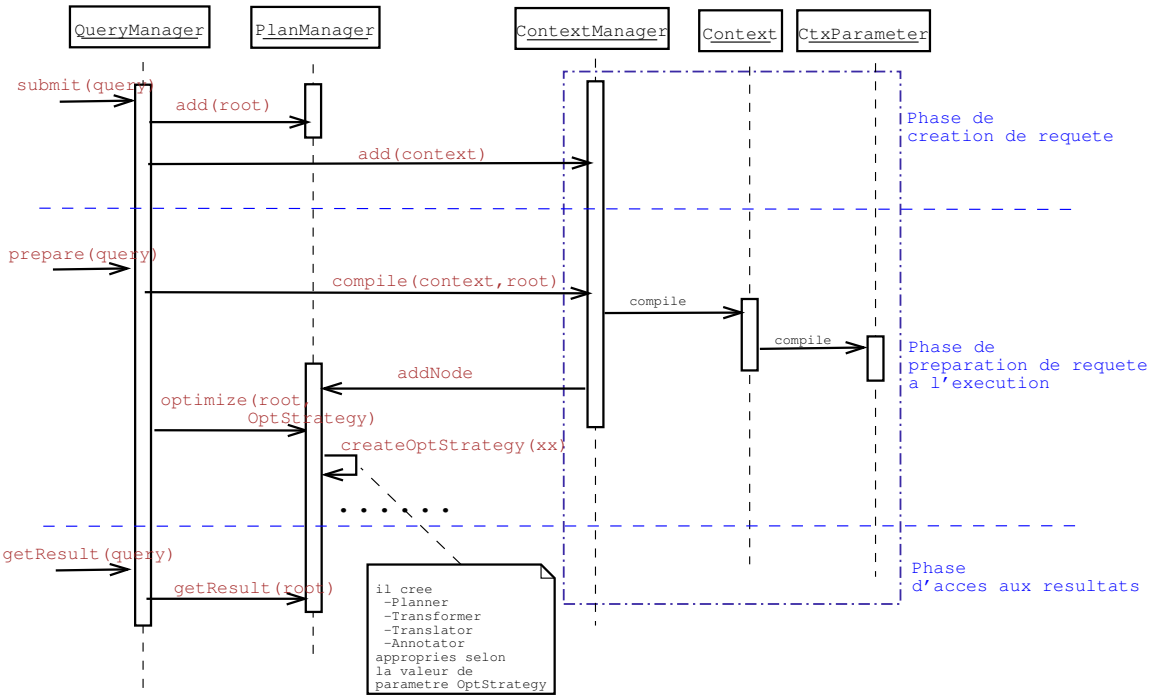



Figure 4.13: Interaction pour l'évaluation personnalisée de requête

Le traitement de requête personnalisé se compose de trois phases de création, de préparation et d'exécution, de façon identique au traitement classique. Pourtant, il nécessite une extension dans les deux phases de *création* et de *préparation* afin d'associer le contexte à la requête, et d'en tenir compte par la suite. La figure 4.13 présente les trois phases de traitement en montrant les interactions entre différents composants de QBF afin d'assumer une évaluation personnalisée.

**Création de requête** A la réception d'une requête composée d'un plan et d'un contexte (par appel de la méthode `submit` sur le QueryManager), le plan de requête représenté par le nœud racine de type IOperNode et le contexte de requête représenté par une instance du type IContext sont créés par appel des méthodes `add` sur le PlanManager et le ContextManager, respectivement.

**Compilation de requête** La compilation est assurée par les deux composants de gestion de contexte et plan de requête. Les paramètres de contexte de requête sont traités, un par un, afin de créer et ajouter un ou plusieurs opérateur(s) dans le plan de requête associée et/ou déterminer une stratégie d'optimisation appropriée pour la requête donnée. L'optimisation de requête assurée par le PlanManager selon la stratégie guidée par le ContextManager. Il s'agit, en effet, du choix d'un Transformer, Translator, Annotator et Planner appropriés à la requête personnalisée.

A l'implémentation de QBF, le programmeur fournit l'implémentation complète ou partielle de ces interfaces ainsi que les structures de données concernant les gestions de données (e.g. contexte, paramètres, etc.). L'ensemble des paramètres de contexte que peut traiter un évaluateur concret ne peut être définis quie lors de l'instanciation de QBF, i.e. implémentation d'un évaluateur ou d'un *Query Broker*.

## 4.5 QBF pour une évaluation dynamique de requête

### 4.5.1 Analyse

Par évaluation dynamique, nous entendons la capacité d'adapter l'exécution d'une requête à des "changements" se produisant pendant son exécution. Les changements peuvent provenir des sources sous-jacentes ou de l'utilisateur, et nous parlons respectivement d'évaluation adaptative et d'évaluation interactive (cf. la section 3.4.2). Que ce soit l'adaptation aux (propriétés des) sources ou à l'utilisateur, il faut être capable d'observer l'exécution de requête ainsi que son environnement afin de détecter une situation qui va déclencher l'adaptation. Par ailleurs, il faut savoir décider comment réagir aux situations détectées.

A la différence des travaux existants sur l'évaluation dynamique, notamment ceux de l'évaluation adaptative, nous cherchons à identifier et à isoler ses principales fonctionnalités, et puis à les placer dans un cadre uniforme qu'est QBF. Cela permettrait, d'une part, l'intégration et l'utilisation de différentes techniques d'une manière uniforme, et d'autre part, d'exploiter chacune de ces fonctionnalités indépendamment afin de faciliter leur réutilisation et leur déploiement dans différents contextes. Les fonctionnalités principales d'une adaptation dynamique sont (i) le déclenchement, (ii) la décision et (iii) la réalisation (cf. la section 3.1).

Pour détecter les situations qui déclenchent une adaptation, i.e. l'origine de l'adaptation, nous devons observer des *propriétés* telles que l'utilisation de mémoire, le débit de données arrivées, la cardinalité, etc. L'état de ces propriétés est vérifié à une *fréquence* donnée. L'observation va donner lieu à des *événements* représentant des *conditions non-souhaitées* (ou changements) auxquelles on doit réagir. Quant à la décision d'adaptation, elle peut être définie en terme de *règles* d'adaptation de types "**E***vénement*-**A***ction*" (EA) ou "**E***vénement*-**C***ondition*-**A***ction*" (ECA), auxquelles nous associons une *stratégie* déterminant l'ensemble et l'ordre des règles à considérer, ainsi que la façon selon laquelle ces règles doivent être exécutées. La réalisation de l'adaptation consiste donc à exécuter des règles, dont la partie **A***ction* définit les modifications à réaliser. Elles peuvent concerner plusieurs niveaux : le niveau de l'opérateur (e.g. l'allocation de ressources, le changement de comportement d'un algorithme ou de choix

d'un algorithme), le niveau de plan de requête (e.g. l'ordre des opérateurs à exécuter ou les opérateurs eux-mêmes) ou le niveau du "paramétrage" de l'évaluateur (e.g. la priorité des opérateurs, le contrôle de l'exécution de certains opérateurs, la mise à jour des méta-données, etc.). Afin de pouvoir réaliser une telle adaptation, il faut donc connaître l'état des opérateurs de requête en cours d'exécution qui est effectivement présenté dans la structure de requête de QBF (cf. section 4.2.2). Sauf adaptation par l'algorithme lui-même, l'adaptation nécessite de repasser par la phase d'optimisation, i.e. appeler les méthodes de PlanManager et de ContextManager, pour re-déterminer le plan d'exécution. En plus, l'adaptation de plan de requête durant son évaluation peut nécessiter de modifier les flux de données déjà initialisés entre des opérateurs du plan de requête. Suivant les modifications effectuées, l'évaluateur peut continuer l'exécution avec les données en cours de traitement, attendre de finir le traitement de ces données avant de continuer, ou ré-exécuter certains opérateurs afin d'obtenir un état de données cohérent.

La réalisation de l'adaptation concerne essentiellement les composants PlanManager, le ContextManager de QBF pour les modifications dans la stratégie d'exécution de requête. Nous séparons les éléments de *déclenchement* de ceux de *décision*. Ces deux éléments sont conçus comme deux composants, un pour le déclenchement d'adaptation (Monitor) et l'autre pour la décision de l'adaptation (RuleManager) et nous les décrivons ci-dessus.

### 4.5.2 Surveillant

Le Monitor est susceptible d'allouer les ressources pour créer les éléments d'observation (représenté par IPropertyMonitor) permettant de détecter des changements *significatifs* dans l'état de certaines propriétés de l'exécution.

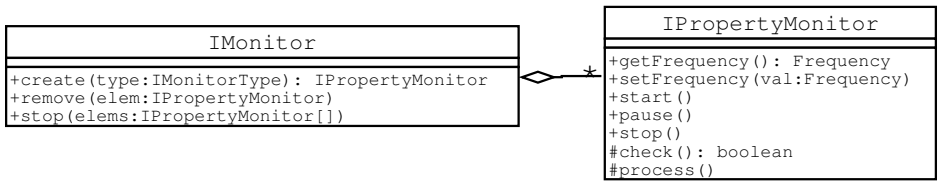



Figure 4.14: Structure de Monitor

Chaque PropertyMonitor est susceptible de surveiller l'état d'une ou quelques propriété(s) de l'exécution de requête dont le changement d'état peut influencer l'évaluation de requête[6]. Un changement est dit significatif s'il satisfait à une condition donnée (définie dans la méthode `check`). La vérification de cette condition se fait à une fréquence prédéfinie (accédé par les méthodes `get/setFrequency`). Lors qu'un changement est détecté, il procède à des traitements appropriés (par appel de la méthode `notify`) notamment pour produire un événement afin de signaler le changement.

[6]Notons que, aucune restriction sur l'organisation des éléments d'observation est imposée. Ils peuvent être incorporés dans les opérateurs du plan de requête, être organisés d'une manière hiérarchique, etc.

### 4.5.3 Gestionnaire de règles

Le changement de l'état des propriétés d'observation sont à l'origine des événements envoyés au RuleManager qui déclenche une ou plusieurs règles d'adaptation en appliquant une stratégie.

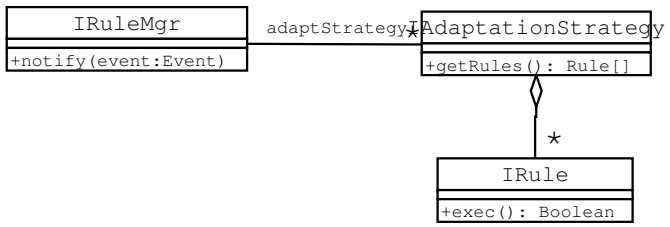



Figure 4.15: Structure de RuleManager

La figure 4.15 présente la structure du gestionnaire de règles. En fait, l'interface IRuleManager pourrait être plus complexe. Pour notre travail, nous nous intéressons seulement à la méthode `notify` susceptible de recevoir des événements provenant de Monitor pour déclencher la règle correspondante. Etant donné un ensemble de règles pouvant être déclenchées lors de la réception d'un événement (appelées *règles déclenchables*), différentes stratégies d'adaptation (représentées par AdaptationStrategy) peuvent être appliquées pour exécuter ces règles. Ces stratégies déterminent, entre autres, le nombre (s'il existe plusieurs règles déclenchables) et l'ordre des règles à exécuter.

### 4.5.4 Evaluation adaptative de requêtes

Pour une évaluation adaptative de requêtes, il est nécessaire de la préparer avant son exécution. Il faut donc *modifier* la phase de compilation de requête, appelée également *préparation*.

**Préparation à l'adaptation** Il s'agit, d'une part, de choisir la stratégie d'évaluation de requêtes en se basant sur la connaissance dont le système dispose et en considérant le contexte de requête (s'il est nécessaire), et d'autre part, de se préparer à une évaluation adaptative, i.e. créer les éléments de surveillance (instances de PropertyMonitor).

La figure 4.16 présente l'interaction entre des composants pendant la phase de préparation d'une requête. La compilation de requête est similaire à celle dans le contexte d'évaluation statique (montré dans la partie grisée du diagramme) sauf une extension vers le composant Monitor afin de créer les éléments d'observation.

**Adaptation** Après avoir compilé (ou préparé) la requête, le système est prêt à retourner les résultats. L'exécution est surveillée par les PropertyMonitor. Les changements détectés durant l'exécution de requête sont signalés par l'envoi d'événements des PropertyMonitor au RuleManager étant susceptible de déclencher les règles afin que le système réagisse aux changements de l'environnement. Les messages

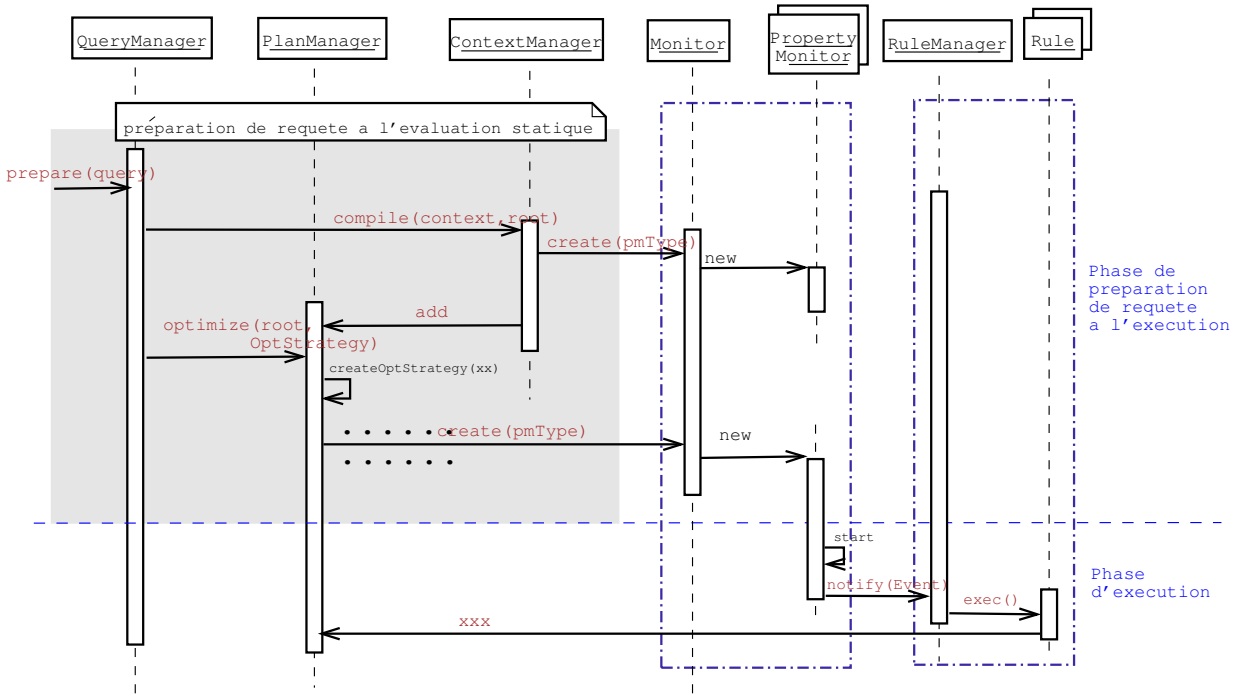



Figure 4.16: Interaction pour l'évaluation adaptative

xxx sont envoyés du RuleManager au PlanManager (voir la figure 4.16) en respectant son interface, i.e. xxx sera remplacé par le nom de méthode définie dans l'interface du PlanManager.

A l'implémentation et/ou l'instanciation de QBF, différents types d'éléments de surveillance peuvent être définis en implémentant l'interface IPropertyMonitor. De même, une ou plusieurs règles sont définies comme des instances de la classe Rule afin de déterminer le comportement du système.

## 4.6 QBF pour une évaluation distribuée de requête

### 4.6.1 Analyse

Dans l'évaluation distribuée de requêtes, plusieurs sites peuvent participer à différentes étapes du traitement, de l'optimisation jusqu'à l'exécution de requête.

Hormis les tâches de l'optimisation classique, il faut décider des sites responsables de l'exécution des opérateurs de requête. De plus, la tâche d'optimisation, elle-même, peut être distribuée sur plusieurs sites ou assurée de la manière centralisée par un seul site. La plupart des systèmes existants choisissent l'approche d'optimisation centralisée dans laquelle un seul site est susceptible de générer la stratégie d'exécution de requête. Cette décision se base, d'une part, sur la localisation de données intéressées par la requête, et d'autre part, sur la communication entre les sites. Cette approche est simple mais

elle requiert les connaissances complètes sur l'ensemble des sites distribués dans le but de prendre la décision sur le mouvement de données et celui de la requête. A contrario, l'approche distribuée n'utilise que les connaissances locales. Elle ne peut donc que résoudre une partie de la requête distribuée. En conséquence, l'optimisation distribuée de la requête nécessite une collaboration entre les sites.

Par ailleurs, l'exécution d'une requête distribuée est naturellement *asynchrone* car elle est assurée par des processus (*threads*) différents, voire des machines différentes. Ce mode d'exécution peut également se réaliser sur un seul site en différentes circonstances, telles que des machines parallèles. Or, le modèle d'exécution de requête adopté par QBF est synchrone. Il faut donc fournir la capacité de *synchroniser* l'exécution asynchrone des sous-requêtes. Une des façons de le réaliser est de *découper* le plan de requête en sous-plans. Au point de découplage, nous *insérons* les tampons pour le stockage de données en cours d'exécution de façon similaire à l'utilisation de l'opérateur *Exchange*, proposé initialement par G. Graefe dans le système Volcano [Gra90]. Ces tampons servent à synchroniser l'exécution de sous-requêtes parallèles et notamment distribuées.

### 4.6.2 Gestionnaire de tampons

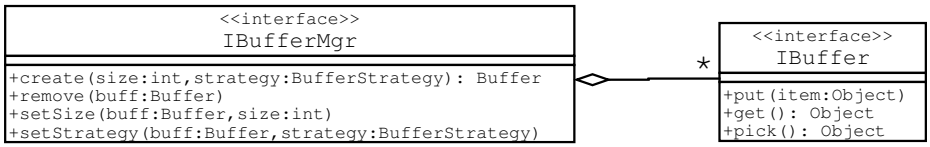



Figure 4.17: Structure de BufferManager

Le BufferManager est susceptible de fournir les outils pour le stockage temporaire de données en cours d'exécution, i.e. allouer et/ou dés-allouer la mémoire pour les tampons (Buffer). Il fournit une interface générique IBufferManager comme le montre la figure 4.17. Les opérations fournies par ce composant (`create`, `remove`, `setSize`, `setStrategy`) permet de créer, de supprimer les tampons et d'ajuster leurs propriétés.

Un tampon peut être impliqué dans l'implémentation de certains opérateurs. Pourtant, il est difficile d'envisager qu'un tel tampon avec ses propriétés (e.g. stratégie de remplacement, taille, etc.) soit défini d'une manière statique avant l'exécution de l'opérateur. D'ailleurs, les tampons peuvent aussi être utilisés en dehors de l'implémentation des opérateurs spécifiques, notamment pour découpler l'exécution d'un plan d'une requête comme présenté précédemment. Pour cette raison, un gestionnaire "centralisée" de tampons est nécessaire et nous le considérons comme un composant indépendant.

### 4.6.3 Interconnexion de *Query Brokers*

La figure 4.18 montre l'interconnexion entre des QBs dans le but d'assurer l'exécution d'une requête distribuée. La communication entre les QBs consiste, d'une part, en l'échange des données, et d'autre part,

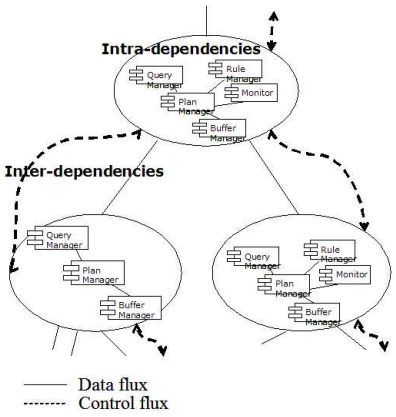



Figure 4.18: Hiérarchie de Query Brokers pour une évaluation distribuée de requête

en la collaboration entre des sites participants à l'exécution de la requête. Dans le cadre de notre travail, nous ne nous intéressons pas à la "découverte" ou la "localisation" de ces sites. Nous supposons que cette connexion s'appuie entièrement sur l'architecture du système cible, e.g. hiérarchie des médiateurs dans le cas du système de médiation, réseau arbitraire de pairs dans le cas du système P2P, etc. Dans le cas de systèmes de médiation, chaque QB correspond à un médiateur de la hiérarchie prédéfinie. Dans le cas de systèmes P2P, la connexion des QBs s'établit d'une manière dynamique lors de la découverte (ou la localisation) des sources et/ou sites précédant l'exécution d'une requête[7]. Dans ce travail, nous nous focalisons sur les échanges (de données et de contrôles) entre des QBs dans une structure hiérarchique prédéfinie.

**Synchronisation de l'exécution de requête distribuée** [8] Elle est assurée par *l'insertion* des tampons dans le plan de la requête distribuée dans le but de synchroniser l'exécution asynchrone assurée par plusieurs sites. Pour cela, nous introduisons un nouvel opérateur nommé *Buffer*. C'est un opérateur unaire capable de stocker des données et implémenté comme un itérateur, c'est-à-dire il fournit les méthodes `open-next-close`. Par conséquent, il peut être inséré, à *n'importe quel point* dans le plan, d'une manière arbitraire comme le montre la figure 4.19(a). Comme tous les opérateurs logiques, il peut exister plusieurs implémentations de l'opérateur *Buffer* qui se différencient par la stratégie de remplacement des données dans le tampon utilisé par cet opérateur, la capacité de stockage de données (dans la mé-

[7] Dans la littérature, nous pouvons trouver différents types de systèmes P2P comme celui basé sur DHT (*Distributed Hash Table*)[GWJD03, HHL+03], les systèmes P2P sémantiques[LIST03, BDK+03, TL04], etc. Ces systèmes se différencient par la façon de représenter et de gérer leurs (méta-)données. En se basant sur les tables de hachage ou les schémas avec des chemins sémantiques (*semantic paths*), le système procède à une phase de localisation afin de déterminer les sites pouvant participer à la réponse de la requête avant de l'évaluer, i.e. effectuer les transferts des données dans la deuxième phase. Notre travail concerne essentiellement la deuxième phase si une telle architecture est prise en considération.

[8] ou désynchronisation de l'exécution de requête centralisée.

moire principale ou sur le disque), ou encore la capacité de compression ou décompression dans le cas d'évaluation de requêtes distribuées que nous décrivons en plus détail ci-dessous.

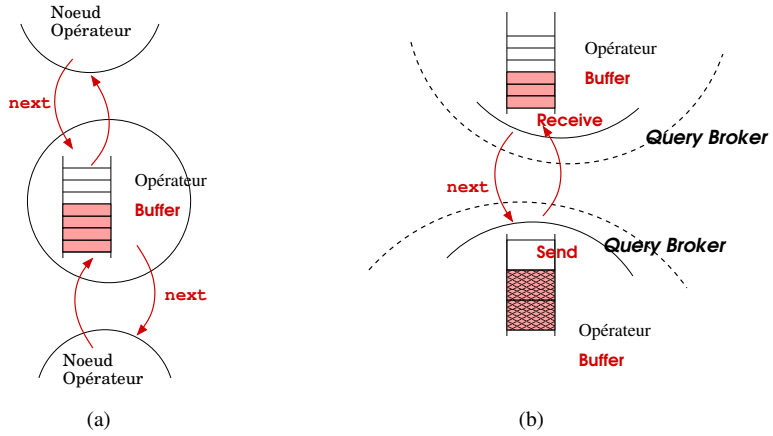



Figure 4.19: Opérateur Buffer

Hormis ce problème de synchronisation de l'exécution de requête sur les différents sites, le problème de transfert de données entre des sites distants influence directement l'efficacité de l'exécution de requête. En effet, l'exécution de requête devient irréaliste si chaque appel `next` sur le site distant ne permet de récupérer qu'un item. Pour cela, nous considérons deux implémentations de l'opérateur *Buffer* : une, appelé *Send*, chargé de "compresser" les données avant de les "envoyer" au site parent, i.e. mis dans le tampon pour attendre d'être consommé par son opérateur parent, et l'autre, appelé *Receive* responsable de "décompresser" d'un item reçu de site distance et de les placer dans le tampon pour être consommé par son opérateur parent (voir la figure 4.19(b)). Les données sont transférées entre les sites par *paquet d'items*, permettant ainsi de limiter le nombre d'appels de méthode à distance et de profiter de la capacité du réseau. La taille d'un paquet est un paramètre de l'opérateur et peut être déterminée d'une manière dynamique.

**Collaboration entre *Query Brokers*** Elle a pour objectif d'échanger les informations de contrôle entre des QBs afin de mieux répondre aux requêtes de l'utilisateur. La collaboration entre les QBs est basée sur l'échange de messages. Plus précisément, un QB peut envoyer les messages (sous forme d'événements) à des QBs directement connectés. L'interconnexion entre des QBs s'appuie sur le composant Monitor, ou plus concrètement PropertyMonitor, responsable de la génération d'événements.

Les messages échangés entre les QB sont : (i) des messages de préparation de l'exécution de requête, appelés *messages d'établissement*, et/ou (ii) des messages de surveillance de l'exécution de requête, appelés *messages de sollicitation*. Les messages d'établissement se produisant avant l'exécution de la requête ont pour objectif de contribuer à la décision de la stratégie d'exécution de requête, e.g. la distribution des sous-requêtes, le coût d'exécution de sous-requêtes, etc. Les messages de sollicitation

consistent à échanger les informations pour la mise à jour des méta-données, les informations concernant l'état d'exécution de sous-requête, etc.

Les messages sont caractérisés par leur type (i.e. établissement vs. sollicitation), l'instant d'émission et d'autres informations spécifiques à chaque type de message, permettant au QB destinataire de décider les traitements.

## 4.7 Travaux connexes

Le résultat présenté dans ce chapitre contribue à la conception d'un outil pour la construction d'évaluateurs adaptables de requêtes. De notre point de vue, l'adaptabilité peut être obtenue à différents moments : à la construction de l'évaluateur (i.e. adaptabilité statique), au début de l'évaluation d'une requête (i.e. personnalisation) et/ou pendant l'évaluation de la requête (i.e. adaptabilité dynamique).

Le tableau 4.2 résume les travaux à comparer avec notre proposition selon les dimensions de l'adaptation.

**Positionnement de QBF selon les systèmes extensibles** Rappelons que l'objectif des systèmes extensibles est d'offrir la capacité d'adaptation à différentes classes d'application.

La plupart des travaux suivant cette voie [GD87, GM93, PHH92] adoptent l'approche de génération de code en proposant les générateurs de l'optimiseur. Les programmeurs, i.e. les utilisateurs de ces générateurs, spécifient le modèle de données souhaité dans un langage de spécification fourni comme interface de générateur. Le résultat de la génération est un optimiseur de requêtes approprié. Dans notre travail, nous ne fournissons pas un langage de haut niveau pour la spécification des besoins de traitement de requêtes mais un niveau abstrait des fonctionnalités de traitement de requêtes aidant les programmeurs à construire les applications avec les fonctionnalités d'interrogation souhaitées. Nous croyons que cette approche est appropriée à la construction d'évaluateurs de requêtes en différentes circonstances, en raison de la diversité des besoins des applications vis-à-vis du traitement de requêtes et de la complexité des fonctionnalités de traitement de requêtes. Cependant, nous sommes conscients que l'utilisation de QBF est moins aisée que celle des générateurs d'optimiseur. Les utilisateurs de QBF doivent, d'une part, comprendre le principe de canevas, et d'autre part, écrire eux-mêmes une partie de code. Néanmoins, nous croyons qu'une spécification claire avec des cas d'utilisation illustrés permettent la compréhension de QBF et facilitent son utilisation.

En positionnant particulièrement QBF par rapport à Volcano[Gra90], nous avons adopté l'idée de la représentation minimale de données et nous fournissons également l'abstraction d'autres fonctionnalités de traitement de requêtes afin de permettre l'extension de la stratégie de recherche et de l'estimation de coût. Par ailleurs, Volcano permet d'encapsuler le parallélisme dans les opérateurs *Exchange*, ce qui est comparable avec l'utilisation de tampon dans notre QBF. Toutefois, le BufferManager de QBF ayant la capacité de générer les tampons adoptant différents modes de fonctionnement permet l'utilisation adaptée des tampons dans différentes situations ainsi que l'optimisation de la communication dans le cas d'évaluation distribuée de requêtes.

Nous adoptons une approche proche de celle considérée les optimiseurs de troisième génération

<table>
<tr><th rowspan="2">Travaux</th><th colspan="3">Adaptabilité statique</th><th colspan="2">Personnalisation</th><th colspan="3">Adaptabilité dynamique</th></tr>
<tr><th>Espace</th><th>Coût</th><th>Stratégie</th><th>Spectre</th><th>Mécanisme</th><th>Origine</th><th>Sujet</th><th>Mécanisme</th></tr>
<tr><td>EXODUS [GD87]</td><td>x</td><td>x</td><td>-</td><td colspan="6" rowspan="4">NON CONSIDERE</td></tr>
<tr><td>Volcano [GM93]</td><td>x</td><td>x</td><td>-</td></tr>
<tr><td>Starburst [PHH92]</td><td>x</td><td>x</td><td>-</td></tr>
<tr><td>[LV91]</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>DISCO [Bon99]</td><td colspan="3" rowspan="2">NON CONSIDERE</td><td rowspan="2">type de résultat partiel<br>préférence[b]</td><td rowspan="2">réécriture de la requête<br>ordonnancement des n-uplets</td><td colspan="3">NON CONSIDERE</td></tr>
<tr><td>JuggleEddy [RH02]</td><td>délai</td><td>n-uplet</td><td>routage</td></tr>
<tr><td>Optimisation paramétrée [CG94, INSS97]</td><td colspan="5" rowspan="3">NON CONSIDERE</td><td>manque de statistique</td><td>le choix d'algorithme, ordre des entrées des opérateurs</td><td>paramétrage (Choose-Plan/liste d'alternatives)</td></tr>
<tr><td>Tukwila [ILW+00]</td><td>tous (par événement)</td><td>tous</td><td>règle</td></tr>
<tr><td>Eddy [AH00]</td><td>délai</td><td>n-uplet</td><td>routage</td></tr>
<tr><td>QBF</td><td>x</td><td>x</td><td>x</td><td>paramètres de contexte</td><td>stratégies alternatives, paramétrables</td><td>tous (événement)</td><td>tous</td><td>règle</td></tr>
</table>

Table 4.2: QBF et les travaux connexes

[a] Rappelons que dans ce système, l'utilisateur spécifie ses préférences sur données en indiquant le dégré de préférence. Ce dernier peut être changé durant l'exécution de requête.

[b] Rappelons que dans ce système, l'utilisateur spécifie ses préférences sur données en indiquant le dégré de préférence. Ce dernier peut être changé durant l'exécution de requête.

[LV91, KD99]. Toutefois, QBF est différent en fournissant une spécification en un niveau plus fin (e.g. l'abstraction de la représentation de données et de requêtes) facilitant l'extension de l'espace de recherche et fournissant, de plus, l'extension de l'estimation de coût. Au-delà, QBF est adaptable dans le sens qu'il permet de modifier ces fonctions dynamiquement (par la composition).

En outre, QBF tient compte des aspects de la personnalisation et de l'adaptabilité dynamique que ne couvre aucun des travaux présentés ci-dessus. Nous croyons que notre approche permet une meilleure extensibilité, réutilisation et adaptabilité.

**QBF et les techniques d'évaluation adaptative et interactive de requêtes** La plupart des travaux sur l'évaluation adaptative et interactive de requêtes[CG94, INSS97, Bon99, ILW+00, AH00, RH02] proposent, d'une manière spécifique, les mécanismes pour résoudre un ou quelques besoins de l'évaluation de requêtes. Plus précisément, les mécanismes d'évaluation adaptative et interactive de requêtes sont proposés, conçus et implémentés dans les systèmes ayant des caractéristiques spécifiques, e.g. modèle de données, opérateurs, stratégie d'optimisation, etc. Par conséquent, il n'est pas équitable de comparer directement QBF avec ces derniers.

En effet, adoptant l'approche de canevas, notre objectif principal n'est pas de proposer les mécanismes spécifiques pour une telle évaluation, mais de séparer les éléments des solutions (i.e. mécanismes) afin de les (re-)intégrer d'une manière uniforme. Cette approche permet, premièrement, d'exploiter l'ensemble de ces mécanismes, et deuxièmement, de combiner les différents éléments de ces mécanismes dans le but d'offrir une solution globale. QBF généralise les aspects liés à l'évaluation personnalisée de requêtes en considérant un contexte associé à chacune des requêtes. Cela permet une plus grande possibilité de personnalisation. Les différents mécanismes d'évaluation adaptative sont également intégrés dans QBF par l'utilisation des règles EA ou ECA.

Par rapport à Tukwila[ILW+00], utilisant également des règles pour intégrer différents mécanismes d'évaluation adaptative de requêtes, QBF diffère par son spectre d'adaptation, ainsi que par sa conception. Tukwila dispose d'un module de surveillance (appelé Status Monitor) permettant de détecter des conditions non-souhaitées se produisant pendant l'exécution d'une requête. Lors de la détection, le système retourne à la phase d'optimisation par le module Event Handler pour effectuer la ré-optimisation de la requête selon les nouvelles connaissances acquises. Dans tous les cas, la ré-optimisation de requêtes dans Tukwila ne tient pas compte de l'état actuel de l'exécution du plan de requête en cours. Un nouveau plan est généré. L'ancien plan avec toutess les données traitées est "matérialisé" pour les traitements ultérieurs. A la fin de l'exécution, le système procède à une phase de "nettoyage" (*Cleanup phase creation*) pour combiner les plans et les données de toutes les étapes afin de compléter le résultat de la requête[Ive02, ILW04]. En disposant d'une conception plus fine au niveau de l'optimiseur ainsi que la possibilité d'annoter les nœuds opérateurs, QBF permet la prise en compte de l'état de l'exécution de requêtes afin de permettre la mise en œuvre des mécanismes d'évaluation adaptative tenant compte de l'état du plan d'exécution. Par ailleurs, dans Tukwila, il n'est pas montré clairement si l'ajout, l'omission ou la combinaison d'événements et de règles est possible. Finalement, Tukwila n'était pas conçu dans le but de permettre l'exploitation de ses différents modules d'une manière indépendante et aisée.

## 4.8 Conclusion du chapitre

Dans ce chapitre, nous avons présenté QBF, un canevas pour la construction des évaluateurs adaptables de requêtes. Nous avons focalisé notre présentation sur la spécification de QBF, i.e. l'ensemble des interfaces représentant les fonctionnalités de base de l'évaluation de requêtes. Nous avons également montré les interactions entre ces composants dans le but d'assurer les tâches bien définies de l'évaluation, à savoir l'optimisation, la personnalisation, l'adaptation dynamique et la distribution de l'évaluation de requêtes.

L'optimisation de requêtes de façon classique est assurée principalement par le composant PlanManager. La séparation des tâches de l'optimisation (i.e. génération de l'espace de recherche, stratégie de recherche, estimation de coût) permet une extensibilité de tous les aspects de l'optimisation et offre une *adaptabilité statique*. De plus, une telle séparation est utile dans l'*évaluation personnalisée* où la prise en compte de contexte associé à la requête d'utilisateur nécessite des modifications dans plusieurs phases de l'évaluation. Pour cela, il faut que le contexte soit défini à l'aide de ContextManager. L'*adaptabilité dynamique* est obtenue grâce aux composants Monitor et RuleManager. Ces deux composants permettent de définir les conditions qui sont à l'origine des déclenchements de l'adaptation (*événements*), des décisions de l'adaptation (*règles*).

Rappelons que notre objectif au départ est d'offrir un support adaptable pour l'évaluation de requêtes. Pour cela, nous avons choisi une approche de séparation et d'abstraction des fonctionnalités de l'évaluation de requêtes. Une telle séparation permet, premièrement, d'identifier clairement les éléments de l'évaluation de requête et de mieux la comprendre. Deuxièmement, elle permet de recomposer quelques uns de ces éléments afin d'offrir les solutions plus adaptées à différentes situations de l'évaluation de requêtes. Troisièmement, grâce à cette séparation, il est possible de définir leurs interactions d'une manière indépendante de leur implémentation, permettant une meilleure réutilisation.

Nous avons vu que l'approche par canevas permet de faciliter la construction d'évaluateurs de requêtes ayant des caractéristiques différentes. Cette approche offre une grande possibilité de réutilisation, d'extension et donc d'adaptation, mais souffre, sans aucun doute, de quelques "*surcoûts*". Par le terme de surcoût, nous entendons les difficultés liées à la séparation des tâches de QBF et donc à son utilisation, à la mise en œuvre des algorithmes existants au sein de QBF, ou encore le coût en terme de performance (e.g. temps, utilisation de CPU, de mémoire) de l'implémentation de QBF et ses instances (e.g. le surcoût de la surveillance, le coût de l'adaptation, etc.). Ces points sont étudiés dans le chapitre suivant.

# CHAPITRE 5

# Mise en œuvre de QBF

"*Quels sont les algorithmes supportés par QBF ?*", est la question que peuvent se poser les lecteurs. C'est à cette question que ce chapitre tente de répondre. Il permet de comprendre la mise en œuvre (*partielle, parfois*) de différents algorithmes d'évaluation (statique et dynamique) de requêtes au sein des composants de QBF.

## 5.1 De la spécification à la mise en œuvre

La spécification de QBF présentée dans le chapitre précédent consiste en un ensemble d'interfaces représentant les fonctionnalités de base des évaluateurs de requêtes (selon notre analyse). La mise en œuvre de QBF correspond à la deuxième étape de notre approche (cf. la section 4.1). Il s'agit d'une implémentation (partielle) des interfaces de QBF. Par implémentation partielle, nous entendons les classes, notamment abstraites, définissant les structures de données à manipuler ainsi que leurs traitements sous forme des méthodes dont certaines peuvent être abstraites. Concrètement, les structures de données OperNode, Annotation, Contexte, QueryManager, PlanManager, ContextManager, etc. doivent être fournies dans l'implémentation de QBF. Par ailleurs, quelques mécanismes d'optimisation, les mécanismes de surveillance (de la cardinalité, de la taille de données, de temps d'exécution, etc.), les mécanismes de notification ou concrètement la communication entre Monitor et RuleManager, les mécanismes d'exécution des règles, les opérateurs de base de l'adaptation (i.e. ceux constituant la définition de la partie "action" des règles) peuvent être également inclus dans l'implémentation de QBF. Les utilisateurs de QBF (i.e. programmeurs) peuvent en tirer profit lors de la construction de leurs évaluateurs selon leurs besoins.

Rappelons que notre approche se base sur le principe de séparer et d'abstraire les fonctionnalités de base des évaluateurs de requêtes afin de rendre ces derniers adaptables. Cependant, les algorithmes existants ne sont pas conçus dans cet esprit. Alors, la complexité de la mise en œuvre de QBF se trouve, d'une part, dans la séparation des tâches au sein des algorithmes (existants) afin de les "attribuer" aux

éléments de QBF *d'une manière correcte*, et d'autre part, dans la réutilisation de ces différents éléments afin de fournir plusieurs stratégies de traitement de requêtes. Dans la suite de ce chapitre, nous soulignons la mise en œuvre de QBF selon ces divers points.

**Plan du chapitre** La suite de ce chapitre est organisé de la manière suivante : tout d'abord, nous expliquons comment représenter et évaluer les requêtes dans QBF (section 5.2) ; les sections de 5.3 à 5.5 présentent la mise en œuvre des mécanismes liés aux différents niveaux de complexité de l'évaluation de requêtes dans QBF, à savoir l'optimisation statique, l'évaluation personnalisée et l'évaluation adaptative ; la section 5.6 évoque nos retours d'expérience dans la réalisation de QBF ; et finalement, la section 5.7 conclut ce chapitre.

## 5.2 Opérateurs de requête

La représentation de requêtes dans QBF concerne principalement les interfaces IOperNode, IAlgorithm, IAnnotation (cf. section 4.2.2). Leur implémentation sont fournies sous forme des classes abstraites, constituant les points d'accès communs aux informations de tous les opérateurs de requête, de tous les algorithmes implémentant ces opérateurs et de tous les propriétés associées.

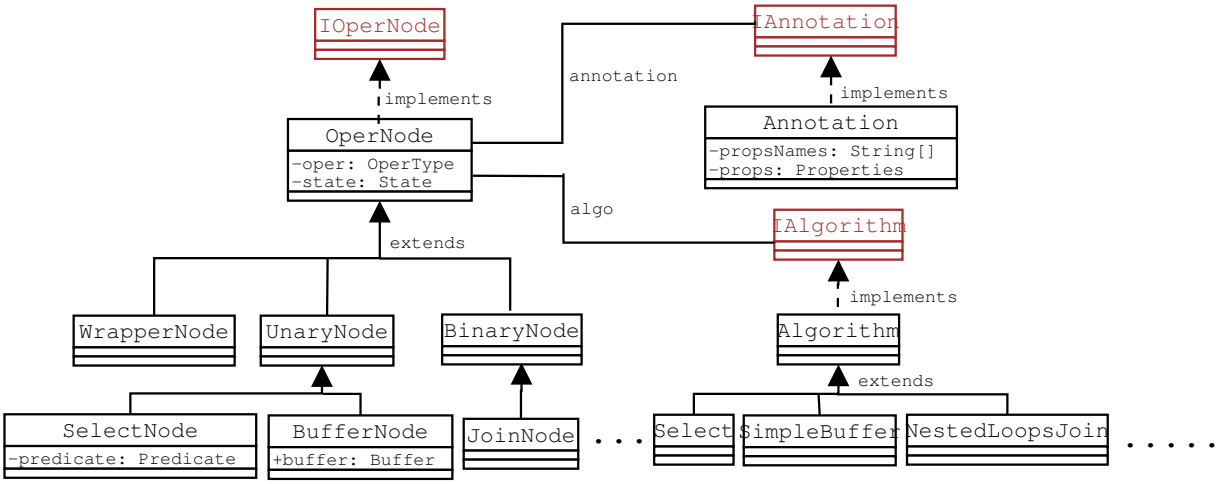



Figure 5.1: Opérateurs de requête dans QBF

Puisque QBF est basé sur une évaluation ensembliste, l'implémentation de QBF inclut les opérateurs algébriques suivants : *Select*, *Join*, *Product*, *Union*, *Intersection*, *Difference*,*Group*, *Distinct*, *Sort*, *Mapper*. Ces opérateurs sont représentés par les classes SelectNode, JoinNode, etc. spécialisant la classe OperNode ou une de ses sous-classes comme le montre la figure 5.1[1]. Nous entendons par spécialisation *l'ajout* de nouveaux attributs spécifiques de l'opérateur et/ou le remplacement des traitements

[1]Les types Properties, Predicate sont définis comme classes dans Java.

génériques fournis par OperNode par des traitements appropriés à chaque opérateur. Par exemple, il faut ajouter l'attribut `predicate` qui représente la condition de sélection pour l'opérateur *Select*. Ainsi, les algorithmes sont fournis par des classes implémentant l'interface IAlgorithm. Quelques exemples d'algorithme sont *select*, *nested-loops-join*, etc. Pour une description plus détaillée et complète des opérateurs et algorithmes fournis dans QBF, les lecteurs sont invités à consulter l'annexe A.

Il est à noter que tous les opérateurs de QBF sont implémentés avec le modèle itérateur (représenté par l'interface IAlgorithm). En conséquence, l'exécution de requêtes est relativement simple. Elle est assurée en un seul processus d'une manière synchrone. Par contre, ce modèle est relativement "pauvre" car il ne permet pas de traitement en parallèle. Lorsqu'un opérateur est bloqué (i.e. il ne peut pas retourner les données), les autres opérateurs sont également bloqués. Afin de pouvoir désynchroniser une telle exécution de requête, nous utilisons l'opérateur *Buffer* (cf. la section 4.6). Ayant capacité de stockage de données, l'opérateur *Buffer* est implémenté par deux *threads* dont un pour la lecture de tampon, i.e. correspondant à l'exécution de l'opérateur parent, et l'autre pour l'écriture dans le tampon, i.e. correspondant à l'exécution de l'opérateur fils. L'exécution peut alors passer au mode *multi-thread* par une simple utilisation de BufferNode dans le plan. Cela permet de choisir librement le mode de fonctionnement de l'exécution selon les requêtes, les ressources du système et de désynchroniser le modèle itérateur si nécessaire (e.g. traitement distribué).

A l'instanciation de QBF, de nouveaux opérateurs et algorithmes peuvent être ajoutés par l'extension d'une de ces classes. Par ailleurs, le programmeur peut aussi définir une structure spécifique pour leurs données en fournissant l'implémentation de l'interface IMetaData.

## 5.3 Optimisation de requête

L'optimisation de requêtes concerne principalement le composant PlanManager qui se compose des sous-composants Transformer, Translator, Annotator et Planner, susceptible de réaliser les fonctions principales de l'optimisation, à savoir la génération de l'espace de recherche, l'estimation de coût et la stratégie de recherche. Cependant, nous avons remarqué que ces trois fonctions ne sont pas souvent séparées dans les algorithmes d'optimisation existants. Cette section propose une adaptation de ces algorithmes afin qu'ils soient applicables dans QBF.

### 5.3.1 Espace de recherche

Nous rappelons que l'espace de recherche d'une requête contient l'ensemble des plans équivalents de la requête donnée. Ils sont générés par application d'opérations sur le plan initial de requête et sur ceux intermédiaires. Dans QBF, nous distinguons deux types d'opérations, un pour la génération de l'espace logique et l'autre pour la génération de l'espace physique.

**Espace logique** L'optimisation logique s'appuie, dans la plupart des cas, sur les propriétés des opérateurs comme (i) la commutativité des opérateurs de sélection, de jointure, d'union ; (ii) l'associativité

des opérateurs de jointure, d'union, d'intersection, de différence ; (iii) la distributivité des opérateurs de sélection sur les opérateurs binaires ci-dessus.

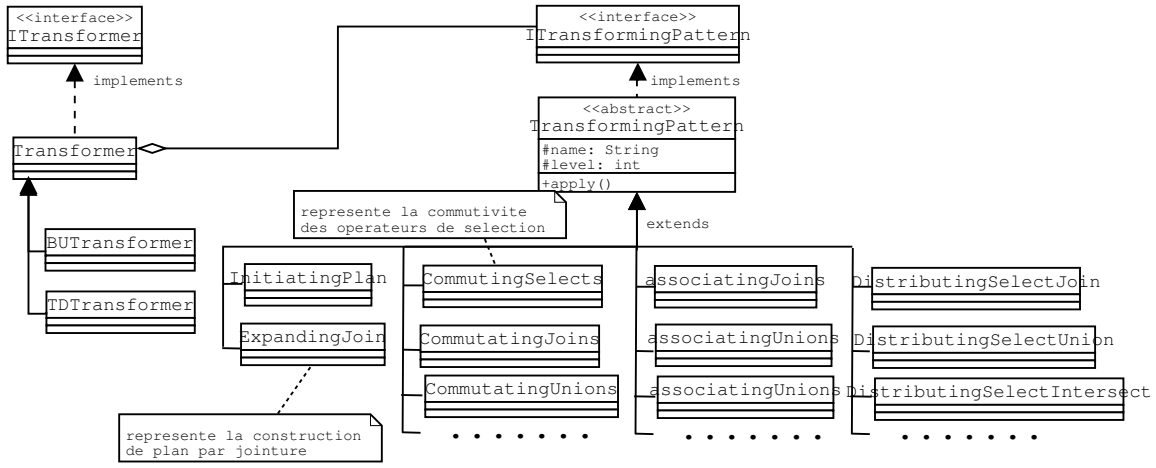



Figure 5.2: Eléments pour générer l'espace logique

La génération de l'espace logique se fait par l'application des opérations d'optimisation logique représentées dans QBF comme les instances de TransformingPattern. La figure 5.2 montre une partie de diagramme de classes les représentant. En fait, on y trouve deux hiérarchies de classes, une pour les opérations d'optimisation (TransformingPattern) et l'autre pour les espaces groupant plusieurs opérations(Transformer). Par exemple, la figure 5.2 montre deux espaces représentés par BUTransformer (*bottom-up*, pour une approche par construction), TDTransformer (*top-down*, pour une approche par transformation.

Il est important de noter que les opérations d'optimisation peuvent se combiner entre elles. Plus précisément, un patron de transformation peut être défini par l'application d'un ensemble d'autres patrons de transformation. Par exemple, les patrons de transformation représentant la distributivité de l'opérateur de sélection avec les opérateurs binaires peuvent être regroupés en un seul patron de transformation représentant l'heuristique "*sélection d'abord*". Nous détaillons l'exploitation des opérations sur le(s) plan(s) lors de l'explication de la mise en œuvre des stratégies de recherche.

**Espace physique** Il se compose de plans de requête se différenciant les uns des autres par les algorithmes implémentant les opérateurs de requête. La génération de cet espace s'appuie généralement sur les implémentations disponibles des algorithmes dans le système d'évaluation. Nous considérons par exemple les algorithmes de sélection séquentielle ou par index, les algorithmes de jointure par boucle imbriquée (*Nested-Loop-Join*), par hachage (*Hash-Join*, *Symmetric-Hash-Join*, *XJoin*, etc.). La génération de l'espace physique applique les opérations d'optimisation physique représentant par MappingOperator susceptible de générer les plans physiques de la requête.

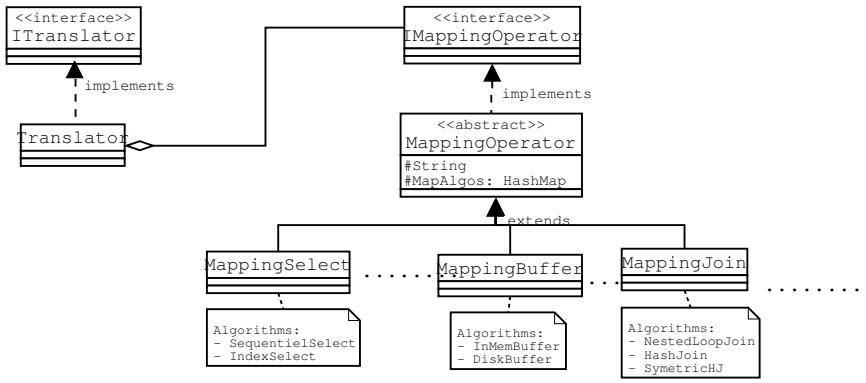



Figure 5.3: Eléments pour générer l'espace physique.

De même que les opérations d'optimisation logique, les opérations d'optimisation physique peut se grouper par de Translator pour assurer quelques caractéristiques spécifiques des plans générés.

L'ensemble des opérations que nous venons de décrire permet de générer l'espace de recherche d'une requête. L'optimisation a donc pour but d'examiner cet espace afin de trouver le meilleur plan d'exécution. Pourtant, il est coûteux de générer l'espace et de le parcourir au cours de deux étapes différentes. La plupart des algorithmes d'optimisation se basent donc sur l'application de cet ensemble d'opérations ou d'une partie de cet ensemble dans le but d'énumérer les plans et d'en choisir un dans le même temps. Ces algorithmes sont bien connus comme stratégies de recherche qui se différencient par l'ensemble d'opérations de plans à appliquer et l'ordre d'application de ces opérations. La section suivante détaille comment les stratégies de recherche sont définies sur l'ensemble de ces opérations.

### 5.3.2 Estimation de coût

Rappelons que le coût d'un plan de requête, s'il existe, est considéré comme une des propriétés de plan ou plus précisément des nœuds de plan représentés comme annotation du nœud (Annotation) . D'une manière générale, l'annotation peut contenir n'importe quelle information guidant le choix de meilleur plan. Le calcul de ces propriétés est assuré par Annotator composé d'une liste des fonctions dont chacune est susceptible de calculer (estimer) la valeur d'une propriété à annoter.

Parmi les propriétés possibles d'un nœud, nous trouvons que la taille de données produites par un nœud est souvent un facteur indispensable de l'estimation de coût. Son calcul est assuré par SizeFunction implémenté dans QBF comme illustre la figure 5.4.

Il est important de noter que la liste des propriétés n'est pas limitée. A l'instanciation, le programmer peut ajouter d'autres propriétés en fournissant une fonction de type PropFunction.

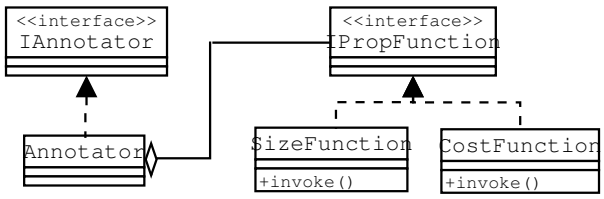



Figure 5.4: Annotation

### 5.3.3 Stratégie de recherche

L'objectif de la stratégie de recherche est de définir la façon selon laquelle l'espace de recherche est examiné. Comme présenté dans le chapitre précédent, le composant Planner de QBF est susceptible de fournir la stratégie de recherche pour l'optimisation de requêtes. En effet, il coordonne les autres composants définissant l'espace de recherche (Transformer et Translator) et l'estimation de coût (Annotator). La stratégie de recherche est implémentée par la méthode `exec` de Planner. Ci-après, nous présentons les points principaux de la mise en œuvre de deux stratégies de recherche correspondant à deux algorithmes d'optimisation présentés dans la section 2.2.3, une par transformation (TopDown) et l'autre par construction (BottomUp) (cf. la figure 5.5).

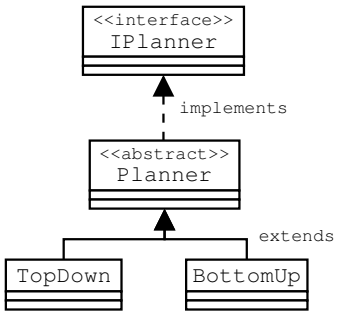



Figure 5.5: Stratégie de recherche

**Optimisation par transformation** Cette approche s'appuie sur la transformation d'un plan de requête d'une manière récursive jusqu'à l'obtention du plan optimal. Il commence par application des transformations (logique-logique) sur le plan initial (par appel `apply` sur le `Transformer` contenant un ensemble des opérations de transformation logiques), puis des translations (logique-physique) qui sont susceptibles d'utiliser les fonctions d'estimation de coût fournies dans Annotator pour choisir le plan optimal.

Le pseudo-code de l'implémentation de cette stratégie est montré dans la figure 5.6.

```
         ITransformer transformer;
         ITranslator translator;
         IAnnotator   annotator;

         List  space;
1:  IOperNode exec(IOperNode root){
2:     space.add(root);
2:     if (translator != null)
3:        root = translator.apply(root,annotator);
4:     if (transformer != null)
5:        root = transformer.apply(root,annotator);
6:     return root;
7:  }
```

Figure 5.6: Pseudo-code de l'algorithme d'optimisation par transformation

Il existe plusieurs variantes de l'algorithme suivant cette approche. Elles se différencient par le nombre des solutions retenues à chaque étape et bien sûr le critère du choix. Par ailleurs, ces variantes peuvent considérer différentes formes de plan, e.g. arbre de type linéaire gauche (*left-deep tree*), arbre de type linéaire droit (*right-depth tree*) ou arbre de type *bushy* (*bushy tree*). Ces derniers se définissent par l'espace de recherche considéré ou plus précisément, par les transformations appliquées. Ces algorithmes peuvent être mis en œuvre au sein de QBF avec légères modifications par rapport à l'algorithme présenté dans la figure et/ou par la combinaison de différents éléments Transformer, Translator et Annotator. Une telle combinaison est aisée car les éléments Transformer, Translator et Annotator sont considérés comme paramètres de Planner et les interactions entre ces éléments sont entièrement définies sur leur définition de l'interface (i.e. les méthodes).

**Optimisation par construction** Selon l'approche de construction, l'espace logique et l'espace physique sont construits au fur et à mesure en élargissant le(s) plan(s) considéré(s). Plus précisément, de nouveaux plans sont construits en joignant ceux construits antérieurement en appliquant l'opération de construction de plan représentée par *ExpandingJoin*. Ce dernier est une opération binaire prenant deux plans en entrée et produisant un plan composé par jointure des deux entrées. Une version simple de cet algorithme est présentée en pseudo code dans la figure 5.7.

## 5.4 Evaluation personnalisée de requêtes

L'objectif de l'évaluation personnalisée de requêtes est de prendre en compte les besoins spécifiques de l'utilisateur vis-à-vis du traitement de requêtes. De tels besoins spécifiés dans le contexte de requête influencent la stratégie d'évaluation de requête, en particulier l'espace de recherche, l'estimation de coût et/ou le choix d'une stratégie de recherche à appliquer. Dans cette section, nous essayons de montrer comment l'implémentation de QBF fournit les traitements de base pour permettre une telle évaluation.

```
        ITransformer transformer;
        ITranslator translator;
        IAnotator   annotator;

1:  IOperNode exec(IOperNode root){
2:     List space;
       // get inputs of the query
3:     List inputs = root.getInputs();
3:     int input_num = inputs.size();
       // find best 1-relation plan for each relation
4:     int n=1;
5:     for (int i=0; i<input_num; i++)
6:        space[i] = translator.apply(inputs[i],
                                     annotator)
       // find best way to join result of 1-relation plan (as outer) to
       // another relation (All 2-relation plans)
7:     while (n < input_num){
8:        List LeftSubGoals = space;
9:        int LeftSize = LeftSubGoals.size();
10:       for (int i=0; i<LeftSize; i++){
11:          RightSubGoals[i] = inputs - LeftSubGoals[i].getInputs();
          }
12        for (int i=0; i<LeftSize; i++){
             RightSize = RightSubGoals[i].size();
13           for (int j=0; j<RightSize; j++)
14:             results[i].add(transformer.apply({LeftSubGoals[i],
                                                  RightSubGoals[i][j]},
                                                  expandingJoin));
15:       for (int i=0;i<results.size(); i++){
16:          for (int j=0; j<RightSize; j++)
17:             results[i][j] = translator.apply(results[i][j],
                                                 annotator)
             // keep the best one
18:          prune(results[i]);
          }
19:       n++;
20:    }
21: }
```

Figure 5.7: Pseudo code de l'algorithme d'optimisation par construction

Plus précisément, nous présentons la gestion de contexte de requête au sein de QBF et la possibilité de "configurer" les tâches d'optimisation dans différentes conditions. Nous détaillerons les problèmes de l'évaluation personnalisée dans le chapitre 6.

### 5.4.1 Représentation de contexte de requête

Les contextes de requête sont représentés dans QBF comme une liste des couples d'attribut et de valeur. Un paramètre de contexte ne peut être traité que s'il est pris en compte par l'évaluateur. Pour cela,

l'implémentation de ContextManager contient un attribut `ContextParameters` définissant la liste des paramètres admissible pour un évaluateur. La méthode `IsValide` est susceptible de vérifier s'il un paramètre est admissible. La figure 5.8 montre l'implémentation des composants concernant essentiellement l'évaluation personnalisée.

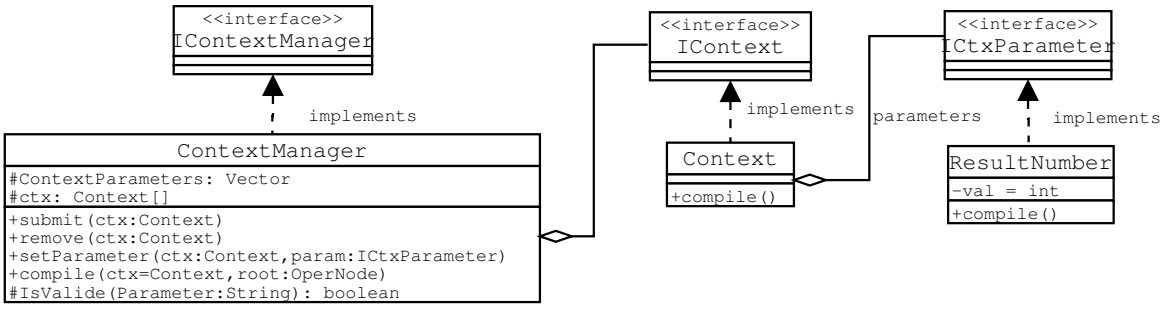



Figure 5.8: Gestionnaire de contexte dans QBF

Nous implémentons un paramètre de contexte *ResultNumber* qui spécifie le nombre de résultats maximal accepté par l'utilisateur. En utilisant ce type de paramètre de contexte, nous montrons dans la suite, comment il est pris en compte par la phase d'optimisation.

### 5.4.2 Optimisation de requête en présence de contexte

La compilation des paramètres de contexte conduit à l'ajout d'opérateurs spécifiques dans le plan de requête initial et/ou de "configurer" les fonctions d'optimisation, e.g. la génération de l'espace de recherche, l'estimation de coût, etc. Il n'est pas possible de traiter les paramètres d'une manière générique car chaque paramètre a une signification particulière et seulement la personne qui le définit est capable de l'expliciter. Ainsi, le traitement des paramètres de contexte est identifié de manière *ad hoc*. QBF ne fournit que le gestionnaire de contexte et facilite le traitement de contexte grâce à sa séparation des tâches d'optimisation.

Prenons l'exemple de paramètre *ResultNumber* étant équivalent à des requête *Top/Bottom k*, nous présentons comment les solutions pour ce paramètre telles que celles proposées dans [CK97] sont mises en œuvre. Nous introduisons un opérateur nommé *MaxSize*, étant un opérateur un-aire qui renvoie les données inchangées en entrée à la sortie jusqu'à l'obtention de l'item $n$-ième.

**Opérateur** *MaxSize* Cet opérateur dispose un "compteur de résultat" pour compter le nombre d'items produits. Lors de l'obtention de l'item $n$-ième, cet opérateur arrête tout de suite l'exécution de requête par appel de sa méthode close lui-même ou suspend l'exécution de requête en attendant d'autre "commande" (ou d'autres traitements). Lors de la compilation de ce paramètre, l'opérateur *MaxSize* est ajouté à la racine du plan de requête.

Selon les deux stratégies d'optimisation de cet opérateur proposé dans [CK97], nous définissons les opérations permettant de propager cet opérateur soit de manière "conservative " ou "agressive" (cf. chapitre 3.3.1). Ces deux stratégies se différencient en fait par l'ensemble des opérations d'optimisation logique à appliquer.

En considérant des besoins tels que ceux du traitement du paramètre *ResultNumber*, nous définissons les différents espaces de recherche considérés selon les critères de plans générés. Pour cela, nous définissons les Transformer et Translator regroupant les opérations de manipulation de plan approprié dans de différentes circonstances et de regrouper les éléments appropriés dans les OptStrategy que peut utiliser le QueryManager suivant le résultat de la compilation de contexte de requête. Par exemple, NonBlockingTranslator ne génère que des plans constitués par les opérateurs non-bloquants.

Il est important de noter que les espaces de recherche spécifiques constitués par les Transformer et Translator spécifiques peuvent partager les opérations de manipulation, i.e. des TransformingPattern et MappedOperator.

## 5.5 Evaluation adaptative de requêtes

Hormis le composant PlanManager qui est indispensable à tous les évaluateurs de requêtes, l'évaluation adaptative concerne essentiellement deux composants Monitor et RuleManager, qui recouvrent les étapes du déclenchement à la réalisation de l'adaptation. Cette section présente les points principaux de la mise en œuvre des mécanismes liés à ces différentes étapes au sein des composants Monitor et RuleManager (les sous-sections 5.5.1 et 5.5.2, respectivement).

### 5.5.1 Déclenchement de l'adaptation

Afin de déclencher une adaptation, il faut être capable de connaître l'état *actuel* des propriétés de l'exécution (e.g. la cardinalité de données d'entrée et de sortie, la sélectivité, l'utilisation de mémoire, le débit réseau, etc.), qui peut influencer l'efficacité de l'exécution de requête et de détecter si un changement *significatif* dans l'état a lieu. Nous devons donc savoir ce qu'il faut surveiller (i.e. propriétés surveillées), comment collectionner les informations (ou état) sur ces propriétés, et quand un changement doit être signalé. D'ailleurs, l'action de collectionner les informations et de vérifier leur état peut se répéter à différentes fréquences. Le tableau 5.1 résume les principales valeurs de la fréquence dans le cadre d'évaluation de requête. Nous laissons le tableau ouvert car l'ajout de nouveaux opérateurs lors de l'instanciation peut nécessiter de définir des fréquences de surveillance particulières.

**Propriétés surveillées** Il s'agit des informations collectionnées pendant l'évaluation de requête afin de pouvoir détecter les changements qui requièrent des modifications de la stratégie d'exécution de requête. Ces informations sont de différentes natures : certaines sont les informations statistiques sur les données telles que la cardinalité des données d'entrée et/ou de sortie, le temps d'accès aux données, le temps de traitement, etc.; d'autres sont les informations sur les ressources telles que la mémoire, le débit du réseaux, etc.; d'autres concernent le coût d'exécution d'ine requête ou de ses sous-requêtes. Selon la

| num | Fréquence | Description |
|---|---|---|
| 1. | *per_item* | pour chaque item |
| 2. | *per_operator* | pour chaque opérateur (à un moment spécifique de l'exécution de l'opérateur) |
| 3. | *per_query* | pour chaque requête (à la fin de l'exécution de la requête) |
| 4. | *every_items(n)* | pour tous les n items |
| 5. | *periodic(t)* | à tous les intervalles de temps t (durée) |
| ... | | |

Table 5.1: Fréquence de surveillance

nature des propriétés surveillées, différents mécanismes doivent être mis en place pour surveiller leur état. Le tableau 5.2 montre les informations que l'on peut collectionner pendant l'exécution de requête et décrit brièvement les mécanismes de surveillance correspondants. Notons que cette liste n'est pas exhaustive. D'autres propriétés peuvent être ajoutées au niveau de l'implémentation de QBF et/ou ses instances.

**Collection des informations** Pour cela, il est nécessaire d'intégrer des "éléments spéciaux" dans l'évaluation de requête afin de recueillir les informations telles que celles décrites ci-dessus. La surveillance de plusieurs propriétés, e.g. cardinalité, mémoire, temps, etc., nécessite de "regarder" le fonctionnement des opérateurs étape par étape, i.e. appels des méthodes `open`, `next`, `close`. Ces éléments de surveillance doivent être intégrés dans le plan de requête soit par l'incorporation des mécanismes de surveillance dans l'implémentation des opérateurs de requête, i.e. extension des implémentations des opérateurs, soit par l'association des "opérateurs de surveillance" à des opérateurs de requête, i.e. maintenance d'une séparation de l'implémentation d'opérateur et des mécanismes de surveillance.

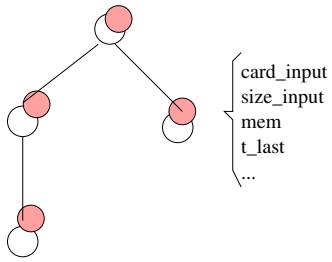



Figure 5.9: Plan de requête en présence d'éléments de surveillance

Compte tenu notre objectif d'offrir des outils d'aide à la construction d'évaluateurs adaptables de requête, nous essayons de séparer les mécanismes de surveillance de l'implémentation des opérateurs de

| num | Propriété (Symbole) | Fréquence | Description | Mécanisme |
|---|---|---|---|---|
| 1. | $card_{input}$ | *per_item* | le nombre d'items en entrée | *compter* le nombre d'appel de la méthode `next` de l'opérateur fils |
| 2. | $card_{output}$ | *per_item* | le nombre d'items produits | *compter* le nombre d'appel de la méthode `next` de l'opérateur |
| 3. | $sel$ | *per_operator*, *per_query*, *every_items*, *periodic* | la sélectivité | calculer à partir du nombre d'items d'entrée et celui produit par l'opérateur. |
| 4. | $size_{input}$ | *per_item* | la taille d'un item en entrée | mesurer la taille effective d'un item. |
| 5. | $size_{output}$ | *per_item* | la taille d'un item produit (résultat) | *idem.* |
| 6. | $t_{begin}$ | *per_item*, *per_operator*, *per_query*, *every_items(n)* | le temps écoulée depuis le commencement de l'exécution | *mesurer la durée* depuis l'appel de la méthode `open` |
| 7. | $t_{last}$ | *per_item*, *every_items(n)*, *periodic(t)* | le temps d'attente depuis le dernier élément en entrée | *mesurer la durée* depuis la réception du dernier item en entrée |
| 8. | $t_{open}$ | *per_operator* | le temps pour initier l'évaluation de requête | *mesurer la durée* d'exécution de la méthode `open` |
| 9. | $t_{next}$ | *per_item*, *every_items(n)*, *periodic(t)* | le temps pour produire un item | *mesurer la durée* d'exécution de la méthode `next` |
| 10. | $mem_{used}$ | *per_operator*, *périodic(t)* | la mémoire utilisée | mesurer l'espace occupé des tampons, tables de hachage, etc. |
| 11. | $rate_{input}$ | *per_operator*, *per_query*, *every_items(n)*, *périodic(t)* | le débit du flux de données en arrivée | pouvant être calculer à partir de la taille de données et le temps écoulé |
| 12. | $rate_{output}$ | *per_operator*, *per_query*, *every_items(n)*, *périodic(t)* | le débit de la production de résultats | *idem.* |
| ... | | | | |

Table 5.2: Propriétés surveillées

requête. En effet, l'adaptabilité à des besoins de l'application peut requérir ou ne pas requérir la capacité de surveillance. La figure 5.9 illustre le plan d'exécution en présence des éléments de surveillance considérés comme opérateurs de surveillance. L'appel des méthodes `open`, `next`, `close` des opérateurs auxquels associe l'opérateur de surveillance fait appel aux méthodes correspondantes de l'opérateur de surveillance.

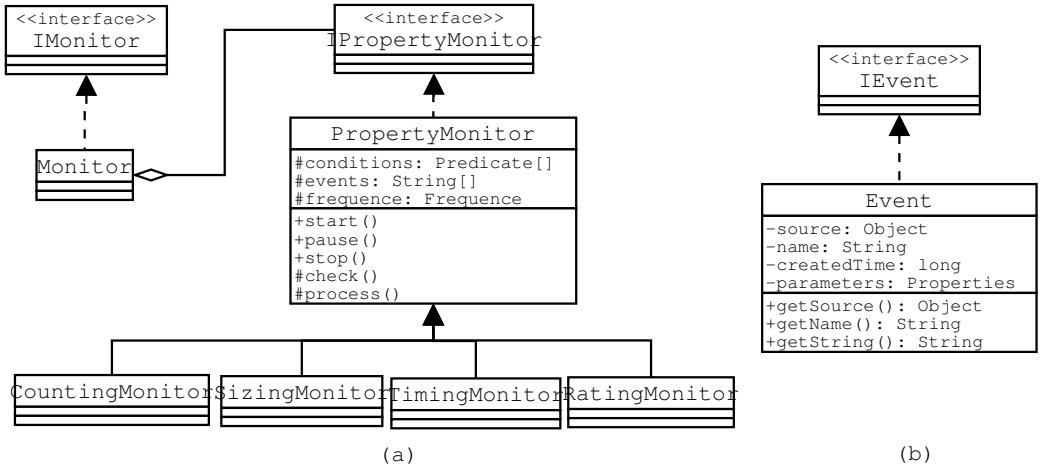



Figure 5.10: Classes pour la détection des conditions non-souhaitées

**Détection des conditions non-souhaitées** Les changements pendant l'exécution, e.g. les statistiques sur les données, l'utilisation de ressources, le débit réseau, etc., doivent être signalés afin que le système puisse y réagir. Un changement d'état d'une propriété est dit significatif si la différence entre son état actuel et celui estimé (par le système en utilisant les méta-informations) dépasse un seuil donné. Dès lors, ce changement requiert une modification, nécessaire pour permettre l'exécution adéquate dans la nouvelle situation. Il est difficile de déterminer, d'une manière générique, à partir de quelle différence un changement est considéré significatif et les adaptations à réaliser. Celles-ci dépendent des types de propriété surveillée, de l'environnement dans lequel la requête est exécutée. C'est pour cela que l'approche par canevas est intéressante. Elle permet de séparer les différents éléments liés à la surveillance, à l'adaptation, et de retarder l'implémentation de certains éléments au plus tard sans avoir besoin de modifier d'autres éléments déjà implémentés.

Dans QBF, la surveillance des propriétés décrites ci-dessus est assurée par les PropertyMonitor dont la structure est présentée dans la figure 5.10(a). Ces éléments sont intégrés dans le plan de requête comme des opérateurs de surveillance comme présenté précédemment. L'implémentation de QBF contient une implémentation partielle de PropertyMonitor. Concrètement, elle fournit les mécanismes pour collectionner les informations pendant l'exécution de requête. Les conditions considérées comme étant non-souhaitées sont définies à l'instanciation de QBF, i.e. à la construction d'un évaluateur, par la

méthode `check` susceptible de vérifier la condition et la notification d'un changement, par la méthode `process`, susceptible de produire les événements (représenté par Event, voir la figure 5.10(b)).

La figure 5.3 montre quelques exemples de type d'événement pouvant être produit suite à des changements d'état dans les propriétés de surveillance ci-dessus.

| num | Evénement | Paramètre(s) | Description |
|---|---|---|---|
| 1. | *timeout* | t - délai | délais de garde dépassé |
| 2. | *every_items* | $n$ - nombre d'items | réception de (ixn)-ième item |
| 3. | *threshold* | $n$ - nombre d'items | obtention de n-ième item |
| 4. | *selectivity* | $new$ - la nouvelle valeur de la sélectivité | changement de la sélectivité |
| 5. | *rate* | $new$ - la nouvelle valeur du débit de flux observé | changement de débit de données arrivé |
| ... | | | |

Table 5.3: Evénements

### 5.5.2 Décision de l'adaptation

La décision de l'adaptation est assurée principalement par le composant RuleManager qui reçoit les événements produits par les éléments de surveillance PropertyMonitor, et déclenche une ou plusieurs règles d'adaptation. Ce composant doit fournir des mécanismes pour la réception des événements provenant de Monitor et pour l'exécution des règles. La figure 5.11 montre le diagramme de classe de l'implémentation de règles et du gestionnaire de règles.

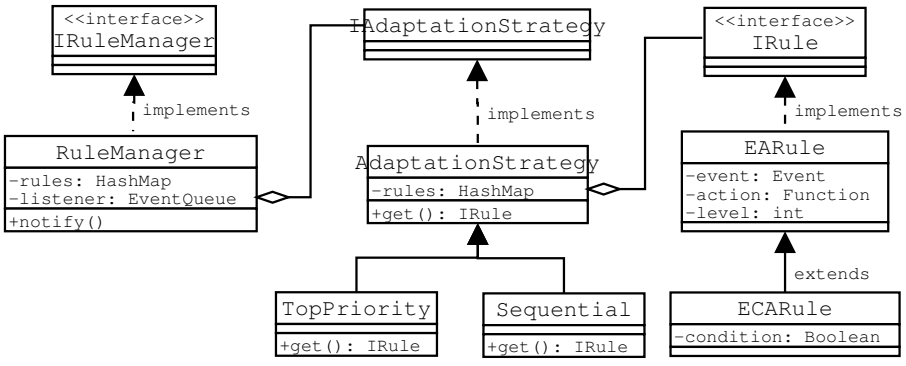



Figure 5.11: Implémentation de RuleManager

**Réception d'événements** Le composant RuleManager dispose d'une une file d'attente (EventQueue) stockant tous les événements envoyés par les PropertyMonitor. La classe EventQueue fournit les

opérations pour la gestion des événements dont les méthodes : (i) `put` qui met un événement dans la file; et (ii) `get` qui retourner un événement de la file à traiter. Afin que la réception des événements et leur traitement soient indépendants, RuleManager est implémenté par un *threads* susceptible de recevoir les événements et une autre *thread* chargé de consommer les événements et de déclencher les règles correspondantes.

| num | Opération d'adaptation | Paramètre(s) | Description |
|---|---|---|---|
| 1. | realloc_mem | $delta$, $increase$ | modifier (i.e. augmenter si la valeur de $increase$ est vrai; sinon, réduire) la taille de mémoire allouée un $delta$. |
| 2. | replace_algo | $new\_algo$ | modifier l'algorithme implémentant l'opérateur |
| 3. | replace_oper | $new\_oper$ | remplacer l'opérateur par un nouveau $new\_oper$ |
| 4. | reschedule | | réordonner l'exécution des opérateurs |
| 5. | reorder_join | | réordonner les opérateurs de jointure |
| ... | | | |

Table 5.4: Opérations de base pour l'adaptation

**Opérations de modification** Lors de la détection d'un changement significatif dans l'exécution de requête, le système essaie de modifier la stratégie d'exécution de requête. Ces modifications peuvent être très différentes d'un système à l'autre mais elles s'appuient sur les modifications élémentaires appelées *opérations d'adaptation* que résume le tableau 5.4. L'implémentation de QBF fournit l'implémentation de ces opérations de base.

**Décision d'adaptation** Lors de la réception d'un événement, le RuleManager détermine les règles déclenchables. Une règle est dite déclenchable si l'événement reçu correspond à celui de la partie "Event" de la règle. Il peut exister une ou plusieurs règle(s) déclenchable lors de la réception d'un événement. L'exécution des règles retenues peut être différente selon la sémantique des règles et/ou les besoins de l'application. Pour cela, nous définissons différentes stratégies d'adaptation représentées par la classe AdaptationStrategy. En effet, ces stratégies définissent l'ordre d'exécution et le mode d'exécution de règles. Dans notre implémentation, nous fournissons quelques stratégies simples :

1. *TopPriority* : Seule la règle déclenchable ayant la priorité (présenté par l'attribut `level`) la plus élevée est exécutée.

2. *Sequential* : Toutes les règles déclenchables sont exécutées, les unes après les autres selon leur priorité.

D'autres stratégies d'exécution de règles peuvent être ajoutées par le programmeur. Nous pouvons constater qu'aucune règle d'adaptation concrète n'est fournie. En effet, cela reste conforme à notre objectif initial qui est de fournir aux programmeurs des mécanismes de base pour la construction de leurs évaluateurs de requêtes mais non de fournir les évaluateurs.

## 5.6 Critères d'évaluation

### 5.6.1 Séparation de tâches au sein de QBF

La séparation de tâches est retenue dans la mise en œuvre des algorithmes d'optimisation statique (présentés dans le chapitre 2) et les algorithmes pour une évaluation adaptative (présentés dans le chapitre 3) au sein des composants PlanManager, Monitor, RuleManager.

En ce qui concerne l'optimisation statique, il s'agit de déterminer les opérations de manipulation de plans permettant de générer l'espace de recherche. Nous considérons deux types d'opération que sont l'opération de transformation (logique-logique) et celle de traduction (logique-physique). Nous les identifions dans les algorithmes d'optimisation existants afin de les "sortir" et les "distribuer" respectivement dans les composants Transformer et Translator. Il n'est peut-être pas très objectif de dire que cette séparation est facilement réalisable (!!) mais nous pouvons affirmer qu'elle est possible. Grâce à cette séparation les modifications dans un des éléments n'impliquent aucune modification dans d'autres éléments. De plus, l'optimisation est assurée par une composition de ces éléments, pouvant se faire à des moments différents et rendant l'optimisation adaptée à des contextes spécifiques. Cela est très significatif pour les adaptations ultérieures que nous aborderons dans les prochains chapitres.

En ce qui concerne l'évaluation adaptative, les techniques existantes groupent à la fois des mécanismes de surveillance, de décision et fournissent une ou plusieurs adaptations spécifiques. Nous avons proposé de les séparer et les distribuer dans les composants Monitor et RuleManager. Certes, cette séparation implique l'ajout de communication entre ces composants mais elle permet "aisément" l'utilisation des techniques proposées dans les contextes différents, e.g. l'environnement distribué où les éléments de surveillance (i.e. PropertyMonitor), de prise de décision (i.e. Rule et RuleManager) et l'effet de l'adaptation (i.e. les changements dans le plan ou dans quelques fragments du plan) peuvent être dispersés sur le réseau. De plus, elle facilite la définition de nouvelles formes d'adaptation tout en profitant des mécanismes de surveillance déjà mis en place.

En résumé, nous croyons que la séparation des tâches de l'évaluateur de requêtes est importante et significative. Elle favorise l'extensibilité, la réutilisation et l'adaptation que nous avons motivées dès début de ce document (cf. le chapitre 1) et que nous avons montrées (et allons montrer) tout au longue de ce document. Reste à savoir si le coût ajouté (pendant l'exécution du système, i.e. *runtime*) dû à cette séparation est important (?). En l'état actuel du travail, il est impossible d'évaluer ce surcoût. En effet, il n'existe pas un modèle que nous puissions utiliser pour estimer (théoriquement) ce coût. Pour une évaluation expérimentale, il nous faudrait disposer des implémentations (sans séparations) de même caractéristiques et fonctions que le nôtre pour que la comparaison soit équitable. Cependant, nous pensons que ce problème, s'il existe vraiment, est en dehors du traitement de requêtes. Il peut

bénéficier des techniques telles que l'optimisation de code [BBPH03] proposées dans les travaux sur la programmation par aspects [EFB01], la composition [GKAF01, AFGK02], etc.

### 5.6.2 Surcoûts de la surveillance

Comme nous l'avons présenté, la surveillance nécessite d'intégrer des éléments spéciaux dans l'évaluation de requête. Ces éléments susceptibles de recueillir les informations sont indispensables pour l'évaluation adaptative mais ajoutent, sans aucun doute, un coût supplémentaire à l'exécution. Cette section présente quelques résultats pratiques de notre évaluation de ce surcoût dans le but de fournir un moyen d'analyser le gain de l'adaptation.

En effet, il est très difficile de connaître le bénéfice de l'adaptation car celui-ci dépend de la "stabilité" de l'environnement d'exécution. Quand on ajoute les éléments de surveillance nécessaires pour l'adaptation si aucun changement ne se produit, la surveillance est vaine. De ce fait, la mesure du coût de la surveillance est importante et significative. Ce coût peut être considéré comme un des facteurs pour la comparaison des systèmes et des techniques sur l'évaluation adaptative. Ainsi, cette mesure donne des indications importantes pour le choix de l'adaptation. A notre connaissance, cette mesure est très peu explicitée dans la littérature et nous ne pouvons que citer les travaux réalisés dans le système DB2 [SLMK01].

Pour les besoins de cette section, les plans d'exécution de requête sont construits "à la main" et nous comparons ceux avec et sans éléments de surveillance. Nous implémentons un opérateur *Source* qui génère, d'une façon aléatoire, des items de taille variée entre 16 bytes et 2000 bytes. La cardinalité de chaque entrée des requêtes est fixée à 10000 (items). Pour l'évaluation, nous mesurons le coûts et le surcoût de chacun des opérateurs et reportons ici les résultats des opérateurs *Select*, *Join*, de *Buffer* et *Map*. Ce choix s'est basé sur les caractéristiques de ces opérateurs[2]. En effet, ces opérateurs se différencient par les traitements liés à leurs paramètres : certains nécessitent d'évaluer les prédicats de sélection sur les données en entrée (e.g. la sélection, la jointure) alors que d'autres doivent réaliser une fonction plus ou moin complexe pour construire les items du résultat. Pour chaque mesure dont la valeur est reportée dans la table 5.5, l'exécution de la requête se répète 10 fois et nous retenons les neufs mesures excepté celle de valeur la plus élevée pour calculer la moyenne.

Les mesures ont été réalisées sur un portable DELL avec une capacité de mémoire de 512MB, un CPU de 1.4GB, en utilisant le système d'exploitation Redhat 9.1, la machine virtuelle de Java JDK 1.4.

**Remarques** Nous trouvons que le surcoût dû à la mesure de la cardinalité (par un compteur) et de temps (par l'utilisation de *time-stamp*) est négligeable alors que ce n'est pas le cas pour la mesure de taille des items. L'ajout des éléments de surveillance de ce dernier type est donc critique. Cette différence importante peut s'expliquer par la différence des mécanismes utilisés pour la surveillance.

Par ailleurs, pour un même type de surveillance (cardinalité, temps, taille), les surcoûts des opérateurs différents sont différents et cette différence est parfois très importante, notamment dans le cas de la

[2]En fait, les mesures d'autres opérateurs ne sont pas présenté car leur "comportement" est similaire à un des opérateurs présentés dans ce chapitre.

| **Opérateur** | **coût de l'exécution (per item)** (en milli seconde) | **surcoût de la mesure de card.** (en %) | **surcoût de la mesure de temps** (en %) | **surcoût de la mesure de taille d'item** (en %) |
|---|---|---|---|---|
| *Select* | 0.0005 | 1.43 | 1.55 | 523.75 |
| *Join* (par boucle imbriquée) | 1.0314 | 1.94 | 2.05 | 134.69 |
| *Buffer* | 0.1233 | 1.01 | 2.47 | 171.67 |
| *Map* | 0.0007 | 1.5 | 1.73 | 235,62 |

Table 5.5: Surcoût de la surveillance

mesure de taille des items. Cette mesure dépend généralement de la taille d'item. Nous notons que le surcoût de la surveillance de la taille du résultat de sélection est *excessivement* important. Ce phénomène peut être expliqué par la proportion du coût de surveillance et du coût de la sélection. En effet, le coût de mesure d'objet en mémoire est toujours important et il ne dépend que de la taille d'objet pendant que le coût de la sélection est très petit par rapport à d'autres opérateurs. Un phénomène similaire est également observé pour l'opérateur *Map*. Il est donc déconseillé d'utiliser ces éléments de surveillance dans certains cas, notamment le cas de sélection.

Nous travaillons actuellement sur d'autres mesures, toujours concernant ces trois types de surveillance et sur ces opérateurs, en simulant une variation du coût de la réalisation des prédicats et des fonctions que sont les paramètres de ces opérateurs. Aussi, nous considérons les différentes fréquences de la surveillance. Ces études nous permettraient de mieux connaître les influences de la surveillance sur la performance de l'exécution de requête afin de fournir des indications effectives aux programmeurs.

## 5.7 Conclusion du chapitre

Dans ce chapitre, nous avons présenté la mise en œuvre de QBF, notamment les algorithmes d'optimisation, les mécanismes de surveillance et d'adaptation, nécessaires pour la construction des évaluateurs de requêtes. Ce chapitre permet, d'une part, de voir concrètement comment les techniques d'évaluation de requête sont réalisées au sein de QBF, et d'autre part, de comprendre comment utiliser une implémentation de QBF pour construire des évaluateurs de requêtes.

Nous avons évoqué la faculté de la séparation et de l'abstraction des tâches selon la spécification de QBF. Grâce à cette séparation, nous avons montré comment QBF permet l'intégration des plusieurs techniques d'évaluation de requête de manière uniforme dans le but de simplifier leur utilisation dans un seul évaluateur. Nous avons également présenté quelques évaluations quantitatives concernant le surcoût de la surveillance de l'exécution. Cela donne une idée du coût de l'adaptation et aide les programmeurs dans le choix des éléments de surveillance à utiliser dans leur évaluateur.

En reprenant notre définition de trois niveaux de l'adaptabilité (cf. section 1.2), ce chapitre concerne essentiellement l'adaptabilité statique obtenue au moment de la construction d'un évaluateur de requête.

En effet, il décrit la mise en œuvre d'une implémentation de QBF que nous avons réalisé et donne des indications concernant l'utilisation de cette *base de code* pour construire un évaluateur de requête spécifique ayant les différentes capacités d'évaluation de requête (e.g. personnalisée, adaptive, etc.). En effet, pour assurer une évaluation personnalisée et/ou adaptative, il faut déjà construire l'évaluateur ayant ces capacités. Plusieurs techniques ont été proposées dans ce but et notre analyse de ces techniques (cf. le chapitre 3) nous a conduit à la proposition de QBF.

Cependant, la personnalisation et l'adaptabilité dynamique définies dans notre approches nécessitent l'utilisation, entre autres, des mécanismes d'évaluation complexes proposés dans la littérature et que nous re-examinons dans les chapitres suivants.

# CHAPITRE 6

# Évaluation personnalisée de requêtes

## 6.1 *re*-Définition du problème

*La personnalisation est l'action de rendre personnel, d'individualiser, d'adapter à chaque client*[1]. Dans le cadre de l'évaluation de requêtes, nous définissons **la personnalisation comme étant la capacité du système d'adapter son fonctionnement répondre à une requête utilisateur ayant des contraintes spécifiques**. En effet, l'objectif de l'évaluation personnalisée est d'offrir à chaque utilisateur, voire à chaque requête utilisateur, un accès adapté aux données. D'une manière générale, un système d'évaluation personnalisée reçoit une requête composée d'une expression de besoin (i.e. les données d'intérêt) et d'une expression de contexte (i.e. les contraintes spécifiques[2]). Le système accède aux données, évalue la requête et ajuste, si nécessaire, la représentation des résultats selon le contexte.

La personnalisation suscite aujourd'hui beaucoup d'attention dans plusieurs domaines d'application modernes, tels que les systèmes d'intégration des données sur le Web ou les GRID, le commerce électronique, les bibliothèques électroniques, etc. [Bou04]. Quelques travaux récents [CK97, CK98, Bon99, RRH99, PY00, PY01, RH02, KK02, BO03, KI04] proposent des mécanismes pour répondre à certains besoins de l'évaluation personnalisée dans des systèmes spécifiques, e.g. filtrer les données, ordonner les résultats selon le degré de préférence, retourner des résultats incomplets, etc. Néanmoins, la plupart d'entre eux proposent de construire des *systèmes de personnalisation comme une "couche" au dessus des SGBD existants*. Selon une telle approche, les données interrogées sont traitées (calculées) en totalité

[1] cf. Dictionnaire, *Le nouveau petit Robert*.

[2] Notons que les contraintes peuvent être spécifiées explicitement dans la requête par l'utilisateur ou être intégrées implicitement dans la requête par le système selon un *profil utilisateur* défini antérieurement et stocké par le système.

(ou presque) par les SGBD sous-jacents avant d'être *adaptées* par la couche de personnalisation chargée de présenter au client le résultat conformément à son contexte. En conséquence, certains calculs *inutiles* pour répondre à la requête peuvent *déjà* avoir été réalisés par le SGBD.

Nous proposons d'*incorporer les traitements de personnalisation à l'intérieur du processus de traitement de requêtes dans les SGBD* afin d'éviter les calculs inutiles aussitôt que possible. Pour cela, il est nécessaire d'identifier l'impact de la personnalisation sur *chacunes* des phases du processus de traitement. Nous considérons un scénario d'interrogation de données bibliographiques qui nous sert à identifier les problèmes à résoudre et à illustrer notre solution.

**Scénario** Un système de consultation de documents offre à ses utilisateurs un accès aux informations bibliographiques (`Bib`), aux prix des documents (`Price`), et aux évaluations de ces document (`Review`). Le schéma du système est décrit comme suit :

```
Bib(id, title, author, year, booktitle,
    abstract, keyword, type, category,source)
Price(id, title, author, year, price, publisher)
Review(id, title, author, year, rate, comment)
```

Pour simplifier la présentation, nous choisissons de présenter les données sous forme relationnelle. Ainsi, les sources participantes au système ont les schémas identiques à des parties du schéma global qu'elles peuplent. Concrètement, nous considérons les sources (i) `Bib` contenant le catalogue des documents consultables ; (ii) `Price_`*xx* (e.g. `Price_1`, `Price_2`) contenant les prix des document et ayant le même schéma de `Price`; et (iii) `Rev_`*xx* (e.g. `Rev_1`, `Rev_2`) contenant les évaluation des documents et ayant le même schéma de `Rev`. Nous considérons maintenant un exemple de requête de recherche de *toutes* les informations concernant tous les documents. La requête, exprimée en SQL, est la suivante :

```
select *
from   Bib, Price, Review
where  Bib.id = Price. id
  and  Bib.id = Review.id
```

La figure 6.1(a) montre un plan possible pour la requête sur le schéma global. Le plan réécrit sur l'ensemble des sources sous-jacentes est présenté dans la figure 6.1(b).

Nous considérons maintenant quelques exemples de personnalisation de la requête décrite ci-dessus en supposant que la source `Price_3` est maintenue à jour tous les ans et que l'accès à `Rev_3` est payant. Les informations sur la dernière mise à jour (i.e. la date de mise à jour) sont disponibles pour chacune des sources. L'utilisateur *Toto* ne s'intéresse qu'aux publications du domaine de "bases de données", comme il a spécifié dans son profil d'accès (le fichier `toto.profile` : `Bib.category contains 'Database'`). D'ailleurs, il ne souhaite que des données "fraîches" (e.g. mise à jour après '1/10/2004'). Le plan d'exécution approprié à cette requête, en tenant compte de ces contraintes, pourrait être celui de la figure 6.1(i) : un prédicat de sélection est ajouté et la source `Price_3` dont la

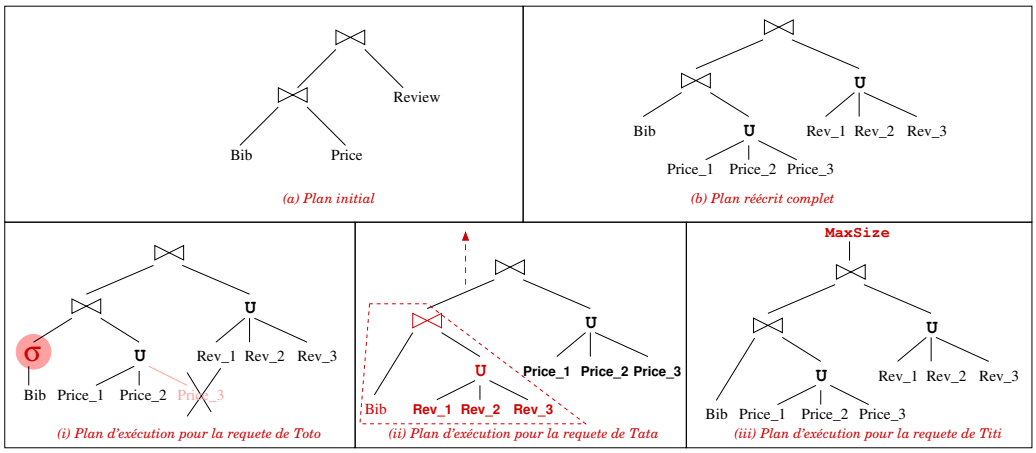



Figure 6.1: Exemple de plans de requête

date de mise à jour est avant '1/10/2004' n'est pas considérée. Un autre utilisateur *Tata* formule la même requête mais il a des *préférences* sur les données concernant la bibliographie (`Bib`) et leur évaluation (`Review`). Plus précisément, *Tata* accepte des résultats, même incomplets, si ces derniers concernent des données bibliographiques et des évaluation. L'idée est que le système lui retourne un résultat partiel en cas d'impossibilité de construction du résultat complet (e.g. `Price` n'est pas disponible). Le plan approprié pour cette de requête est celui de la figure 6.1(ii). Un autre utilisateur *Titi* veut limiter le nombre de résultats reçus à 100 et spécifie les contraintes sur le coût (économique) d'exécution de requête. La figure 6.1(iii) montre ce que pourrait être le plan d'exécution de sa requête.

Ces quelques exemples simples montrent l'influence du contexte de requête (e.g. le profil d'utilisateur, la qualité des résultats obtenus, le critère de traitement, etc.) sur la solution proposée par l'évaluateur pour évaluer la requête utilisateur, et donc la nécessité d'en tenir compte dans les différentes phases du processus de traitement de requête.

**Problèmes à résoudre** D'une manière générale, le problème d'évaluation personnalisée de requête est formulé comme suit :

> *Etant donné une requête (appelée requête personnalisée) composée d'une expression de besoin (i.e. les données d'intérêt) et d'un contexte (i.e. les contraintes spécifiques), déterminer une stratégie d'exécution optimale, i.e. un plan d'exécution de requête et des contrôles nécessaires, pour répondre à la requête donnée, i.e. retourner les données d'intérêt en satisfaisant le contexte.*

La figure 6.2 rappelle le processus du traitement de requête classique en indiquant les problèmes posés pour chacune des phases dans le cadre de la personnalisation. Ces problèmes peuvent être résumés par les questions suivantes :

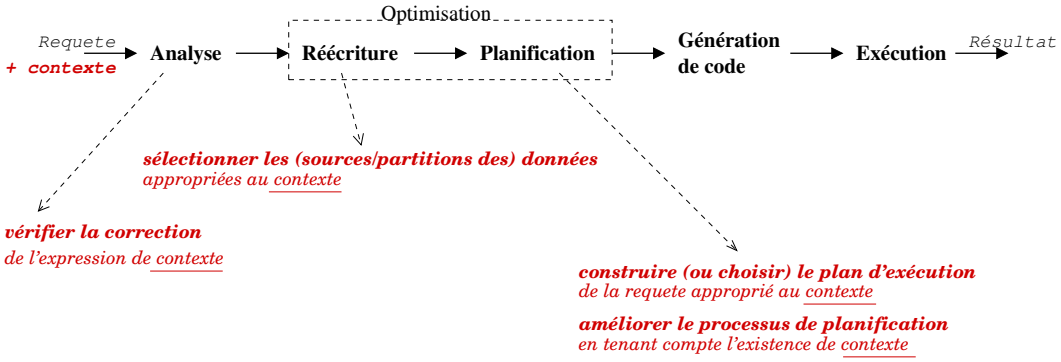



Figure 6.2: Processus de traitement de requêtes personnalisées

(1) *Que peut-on personnaliser ?*

Il s'agit de déterminer les paramètres de personnalisation pouvant être pris en compte par l'évaluateur.

(2) *Comment choisit-on les (sources de) données pertinentes pour répondre à une requête personnalisée ?*

Il s'agit de déterminer les critères pour sélectionner des sources selon les paramètres de personnalisation. Cette question concerne essentiellement la phase de la *réécriture*.

(3) *Quel est le critère de choix de plan optimal ?*

Dans les systèmes d'évaluation personnalisée, le critère de choix d'un plan optimal est souvent *multiple* (e.g. temps, coût économique, etc.). Par ailleurs, ce critère peut être différent d'une requête à une autre. Il faut alors que le système soit capable de choisir des critères d'optimisation pour chacune des requêtes en fonction de leur contexte et de trouver le plan optimal selon ces critères multiples. Cette question concerne principalement la fonction d'*estimation de coût*.

(4) *Comment planifie-t-on une requête personnalisée ?*

En présence d'un contexte de requête, il est souvent souhaitable que la solution proposée pour l'exécution de cette requête dispose de quelques propriétés particulières (e.g. être capable de produire les premiers résultats aussitôt que possible, limiter le nombre de résultats produits, etc.). Ainsi, la non-satisfaction d'(une partie) de contexte de requête d'une solution conduit à son rejet. Cela concerne, alors :

(4.1) la génération de l'*espace de recherche*, et

(4.2) la *stratégie de recherche*, et plus particulièrement l'*élimination des plans non-candidats* pendant l'exploration de l'espace de recherche.

**Plan du chapitre** La suite de ce chapitre donne des réponses à ces questions. Elle est organisée de la manière suivante : tout d'abord, nous introduisons une classification de paramètres de personnalisation (la section 6.2) ; les mécanismes à intégrer dans les différentes phases du processus de traitement de requêtes sont présentés dans la section 6.3 ; ensuite, nous évoquons quelques points importants de la mise en œuvre de ces mécanismes avec QBF (la section 6.4) ; nous comparons notre proposition avec les travaux connexes dans la section 6.5 ; enfin, la section 6.6 conclut le chapitre.

## 6.2 Classification des paramètres de personnalisation

En proposant une classification des paramètres de personnalisation, nous essayons de répondre *partiellement* à la première question (*Que peut-on personnaliser?*). Cette classification nous aide à identifier l'*impact* de la personnalisation sur l'évaluation de requête et les extensions des mécanismes de traitement nécessaires afin de pouvoir assurer une évaluation personnalisée. Par ailleurs, une telle classification aide le programmeur à définir la liste des paramètres pouvant être traités lors de l'implémentation de son évaluateur.

Dans un système d'évaluation de requêtes classique, l'utilisateur soumet une requête spécifiant les données qui l'intéressent et attend le résultat provenant du système responsable du choix d'une stratégie d'évaluation optimale et de son exécution. La personnalisation peut concerner les *données* et/ou le *traitement*. En ce qui concerne les données, il s'agit, d'une part, des données interrogées (i.e. les données pertinentes pour la réponse), et d'autre part, de leur calcul (i.e. la construction des résultats). Quant aux traitements du système, deux aspects principaux à considérer sont l'objectif du traitement (i.e. le(s) critère(s) de choix d'une stratégie optimale), et le coût effectif du traitement. Nous classons donc les paramètres de personnalisation en quatre groupes : (i) les données pertinentes (dénoté $\mathcal{PD}$, *pertinent data*), (ii) la qualité de résultats obtenus (dénoté $\mathcal{QR}$, *quality of results*), (iii) le coût d'exécution (dénoté $\mathcal{QC}$, *query cost*) et (iv) l'objectif d'optimisation (dénoté $\mathcal{OO}$, *optimization objectif*).

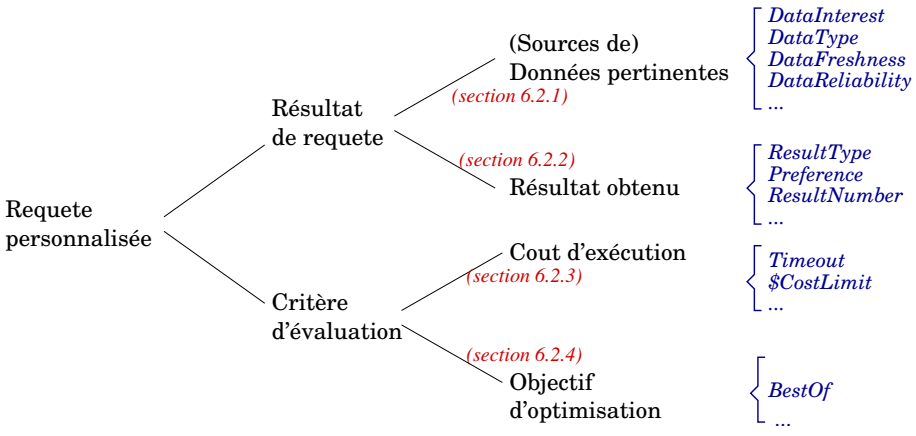



Figure 6.3: Classification de paramètres de personnalisation

La figure 6.3 montre quelques exemples de paramètres appartenant aux quatres catégories de paramètres que nous décrivons en détail ci-après.

### 6.2.1 Données pertinentes

Les paramètres de cette catégorie permettent de déterminer les données qui sont *éventuellement* pertinentes pour répondre à *une requête utilisateur*. Ces paramètres permettent au système de sélectionner les données à accéder en tenant compte des attentes de l'utilisateur vis-à-vis des résultats de la requête (voir la section 6.1 pour les exemples). Cela permet souvent de réduire le volume de données à traiter lors de l'interrogation de sources multiples, tout en répondant à la requête utilisateur.

Quelques exemples des paramètres $\mathcal{PD}$ sont : (i) *DataInterest* qui est souvent défini par un ensemble des prédicats ou même par des requêtes complexes afin de "spécifier" les données d'intérêt pour l'utilisateur. Reprenons notre exemple de la requête de *Toto*, qui ne s'intéresse qu'aux documents stockés dans des bases de données, son fichier de profil (`toto.profile`) peut contenir la ligne "*DataInterest* = `category contains 'Database'`" ; (ii) *DataFreshness* qui permet à l'utilisateur de spécifier la qualité de données interrogées vis-à-vis de leur fraîcheur. Dans notre exemple, il est exprimé par la date de mise à jour (e.g. *DataFreshness* > '1/10/2004', c'est-à-dire que les données sont mises à jour après le 1/10/2004) ; ou encore d'autres paramètres comme (*DataType*) permettant de spécifier les types de données acceptés par l'utilisateur (e.g. il ne veux que les données non-multimédia, e.g. *ppt*, *pdf*, et les données de types *mp3*, *avi*, *divX*, *mpeg4* devront être "filtrées" implicitement), etc.

La prise en compte de ces paramètres nécessite des méta-informations sur les (sources de) données afin de connaître leurs propriétés (e.g. fraîcheur, type, etc.). Elle concerne essentiellement la phase de réécriture du processus de traitement de requête (e.g. l'ajout des prédicats de sélection, de jointure implicite, etc.) et requiert l'*annotation des opérateurs* par ces propriétés, qui seront utilisées par la suite pour éliminer les données non-pertinentes.

### 6.2.2 Résultat obtenu

Cette catégorie regroupe les paramètres liés à la qualité du résultat obtenu. Ces paramètres "guident" le système dans la construction du résultat, notamment en cas d'impossibilité d'obtention du résultat (complet) tel qu'il est spécifié dans la requête utilisateur. Cela permet d'augmenter la satisfaction de l'utilisateur tout en améliorant le fonctionnement global du système.

Quelques paramètres $\mathcal{QR}$ sont : *ResultType* qui concerne la complétude des résultats obtenus, définie par la relation entre le résultat complet et le résultat obtenu. Hormis de *résultat complet* (complete), nous distinguons trois types des résultats partiels : le *résultat partiel horizontal* (horizontal), le *résultat partiel vertical* (vertical) et le *résultat partiel généralisé* (generalized). Le résultat partiel horizontal est un sous-ensemble (ayant toutes les propriétés de données mais de cardinalité inférieure) du résultat de l'évaluation de la requête (i.e. résultat complet). Le résultat partiel vertical est le résultat de l'évaluation ne tenant pas compte de toutes les propriétés des données. Le résultat partiel généralisé est la combinaison de deux derniers types. Nous omettons la définition formelle de ces types de résultats et les illustrons

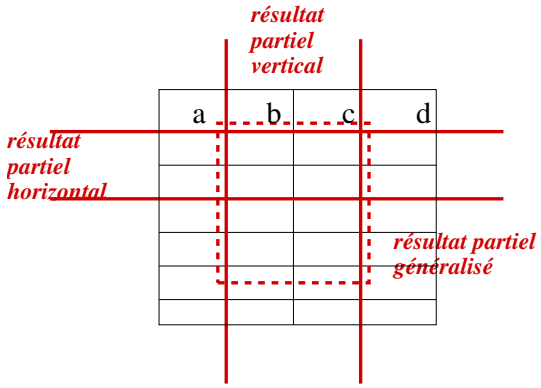



Figure 6.4: Exemple de différents types de résultats partiels

dans le contexte de données relationnelles dans la figure 6.4 ; concernant toujours la complétude du résultat, l'utilisateur peut spécifier plus précisément la *cardinalité* du résultat (*ResultNumber*) qu'il souhaite recevoir (e.g. *Titi* limite le nombre de résultats reçus à 100) ; sa *préférence* (*Preference*) sur un sous-ensemble des données interrogées (e.g. *Tata* a des préférences sur les informations bibliographiques et celles sur les critiques), etc.

Ces paramètres étant exprimés sur le résultat de la requête, ils influencent directement la construction de résultat, qui dépend du plan optimal généré, et nécessite des traitements spécifiques pendant l'exécution. Ces traitements peuvent correspondre à l'arrêt de l'exécution de la requête lors de l'obtention du n-ième élément du résultat (paramètre *ResultNumber*), au retour des données de préférence (paramètre *Preference*) en cas d'impossibilité d'évaluation de la requête globale, ou au choix d'un plan de requête permettant la production de type de résultat accepté par le client (paramètre *ResultType*).

### 6.2.3 Coût d'exécution

Il s'agit de paramètres définissant la limite du "budget" attribué pour la requête utilisateur par l'utilisateur. Effectivement, l'évaluation de requête sur des sources multiples peut être coûteuse en terme de temps d'exécution, de coût économique, etc. L'accès à certaines sources, ainsi que les services offerts par des sites, peuvent être payants. Lorsque le coût effectif d'exécution de requête dépasse le budget prévu, le système peut arrêter la requête afin de mieux satisfaire les besoins de l'utilisateur tout en "évitant" les surcharges inutiles dans le système, notamment dans le cas multi-utilisateurs. Au-delà, le système peut aussi s'appuyer sur la valeur de ces paramètres pour proposer une solution adaptée au mieux à la requête personnalisée (e.g. retourner un nombre maximal de résultats dans un délais limité ou pour un coût économique limité).

Quelques exemples de paramètres sont : (i) *Timeout* qui spécifie la durée maximale d'évaluation ; (ii) *$CostLimit* qui spécifie la limite de budget économique pour l'exécution de requête. Ces paramètres sont de type numérique (e.g. entier, réel, etc.)

La prise en compte de ces paramètres concerne, d'une part, la génération du plan d'exécution optimal, et d'autre part, la surveillance et l'adaptation de l'exécution lorsque les contraintes ne sont plus satisfaites. Une adaptation simple consisterait à arrêter l'exécution de la requête en cours.

### 6.2.4 Objectif d'optimisation

Comme nous l'avons présenté, le coût de l'évaluation peut se composer de plusieurs facteurs *"visibles"* pour l'utilisateur alors que l'objectif d'optimisation est souvent multiple et peut être différent d'une requête à l'autre. Ainsi, l'utilisateur peut vouloir expliciter le critère d'optimisation de sa requête. D'ailleurs, la présence de plusieurs critères d'optimisation (e.g. temps, coût économique) amène souvent à déterminer un compromis (*trade-off*) entre ces critères. Un tel compromis peut être défini par le client ou par le système lui-même.

Pour spécifier l'objectif d'optimisation de requête, l'utilisateur peut spécifier le paramètre *BestOf*. Considérons par exemple les deux facteurs de coût que sont le temps et le coût économique, ce paramètre peut avoir une des valeurs suivantes : (i) $T_{first}$ (i.e. l'objectif d'optimisation est de retourner les premiers résultats dans le meilleur délai) ; (ii) $T_{all}$ (i.e. l'objectif est de retourner tous les résultats dans un meilleur délais) ; (iii) $\$C_{first}$ ; (iv) $\$C_{all}$ (i.e. l'objectif est de retourner les premiers ou tous les résultats à un coût minimal) ; ou encore (v) une fonction $f$ définissant le compromis entre des facteurs de coût.

Il faut donc que le module d'estimation de coût soit capable de calculer le coût de l'exécution de requête selon ces différents facteurs et que le critère de choix soit adapté à la valeur du paramètre *BestOf*.

Le tableau 6.1 résume les classes des paramètres personnalisation en montrant l'influence de ces paramètres sur les différentes fonctions du traitement de requête que nous présentons dans la section suivante.

## 6.3 Mécanismes d'évaluation personnalisée

Cette section tente de donner une réponse aux questions de (2) à (4) relevées au début du chapitre (cf. la section 6.1). Concrètement, nous évoquons les extensions des différentes fonctions du traitement de requêtes influencées par la personnalisation : la sélection des sources pertinentes (la section 6.3.1), la génération de l'espace de recherche (la section 6.3.2), l'estimation de coût (la section 6.3.3), l'élimination des plans non-candidats (la section 6.3.4). Pour chacune de ces fonctions, nous abordons d'une manière générale l'influence de la personnalisation, puis les mécanismes concrets pour les paramètres décrits ci-dessus (cf. la table 6.1) et l'extension de cette fonction. Nous montrons également comment ces différents mécanismes s'intègrent dans une solution globale, c'est-à-dire un algorithme d'optimisation, afin d'assurer une évaluation personnalisée de requête (la sections 6.3.5).

| Classe | Paramètre | Fonction du traitement de requête concernée | |
|---|---|---|---|
| **Données pertinentes** ($\mathcal{PD}$) | *DataInterest*, *DataFreshness*, *DataType*,... | - sélection des sources | *section 6.3.1* |
| **Résultat obtenu** ($\mathcal{QR}$) | *ResultNumber*, *Preference*, *ResultType*,... | - génération d'espace de recherche | *section 6.3.2* |
| **Coût d'exécution** ($\mathcal{QC}$) | *Timeout*, *$CostLimit*,... | - estimation de coût<br>- élimination des plans non-candidats | *section 6.3.3*<br>*section 6.3.4* |
| **Objectif d'optimisation** ($\mathcal{OO}$) | *BestOf* | - estimation de coût<br>- élimination des plans non-candidats | *section 6.3.3*<br>*section 6.3.4* |

Table 6.1: Les paramètres de la personnalisation et les fonctions du traitement de requête

### 6.3.1 Sélection des (sources de) données

Etant donné une requête sur le schéma logique d'application, il est souvent nécessaire de procéder à une phase susceptible de trouver les sources/partitions de données pertinentes pour répondre à la requête entrée. Dans les systèmes classiques, cette phase consiste en la réécriture de la requête, qui s'appuie sur les déclarations statiques comme la définition des schémas, les correspondances sémantiques entre les schémas ou encore les contraintes d'intégrité dans les sources ou entre les sources. Elle ne tient pas compte des caractéristiques de (l'implémentation de) données interrogées (e.g. taille de données, coût d'évaluation, etc.) qui, dans le cas de l'évaluation personnalisée, sont importantes pour satisfaire les paramètres de personnalisation (i.e. contexte de requête), notamment ceux de type $\mathcal{PD}$ dans notre classification. L'objectif de la fonction de sélection de sources est de restreindre le plus tôt possible et autant que possible les données ne satisfaisant pas les contraintes de la requête.

Pour le paramètre *DataInterest*, il s'agit d'*incorporer* ces prédicats dans la requête utilisateur. Cela se fait par l'ajout d'opérateurs de sélection, de jointure correspondants ou dans le cas le plus complexe, elle peut s'appuyer sur les techniques de réécriture proposées dans la littérature [GMHI$^+$95, LRO96, PL00]. Quand il s'agit des contraintes sur la fraîcheur de données (*DataFreshness*), les types de données (*DataType*), il est nécessaire que ces informations se trouvent dans les méta-informations faisant partie de la description des sources. Dans un environnement hétérogène, distribué et dynamique, ces informations peuvent être obtenues par l'application des mécanismes proposés dans [ROH96, GRZZ00, CGM03]. Pour l'utilisation de ces informations, nous proposons de considérer un *plan de requête annoté* par

les propriétés correspondantes à des paramètres de personnalisation. Le système accède aux méta-informations, contacte les sources, si nécessaire, pour annoter le plan. Enfin, le système procède à une phase de *vérification* pour éliminer toutes les sources (e.g. les feuilles du plan) ne satisfaisant pas les paramètres $\mathcal{PD}$ de la requête.

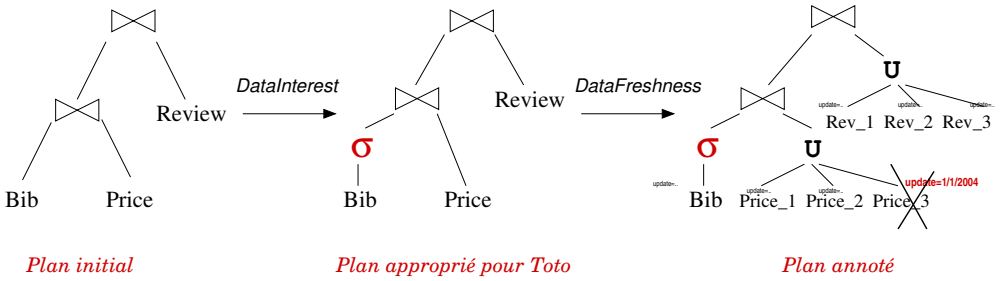



Figure 6.5: Exemple de la sélection de données pertinentes

La figure 6.5 montre l'exemple de la requête de *Toto* qui ne s'intéresse qu'aux documents stockés dans des bases de données, et qui souhaite que les données du résultat de sa requête soit à jour après le '1/10/2004' (cf. la section 6.1). Dans une première étape, la condition de filtrage sur `Bib` est ajoutée pour sélectionner seulement les documents stockés dans des bases de données. Après une réécriture permettant d'identifier les sources correspondantes, le plan est annoté par la propriété *update* représentant la date de mise à jour. Se basant sur cette annotation, le système élimine tous les opérateurs primitifs (i.e. les feuilles) qui ne satisfont pas les paramètres de la requête. Dans cet exemple, la source `Price_3` n'est pas pertinente.

### 6.3.2 Génération de l'espace de recherche

D'une manière générale, l'espace de recherche contient *tous* les plans équivalents pour répondre à une requête donnée. Ces plans se différencient par l'ordre des opérateurs à exécuter, notamment celui des opérateurs de jointure, par le niveau de parallélisme de l'exécution et par les algorithmes implémentant chacun des opérateurs. L'optimiseur explore cet espace afin de choisir le plan optimal. Puisque l'espace de recherche est souvent grand, son exploration peut être coûteuse. Les algorithmes d'optimisation peuvent donc utiliser les heuristiques pour éviter le *parcours exhaustif* de l'espace de recherche ou, plus précisément, pour limiter le nombre de plans à examiner. Dans ce même esprit, nous proposons de considérer les paramètres de personnalisation comme des critères pour réduire l'espace de recherche considéré par l'optimiseur, dans le but d'améliorer sa performance. La suite de cette section analyse concrètement l'impact des paramètres présentés dans la table 6.1 sur l'espace de recherche effectif de la requête et montre, d'une manière générale, l'influence de la personnalisation sur la génération de l'espace de recherche.

Le premier paramètre considéré est *ResultNumber* ayant pour but d'expliciter le nombre de résultats

reçus. Pour l'assurer, nous introduisons un nouvel opérateur appelé *MaxSize*. Cet opérateur a un seul paramètre *max* définissant le nombre maximal de résultats souhaité. L'opérateur produit en sortie les données d'entrée inchangées jusqu'à l'obtention du *max*-ième élément. Au cas où l'utilisateur souhaite recevoir les résultats ordonnés, l'opérateur *Sort* est appliqué sur le flux d'entrée de l'opérateur *MaxSize*. Cette requête est similaire à une requête *Top N* [CK97, IAE03] et les différentes stratégies d'optimisation présentées dans la section 3.3.1 peuvent être utilisées pour déterminer la (les) position(s) de l'opérateur *MaxSize* dans le plan. Par ailleurs, dans le contexte de sources multiples, où les données concernant un même sujet peuvent se trouver dans plusieurs sources, ce paramètre peut influencer également la priorité d'accès à ces sources.

En définissant le paramètre *ResultType*, l'utilisateur indique le type de résultat (souvent, *partiel*) qu'il peut gérer en laissant au système de décider des résultats produits. Comme nous l'avons présenté, le type de résultats est défini par la relation entre les résultats acceptés et les résultats complets. Il peut avoir une des valeurs suivantes : *complete*, *vertical*, *horizontal*, *generalized* (cf. la section 6.2). Afin de garantir la possibilité de retourner les résultats de type approprié, le plan retenu par l'optimiseur doit donc disposer de quelques caractéristiques spécifiques. En tirant profit des caractéristiques exigées, l'optimiseur considère seulement les plans le satisfaisant, permettant de réduire l'espace de recherche effectif. En s'appuyant sur le modèle itérateur pour l'exécution de requête, les résultats partiels horizontaux sont obtenus naturellement, du moins si les opérateurs exécutés sont non-bloquants. En effet, dans ce cas, les données sont traitées item par item et les résultats sont produits aussitôt que possible. Par conséquent, seuls les plans constitués par les algorithmes non-bloquants sont considérés et l'espace de recherche se réduit à ce que nous appelons l'*espace non-bloquant*. Lorsque l'utilisateur souhaite recevoir les résultats partiels verticaux, il est nécessaire de considérer des mécanismes capables de retourner les résultats des sous-plans à l'utilisateur. Ces mécanismes consistent souvent à acheminer un ou quelques flux vers le client d'une manière irrégulière. La réalisation de ces mécanismes peut se faire au niveau de l'implémentation des opérateurs en considérant les algorithmes spécifiques et/ou par des opérateurs spécifiques. Nous supposons ici que ces opérateurs existent et qualifions l'espace composé des plans capables de retourner les résultats partiels verticaux l'*espace flexible* (en comparaison avec "non-bloquant"). La présentation détaillée de ces opérateurs est reportée en chapitre prochain (la section 7.2 du chapitre 7). Dans le cas où l'utilisateur accepte les résultats partiels généralisés (i.e. correspond à la valeur *generalized*), l'optimiseur considère alors seulement les plans flexibles et non-bloquants. Dans le cas où l'utilisateur n'accepte que les résultats complets, nous utilisons un opérateur *Buffer* à la racine du plan de requête pour stocker l'intégralité du résultat de la requête, exécutée dans le modèle itérateur, avant de le retourner au client. En se basant sur ces caractéristiques, le système considère les différents espaces de recherche et, au moment de l'optimisation de requête, le système génère l'espace approprié selon les paramètres de la requête entrée. En considérant nos paramètres de personnalisation, nous pouvons distinguer les espaces suivants :

(i) l'**espace non-bloquant** : qui contient seulement les plans permettant la production du premier résultat aussitôt que possible. Concrètement, la génération de cet espace considère seulement les algorithmes non-bloquants.

(ii) l'**espace flexible** : qui contient les plans permettant la production des résultats partiels, notamment les résultats verticaux. Concrètement, la génération de cet espace considère les implémentations autorisant l'acheminement irrégulier des flux de données vers le client dans certaines conditions.

Avant d'illustrer l'utilisation de ces caractéristiques, nous considérons le dernier paramètre qu'est la *Preference*. Ce paramètre permet à l'utilisateur de spécifier les données d'intérêt parmi les données de la requête. Comme nous l'avons présenté, ce paramètre contient la liste des sources préférées par l'utilisateur. Le système essaie de retourner les résultats le concernant (i.e. résultats partiels verticaux) dans le cas d'impossibilité de retour des résultats complets. Pour cela, nous proposons de *regrouper les sources selon leur préférence*. L'idée principale est de construire le(s) sous-plan(s) de taille maximale pour les sources préférées, l'optimisation est réalisée pour chaque sous-plan, puis le système construit le plan global en considérant ces sous-plans. Le principe de l'algorithme est le suivant :

**Entrée** la requête de jointure entre les relations $\{R_i\}$
liste des sources préférées : *Preference* = $\{R_j\}$ où $\{R_j\} \subset \{R_i\}$

**Sortie** plan d'exécution de la requête en entrée

(i) considérer l'ensemble de plans préférés $\{R_j\}$, identifier les ensembles maximaux des relations *joignables*. Les relations joignables sont les relations dont les conditions de jointure font partie de la requête utilisateur. Ces ensembles sont dénotés $\{P_k\}$ dont chaque $P_k$ représente une requête de jointure entre des relations joignables identifiées précédemment.

(ii) Pour chaque $P_k$, appeler l'optimiseur pour générer le plan d'exécution de $P_k$. Dès lors, le plan $P_k$ est considéré *stable*, i.e. il ne peut pas être changé lors de l'optimisation de plan plus large dont il fait partie (aucune optimisation peut être appliquée à $P_k$).

(iii) considérer l'ensemble des sous-plans identifiés dans l'étape précédent et les relations non-préférées ($\{R_i\} \setminus \{R_j\} \cup \{P_k\}$), appeler l'optimiseur pour générer le plan d'exécution de requête initiale en considérant l'espace flexible, i.e. les algorithmes des opérateurs, notamment la jointure, permettant le retour des résultats partiels concernant les sources de préférence.

Notons qu'à ce stade, l'optimiseur considère les $P_k$ comme plans primitifs et aucune optimisation n'est réalisée sur ces sous-plans.

L'objectif de cet algorithme est de réduire l'espace de recherche d'une requête donnée en favorisant la production des résultats partiels concernant les données de préférence. En effet, le regroupement des entrées de la requête pour appliquer l'optimisation en plusieurs niveaux[3] fait que le nombre de combinaisons possibles entre les relations est réduit et donc le nombre de plans à examiner l'est aussi. La réduction de la taille de l'espace de recherche généré améliore, sans aucun doute, l'efficacité de l'optimisation et donc de l'évaluation de requête en général.

[3]Les sous-requêtes $P_k$ sont optimisés indépendants les unes des autres avant que leur combinaison soit optimisée.

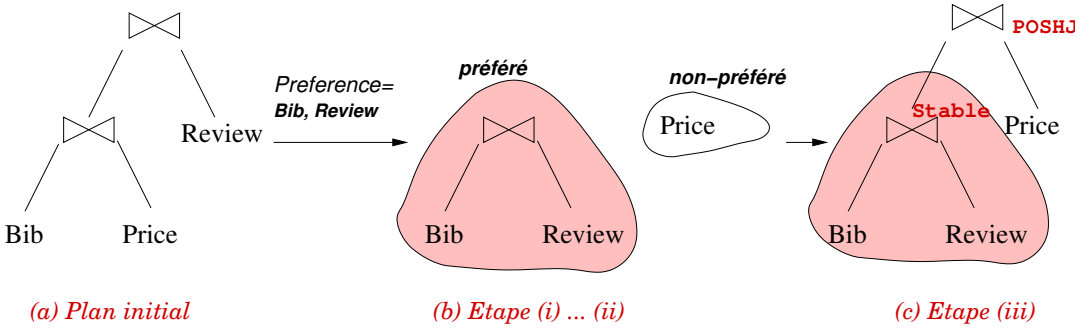



Figure 6.6: Exemple de la génération de plan

Nous avons maintenant tous les "éléments" nécessaires pour expliquer le choix du plan optimal de la requête de *Tata* dans notre scénario présenté dans l'introduction du chapitre (cf. la setion 6.1). La figure 6.6 résume le processus de génération de plan de requête. Rappelons que l'utilisateur *Tata* a des préférences sur les données concernant `Bib` et `Review`. Pour simplifier la présentation, nous omettons la sélection des sources que nous avons présentée dans la section précédente (cf. la figure 6.5). Dans la première étape, l'algorithme identifie l'ensemble maximal des relations joignables que sont `Bib` et `Review` dans notre exemple et le plan d'exécution est généré durant la deuxième étape (voir la figure 6.6(b)). Enfin, les plans de requête complets sont générés en considérant les plans de sous-requêtes générés par l'étape précédente et seulement les algorithmes permettant la production de résultats partiels. Dans cet exemple, nous considérons l'algorithme *POSHJ* (*Progressive Outer Symmetric Hash Join*) pour la production des résultats partiels, et que nous présenterons dans le chapitre 7.

### 6.3.3 Estimation de (facteurs de) coût

Le choix du plan optimal s'appuie souvent sur le coût (estimé) de l'exécution de requête, qui concerne directement l'objectif de l'optimisation. Les deux objectifs d'optimisation les plus considérés dans les SGBD sont de *minimiser le coût de requête* ou de *maximiser le débit de production de résultats* [OV99].

Dans le cas de l'évaluation personnalisée de requête, l'optimisation *mono-objectif* n'est plus adaptée. En effet, dans ce contexte, le coût d'exécution de requête dépend non seulement du système local[4] mais aussi des systèmes sous-jacents qui peuvent disposer de mesures de coût différentes, e.g. le temps d'exécution, le coût économique. Ainsi, ce critère peut être différent d'une requête à l'autre selon le contexte de requête, e.g. quelques utilisateurs veulent optimiser le temps d'exécution pendant que d'autres veulent optimiser le coût économique de l'exécution ou encore une combinaison de ces deux, etc. Par conséquent, il faut que l'évaluateur de requêtes soit capable d'estimer le coût d'une requête en différentes mesures (e.g. temps, coût économique, etc.). Ainsi, en cas de présence de plusieurs facteurs de coût, il

[4]Par système local, nous entendons le système qui reçoit la requête et qui coordonne d'autres systèmes, si nécessaire, pour répondre à la requête donnée.

nécessite une fonction de coût global permettant la comparaison de coût composé de plusieurs facteurs.

Il faut donc pouvoir estimer les facteurs de coût d'une manière indépendante, puis les combiner par une fonction de coût globale, si cette dernière existe. Par ailleurs, il serait avantageux de maintenir les informations concernant les facteurs de coût élémentaires ainsi que le coût global d'une manière explicite, dans le but de les utiliser à n'importe quel moment du processus d'évaluation.

Comme mentionné précédemment, dans le contexte de requête personnalisée, les facteurs de coût sont multiples, lesquels dépendent du système et de la requête. Il est alors impossible de mentionner, en dehors d'un système (une application) concret(e), tous les facteurs de coût, ainsi qu'un modèle de coût global. Ce que nous présentons ci-après ne sont que des exemples de l'estimation de coût dans ce contexte.En considérant les paramètres présentés dans la section 6.2, les facteurs de coût suivants peuvent être considérés :

- la cardinalité de résultat (`Card`) qui correspond au nombre d'items produits par un opérateur (ou requête).

  Ce facteur est estimé en utilisant les informations sur la sélectivité, la distribution de valeur, etc. comme présenté dans [Ioa00].

- le temps de production de premier résultat ($T_{first}$) qui correspond au temps découlé depuis le lancement de la requête jusqu'au retour du premier élément de résultat.

  En s'appuyant sur le modèle itérateur, le temps de production du premier résultat peut être considéré *approximativement* comme le temps de réalisation de la phase `open`.

- le temps de production de tous les résultats ($T_{last}$) qui correspond au temps total de l'exécution.

  En s'appuyant sur le modèle itérateur, il peut être considéré *approximativement* comme le temps jusqu'à la fin de réalisation de la phase `close`.

- le coût économique pour produire le premier résultat ($\$C_{first}$) qui correspond au coût d'exécution jusqu'au retour du premier item de résultat.

  Généralement, il est la somme du coût de connexion et le calcul du premier résultat et peut être approximatif au coût de connexion.

- le coût économique pour produire tous les résultats ($\$C_{last}$) qui correspond au coût d'exécution jusqu'au retour du dernier item de résultat.

  Généralement, il est la somme du coût de connexion et le calcul des résultats.

Le choix d'un plan optimal peut se baser sur les critères individuels et/ou sur l'ensemble de ces critères ce qui amène souvent à déterminer un compromis (*trade-off*) entre ces critères. Plus précisément, il revient souvent à définir un modèle de coût global, plus ou moins complexe, basé sur les facteurs élémentaires. Un moyen de définir ce compromis est l'utilisation d'une fonction pondérée :

$$f() = \Sigma(w_i * c_i)$$

où $c_i$ est un composant de coût et $w_i$ est le poids du composant de coût correspondant que peut spécifier le client ou le programmeur d'application. De nouveaux facteurs de coût peuvent être ajoutés et combinés avec d'autres facteurs de coût dans la fonction globale.

### 6.3.4 Elimination des plans non-candidats

Dans une évaluation personnalisée de requête, le plan optimal choisi par le système devra satisfaire les contraintes définies par les paramètres de personnalisation, tels que la limitation de temps, de coût, etc. Pendant la génération de l'espace de recherche, tous les plans énumérés devront être vérifiés au regard de ces contraintes et seulement ceux satisfaisant les contraintes sont gardés pour les traitements ultérieurs. En conséquence, les plans non-candidats sont éliminés aussitôt possible.

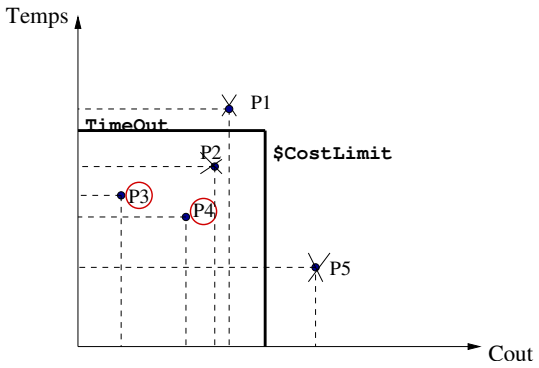



Figure 6.7: Exemple de la relation entre l'espace de recherche et l'espace de la personnalisation

Considérons un exemple de requête dont le coût est exprimé dans l'espace à deux dimensions de temps et de coût économique. La figure 6.7 présente quelques exemples de plans (dénotés *P1*, *P2*, *P3*, *P4*, *P5*) dans cet espace. Nous supposons que l'utilisateur spécifie deux valeurs maximales de temps et de coût économique alloués pour l'exécution de sa requête, les plans *P1* et *P5* se trouvent en dehors de l'espace et sont éliminés sans être évalués complètement.

Dès lors, chaque dimension de coût des plans dans l'espace est comparée individuellement afin d'éliminer tous les *plans dominés*. Un plan domine un autre plan si le coût de ce premier plan est supérieur à celui de deuxième plan dans toutes ses dimensions [PY01]. Reprenons l'exemple présenté dans la figure 6.7, le plan *P2* est éliminé car il domine *P3* et *P4* qui, quant à eux, sont incomparables dans l'ensemble de dimensions.

Il ne reste donc que les plans incomparables dans l'espace personnalisé. Ces plans se comparent alors par la fonction de coût global.

### 6.3.5 Algorithme d'optimisation

Les cinq sections précédentes ont apportées différents éléments de réponse à la question "*Comment optimiser une requête en présence des contraintes spécifiques ?*". Cette section reprend ces éléments et les intègre dans un algorithme d'optimisation. Le principe de cet algorithme est décrit comme suit :

**Entrée** Une requête de jointure des relations $R_i$
l'ensemble des paramètres de personnalisation ($pd_i$, $qr_j$,$qc_k$,$oo_p$) où $pd_i \in \mathcal{PD}$, $qr_j \in \mathcal{QR}$, $qc_k \in \mathcal{QC}$ et $oo_p \in \mathcal{OO}$.

**Sortie** Plan d'exécution de la requête

(1) réécrire la requête en fonction des sources sous-jacentes

(2) sélectionner les sources pertinentes en considérant les paramètres $pd_i$ (cf. la section 6.3.1)

(3) générer l'espace de recherche en considérant les paramètres $qr_j$ (cf. la section 6.3.2)

(3.1) identifier sous-plans à optimiser, si nécessaire (e.g. le cas de paramètre *Preference*)

(3.1) déterminer (le type de) l'espace de recherche pour chacun des sous-plans et pour le plan global.

(4) examiner le plan (partiel) de l'espace en éliminant (cf. la section 6.3.4)

(4.1) ceux qui n'appartient pas à l'espace personnalisé déterminé par les paramètre $qc_k$

(4.2) ceux qui sont non-dominé en considérant les paramètres $oo_p$

(5) sélectionner le plan optimal en basant sur l'estimation de coût global.

Intuitivement, nous trouvons que les étapes (1) et (2) concernent principalement les déclarations statiques dans le système et les techniques de réécriture classiques, telles que celles proposées dans les approches GaV et LaV, qui peuvent être utilisées. Les étapes de (3) à (5) correspondent à l'étape de planification dans l'optimisation classique. Elles correspondent donc à étendre des algorithmes d'optimisation ( e.g. l'algorithme de programmation dynamique dans System R, l'algorithme basé sur les règles de transformation dans Volcano),etc. selon les points présentés dans la section précédente.

## 6.4 Mise en œuvre avec QBF

Cette section présente les points importants de la mise en œuvre des techniques d'évaluation personnalisée avec QBF. Les composants de QBF les plus concernés sont le ContextManager et le PlanManager susceptible de fournir respectivement les mécanismes pour traiter les paramètres de la personnalisation (considérés comme contexte de requête) et pour optimiser la requête (en couvrant les trois aspects principaux que sont la stratégie de recherche, l'espace de recherche et l'estimation de coût).

### 6.4.1 Gestion de contextes de requête

Avec QBF, les contextes de requête sont gérés par le composant ContextManager responsable de la compilation de la partie de contexte de requête précédant la phase d'optimisation du plan de requête. L'objectif de la compilation de contexte de requête est, d'une part, d'ajouter les opérateurs susceptibles de contrôler l'exécution de requête afin de satisfaire les contraintes posées par le contexte et, d'autre part, de choisir les composants appropriés pour l'optimisation de requête personnalisée.

Selon les paramètres, un choix de stratégie d'optimisation, représentée par OptStrategy, est fait. Ce choix peut concerner l'annotation de plan (Annotator, e.g. les propriétés concernant le temps d'exécution, la cardinalité, la préférence, etc.), la génération de l'espace physique (Translator, e.g. comportant des opérations de translation logique-physique permettant de générer les plans non-bloquants), la génération de l'espace logique (Transformer, e.g. comportant des opérations de transformation logique-logique permettant de propager les opérateurs comme *MaxSize*, *TimeOut* selon les stratégies différentes). Cela permet d'identifier une stratégie d'optimisation appropriée à la requête en entrée.

### 6.4.2 Génération de plan de requête

Le choix de la stratégie d'optimsation fait par le gestionnaire de contextes permet au composant *PlanManager* de créer une stratégie d'optimisation appropriée à la requête personnalisée à traiter. Cela consiste à instancier un Planner approprié avec un Transformer et un Translator comportant des opérations d'optimisations appropriées, et un Annotator permettant de "calculer" les annotations appropriées pour l'exécution de la requête. La génération de plan optimal de requête est assurée par le composant PlanManager qui coordonne ces composants Planner, Transformer, Translator et Annotator, permettant de choisir le meilleur plan exécution tout en tenant en considération les paramètres de contexte.

**Discussion** Nous venons d'évoquer la façon par laquelle, les composants de QBF tiennent compte les requêtes personnalisées. Il s'agit, principalement, de déterminer les éléments appropriés de l'optimisation, à savoir l'espace de recherche, l'estimation de coût et la stratégie de recherche si nécessaire. Il est donc nécessaire de pouvoir supporter, plusieurs stratégies d'optimisation afin de pouvoir choisir une stratégie la plus adaptée au contexte de requête. Avec QBF, le choix d'une stratégie d'optimisation est relativement aisé car tous les éléments de l'optimisation sont séparés et exportent une interface bien définie que d'autres composants utilisent pour interagir avec lui.

## 6.5 Travaux connexes

Le résultat présenté dans ce chapitre contribue à l'évaluation personnalisée de requêtes. Concrètement, l'approche proposée vise à reconsidérer le processus de traitement de requêtes, phase par phase, afin d'identifier les influences de la personnalisation sur chacune des phases. Cela conduit par la suite à déterminer les extensions de ces phases dans le but d'améliorer le fonctionnement global du système ou plus concrètement de l'optimiseur. Dans son ensemble, la proposition va vers le but de fournir la person-

nalisation *systématique* dans les évaluateurs de requêtes. Par le terme "systématique", nous entendons la capacité de prendre en compte des besoins de la personnalisation dans chacune des fonctionnalités de l'évaluateur de requête[5] ainsi que la faculté d'étendre le spectre de la personnalisation. A notre connaissance, il n'existe à ce jour aucun travail ayant cet objectif. Quelques travaux récents proposent des mécanismes pour répondre seulement à certains problèmes de l'évaluation personnalisée dans des domaines d'application spécifiques. Dans la suite de cette section, nous essayons de situer ces travaux dans notre approche et les comparons avec les mécanismes proposés dans ce chapitre lorsque c'est possible[6].

**Requête *TopN* [CK97, CK98]** L'objectif est de retourner seulement les *N* premiers résultats de la requête d'utilisateur. Les résultats peuvent être ordonnées selon une fonction de tri si cette dernière existe. Pour cela, les auteurs proposent un nouvel opérateur logique *Stop* avec différentes implémentations *Scan-Stop* et *Sort-Stop* pour limiter le nombre de résultats produits.

Ces travaux concernent le paramètre *ResultNumber* dans notre travail et les mécanismes d'optimisation proposés dans [CK97, CK98] peuvent être intégrés dans notre canevas comme opérations d'optimisation. L'opérateur *Stop* est similaire à l'opérateur *MaxSize* considéré dans notre travail et l'opérateur *Sort* dans le cas de tri. Nous séparons ces deux opérateurs pour définir clairement la sémantique des opérateurs et faciliter leur utilisation dans les contextes différents.

**DISCO [BT98]** En ce qui concerne la personnalisation, les caractéristiques les plus importantes dans DISCO sont la capacité de générer les requêtes parachutes selon l'indication fournie par l'utilisateur et la capacité d'optimisation de requête sous contrainte, au cas où l'utilisateur fournirait les requêtes parachutes à priori. Dans les deux cas, l'évaluateur de DISCO permet de retourner le résultat partiel qui intéresse l'utilisateur. Les mécanismes proposés dans DISCO s'appuient, d'une part sur la matérialisation des données, et d'autre part sur la réécriture de la requête d'entrée en utilisant les relations temporaires matérialisées (cf. la section 3.3.2).

Le problème abordé dans DISCO correspond aux paramètres de type $\mathcal{QR}$ dans notre travail, notamment les paramètres *ResultType*, comparable avec le premier cas, et *Preference*, comparable avec le deuxième cas dans DISCO, mais les solutions proposées se différencient sur plusieurs points. Premièrement, nous ne considérons pas la matérialisation explicite des données mais plutôt le *plan partiellement évalué*. L'avantage de notre approche est que l'exécution de la requête est continue. En effet, dans DISCO l'indisponibilité des sources ne peut être détectée qu'au début de l'évaluation de la requête. Lorsqu'il existe une ou plusieurs source(s) indisponible(s), le système procède à une phase de matérialisation et puis à une phase d'extraction de données des relations matérialisées. Le retour de résultats partiels ne se fait qu'après la matérialisation, et à condition que les sources indisponibles soient détectées dès début de l'évaluation de requête. En s'appuyant sur le modèle itérateur, notre solution permet de retourner les résultats au fur et à mesure (bien sûr, si l'utilisateur les accepte) selon la disponibilité

[5] A contrario, la personnalisation peut être traité par une "couche" construite au-dessus de l'évaluateur.

[6] C'est le cas où les exemples de paramètres que nous avons considérés dans les sections précédentes correspondent au problème traité par le travail en question.

des sources pendant l'exécution. Par contre, il faut dire que nous devons modifier et/ou implémenter quelques opérateurs de requête, alors que tous les opérateurs dans DISCO sont plutôt classiques[7].

***Preference SQL* [KK02]** L'idée de ce système est de prendre en compte des contraintes floues de l'utilisateur concernant les conditions de sélection ou encore, les valeurs des attributs. En fait, les requêtes dans *Preference SQL* sont exprimées dans un langage SQL étendu. Elles peuvent être comparables à des requêtes *Top N* mais son expression de tri est plus complexe et autorise les contraintes floues. Par ailleurs, le système *Preference SQL* est considéré comme un pré-processeur et construit au-dessus d'un SGBD commercial.

Dans notre travail, les paramètres de personnalisation définissent les contraints *exactes* de l'utilisateur. Il serait intéressant de voir comment on peut relaxer ces contraintes et/ou étendre la définition des paramètres pour autoriser les contraintes floues.

**[KI04, KI05]** Un travail tout nouveau de G. Koutrika et al. proposent un modèle de représentation des préférences et des algorithmes pour les intégrer dans la requête utilisateur. La proposition consiste à enrichir la requête avec le profil utilisateur afin que l'évaluation de requête tienne compte de ses préférences. Leur système est conçu comme étant une couche au dessus d'un SGBD.

Il serait intéressant de voir si un tel modèle peut être intégré dans notre canevas pour enrichir et faciliter la représentation de contexte de requête. Cela est également à considérer pour les travaux futurs.

## 6.6 Conclusion du chapitre

Dans ce chapitre, nous avons analysé les problèmes autour de l'évaluation personnalisée de requête et proposé des mécanismes à intégrer dans le processus de traitement de requête. La présentation du chapitre met en évidence l'influence de la personnalisation sur les différents aspects de l'évaluation de requêtes, et montre également leur mise en œuvre avec QBF.

Nous avons également montré à travers des exemples comment les paramètres de la personnalisation sont pris en compte et comment les fonctions de l'optimisation sont "configurées" pour offrir une stratégie appropriée à une requête personnalisée d'entrée. En retour, l'analyse de la mise en œuvre des mécanismes d'évaluation personnalisée montre la nécessité de support des plusieurs stratégies de traitement de requête dans un évaluateur de requêtes et les bénéfices de la séparation de code dans QBF face à l'évaluation personnalisée de requête.

Nous pouvons noter que, l'évaluation personnalisée que présente ce chapitre est statique. Plus précisément, les mécanismes proposés ne concernent que le choix de la stratégie d'optimisation et donc du plan optimal à la compilation de requête, en supposant que toutes les estimations sont correctes, et permettent de déterminer d'un plan optimal pour un contexte donné. Pour la même raison que pour l'optimisation classique, cette hypothèse est très contraignante dans le contexte de système multi sources

[7] C'est au moins ce que P. Bonnet a écrit dans sa thèse [Bon99].

où l'environnement d'exécution est souvent imprévisible. L'utilisateur formule la requête avec des contraintes spécifiques, mais si ces dernières ne sont pas satisfaites, il peut envisager d'intervenir pour les modifier. L'exécution de requête devient donc interactive afin de permettre l'utilisateur de contrôler l'exécution de sa requête. Ce sujet est évoqué dans le chapitre suivant.

# CHAPITRE 7

# Évaluation interactive de requêtes

## 7.1 *re*-Définition du problème

Généralement, un système informatique est interactif s'il est capable (i) de produire, au cours de son exécution, des informations perceptibles de son état interne, et (ii) de prendre en compte, au cours de son exécution, les informations communiquées par le(s) utilisateur(s) du système. Dans le cadre de l'évaluation de requêtes, les informations perceptibles sont souvent les *résultats partiels*, qui permettent à l'utilisateur de connaître le déroulement de l'exécution de sa requête. Au vu des premiers résultats (incomplets), l'utilisateur peut intervenir pour contrôler l'exécution de sa requête, voire pour l'affiner (i.e. la faire évoluer). Ainsi, **un évaluateur est dit interactif s'il est capable de produire des résultats partiels pendant l'exécution d'une requête (longue) et d'adapter son exécution aux changements provoqués par l'interaction avec l'utilisateur**. La figure 7.1 montre le cadre général de l'évaluateur interactif.

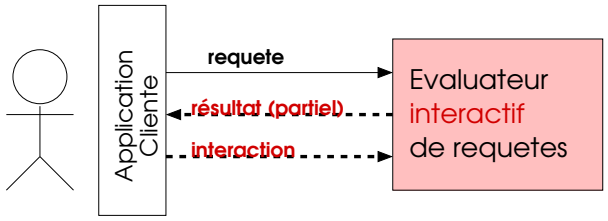



Figure 7.1: Vue générale de l'évaluateur interactif de requêtes

Pour une évaluation interactive efficace, il faut d'une part disposer les mécanismes d'évaluation répondant aux besoins décrits ci-dessus et, d'autre part, offrir à l'utilisateur une interface facilitant ses interactions (i.e. recevoir les résultats partiels et soumettre les contrôles et/ou les demandes d'affinement). Dans ce travail, nous ne nous intéressons qu'aux mécanismes d'évaluation de requêtes permettant l'interaction. Nous définissons par ailleurs une interface de programmation pour permettre l'interaction avec l'utilisateur. La suite de ce chapitre aborde les problèmes suivantes :

(1) la construction de résultats partiels, qui consiste à répondre aux questions suivantes :

(1.1) *Quels résultats incomplets peuvent être retournés à l'application cliente ?*

Il s'agit de définir les propriétés des résultats partiels afin qu'ils soient significatifs et compréhensibles par l'utilisateur.

(1.2) *Comment les construit-on ?*

Il s'agit des mécanismes d'évaluation partielle pour le cas d'exécution d'une requête longue et/ou en cas d'indisponibilités des sources.

(2) l'adaptation de l'exécution de requête, ce qui revient à répondre aux questions suivantes :

(2.1) *Quelles sont les interactions autorisées par le système ?*

Elles concernent les informations communiquées par l'utilisateur pendant l'exécution de sa requête. En considérant une interface d'interaction simple, nous définissons des *opérateurs d'interaction*.

(2.2) *Comment réagit le système aux interactions de l'utilisateur ?*

Il s'agit d'un processus d'adaptation nécessaire pour rendre l'exécution d'une requête conforme aux changements issus de l'interaction de l'utilisateur. Ce que l'on peut adapter ici sont les stratégies d'exécution de la requête, voire la requête elle-même (i.e. la sémantique de la requête). Dans le premier cas, les mécanismes proposés dans l'évaluation adaptative peuvent être utilisés (cf. la section 3.4.2). Dans le deuxième cas, il est nécessaire de considérer un *processus de transformation de requête* en cours d'exécution afin de tenir compte des nouveaux besoins. Les travaux présentés dans ce chapitre se focalisent sur ce dernier cas, en supposant l'utilisation des mécanismes d'évaluation adaptative.

**Plan du chapitre** La suite de ce chapitre est organisée de la manière suivante : les sections 7.2 et 7.3 présentent respectivement deux des techniques clés de l'évaluation interactive que sont la construction des résultats partiels et la transformation interactive de requête ; nous évoquons ensuite quelques points importants de la mise en œuvre de ces techniques avec QBF (la section 7.4) ; finalement, la section 7.5 compare notre travail avec les travaux existants, et nous concluons le chapitre dans la section 7.6.

## 7.2 Construction de résultats partiels

Rappelons que le résultat partiel d'une requête est le résultat de son évaluation sur un *sous-ensemble* des données en entrée. Ce sous-ensemble peut contenir, ou non, de toutes les propriétés des données spécifiées dans la requête. Par conséquent, leur évaluation peut assurer, ou non, tous les calculs nécessaires (i.e. les conditions de sélection, de jointure, etc.) de la requête utilisateur, et la cardinalité du résultat partiel est souvent différente de celle du résultat complet. Quelques techniques ont été proposées mais elles ne couvrent pas tous les cas possibles. Par exemple, la production de résultat incomplet en l'absence de quelques propriétés des données du résultat est très peu abordée dans la littérature[1]. C'est ce créneau que vient combler notre travail. Afin de situer le cas considéré par notre travail dans l'ensemble, nous introduisons tout d'abord quelques hypothèses (la sous-section 7.2.1). Ensuite, nous présentons les propriétés du résultat partiel que nous souhaitons garantir afin qu'il soit significatif pour l'utilisateur (la sous-section 7.2.2). Enfin, nous présentons les mécanismes proposés pour la construction de résultats partiels (la sous-section 7.2.3).

### 7.2.1 Hypothèses

Tout d'abord, nous supposons que le système d'évaluation de requêtes retourne *continuellement* les résultats (incomplets) à l'application cliente, qui est susceptible de les afficher *à sa manière*. Aucune information supplémentaire n'est fournie avec les résultats incomplets. Par ailleurs, aucune contrainte d'intégrité entre les sources de données n'est considérée.

Au regard du type de résultats partiels, nous distinguons le résultat partiel horizontal, vertical et généralisé (cf. la section 6.2). Nous avons, par ailleurs, remarqué que le résultat partiel horizontal est obtenu naturellement par l'exécution basée sur le modèle itérateur en utilisant des opérateurs non-bloquants [WA91, HH99, IFF+99, UF00] (cf. la section 3.4.3). Le résultat partiel vertical d'une requête est considéré comme un cas particulier du résultat partiel généralisé en utilisant les techniques de "matérialisation". Nous considérons donc seulement le cas de la *construction de résultat partiel généralisé* qui est le cas le plus complexe. Le problème qui nous intéresse est donc reformulé comme suit : *Comment peut-on retourner à l'utilisateur un résultat (incomplet, bien sûr) lorsqu'une ou plusieurs entrée(s) indispensable(s) pour sa requête est (sont) indisponible(s) (ou son accès est lent)?*[2]. Une entrée est *indispensable* si elle contient les propriétés des données intéressantes pour la requête utilisateur qui ne sont contenues dans aucune autre source. Ce problème concerne essentiellement les opérateurs ayant au moins deux entrées, dont les structures ne sont pas identiques, notamment le cas de la *jointure*.

Par ailleurs, un opérateur peut être bloqué du fait de l'absence d'une ou de plusieurs de ses entrées, dans le cas où (i) l'entrée est une source indisponible, ou bien (ii) l'entrée est un opérateur bloquant[3], comme les opérateurs d'agrégation. Dans le premier cas, le problème devra être résolu au niveau de

[1] Nous en rediscuterons dans la section 7.5, lors de la comparaison de notre proposition avec des travaux similaires.

[2] Bien entendu, il doit avoir au moins une entrée disponible.

[3] Rappelons qu'un opérateur bloquant nécessite la lecture de *toutes* les données en entrée avant de commencer à produire les résultats (cf. section 3.4.3).

l'opérateur lui-même. Dans le deuxième cas, il est nécessaire de débloquer l'opérateur bloquant. Ce dernier cas consiste à concevoir des algorithmes pour les opérateurs d'agrégation (e.g. le somme, le moyen, etc.) permettant de produire des résultats partiels et de les mettre à jour (si nécessaire) durant l'exécution de la requête. Considérons par exemple l'opérateur `sum`, qui doit lire les données en entrée en totalité avant de produire le résultat exact. Si on veut produire le résultat avant la fin de la lecture des données, le résultat n'est qu'approximatif. Les algorithmes non-bloquants de ces opérateurs sont appelés agrégations en ligne (*online aggregation*) et ont fait l'objet de travaux présentés dans [HHW97, TGO99]. L'utilisation de ces algorithmes permet de débloquer les opérateurs en question. Nous nous focalisons alors sur le cas de blocage dû à l'indisponibilité de données ou lorsque l'accès aux données est lent, encore une fois dans le cas de la *jointure*.

Ainsi, le problème de la construction des résultats partiels pour une requête générale est réduit à la construction de résultat partiel pour la jointure.

### 7.2.2 Propriétés souhaitées des résultats partiels

Comme nous l'avons mentionné, le résultat partiel d'une requête n'est pas *forcement* un sous-ensemble du résultat complet[4]. En effet, cela est dû à l'impossibilité de réaliser tous les calculs requis par la requête utilisateur, notamment les vérifications des condition de sélection ou de jointure concernant des données indisponibles. Considérons l'exemple de requête de jointure de $\mathcal{R}$, $\mathcal{S}$ et $\mathcal{T}$ (voir la figure 7.2), si l'accès à $\mathcal{S}$ est bloqué et que l'on souhaite retourner les données concernant $\mathcal{R}$ et $\mathcal{T}$, il sera impossible de vérifier les conditions de jointure sur les attributs $b$ et $d$ pour garantir que tous les items partiels produits font partie du résultat final complet[5].

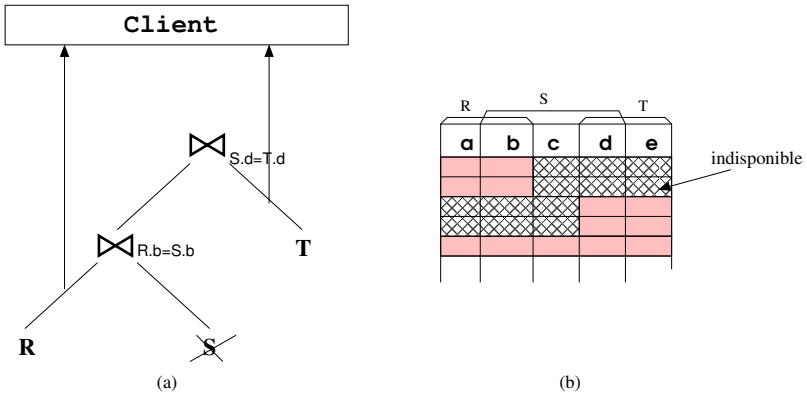



Figure 7.2: Exemple de résultats partiels

Par conséquent, nous définissons les propriétés du résultat partiel attendu que doit garantir le système.

[4]En particulier, le cas des résultats partiels verticaux.

[5]Excepté dans le cas où il existe des contraintes d'intégrité quelconques que nous ne considérons pas dans ce travail.

Elles sont :

(P1) **Maximum** : le résultat d'un opérateur doit être produit aussitôt que possible lorsqu'il existe au moins une entrée disponible.

Cela signifie qu'un maximum de données entrantes doit être "acheminé" vers le client.

(P2) **Non-dupliquée** : un élément incomplet est retourné comme résultat partiel une et une seule fois.

Cela signifie qu'une fois qu'un élément incomplet est acheminé vers le client, il n'est pas envoyé de nouveau jusqu'au moment où il est combiné avec d'autres données pour retourner les nouveaux items du résultat plus significatifs. Cette propriété a pour but d'éviter les produits cartésiens si ces derniers ne font pas partie de la requête utilisateur.

Ces deux propriétés permettent de garantir que l'utilisateur peut voir autant de données concernant sa requête lorsque celles-ci sont disponibles, mais seulement les données *pouvant* participer au résultat final.

Reprenons notre exemple de requête de jointure de $\mathcal{R}$, $\mathcal{S}$ et $\mathcal{T}$. Comme montré dans la figure 7.2(a), il est souhaité que les données provenant de $\mathcal{R}$, $\mathcal{T}$ deviennent visibles à l'utilisateur lorsque $\mathcal{S}$ est indisponible (i.e. le cas où l'exécution de requête est naturellement bloquée). Nous notons qu'il n'existe aucune condition de jointure entre $\mathcal{R}$ et $\mathcal{T}$. Ainsi, le produit cartésien n'est pas spécifié dans la requête d'entrée. En garantissant les deux propriétés décrites ci-dessus, les résultats que nous souhaitons rendre à l'application cliente sont ceux illustrés dans la figure 7.2(b).

### 7.2.3 Alternatives de la construction de résultat partiel

Rappelons que l'exécution de requête basée sur le modèle itérateur est naturellement assurée par un seul processus et donc la production du résultat d'une requête dépend de l'accès (ou du traitement du nœud) le plus lent dans le plan. Ainsi, la construction de résultat partiel nécessite de désynchroniser l'exécution de requête dans le but de favoriser les calculs de certaines sous-requêtes dont le résultat peut être retourné à l'utilisateur. Cela peut être assurée par le découpage du plan en utilisant un support de stockage temporaire.

Nous proposons donc d'explorer deux alternatives : la première consiste à insérer explicitement des tampons dans le plan et la seconde consiste à étendre les implémentations des opérateurs, les dotant d'une capacité de stockage interne. Pour le premier cas, il faut déterminer les endroits où les tampons doivent être insérés ainsi que rediriger les flux de données entre les nœuds opérateurs. L'implémentation des autres opérateurs reste inchangée. Pour la deuxième alternative, nous considérons une extension des algorithmes, notamment la famille d'algorithmes de jointure par hachage. Nous détaillons ces deux alternatives dans la suite de cette section.

#### 7.2.3.1 Plan de requête pour la production des résultats partiels

La figure 7.3(b) montre ce que peut être le plan de requête pour la construction du résultat partiel de notre exemple. Les tampons sont insérés dans le plan à chaque point où l'on souhaite que les données

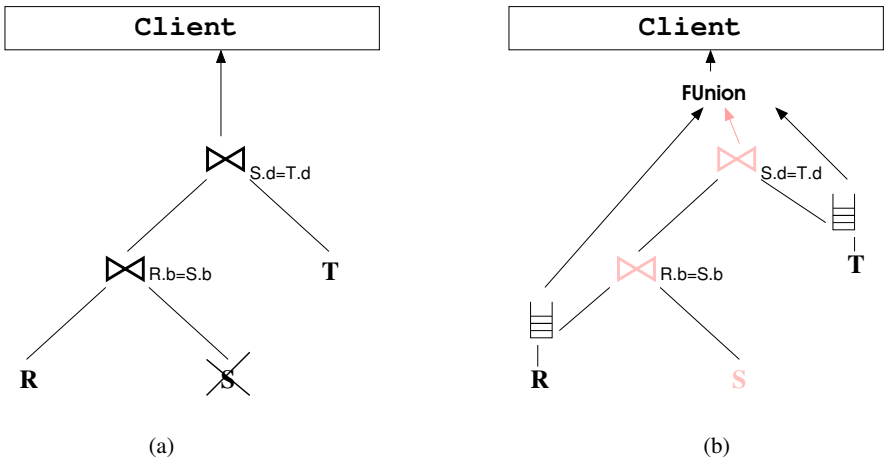



Figure 7.3: Plan pour production des résultats partiels

soient acheminées vers le client. Cela dépend des préférences de l'utilisateur et/ou des propriétés des sources de données. La taille de tampon est dynamiquement modifiée selon le taux d'arrivée des données. L'opérateur *FUnion* est susceptible de retourner les items provenant d'une de ses entrées.

#### 7.2.3.2 Extension des algorithmes pour la production des résultats partiels

L'idée principale de cette approche est d'acheminer les données vers le client en traversant tous les opérateurs du plan de requête. Lorsqu'une ou plusieurs entrée(s) est(sont) indisponible(s), les jointures se comportent comme une jointure externe, en considérant une structure de données intermédiaire différente de la structure prévue. L'extension a pour but de tenir compte de cette "anomalie" en produisant au client le maximum des données intéressantes.

Nous détaillons ici seulement le cas de l'opérateur de jointure et particulièrement l'algorithme de *jointure par hachage symétrique* [WA91]. La raison de ce choix est double : premièrement, l'opérateur de jointure est complexe du fait de son traitement de plusieurs flux de données en entrée, composé de plusieurs étapes ; deuxièmement, la jointure par hachage est souvent utilisée dans les plans de requête complexe et pour combiner les données de sources multiples. L'extension consiste à considérer un élément spécial *any* dans la table de hachage, correspondant à tous les éléments auxquels il peut être comparé pour construire un élément du résultat (partiel). Nous appelons cette extension *Progressive Outer Symmetric Hash Join* (dénoté *POSHJ*).

La figure 7.4 illustre le principe de l'algorithme en considérant la jointure de $\mathcal{R}$ et $\mathcal{S}$. A la réception d'un item $tr$ de $\mathcal{R}$, $tr$ est comparé avec tous les items correspondants dans la table de hachage de $\mathcal{S}$. Si le résultat est vide, i.e. il n'existe aucun item déjà lu de $\mathcal{S}$ correspondant à $tr$, et que $tr$ fait partie des données de préférence de l'utilisateur, un item $tr'$ équivalent au résultat de la jointure externe (*outer-join*) est produit. Concrètement, la structure de $tr'$ est la même que celle de la jointure de $\mathcal{R}$ et $\mathcal{S}$ excepté la

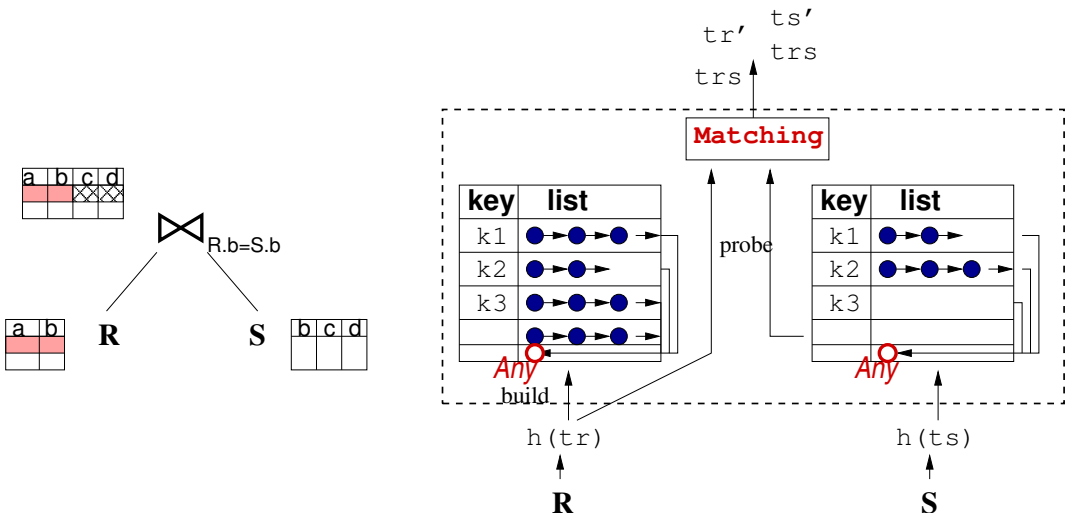



Figure 7.4: Algorithme de jointure pour produire les résultats partiels

valeur *Any* de tous les attributs provenant de $ts$. Enfin, l'item $tr$ est inséré dans la table de hachage pour être joint ultérieurement avec les items de $\mathcal{S}$. La valeur *any* est un type spécial de valeur *null* sauf qu'elle n'est utilisée que pour le retour des résultats. Cela permet de garantir la propriété (P1).

Dans le cas où les données indisponibles se trouvent plus en profondeur dans l'arbre de requête, e.g. ($\mathcal{R}$ x $\mathcal{S}$) x $\mathcal{T}$ où $\mathcal{S}$ est indisponible, les items en entrée d'un opérateur de jointure contiennent déjà des attributs dont la valeur est *any*. Dans ce cas, ces items sont comparés avec les items dans la table de hachage de l'autre entrée (i.e. réaliser la phase *probe*) mais ils ne sont pas insérés dans la table de hachage correspondante (i.e. ne pas réaliser la phase *build*). Cela permet de garantir que l'item partiel $tr$ n'est pas retourné à nouveau à l'application cliente jusqu'à l'obtention d'un item joignable de $\mathcal{S}$. Cela permet de garantir la propriété (P2).

La figure 7.5 montre le principe de cet algorithme. Notons que ce principe peut aussi être appliqué à des algorithmes tels que XJoin [UF00], DPHJ [ILW+00], ou d'autres basés sur SHJ [WA91].

## 7.3 Transformation interactive de requête

### 7.3.1 Opérateurs d'interaction

En considérant les requêtes composées d'une expression de besoins et d'un contexte (cf. la section 6.1), les interactions peuvent concerner ces deux parties de la requête.

**Modification de contexte** Il s'agit de changements de paramètres du contexte, comme la cardinalité de résultat souhaitée (*ResultNumber*), la limite de temps d'évaluation (*Timeout*), etc. En général, cette modification n'entraîne aucune modification dans la sémantique de la requête. Toutefois, elle nécessite

```
function POSHJ(Input input1, Input input2, Predicate joined)
        table1 = create_HT();
        table2 = create_HT();
        init_MDResult(input1,input2);

        while(true)
                if (!input1.hasNext() && !input2.hasNext())
                        break;
                available_input = getAvailableInput();
                if (available_input == 1)
                        item = input1.next();
                        if (fullItem(item)) //i.e. checking any value for processing build phase
                                key = table1.Hash(item);
                                table1.add2HT(key,item);
                        if (table2.isInHT(key,item,joined))//i.e. probe phase
                                join(item,table2.objectFound()) //i.e. join 2 items
                        else
                                join_empty(item) //i.e. producing external join result
                else
                        ... idem... ...
                        .....
end_function
```

Figure 7.5: Extension de la jointure par hachage symétrique pour la construction des résultats partiels

des modifications des paramètres de certains éléments de contrôle (e.g. l'observation de temps *Timing-Monitor*) et/ou de certain opérateurs spécifiques (e.g. *MaxSize*). La partie gauche de la figure 7.6 montre la syntaxe de l'opération de modification de contexte où `CtxParameter` est le paramètre de contexte à modifier.

| | |
|---|---|
| | (1) **setContext** ({*ResultNumber*, *100*}) |
| **setContext** (`CtxParameter`) | (2) **setContext** ({*Timeout*, *300*}) |

Figure 7.6: Opération pour la modification de contexte de requête

Quelques exemples sont donnés dans la partie droite de la figure 7.6 : la ligne (1) montre que l'utilisateur veut modifier le nombre maximal des résultats (*ResultNumber*) ; la ligne (2) signifie que le temps d'exécution est limité à 300 secondes. Bien sûr, le paramètre de cette opération doit appartenir à la liste des paramètres acceptés par le système.

**Modification de requête** Elle consiste généralement à ajouter, ou à supprimer, un ou plusieurs opérateur(s) algébriques dans le plan de requête. La figure 7.7 présente la syntaxe des opérations pour la modification de requête (la partie gauche) et quelques exemples (la partie droite de la figure[6]).

| | |
|---|---|
| | (1) **addOperator** `(Select(R, greater(e,4)))` |
| **addOperator** `(oper)` | (2) **addOperator** `(getData(U))` |
| **removeOperator** `(oper)` | (3) **removeOperator** `(getData(S))` |

Figure 7.7: Opération pour la modification de requête

### 7.3.2 Implication de l'interaction

Les modifications décrites ci-dessus impliquent souvent des changements dans le plan de requête. Toutefois, certaines modifications sont faciles à réaliser alors que d'autres sont plus difficiles et coûteuses. En fait, cela dépend de l'impact des modifications sur la structure des résultats intermédiaires et finaux d'une requête. Les opérations d'affinement de requêtes peuvent donc être classifiées en deux catégories :

1. **affinement sans restructuration** : l'affinement de requête n'entraîne aucune modification de la structure des données du résultat. L'impact de cet affinement sur le processus de traitement est :

   1.1 peu coûteux : Ce sont les cas où l'utilisateur veut modifier les conditions de sélection, comme ajouter, supprimer un opérateur *Select*.

   1.2 coûteux : Ce sont les cas où l'utilisateur veux ajouter ou supprimer des opérateurs comme *Union*, *Intersection*, *Difference*, qui ne changent pas la structure de données intermédiaires ni en sortie, mais qui peuvent nécessiter un traitement (optimisation) supplémentaire.

2. **affinement avec restructuration** : l'affinement de requêtes influence la structure du résultat. Nous distinguons également l'affinement :

   2a. peu coûteux : Ce sont les cas où la restructuration consiste à réduire les propriétés de données, e.g. ajouter un opérateur de projection.

   2b. (très) coûteux : Ce sont les cas où l'utilisateur veut ajouter/supprimer une ou plusieurs sources de données (e.g. ajouter un opérateur *Source*, *Join*, *Product*).

Ces critères peuvent être utilisés pour décider les interactions autorisées lors de la construction d'un évaluateur ou pour différents utilisateurs.

[6] *getData* représente un opérateur générique pour l'accès aux données.

### 7.3.3 Processus de transformation

Tous les affinements nécessitent de modifier les flux de données entre des opérateurs en cours d'exécution de façon continue, i.e. le système doit être capable de re-diriger des flux de données entre des opérateurs en évitant de les réinitialiser, si possible. Cela est possible parce que tous les opérateurs sont implémentés selon le modèle itérateur et que nous maintenons explicitement la relation entre les opérateurs à tout moment de leur cycle de vie.

Dans le cas de l'affinement de requête sans restructuration des résultats, le processus de transformation de requête se compose des étapes suivantes :

1. signaler la demande de modifier le plan de requête.

   Tous les nœuds d'opérateur du plan de requête sont mis en état "*halted*". La modification est propagée à partir du nœud racine jusqu'aux feuilles.

2. déterminer les changements qui devront être effectués, i.e. quels paramètres de l'évaluateur doivent être changés ou quels opérateurs, dont les paramètres doivent être changés, ou quel opérateur du plan doit être éliminé ou à quelle position du plan le nouvel opérateur doit-il être inséré.

3. modifier le plan, i.e. modifier les flux de données entre des nœuds.

4. "continuer" l'exécution du plan affiné.

Dans le cas de l'affinement de requête avec restructuration des résultats, le processus d'affinement de requête se compose de principales étapes suivantes :

1. signaler la demande de modifier le plan de requête.

   Tous les nœuds d'opérateur feuilles sont mis en état "*halted*". D'autres nœuds d'opérateur continuent leur exécution normale jusqu'à vider les flux et passent à l'état "*stalled*".

2. déterminer les changements qui devrons être effectuées, i.e. quels paramètres de l'évaluateur devront être changés ou quels opérateurs, dont les paramètres devront être changés, ou quel opérateur du plan devra être éliminé ou à quelle position du plan le nouvel opérateur devra-t-il être inséré.

3. modifier le plan, i.e. modifier les flux de données entre des nœuds. Cette modification est propagée des feuilles jusqu'à la racine (selon l'odre postfix) en vérifiant la structure de données interne de chaque nœud. Cela permet de modifier explicitement les nœuds dont la structure de résultat devra être changé.

4. valider ou annuler l'affinement selon le résultat de la phase précédente. Pour valider, les flux de données entre des nœuds d'opérateur sont (re-)initialisés si nécessaire, e.g. dans le cas où on ajoute un opérateur, il faut initialiser le cursor de ce nœud et réinitialiser le cursor du nœud parent pour qu'il prenne en entrée les données produites par le nouvel opérateur.

La suppression d'une ou plusieurs sources de données dans une requête peut être considérée comme un cas où cette (ces) source(s) est (sont) indisponible(s), i.e. l'utilisation de l'opérateur *dummy*. Le processus d'affinement est donc simplifié.

## 7.4 Mise en œuvre avec QBF

Cette section évoque quelques points importants de la réalisation des mécanismes présentés précédemment avec les composants de QBF. Nous commençons par l'implémentation des opérateurs pour la production des résultats partiels, puis décrivons l'interface pour l'interaction. Enfin, nous évoquons les aspects dynamiques concernant l'interaction.

### 7.4.1 Implémentation des opérateurs

Dans QBF, les opérateurs et les algorithmes sont respectivement modélisés par OperNode et Algorithm. Par conséquent, les opérateurs et algorithmes des évaluateurs basés sur QBF implémentent les interfaces correspondantes. Pour l'utilisation des tampons, nous utilisons l'implémentation générique de l'opérateur *Buffer* fourni dans QBF.

**Opérateur** *POSHJ* Dans QBF, l'opérateur de jointure est considéré comme étant un opérateur binaire, ayant un prédicat de jointure (représenté par l'interface Predicate, cf. le chapitre 4) qui peut être éventuellement vide[7]. L'implémentation de l'opérateur réalise l'interface Algorithm présentée dans la section 4.2 du chapitre 4. Par ailleurs, l'implémentation de la jointure considère aussi une fonction (représenté par l'interface Function, cf. le chapitre 4) définissant la construction des éléments du résultat.

L'élément *Any* pour la production de résultats partiels nécessite de définir les classes de type Predicate et Function capable de traiter cet élément. L'implémentation de *POSHJ* consiste à définir les classes Any_Equal, Any_CreateItem (implémentant respectivement les interfaces Predicate, Function) et à étendre l'implémentation de *SHJoin* qui, quant à elle, n'implique que la modification dans l'implémentation de la fonction de hachage. La suite montre bien que la réalisation de notre algorithme dans QBF est très facile et rapide en utilisant l'algorithme de jointure par hachage symétrique implémenté antérieurement.

```
public class Any{
}
...

public class Any_Equal extends Equal{

   public boolean invoke(Object o1, Object o2){
      if ((o1 instanceof Any) || (o2 instanceof Any))
         return true;
      else
```

[7] Si un prédicat est vide, il retourne toujours la valeur "vrai". C'est le cas du produit cartésien, par exemple.

```
         return super.invoke(o1,o2);
   }
}
...

public class Any_CreateItem extends CreateItem{

   public Object invoke(Object o1, Object o2){
      if ((o1 instanceof Any) || (o2 instanceof Any))

          //créér item ``partiel''
          ...

      else
         super.invoke(o1, o2);
  }
}

public class Any_HashFunction extends HashFunction{

      public static int any_val = 65536;

      public static int hash(Object o){
          if (o instanceof Any)
        return any_val;
          else
                super.hash(o);
      }
}

public class POSHJ extends SHJoin{
     public POSHJ(Any_Equal predicate, Any_CreateItem createFnc, Any_HashFunction hashfnc){
       super(predicate, createFnc, hashfnc);
     }
}
```

### 7.4.2 Interactions avec l'utilisateur

Dans notre implémentation, nous considérons une interface "simple" pour recevoir les interactions de l'utilisateur sous forme des opérations d'interaction, comme ceal a été présenté dans la section 7.3.1. Pendant l'exécution d'une requête, l'utilisateur peut utiliser les commandes basées sur ces opérations pour interagir avec le système.

L'interaction avec l'utilisateur pendant l'exécution de requête est "capturée" par un élément de surveillant InteractionMonitor spécialisé de la classe PropertyMonitor fourni dans QBF. Chaque interaction génère donc effectivement un événement correspondant afin de signaler l'interaction et le système déclenche les traitements nécessaires.

Il serait intéressant d'associer à l'élément de surveillance de l'interaction une interface plus conviviale pour l'utilisateur. Cela permettrait, d'une part, de faciliter l'interaction avec l'utilisateur, et d'autre part, d'ajouter plus aisément les vérifications nécessaires de ces interactions.

### 7.4.3 Adaptation de requête

Nous considérons la transformation de requête issue de l'interaction utilisateur, décrite dans la section 7.3, comme étant une forme d'adaptation, i.e. adaptation aux changements des besoins de l'utilisateur. Les interactions sont alors à l'origine des événements qui correspondent à des opérations d'interaction définies antérieurement (la section 7.3.1). Ces événements peuvent déclencher une ou plusieurs règles définissant les transformations nécessaires comme le décrit la section 7.3.

## 7.5 Travaux connexes

Le résultat présenté dans ce chapitre contribue à deux aspects de l'évaluation interactive de requêtes que sont la construction des résultats partiels et la transformation de requête en cours d'exécution. Cette section compare notre travail avec ceux de ces deux catégories.

**Construction des résultats partiels** La production des résultats partiels a été abordée dans plusieurs travaux, notamment ceux sur les opérateurs adaptatifs [WA91, HH99, IFF$^+$99, UF00]. Pourtant ces travaux consistent à produire les résultats dans le cas d'accès lent aux données, et donc, le seul type de résultat partiel autorisé est horizontal. Ces techniques sont complémentaires à la nôtre. Notre travail se focalise sur la production des résultats partiels en cas d'absence de données et nous considérons le cas le plus général qu'est la construction des résultats partiels généralisés.

Notre travail peut être comparé aux travaux réalisés dans DISCO permettant de retourner les résultats partiels en cas d'indisponibilité des sources [BT98] (cf. la section 3.3.2). Dans le cas où une ou quelques sources sont indisponibles au lancement de l'exécution de requête, les mécanismes proposés dans ce chapitre produisent les mêmes résultats que ceux dans DISCO. Pourtant, notre travail se différencie en plusieurs points : (i) Dans DISCO, la détection de l'indisponibilité des sources ne fait qu'une seule fois au lancement de l'exécution de requête (incrémentielle) tandis que dans notre travail l'indisponibilité des sources peut être détectée à n'importe quel moment par des mécanismes de surveillance (cf. la section 5.5) ; (ii) les résultats partiels de DISCO ne peuvent être retournés à l'utilisateur qu'à la fin de l'exécution des requêtes parachutes et leur matérialisation. Nos mécanismes permettent de produire les résultats (partiels) au fur et à mesure durant la progression de l'exécution de la requête.

**Transformation interactive** Abordant les problèmes de traitement de requêtes interactif, les travaux dans [RH02] proposent d'utiliser les mécanismes Eddy pour produire les résultats partiels selon le degré de préférence d'utilisateur (cf. la section 3.4.5). Malgré la différence profonde dans le modèle d'exécution, nous pouvons comparer ce travail au nôtre selon quelques points : (i) ces deux travaux permettent de produire les résultats (partiels) d'une manière incrémentielle ; (ii) ces deux travaux permettent

l'interaction de l'utilisateur pendant l'exécution de requête. Pourtant ces deux travaux sont par essence différents.

Les travaux dans [RH02] se basant sur Eddy permettent une adaptation à grain très fin, mais ces mécanismes ne peuvent pas être utilisés dans les moteurs d'exécution de requêtes classiques (i.e. arbre de requête), largement utilisés et qui correspondent au cas que nous avons étudié. Par ailleurs, les auteurs de [RH02] considère l'existence de contraintes d'intégrité entre les sources afin de garantir la cohérence des résultats partiels produits. Les mécanismes proposés dans [RH02] garantissent donc que le résultat partiel est effectivement un sous-ensemble du résultat final. Nous pensons que cette hypothèse est forte et difficile à garantir dans le contexte de l'interrogation des sources multiples, hétérogènes et autonomes. Une telle contrainte pourrait conduire à une situation où aucun résultat ne pourrait être produit, jusqu'au moment où toutes les sources deviennent disponibles. En outre, nous considérons un spectre d'interaction plus large que celui de [RH02], qui concerne seulement le degré de préférence.

## 7.6 Conclusion du chapitre

Dans ce chapitre, nous avons présenté quelques mécanismes pour l'évaluation interactive de requêtes, à savoir la construction de résultats partiels et la transformation interactive de requête.

Le mécanisme de construction de résultat partiel consiste en l'utilisation de nouveaux opérateurs et/ou de nouveaux algorithmes implémentant les opérateurs. Nous pensons donc qu'il est facile de les intégrer dans les moteurs d'exécution de requêtes existants. Quant aux mécanismes de transformation de requêtes, ils ne sont possibles que lorsque l'état des opérateurs est accessible. Cela est indispensable pour garantir la correction de l'avancement de l'évaluation de requête. Leur mise en œuvre est donc plus contraignante pour les moteurs d'exécution de requête.

Par ailleurs, il faut noter que le travail présenté dans ce chapitre aborde seulement une partie de l'évaluation interactive. Il consiste en quelques mécanismes de base. Pour une évaluation interactive efficace et effective, plusieurs travaux restent encore à faire et que nous les évoquerons un peu plus loin dans ce document.

# CHAPITRE 8

# Application

Ce chapitre illustre l'utilisation de QBF (i.e. l'instanciation) dans la construction d'évaluateurs de requêtes, ou plus précisément la fonctionnalité de l'évaluation de requêtes dans les systèmes. Tout d'abord, nous présentons le principe de l'instanciation de QBF (la section 8.1). Ensuite, nous évoquons nos expériences dans la réalisation d'un évaluateur de requêtes XML pour le projet MediaGRID (la section 8.2). Enfin, la section 8.3 conclut le chapitre.

## 8.1 Principe d'instanciation (de QBF)

Rappelons que l'approche QBF pour la construction d'évaluateurs de requêtes comporte trois étapes que sont la spécification, l'implémentation et l'instanciation (voir la figure 4.2 du chapitre 4). L'instanciation de QBF consiste à exploiter sa spécification et son implémentation afin de construire un évaluateur qui reçoit des requêtes exprimées dans un *langage spécifique*, qui procède à des *traitements spécifiques* pour leur évaluation, et qui retourne les résultats dans un *format spécifique*. Ce qui doit être défini lors de l'instanciation de QBF concerne le langage d'interrogation, le traitement à assurer, et le format des données retournées, ces trois aspects sont plus ou moins dépendants les uns des autres.

En ce qui concerne le langage d'interrogation, il dépend en effet du modèle de données considéré composé d'une collection de structures de données et d'une collection d'opérateurs définissant les manipulations possibles sur ces structures. En s'appuyant sur QBF, il s'agit d'étendre (ou d'implémenter) les classes (ou les interfaces) MetaData, DataSet pour les structures de données et OperNode, Algorithm pour les opérateurs[1]. Pour autoriser les requêtes personnalisées, il faut aussi définir les paramètres de contexte (représentés par Context, CtxParameter) que peut traiter l'évaluateur. Le format des données retournées (i.e. résultats) est aussi défini. En ce qui concerne les traitements à assurer, hormis l'exécution des opérateurs définis précédemment, il importe, premièrement, de définir les opérations

[1]Nous utilisons le nom des interfaces définies dans la spécification de QBF aussi pour référencer des classes, notamment abstraites, implémentant ces interfaces dans l'implémentation de QBF.

d'optimisation logique et physique appropriées (représentées par TransformingPattern, MappingOperator), les calculs de coûts appropriés (représentés par Annotator, Annotation, Property et PropFunction) et la stratégie d'optimisation (représentée par Planner). Deuxièmement, il faut prévoir les possibilités d'adaptation de l'évaluateur, en identifiant les éléments de surveillance à intégrer (PropertyMonitor et Monitor) et les adaptations pouvant être réalisées (Rule et RuleManager).

Quelques éléments sont déjà fournis dans l'implémentation de QBF, d'autres sont à définir par le programmeur. Par exemple, l'implémentation des opérateurs génériques sur les collections (e.g. *Select*, *Join*, *Union*, etc.) peut être utilisée. Les programmeurs peuvent étendre cette implémentation afin de tenir compte des aspects spécifiques liés à la structure de données (e.g. les prédicats, les fonctions pour la construction des résultats) ; l'implémentation de certaines opérations d'optimisation peut être utilisée. Par ailleurs, les mécanismes de surveillance ont été partiellement implémentés. Concrètement, les mécanismes de collection des informations telles que la cardinalité (StatisticMonitor), le temps (TimingMonitor), etc., présentés dans le chapitre 5 peuvent être utilisés en y ajoutant la définition des conditions dites non-souhaités, i.e. quand un changement doit être signalé. Les opérations d'adaptation de base (`replace_algo`, `replace_oper`, `insert_oper`, `reorder_join`, etc., voir le chapitre 5) peuvent également être utilisées pour définir les adaptations à réaliser lorsqu'un changement est détecté. Tous ces éléments facilitent la définition des règles d'adaptation.

En résumé, les programmeurs peuvent construire leurs évaluateurs en tirant profit de l'implémentation de QBF (cf. le chapitre 5). Dans la suite, nous illustrons ce principe dans le cadre du projet MediaGRID pour lequel nous avons développé un évaluateur de requêtes pour les données XML.

## 8.2 MediaGRID

Le projet MediaGRID[2] se propose de construire une infrastructure de médiation pour l'accès transparent aux données. Une telle infrastructure offre aux applications des facilités d'intégration, d'interrogation des sources de données tout en cachant leur hétérogénéité et leur complexité. L'approche de MediaGRID consiste à définir et réaliser une infrastructure ouverte, c'est-à-dire un ensemble des outils, des éléments réutilisables pour la construction d'un médiateur.

En prenant le domaine des applications biologiques comme contexte de validation, le projet MediaGRID mène trois actions de recherche complémentaires qui visent à :

- faciliter l'intégration de sources de forme quelconque : le défi est d'offrir une aide à l'intégration de données autorisant la mise en correspondance structurelle et sémantique d'un très grand nombre de sources.
- proposer des " techniques " d'évaluation et d'optimisation d'expressions de calcul : le défi est d'offrir des algorithmes, des outils, des modèles pour réaliser le choix des " meilleures " expressions de calcul, autoriser des résultats partiels et la possibilité de raffiner une expression de manière dynamique.

[2]http://www-lsr.imag.fr/mediagrid

- valider les deux composants majeurs de l'infrastructure (i.e. l'intégrateur/générateur et l'évaluateur) en considérant le domaine particulier des applications biologiques dans lesquelles les besoins en matière d'intégration de sources de données biologiques et de facilité d'accès aux données sont très forts.

Notre participation[3] à ce projet concerne principalement l'aspect d'évaluation et d'optimisation des requêtes. Dans la suite de cette section, nous présentons tout d'abord l'architecture d'un système MediaGRID (la sous-section 8.2.1). Nous nous focalisons par la suite sur la fonctionnalité d'interrogation en abordant les traitements en matière de langage de requête et format de données (la sous-section 8.2.2) et le processus d'évaluation adaptative et interactive de requêtes (la sous-section 8.2.3). La sous-section 8.2.4 présente quelques points importants de la mise en œuvre de cet évaluateur en utilisant QBF.

### 8.2.1 Architecture

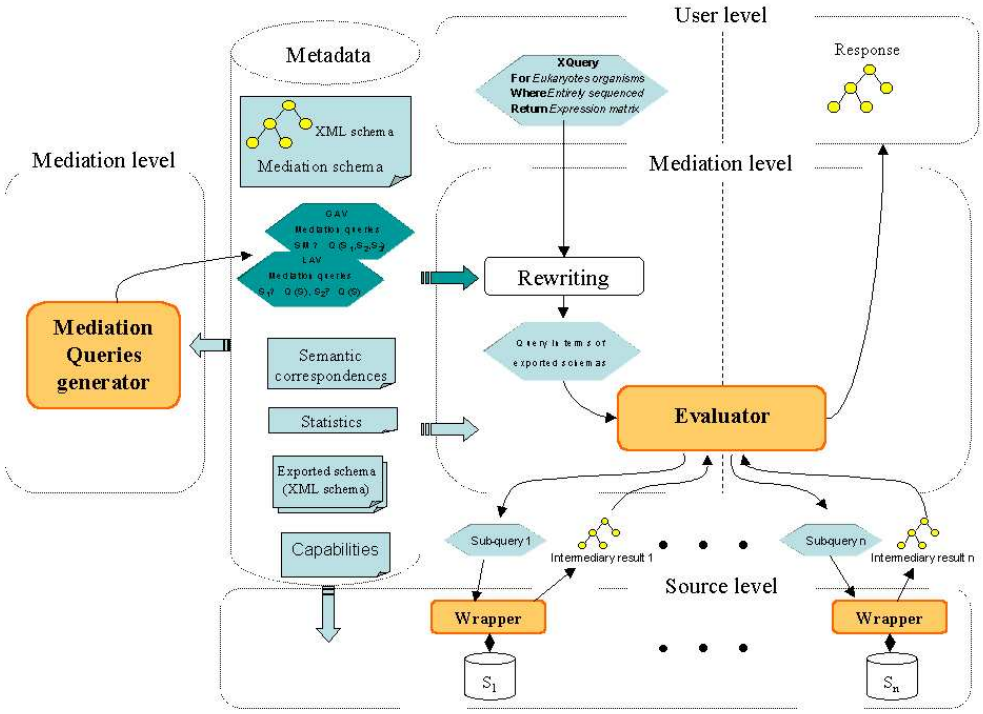



Figure 8.1: Architecture générale de MediaGRID

La figure 8.1 présente l'architecture générale de MediaGRID comme un médiateur basé sur XML. Excepté le module de gestion de meta-données qui est indispensable à tout moment, les autres modules se regroupent autour de deux processus principaux, l'intégration et l'interrogation, qui se déroulent en deux étapes différents.

[3]en ce qui concerne ce travail de thèse, *seulement*.

Le processus d'intégration dans MediaGRID se fait à l'aide du module Mediation Queries Generator lors de la construction du médiateur. La génération de requêtes de médiation repose sur les méta-données composées de la définition des besoins en interrogation (i.e. le schéma global), de la description des sources (i.e. les schémas exportés des sources) ainsi que des correspondances sémantiques entre le schéma globale et les schémas exportés pour générer la (les) requête(s) de médiation. Les requêtes de médiation décrivent les liens de calcul entre le schéma global et les schémas exportés. Elles peuvent suivre aussi bien l'approche *global-as-view* que l'approche *local-as-view* (cf. la section 2.4.2). Dans MediaGRID, les requêtes de médiation sont générées selon l'approche *global-as-view* et stockées comme partie des méta-données. Plus de détails sur la génération de requêtes de médiation sont donnés dans [PRI03].

D'ors et déjà, les requêtes utilisateur (ou expressions de besoins) peuvent être évaluées par le processus d'interrogation. Puisque le modèle de données pivot de MediaGRID est XML, les requêtes d'entrée sont formulées dans le langage XQuery [XQu]. Elle est réécrite (par le module Rewriting) en utilisant la (les) requête(s) de médiation générée(s) précédemment. La requête réécrite, exprimée sur les sources sous-jacentes, est évaluée par le module Evaluator qui, quant à lui, est *une instance de QBF pour les données XML*. Ce module est susceptible d'envoyer des sous-requêtes aux sources (ou plus précisément aux Wrappers), de récupérer des données et de réaliser des calculs nécessaires pour construire les résultats et les retourner à l'utilisateur.

La suite de cette section se focalise sur le module Evaluator qui reçoit en entrée une requête exprimée dans un langage algébrique sur les sources sous-jacentes.

### 8.2.2 Interrogation de données XML

Pour l'interrogation des données XML, il existe deux approches principales : une basée sur l'extension de l'algèbre relationnelle et relationnelle-objet, appelé approche "variable liée" (*variable binding*) [DFF$^+$98, GVD$^+$01, SA02, DN03] et l'autre basée sur la manipulation des collections d'arbres [HLST01]. Dans MediaGRID, nous avons adopté l'algèbre Xtasy [SA02] (une algèbre de l'approche *variable binding*) et y avons apporté quelques modifications afin d'enrichir et de faciliter l'expression de requête. L'idée principale de cette algèbre est de considérer la structure intermédiaire comme une *relation* pour le traitement des données. En conséquence, la plupart des opérateurs algébriques ressemblent à des opérateurs de l'algèbre relationnelle et objet (e.g. *Selection*, *Projection*, *Join*, etc.) excepté deux opérateurs **Source** et **Return**, appelés opérateurs frontières (*border operators*), le premier pour la transformation des données XML en la représentation intermédiaire (i.e. lecture des données XML) et le second pour la transformation inverse (i.e. production des données XML).

Précisément, la représentation interne des données XML est *une collection (***Env***) d'un ensemble de variables de type* **XTree**. *XTree* représente la structure d'arbre des éléments XML. Les éléments XML intéressés par le traitement d'une requête sont représentés comme des variables. La figure 8.2 montre le plan d'une requête simple en XQuery, en illustrant la structure interne *Env* et les opérateurs *Source* et *Return*. Dans ce document, nous omettons la présentation formelle de cette algèbre et fournissons simplement la liste complète des opérateurs algébriques dans l'annexe B (voir le tableau B.1).

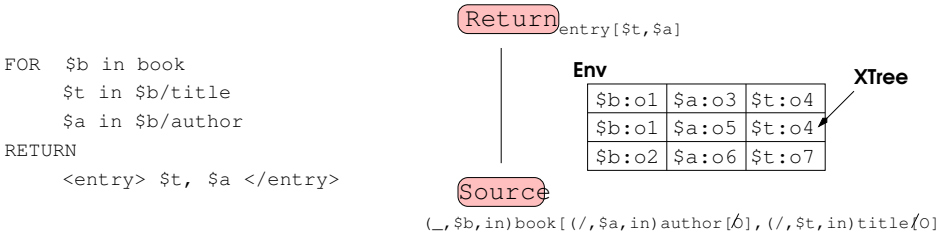



Figure 8.2: Exemple de requête en l'algèbre XML

### 8.2.3 Evaluation de requêtes

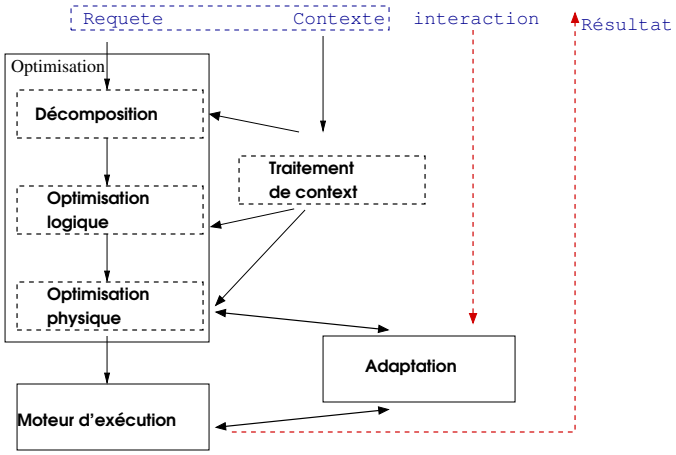



Figure 8.3: Evaluation de requêtes dans MediaGRID

La figure 8.3 montre la vue logique[4] de l'évaluateur de requêtes qui reçoit une requête représentée sous forme de plan de requête auxquelles nous associons un contexte composé des paramètres tels que la cardinalité du résultat, la limite de temps d'exécution, le type des résultats partiels, etc. Le processus d'évaluation de requêtes se compose des phases suivantes :

1. **Décomposition** : L'objectif est d'identifier des opérateurs pouvant être évalués par les sources afin d'y déléguer leur exécution. Les informations concernant la capacité des sources sont stockées dans les méta-données (voir la figure 8.1) en se basant sur le modèle d'opérateur (cf. la section 2.4.2.2). La décomposition se fait en parcourant le plan de requête en commençant par les feuilles en remontant vers la racine ou selon l'ordre post-fixe afin d'annoter les nœuds. Tous les

[4]Par "vue logique", nous entendons l'ensemble des fonctions ainsi que leur relation (i.e. l'ordre et les informations échangées entre les modules) pour accomplir la tâche d'évaluation d'une requête, lesquelles ne correspondent pas forcement à des modules implémentées (i.e. classes).

nœuds pouvant être résolus par une source sont regroupés en un nouvel opérateur appelé *getData*. L'opérateur *getData* a deux paramètres : (i) *url* de la source et (ii) une requête XQuery (*query*) correspondant au sous-plan identifié. Les opérateurs *getData* sont susceptibles d'accéder aux sources distantes pour récupérer les données.

2. **Optimisation logique** : Elle consiste à déterminer l'ordre des opérateurs à exécuter. Pour cela, les transformations de plan se basant sur les propriétés de l'algèbre sont appliquées. Quelques exemples de transformation sont : pousser les sélections, les projections vers les feuilles, fusionner l'opérateur *Project* et l'opérateur *Source*, etc.

3. **Optimisation physique** : Elle vise à choisir le "meilleur" algorithme pour chacun des opérateurs du plan.

   Ce choix peut être "guidé" par les paramètres de contexte (gérés par la fonction du **traitement de contexte**) si ces derniers existent, e.g. choisir des algorithmes pour pouvoir retourner les résultats partiels.

4. **Adaptation** : Cette phase se déclenche lors qu'un "changement significatif" dans l'environnement d'exécution se produit. L'objectif est d'adapter l'exécution de requête à de nouvelles conditions. Nous considérons les interactions avec l'utilisateur comme une forme d'adaptation. Dans ce dernier cas, l'adaptation peut consister en la modification de (la sémantique de) la requête elle-même. Cette tâche se fait souvent en appelant (utilisant) les fonctions d'optimisation.

### 8.2.4 Mise en œuvre avec QBF

Nous pouvons à présent voir comment l'évaluateur de requêtes de MediaGRID est réalisé comme une instance de QBF. Nous décrivons la mise en œuvre des fonctions décrites ci-dessus selon les points principaux suivants : (i) les opérateurs algébriques de XML, (ii) l'optimisation de requêtes[5], et (iii) l'adaptation.

#### 8.2.4.1 Opérateurs algébriques

En s'appuyant sur QBF, l'implémentation de l'algèbre présentée dans la section 8.2.2 consiste premièrement à définir l'interface EnvMetaData pour accéder à la structure *Env* et *XTree* ainsi que la structure de données pour leur stockage (temporaire). Deuxièmement, il faut implémenter les opérateurs pour manipuler cette structure (e.g. SourceNode, ReturnNode, etc.) et leurs algorithmes (e.g. SourceSAX2Env, ReturnSAX, etc.). La figure 8.4 montre une partie du diagramme de classes implémentées[6].

Les opérateurs logiques sont définis par des classes spécialisées de la classe OperNode. Leurs algorithmes implémentent à la fois l'interface Algorithm (ou étendent l'implémentation des opérateurs

[5]Nous évoquons également l'influence des paramètres de contexte sur la fonction d'optimisation sans présenter en détail l'implémentation de ce module qui est relativement simple.

[6]Pour une raison de lisibilité, nous ne montrons pas le diagramme en totalité. La liste complète des opérateurs et des algorithmes est données dans l'annexe B.

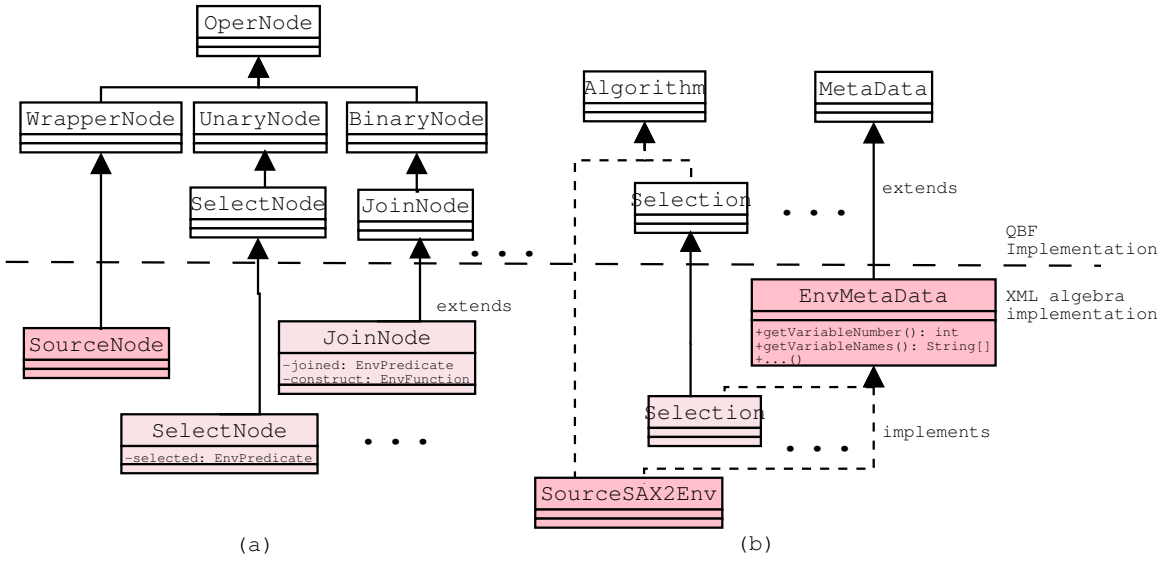



Figure 8.4: Diagramme de classes de l'implémentation de l'algèbre XML

génériques correspondants) et l'interface EnvMetaData pour l'accès à la structure *Env*. Notons que l'extension des opérateurs génériques (e.g. sélection, jointure, etc.) consiste essentiellement à définir de nouveaux types de prédicat (e.g. EnvPredicate pour la comparaison des données de structure *XTree* et *Env*) et de nouvelles fonctions (EnvFunction pour la construction des résultats d'une jointure par exemple). La structure de données de chacun de ces opérateurs ainsi que l'implémentation des algorithmes restent inchangés et sont donc réutilisés.

#### 8.2.4.2 Optimisation

L'optimisation concerne principalement le composant PlanManager de QBF qui recouvre la génération de l'espace de recherche, l'estimation de coût et la stratégie de recherche. L'implémentation de l'optimisation consiste donc à étendre les classes correspondantes comme le montre (partiellement) la figure 8.5[7]

Du fait que l'instanciation de cet évaluateur nécessite l'ajout de nouveaux opérateurs (e.g. *Source* et *Return*), il faut définir des nouvelles opérations d'optimisation les concernant. Par exemple, MergingProjectSource correspond à la fusion des opérateurs *Project* et *Source* lors que ces deux opérateurs sont placés successivement dans le plan de requête ; MappingSourceSAX et MappingReturnSAX correspondent respectivement au choix de l'implémentation des opérateurs *Source* et *Return*.

Le traitement des paramètres de contexte consiste à ajouter quelques opérateurs spécifiques (e.g. l'opérateur *MaxSize* pour limiter le nombre maximal des résultats souhaité) et/ou à guider le Plan-

[7]Les classes en couleur foncée sont spécifiques de cet évaluateur, i.e. instanciation de QBF. D'autres appartiennent à l'implémentation de QBF.

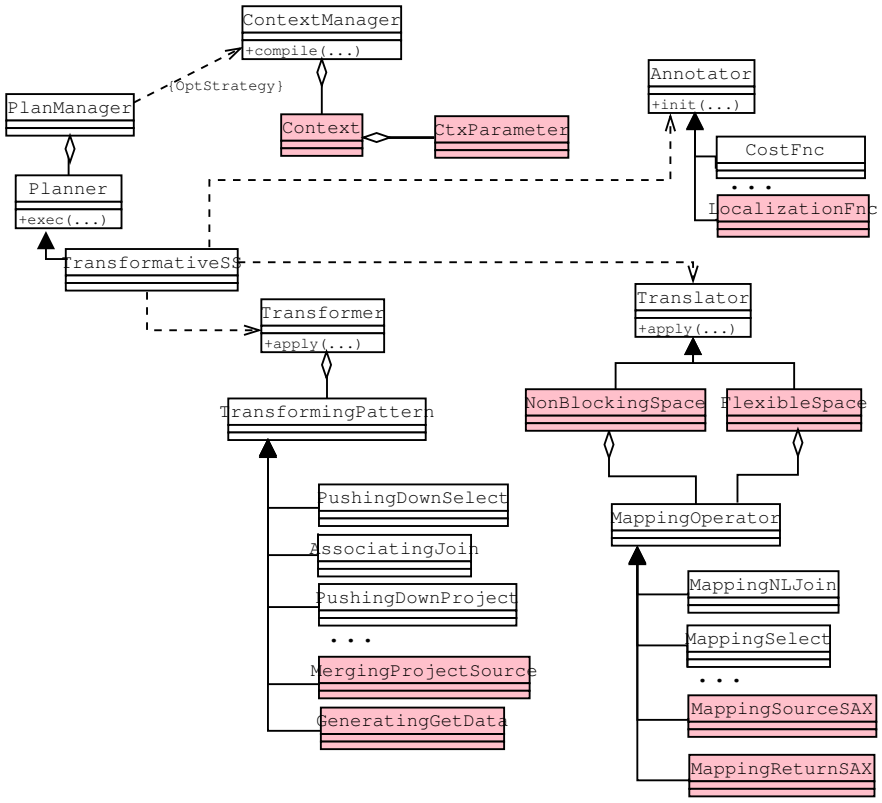



Figure 8.5: Classes implémentant l'optimisation

Manager dans le choix de l'espace de recherche à considérer. Par exemple, deux espaces physiques NonBlockingSpace et FlexibleSpace sont définis pour limiter le nombre de plans énumérés dans les cas différents, l'un ne contient que les plans constitués des algorithmes non-bloquants pour produire les premiers résultats dans un meilleur délai, et l'autre contient les plans constitués des algorithmes facilitant la production des résultats partiels (e.g. l'extension de la jointure symétrique présenté dans la section 7.2.3). Notons que la définition de ces espaces consiste à choisir des opérations d'optimisation à appliquer. En conséquence, ces opérations peuvent être partagées entre les espaces, facilitant leur définition. La stratégie de recherche (implémentée par Planner) peut être utilisée sans aucune modification.

En ce qui concerne la décomposition de requête, il s'agit d'un processus d'annotation de plan (assuré par LocalizationFnc). Concrètement, les nœuds du plan sont annotés par l'identification de la source pouvant le résoudre (*url*). En utilisant la description de la capacité des sources stockée dans les méta-données, le processus d'annotation parcourt le plan selon l'ordre post-fixe (e.g. commençant par les feuilles en remontant le plan vers la racine). Enfin, les opérateurs dont l'exécution peut être assurée par une source sont regroupés en un opérateur *getData* par l'application de GeneratingGetData.

La réalisation du processus d'optimisation décrit dans la figure 8.3 avec QBF peut être résumé d'une manière simple comme suit :

1. appeler la méthode `compile` sur le composant ContextManager (pour le *traitement de contexte*)
   1.1. inserer de nouveaux opérateurs, si nécessaire.
   1.2. déterminer les paramètres pour créer la stratégie d'optimisation.
2. décomposer la requête en commençant par créer une stratégie d'optimisation spécifique `Planner(Transformer,null,Annotator)`[8].
   2.1. appeler la méthode `init(Localization)` sur Annotator (pour l'annotation)
   2.2. appeler la méthode `apply` sur Transformer pour regrouper les opérateurs à envoyer aux sources (pour la *décomposition*). Il consiste, d'une part, à pousser les opérateurs unaires pouvant être résolus par les sources vers les feuilles respectives et à remplacer les sous-plans par des opérateurs *getData* si possible.
3. optimiser la requête en commençant par créer la stratégie d'optimisation approprié `Planner(Transformer,Translator,Annotator)`.
   3.1. appeler la méthode `apply` sur Transformer (pour l'*optimisation logique*), puis
   3.2. appeler la méthode `apply` sur Translator (pour l'*optimisation physique*).
   3.2. créer les éléments de surveillance (PropertyMonitor) nécessaires.

#### 8.2.4.3 Adaptation

En se basant sur QBF, l'adaptation de l'évaluation de requêtes concerne principalement les deux composants Monitor et RuleManager, le premier pour la détection des changements produits pendant l'exécution de requête et le second pour l'adaptation de l'exécution de la requête en cours. La détection des changements utilise les mécanismes de surveillance susceptibles de recueillir les informations pendant l'exécution de requête. Quelques mécanismes sont déjà fournis dans l'implémentation de QBF tels que la collection des statistiques sur les données, le temps, etc. (cf. le chapitre 5). Pour l'instant, nous nous intéressons à la détection des délais d'accès aux données et donc la classe TimingMonitor, déjà implémentée, permettant de collectionner les informations concernant le temps (d'attente dans notre cas) peut être utilisé. Effectivement, nous l'utilisons en ajoutant la définition de la condition dans laquelle un délai (i.e. changement) est constaté (180 secondes[9]). Ainsi, les opérations de base de l'adaptation telles que `replace_oper`, `replace_algo`, `insert_oper`, `reorder_join`, etc. peuvent être réutilisées (comme étant une bibliothèque des opérations d'adaptation). Alors, la spécification des capacités d'adaptation revient à l'utilisation de ces éléments pour définir des règles.

Nous illustrons ici la définition des règles pour réaliser les techniques *Query Scrambling* proposées dans [UFA98] (cf. la section 3.4.2) en considérant les deux règles *Rescheduling* et *Oper_Synthesis* dans la figure 8.6.

[8] La valeur `null` signifie qu'aucun `Translator` n'est choisi. En effet, la décoposition ne cherche qu'à identifier les opérateurs logiques résolus par les sources.

[9] Notons que ce délai n'est donnée qu'à titre d'exemple et sa valeur peut être changée facillement.

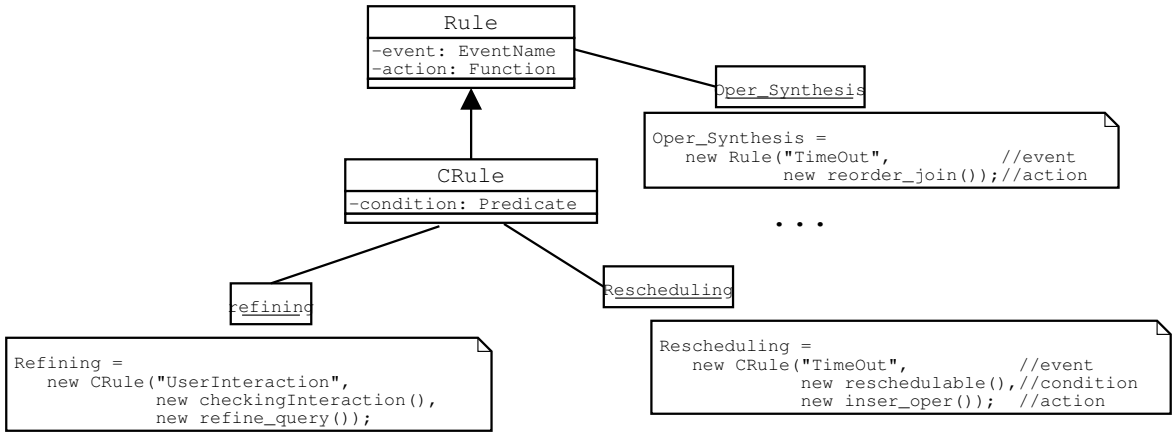



Figure 8.6: Implémentation des règles

Pour la détection des événement `TimeOut`, les éléments TimingMonitor sont créés par la phase d'optimisation pour chacune des entrées (i.e. les noeuds feuilles du plan de requête). Rappelons que chaque événement contient au moins une propriété (*`Source`*) définissant l'origine de l'événement. Dans ce cas, il correspond à la source dont l'accès est bloqué. La condition `reschedulable` vérifie en fait, s'il y a un sous-plan qui peut être exécuté malgré le délai détecté[10]. La première adaptation à réaliser (règle *Rescheduling*) correspond à l'insertion d'un opérateur de matérialisation (BufferNode) comme nœud parent du nœud racine d'un sous-plan qui peut être exécuté pendant que le délai détecté persiste. La deuxième adaptation (règle *Oper_Synthesis*) correspond à réordonner les opérateurs de jointure dans le plan. Il consiste en un retour à l'optimisation en appliquant un sous-ensemble restreint des opérations d'optimisation. Plus précisément dans ce cas, il s'agit seulement les opérations d'optimisation concernant la jointure.

Ici, nous considérons une stratégie d'adaptation où la règle *Oper_Synthesis* ne peut être déclenchée que lorsque la règle *Rescheduling* ne peut plus l'être.

## 8.3 Conclusion du chapitre

Dans ce chapitre, nous avons illustré l'utilisation de QBF ainsi que des mécanismes présentés tout au long de ce document pour construire un évaluateur de requêtes concret. Elle concerne essentiellement l'étape d'instanciation et le niveau d'adaptabilité statique de notre approche. Il est important de rappeler que pour autoriser des adaptations ultérieures (i.e. personnalisation et adaptation dynamique), il est nécessaire au moment de l'instanciation de prévoir les possibilités d'adaptation de l'évaluateur et de faire des choix en conséquence. Ainsi, l'instanciation de QBF implique le choix des mécanismes d'évaluation quels que soient statique, personnalisée, adaptative et/ou interactive, et leur mise en place.

[10]Cette vérification se fait par un parcours du plan.

Il est temps de réviser la proposition vis-à-vis de notre objectif de départ : un support (service) pouvant être adapté aux mieux aux besoins de l'application qui va l'utiliser et au contexte dans lequel il va être utilisé ; l'adaptation peut être réalisée à trois moments différents correspondant aux trois formes de l'adaptation, à savoir l'adaptation statique, la personnalisation et l'adaptation dynamique. Pour ce fait, nous résumons, en guise de conclusion, le processus de développement d'évaluateurs de requêtes basé sur QBF[11].

-1. Spécification de QBF (cf. le chapitre 4).

0. Implémentation de QBF (cf. le chapitre 5).

*... préparation à l'adaptation statique* $\surd$

1. Définition des structures de données (MetaData, DataSet) et des opérateurs de manipulation associés (OperNode, Algorithm).

2. Définition des stratégies d'optimisation possibles (Planner, Transformer, TransformingPattern, Translator,MappingOperator, etc.). Notons que plusieurs stratégies peuvent être utilisées comme étant un moyen d'adaptation de la fonction d'optimisation à des circonstances issues du traitement des requêtes spécifiques.

*... préparation à la personnalisation* [12] $\surd$

3. Définition des possibilités d'adaptation (PropertyMonitor, Rule)

*... préparation à l'adaptation dynamique*[13] $\surd$

4. Compilation du programme (i.e. évaluateur)

*... réalisation de l'adaptation statique* $\surd$

5. Lancement de l'évaluateur. Ce dernier reçoit les requêtes (personnalisées, si elles sont autorisées), procède à des traitements spécifiés antérieurement incluant les adaptations (si nécessaire).

*... réalisation de la personnalisation et de l'adaptation dynamique*[14] $\surd$

[11] Notons que les résultats des phases dénotées (-1.) et (0.) sont considérés comme le point de départ de ce processus et qu'à chaque phase, la "définition" d'un ou de plusieurs élément(s) signifie parfois le *choix* d'un ou de plusieurs élément(s) déjà implémenté(s) dans l'implémentation de QBF ou peut utiliser ces éléments.

[12] Notons que la préparation à la personnalisation est nécessaire seulement si une telle adaptabilité est souhaitée.

[13] De même que la personnalisation, cette phase n'est pas obligatoire.

[14] Seulement si ces adaptations ont été prévues.

# CHAPITRE 9

# Conclusion

Ce mémoire a présenté notre proposition pour la construction d'évaluateurs adaptables de requêtes. Les trois niveaux d'adaptabilité considérés sont l'adaptabilité statique, la personnalisation et l'adaptabilité dynamique (cf. le chapitre 3). L'approche proposée se fonde sur la définition d'un canevas d'évaluation de requêtes nommé QBF (cf. le chapitre 4). La mise en œuvre de ce canevas fournit les mécanismes de base pour réaliser essentiellement l'adaptabilité statique mais aussi la personnalisation et l'adaptabilité dynamique (cf. le chapitre 5). Ces deux dernières ont été examinées de nouveau au regard de leur influence sur l'optimisation et l'exécution de requêtes (cf. les chapitres 6 et 7). Nous rappelons que ces trois niveaux d'adaptabilité ne sont pas exclusifs. Effectivement, la réalisation de l'adaptation statique peut impliquer des choix concernant la réalisation de deux autres formes de l'adaptation comme le montre l'exemple de l'instanciation de QBF (cf. le chapitre 8).

Nous pouvons à présent dresser un bilan de cette thèse (la section 9.1) et évoquer ses quelques perspectives (la section 9.2).

## 9.1 Bilan du travail effectué

Etant donné l'objectif de revisiter l'architecture de l'évaluateur de requêtes, dans le but de le reconstruire sous forme d'un support adaptable de l'évaluation de requêtes, nous avons commencé ce travail par une étude bibliographique. Cette étude couvre aussi bien les aspects propres à l'évaluation de requêtes (i.e. les techniques d'optimisation et d'exécution de requêtes) que les aspects concernant l'architecture logicielle. Elle nous a permis de comprendre comment les travaux existants supportent l'adaptabilité telle que nous l'entendons. Les constats issus de ce premier volet sont à l'origine de la solution proposée permettant les trois niveaux d'adaptabilité souhaités. Les travaux de cette thèse concernent les différents aspects de l'informatique que sont l'architecture logicielle, l'algorithmique ou, plus concrètement, les techniques d'évaluation de requêtes et l'expérimentation. Dans la suite de cette section, nous faisons le bilan des travaux effectués selon ces aspects.

### 9.1.1 Architecture

L'étude des travaux existants concernant l'adaptabilité dans les évaluateurs de requêtes nous a montré un "interstice" entre les travaux axés sur l'architecture (extensible) de l'évaluateur de requêtes et ceux concernant les techniques d'évaluation complexes. Effectivement, les travaux sur l'architecture extensible offrent des facilités de construction d'évaluateurs ayant des caractéristiques spécifiques mais ils considèrent essentiellement les techniques d'évaluation statique et n'autorisent donc aucune adaptation ultérieure. A contrario, les travaux sur les techniques d'évaluation de requêtes avancées, notamment celles de l'évaluation personnalisée, adaptative et interactive, consistent en une collection des techniques conçues de façon isolées plutôt qu'une architecture bien définie et compréhensible. Ces techniques sont complexes et sont souvent conçues ensemble pour répondre à un ou quelques besoin(s) spécifique(s). Elles sont donc, par essence, inextensibles et inadaptables (à différents contextes d'utilisation).

Partant de cette constatation, nous avons proposé QBF (*Query Broker Framework*), un canevas pour la construction d'évaluateurs adaptables de requêtes. En considérant les trois niveaux d'adaptabilité définis, la spécification de QBF couvre aussi bien les aspects de l'évaluation statique que ceux permettant l'évaluation personnalisée, adaptative et/ou interactive. Ces aspects sont représentés dans QBF sous forme d'interfaces, ou composant, et leurs interactions ont été également présentées. Il s'agit de patrons définissant la collaboration entre des composants dans le but de réaliser les tâches de l'évaluation de requêtes, telles que l'optimisation, l'adaptation, etc. tout en considérant l'indépendance de leur réalisation.

A la différence des travaux existants, QBF fournit un cadre uniforme pour la mise en œuvre des techniques, qu'elles soient pour une évaluation statique, personnalisée, adaptative et/ou interactive, toujours dans l'optique de faciliter leur exploitation, leur extension, leur adaptation ainsi que la définition de nouvelles techniques.

### 9.1.2 Techniques d'évaluation de requêtes

Nous avons analysé un nombre important de travaux existants sur les techniques d'évaluation de requêtes, notamment ceux de l'évaluation adaptative et interactive. Nous avons défini un cadre permettant la comparaison de ces techniques qui se distinguent autant par leur forme (i.e. la façon dont les techniques sont mises en œuvre) que par leur capacité (i.e. les cas que les techniques peuvent traiter). Cette comparaison a mis en évidence les éléments plus ou moins orthogonaux de ces techniques. Elle a permis, d'une part, de comprendre la nature des techniques proposées quelque soit leur forme (i.e. architecture, sujet, origine et mécanismes de réalisation d'adaptation, etc.) ce qui est à l'origine de notre proposition de QBF, et d'autre part, elle nous a montré l'absence de certaines considérations dans ces techniques que nous considérons ci-après.

Nous avons revisité le processus d'évaluation de requêtes afin de prendre en compte des besoins spécifiques de l'utilisateur (i.e. évaluation personnalisée) d'une manière systématique. Il consiste essentiellement à réduire l'espace de recherche de la requête évaluée. L'analyse des mécanismes d'évaluation personnalisée nous a montré également l'importance et le bénéfice de la décomposition et de la séparation des aspects au sein de l'évaluateur de requêtes telles qu'elles sont proposées dans QBF.

Pour l'évaluation dynamique de requêtes, nous avons proposé quelques mécanismes pour l'évaluation interactive de requêtes. Il s'agit des mécanismes pour la construction de résultats partiels et pour la transformation de requête en cours d'exécution afin de prendre en compte l'évolution des besoins utilisateur.

Les mécanismes proposés dans cette thèse sont considérés comme des compléments de ceux de l'évaluation adaptative et interactive dans le but de mieux assurer l'adaptabilité de l'évaluateur de requêtes. Certains de ces mécanismes peuvent être intégrés aisément dans les systèmes existants et d'autres permettent l'utilisation des mécanismes existants dans le cadre uniforme qu'est QBF.

### 9.1.3 Expérimentation

Pour valider notre approche, nous avons implémenté les différents composants de QBF et les avons utilisés pour construire des évaluateurs de requêtes. L'expérimentation se réalise surtout pour montrer la faisabilité de notre approche. Nous avons également évoqué les critères d'évaluation de l'approche proposée.

L'implémentation de QBF, quant à elle, peut être considérée comme un évaluateur minimal permettant l'exécution des requêtes basée sur une algèbre de "cursor". Notre prototype est réalisé en java en utilisant la bibliothèque XXL [vdBDS00] contenant quelques algorithmes des opérateurs de requête. Nous avons implémenté quelques algorithmes supplémentaires comme la jointure par hachage (symétrique) et les opérateurs proposés dans cette thèse. Une instance de QBF pour l'interrogation des données relationnelles a été réalisée. Cette instance a été étendue par la suite pour construire un évaluateur de requêtes dans le cadre du projet MediaGRID. Pour cet évaluateur, nous avons implémenté une algèbre XML suivant l'approche "variable liée". Par ailleurs, les mécanismes de surveillance ont été également implémentés. L'utilisation de l'implémentation de QBF pour construire des évaluateurs de requêtes nous a montré le gain important de la réutilisation de QBF. Par exemple, lors de la construction de l'évaluateur pour MediaGRID, nous n'avons du ajouter aucune ligne de code concernant les mécanismes de surveillance. Par ailleurs, une grande partie de l'implémentation des opérateurs génériques sont réutilisée pour les opérateurs spécifiques de l'évaluateur de MediaGRID.

Nous avons également réalisé des mesures expérimentales. Ce que nous voulions mesurer est le coût de l'adaptabilité dans notre approche. A ce propos, nos premières mesures concernent le coût de la surveillance. Il consiste à déterminer le coût (en terme de temps dans notre expérimentation) ajouté à l'exécution d'une requête lorsque les éléments de surveillance se présentent. L'objectif de cette mesure est, d'une part, d'évaluer les mécanismes de surveillance implémentés, et d'autre part, de fournir aux programmeurs les indications sur le coût de la surveillance des différentes propriétés de l'exécution, à différentes fréquences afin de les aider dans le choix des éléments appropriés pour leur évaluateur. Par exemple, notre expérimentation a montré que l'ajout des éléments de surveillance la taille de données est excessivement critique pour la performance du système d'évaluation de requêtes. Cela conduit à l'identification de quelques règles d'utilisation des éléments de surveillance dans les cas différents. D'autres mesures seront l'objet des travaux futurs que nous évoquons ci-après.

## 9.2 Perspectives

Les travaux réalisés dans cette thèse ne sont que les premiers pas vers la définition d'un outil *efficace* pour la construction d'évaluateurs adaptables de requêtes. L'efficacité se manifeste, d'une part, dans la performance des systèmes construits en utilisant cet outil, et d'autre part, dans la possibilité et la facilité de l'utiliser. Nous évoquons donc les perspectives de cette thèse selon ces deux directions.

### 9.2.1 Considération d'expérimentation

Suite aux premières mesures que nous avons réalisées, nous voudrions poursuivre l'évaluation expérimentale de notre approche selon d'autres critères.

**Adaptation dynamique** Dans cette thèse, nous avons implémenté un mécanisme de surveillance de l'exécution de requêtes et réalisé les mesures sur cette implémentation. Il serait intéressant de pouvoir comparer ce mécanisme de surveillance avec d'autres. D'ailleurs, comment mesure-t-on le coût de la prise de décision et de la réalisation de l'adaptation ? Comment compare-on l'efficacité des différentes techniques dans des différents contextes d'utilisation ? Il pourrait être intéressant de définir un banc d'essais pour ce contexte. Quant à l'évaluation interactive de requêtes, la transformation de requête en cours d'exécution (au lieu de redémarrer l'exécution de la requête), est-il *toujours* bénéfique ?

**Personnalisation** En ce qui concerne l'évaluation personnalisée, notre approche consiste à limiter la taille de l'espace de recherche comme étant un moyen d'améliorer la tâche d'optimisation de requête. Cela permet de réduire le nombre de plans à examiner mais ajoute des coûts concernant la prise de décision lors de la génération de l'espace de recherche. Il serait intéressant de mesurer ces coûts, de voir l'avantage (vs. l'inconvénient) de ces choix vis-à-vis l'objectif d'optimisation de requête.

**Adaptation statique** Comme nous l'avons déjà mentionné (cf. le chapitre 5), la séparation des tâches au sein de l'évaluateur de requêtes ajoute des coûts dus aux interactions entre les composants. Ce coût pourrait être plus ou moins important selon la nature de cette communication (e.g. entre des objets, entres des sites distribués, etc.). Il serait intéressant de l'étudier pour évaluer notre approche et d'examiner les possibilités pour l'améliorer en utilisant, par exemple, des techniques d'optimisation de code [BBPH03].

### 9.2.2 Consolidation de l'approche QBF

Nous considérons maintenant les perspectives liées à la définition de notre approche selon les trois niveaux d'adaptabilité souhaités.

**Adaptabilité statique (chapitre 5,8)** *"Faut-il être programmeur chevronné pour utiliser QBF ...?"* a été le commentaire d'un lecteur anonyme sur un article décrivant QBF. Bien que nous ayons montré, dans ce document, l'utilisation de QBF ainsi que les "gains" apportés, il est toujours difficile d'avoir une réponse précise à cette question. En effet, étant concepteur et utilisateur de QBF, notre réponse

quelle qu'elle soit n'est peut-être pas très objective. Nous pensons qu'une documentation sur la conception détaillée de QBF, et bien davantage d'exemples de cas d'utilisation (i.e. une sorte de "livre de recettes"), seraient utiles pour qu'un "simple" programmeur puisse utiliser notre canevas pour construire son évaluateur.

Par ailleurs, l'instanciation de QBF requiert l'extension de ses composants (e.g. les opérations d'optimisation, les éléments de surveillance, les règles d'adaptation, etc.) et leur composition (e.g. l'ordre d'application des opérations d'optimisation, la stratégie d'adaptation), il est donc difficile de garantir la "correction" des codes ajoutés. Il est souhaitable de fournir les indications aux concepteurs et programmeurs d'un système d'interrogation particulier, ou, encore mieux, d'insérer des contrôles, afin de prévenir autant que possibles les erreurs de l'instanciation. Par exemple, le contrôle de la taille de l'espace de recherche généré, des boucles dans le gestionnaire d'événements, etc.

**Personnalisation (chapitre 6)** En ce qui concerne l'évaluation personnalisée de requête, notre travail s'est focalisé sur la nécessité de rendre les différentes tâches de l'évaluation de requête "configurables" et propose des "moyens" pour l'assurer. Quant à la capacité d'expression des besoins de la personnalisation, nous avons adopté la représentation sous forme attribut-valeur qui est relativement simple. Cependant, nous pensons que ce modèle de représentation n'est pas facile à comprendre et à utiliser par les *utilisateurs finals*. Il serait intéressant de considérer d'autres modèles de représentation plus complexes tel que le modèle de graphe proposé dans [KI04, KI05].

**Adaptabilité dynamique (chapitre 7)** Les problèmes liés à l'évaluation interactive de requêtes ne sont que partiellement considérés dans ce travail de thèse. Il est, comme nous l'avons déjà souligné, indispensable d'aborder les aspects de l'interface homme machine, qui, en elle-même, est un sujet de recherche. Comment l'application cliente affiche-t-elle les résultats partiels retournés par le système ? Comment l'utilisateur peut-t-il soumettre les informations aux systèmes au cours d'exécution ? Ce sont les questions pouvant se poser. Une interface graphique où l'utilisateur peut voir les résultats (incomplets), naviguer en utilisant des "scroll bar", en "cachant" certaines parties des données, etc. pourrait faciliter l'interaction d'utilisateur. Les actions perçues par l'interface seraient à l'origine des événements correspondant à ceux produits par nos opérateurs d'interaction afin que ces interactions soient prises en compte par le moteur d'exécution de requêtes pour s'adapter.

En ce qui concerne l'évaluation adaptative, qui n'était pas considérée en priorité dans ce travail (excepté les efforts de le situer, conceptuellement ainsi que pratiquement,dans un cadre uniforme), nous pensons que plusieurs aspects devraient être revus : (i) quels sont les contextes les plus appropriés pour chacune (ou un ensemble) des mécanismes existants ? (ii) la réalisation de ces mécanismes dans l'environnement largement dispersé et dynamique (e.g. les GRIDs) nécessite de définir un protocole de communication complexe entre des sites ; et (iii) comment le système prend-il en compte l'état actuel de stockage de données ? En effet, la plupart des techniques existantes ne considèrent pas l'état de stockage temporaire de données en appliquant à nouveau l'optimisation bien qu'une partie des données aient déjà été lues dans la mémoire et/ou matérialisées localement. Cela pourrait rendre la ré-optimisation

inefficace. Il serait alors intéressant de voir comment les stockages temporaires de données sont pris en compte lors de la ré-optimisation.

Ce sont les perspectives liées étroitement au développement de notre approche. D'autres peuvent aussi s'ouvrir vers l'interaction de notre travail avec d'autres travaux dont certains sont réalisés dans notre groupe : le service de règles [GR00] et le service d'événements [VS00] pour les aspects de l'évaluation dynamique, le service de persistance et le service de cache [Gar03] pour améliorer l'évaluation de requêtes, ou encore les travaux sur la localisation dans les systèmes P2P [dPVRL04] pour le déploiement de QBF et ses instances dans les environnements largement distribués, etc.

# Bibliographie


[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems (TODS)*, 1(2), 1976.

[AFGK02] Luis F. Andrade, José L. Fiadeiro, Joao Gouveia, and Georgios Koutsouko. Separating Computation, Coordination and Configuration. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5), 2002.

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling Query Plans to cope with Unexpected Delays. Dans *Proc. of International Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2000.

[BBPH03] Sara Bouchenak, Fabienne Boyer, Noel De Palma, and Daniel Hagimont. Can Aspects Be Injected? Experience with Replication and Protection. Dans *Proc. of International Symposium on Distributed Objects and Applications (DOA)*, 2003.

[BCD89] François Bancilhon, Sophie Cluet, and Claude Delobel. A Query Language for the O2 Objet Oriented Database System. Dans *Proc. of International Workshop on Database Programming Languages (DBPL)*, 1989.

[BDK91] François Bancilhon, Claude Delobel, and Paris Kanellaskis, editors. *Buildding an Object Oriented Database System: the Story of O2*. Morgan Kaufmann, San Mateo, CA, 1991.

[BDK+03] Ingo Brunkhorst, Hadhami Dhraief, Alfons Kemper, Wolfgang Nejdl, and Christian Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P Systems. Dans *Proc. of International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P)*, 2003.

[BFMV00a] Luc Bouganim, Françoise Fabret, Chandrasekaran Mohan, and Patrick Valduriez. A Dynamic Query Processing Architecture for Data Integration Systems. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2000.

[BFMV00b] Luc Bouganim, Françoise Fabret, Chandrasekaran Mohan, and Patrick Valduriez. Dynamic Query Scheduling in Data Integration Systems. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 2000.

[BKS01] Stephan Borzsonyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 2001.

[BO03] Brian Babcock and Chris Olston. Distributed Top-K Monitoring. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2003.

[Bon99] Philippe Bonnet. *Prise en compte des sources de données indisponibles dans les systèmes de médiation*. Thèse de Doctorat, Université de Savoie, 1999.

[Bou04] Coordinateur : Mokrane Bouzeghoub. Action Spécifique sur la Personnalisation de l'Information". Rapport technique AS98/RTP9, CNRS, 2004.

[BT98] Philippe Bonnet and Anthony Tomasic. Partial Answers for Unavailable Data Sources. Dans *Proc. of International Conference on Flexible Query Answering Systems (FQAS)*, 1998.

[CBB+97] R. G. G. Cattel, Douglas K. Barry, Dirk Bartels, Mark Berler, Jeff Eastman, David Jordan Sophie Garmerman, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard : ODMG 2.0*. Morgan-Kaufmann, 1997.

[CBB+04] Christine Collet, Khalid Belhajjam, Gilles Bernot, Chrsitophe Bobineau, Gennaro Bruno, Béatrice Finance, Fabrice Jouanot, Zoubida Kedad, David Laurent, Fariza Tahi, Genoveva Vargas, Tuyet-Trinh Vu, and Xiaohui Xue. Toward a Mediation System Framework for Transparent Access to largely Distributed Sources. Dans *Proc. of International Conference on Semantic Network Web (ICSNW)*, 2004. Also published in Proc. of "Distribution de données à grande échelle" (DRUIDE) Spring School, 2004.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of Dynamic Query Evaluation Plans. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1994.

[CGM03] Junghoo Cho and Hector Garcia-Molina. Estimating Frequency of Change. *ACM Transactions on Internet Technology (TOIT)*, 3(3), 2003.

[CHS+95] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. Dans *Proc. of International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management (DOM)*, 1995.

[CK97] Michael J. Carey and Donald Kossmann. On Saying "Enough Already!" in SQL. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1997.

[CK98] Michael J. Carey and Donald Kossmann. Reducing the Breaking Distance of an SQL Query Engine. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1998.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communication of the ACM (CACM)*, 6(13), 1970.

[Cod80] E. F. Codd. Data Models in Database Management. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1980.

[DD93] C.J. Date and Hugh Darwen. *A Guide to The SQL Standard*. Addison-Wesley, 1993.

[DD99] Ruxandra Domenig and Klaus R. Dittrich. An Overview and Classification of Mediated Query Systems. *SIGMOD Record*, 1999.

[DFF+98] Alain Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Xml-ql: A query language for xml. Dans *WWW The Query Language Workshop (QL)*, 1998.

[DG97] Olivier M. Duschka and Michael R. Genesereth. Query Planning in Infomaster. Dans *Proc. of ACM Symposium on Applied Computing (SAC)*, 1997.

[DN03] Tuyet-Tram Dang-Ngoc. *Fédération de données semi-structurées avec XML*. Thèse de Doctorat, Université de Versailles Saint-Quentin-en-Yvelines, 2003.

[dPVRL04] Maria del Pilar Villamil, Claudia Rocancio, and Cyrils Labbe. PinS: Peer-to-Peer Interrogation and Indexing. Dans *Proc. of International Database Engineering and Applications Symposium (IDEAS)*, 2004.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming: Introduction. *Communication of the ACM (CACM)*, 44(10), 2001.

[FG91] Beatrice Finance and Georges Gardarin. A Rule-Based Query Rewriter in an Extensible DBMS. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 1991.

[FLMS99] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1999.

[Gar] http://www.almaden.ibm.com/cs/garlic/homepage.html.

[Gar03] Luciano Garcia. *PERSEUS: un canevas logiciel pour la construction des gestionnaires d'objets persistants*. Thèse de Doctorat, Institut National Politechnique de Grenoble, 2003.

[GD87] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1987.

[GKAF01] Joao Gouveia, Georgios Koutsoukos, Luis F. Andrade, and José L. Fiadeiro. Tool support for coordination-based software evolution. Dans *Proc. of Conferences on Technology of Object-Oriented Languages and Systems (TOOLS)*, 2001.

[GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 1993.

[GMHI+95] Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. Dans *Proc. of the AAAI Symposium on Information Gathering*, 1995.

[GMPQ+97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS Approach to Mediation : Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)*, 8(2), 1997.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.

[GPSF04] Anastasios Gounaris, Norman W. Patton, Rizos Sakellariou, and Alvaro A. A. Fernandes. Adaptive Query Processing and the Grid: Opportunities and Challenges. Dans *Proc. of International Workshop on Grid and Peer-to-Peer Comuputing Impacts on Large Scale Heterogeneous Distributed Database System (GLOBE)*, 2004.

[GR00] Helena Grazziotin-Ribeiro. *Un service de règles actives pour fédérations de bases de données*. Thèse de Doctorat, Université de Joseph Fourier, 2000.

[Gra90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1990.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 18(3), 1995.

[Gra04] Jim Gray. The Revolution in Database Systems Architecture. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2004. Keynote. Extended abstract at http://research.microsoft.com/ Gray.

[GRZZ00] Jean-Robert Gruser, Louiqa Rachid, Vladimir Zadorozhny, and Tao Zhan. Learning Reponse Time for Web Sources using Query Feedback and Application in Query Optimization. *The International Jounal on Very Large DataBases (VLDBJ)*, 9(1), 2000.

[GVD+01] Leonidas Galanis, Efstratios Viglas, David J. DeWitt, Jeffrey. F. Naughton, and David Maier. Following the paths of xml data: An algebraic framework for xml query evaluation. University of Wisconsin Madison, 2001.

[GW89] Goetz Graefe and Karen Ward. Dynamic Query Evaluation Plans. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1989.

[GWJD03] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating Data Sources in Large Distributed Systems. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2003.

[HFC+00] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive Query Processing: Technology in Evolution. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 23(2), 2000.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible Query Processing in Starburst. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1989.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1999.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon T. Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2003.

[HHW97] Joseph M. Hellerstein, Peter J. Hass, and Helen J. Wang. Online Aggregation. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1997.

[HLST01] H.V.Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. Dans *Proc. of International Workshop on Database Programming Languages (DBPL)*, 2001.

[IAE03] Ihab F. Ilyas, Walid G. Aref, and Almed K. Elmagarmid. Supporting Top-K Join Queries in Relational Databases. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2003.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc Friedman, et al. An Adaptive Query Execution System for Data Integration. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1999.

[IK90] Yannis E. Ioannidis and Younkyung Kang. Randomized Algorithms for Optimizing Large Join Queries. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1990.

[IK91] Yannis E. Ioannidis and Younkyung Kang. Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1991.

[ILW+00] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive Query Processing for Internet Applications. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 20(2), 2000.

[ILW04] Zachary Ives, Alon Levy, and Daniel S. Weld. Adapting to Source Properties in Processing Data Integration Queries. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2004.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1992.

[INSS97] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. *The International Jounal on Very Large DataBases (VLDBJ)*, 6(2), 1997.

[Ioa00] Yannis E. Ioannidis. *Query Optimization*, chapter 45. CRC Press, 2000.

[Ive02] Zachary Ives. *Efficient Query Processing in Data Intergration Systems*. Thèse de Doctorat, University of Washington, 2002.

[IW87] Yannis E. Ioannidis and Eugene Wong. Query Optimization by Simulated Annealing. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1987.

[JKR99] Vanja Josifovski, Timour Katchaounov, and Tore Risch. Optimizing Queries in Distributed and Composable Mediators. Dans *Proc. of International Conference on Cooperative Information Systems (CoopIS)*, 1999.

[Joh97] Ralph E. Johnson. Framework = Components + Patterns. *Communication of the ACM (CACM)*, 40(10), 1997.

[KCC+03] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel R. Madden, Frederick Reiss, and Mehul A. Shah. TelegraphCQ : An Architectural Status Report. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26(1), 2003.

[KD98] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1998.

[KD99] Navin Kabra and David. J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *The International Jounal on Very Large DataBases (VLDBJ)*, 1999.

[KI04] Georgia Koutrika and Yannis Ioannis. Personnalization of Queries in Database Systems. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 2004.

[KI05] Georgia Koutrika and Yannis Ioannis. Personalized Queries under a Generalized Preference Model. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 2005.

[KK02] Werner Kießling and Gerhard Köstler. Preference SQL - Design, Implementation, Experience. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2002.

[Led01] Coordonator: Thomas Ledoux. Etat de l'art sur l'adaptabilité. Rapport technique D1.1, Projet RNTL ARCAD, 2001.

[leS] Le select. http://www-caravel.inria.fr/LeSelect/.

[Lev01] Alon Levy. Answering Queries using Views : A Survey. *The International Jounal on Very Large DataBases (VLDBJ)*, 10(4), 2001.

[LFL88] Mavis K. Lee, Johann Christoph Freytag, and Guy M. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1988.

[LIST03] Alon Y. Levy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema Mediation in Peer Data Management Systems. Dans *Proc. of International Conference on Data Engineering (ICDE)*, 2003.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-Answering Algorithms for Information Agents. Dans *Proc. of National Conference on Artificial Intelligence (AAAI)*, 1996.

[LV91] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the Search Strategy in a Query Optimizer. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1991.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), 1997.

[Man01] Ioana Manolescu. *Optimization techniques for querying distributed heterogeneous data sources*. Thèse de Doctorat, Université de Versailles Saint-Quentin-en-Yvelines, 2001.

[MBHT96] William J. McKenna, Louis Burger, Chi Hoang, and Melissa Truong. EROC: A Toolkit for Building NEATO Query Optimizers. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1996.

[MED03] MEDIAGRIDTeam. A mediation framework for a transparent access to biological data source. Dans *Poster session at Internal Conference on Conceptual Modeling (ER).*, 2003. Long version published in EMISA Forum.

[NKT+97] Hubert Naacke, Olga Kapitskaia, Antony Tomasic, Philippe Bonnet, Louiqa Raschid, and Remy Amouroux. The Distributed Information Search Component (DISCO) and the World Wide Web. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1997.

[NSS86] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. Simulated Annealing and Combinatorial Optimization. Dans *Proc. of the 23rd ACM/IEEE Conference on Design automation*, 1986.

[OV99] Tamar Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition edition, 1999.

[PGLK97] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. The Complexity of Transformation-based Join Enumeration. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1997.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule-based Query Rewrite Optimization in Startburst. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1992.

[PL00] Rachel Pottinger and Alon Levy. A Scalable Algorithm for Answering Queries Using View. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2000.

[PRI03] PRISM. Génération de requêtes de médiation. Rapport technique, PRISM. Délivrable D2.1 du projet Mediagrid, 2003.

[PY00] Christos H. Papadimitriou and Mihalis Yannakakis. On the Approximability of Trade-offs and Optimal Access of Web sources. Dans *Proc. of IEEE Symposium. on Foundations of Computer Science*, 2000.

[PY01] Christos H. Papadimitriou and Mihalis Yannakakis. Multiobjective Query Optimization. Dans *Proc. of Symposium on Principles of Database Systems (PODS)*, 2001.

[RH02] Vijayshankar Raman and Joseph M. Hellerstein. Partial Results for Online Query Processing. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2002.

[Riv04] Michel Riveill. Service composition. Dans *Lecture at 3rd LAFMI Summer School on Distributed Systems*. LAFMI, 2004.

[ROH96] Mary Tork Roth, Fatma Ozcan, and Laura M. Hass. Cost models do matter: Providing cost information for diverse data sources in a federated system. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1996.

[RRH99] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1999.

[SA02] Carlo Sartiani and Antonio Albano. Yet Another Query Algebra for XML Data. Dans *Proc. of International Database Engineering and Applications Symposium (IDEAS)*, 2002.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1979.

[SFF94] Veronique Smahi, Jerome Fessy, and Beatrice Finance. Query Processing in IRO-DB. Rapport technique 1994/37, PRiSM Laboratory, 1994.

[SG88] Arun Swami and Anoop Gupta. Optimization of Large Join Queries. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1988.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's Learning Optimizer. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2001.

[Swa89] Arun N. Swami. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1989.

[SWKH76] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems (TODS)*, 1(3), 1976.

[TGO99] Kian-Lee Tan, Cheng Hian Goh, and Beng Chin Ooi. Online Feedback for Nested Aggregation Queries with Multi-Threading. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 1999.

[TL04] Igor Tatarinov and Alon Levy. Efficient Query Reformulation in Peer Data Management Systems. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2004.

[UF00] Tolga Urhan and Michael J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 23(2), 2000.

[UF01] Tolga Urhan and Michael J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. Dans *Proc. of International Conference in Very Large DataBases (VLDB)*, 2001.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost Based Query Scrambling for Initial Delays. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 1998.

[VC03] Tuyet-Trinh Vu and Christine Collet. Query Brokers for Distributed and Flexible Query Evaluation. Dans *Actes de la Conférence Internationale Associant Chercheurs Vietnamiens et Francophones en Informatique (RIVF)*, 2003.

[VC04a] Tuyet-Trinh Vu and Christine Collet. Adaptable Query Evaluation using QBF. Dans *Proc. of International Database Engineering and Applications Symposium (IDEAS)*, 2004.

[VC04b] Tuyet-Trinh Vu and Christine Collet. QBF: A Query Broker Framework for Adaptable Query Evaluation. Dans *Proc. of International Conference on Flexible Query Answering Systems (FQAS)*, 2004.

[VCB03] Tuyet-Trinh Vu, Christine Collet, and Christophe Bobineau. Spécification de l'évaluateur de requêtes. Rapport technique, LSR-IMAG. Délivrable D2.2 du projet Mediagrid, 2003.

[vdBDS00] Jochen van den Bercken, Jens-Peter Dittrich, and Bernhard Seeger. javax.XXL: A Prototype for a Library of Query Processing Algorithms. Dans *Proc. of International Conference on Management of Data (SIGMOD)*, 2000.

[VM03] Tuyet-Trinh Vu and MEDIAGRIDTeam. Document état de l'art, chapitre 3 et 4. Rapport technique, LSR-IMAG, PRISM and LAMI. Délivrable D1 du projet Mediagrid, 2003.

[VS00] Genoveva Vargas-Solar. *Service d'événements flexible pour l'intégration d'applications bases de données réparties*. Thèse de Doctorat, Université de Joseph Fourier, 2000.

[Vu04a] Tuyet-Trinh Vu. Evaluation adaptative et interactive de requêtes. Dans *"Distribution de données à grande échelle" (DRUIDE) Spring School*. Exposé avancé, 2004.

[Vu04b] Tuyet-Trinh Vu. Querying of Distributed Data Sources. Dans *Lecture at 3rd LAFMI Summer School on Distributed Systems (ESD)*. LAFMI, 2004.

[WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. Dans *Proc. of International Conference on Parallel and Distributed Information Systems (PDIS)*, 1991.

[XML] Extensible markup language (xml). http://www.w3.org/XML.

[XQu] Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/.

[ZW86] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. Dans *Proc. of the Hawaii International Conference on System Sciences*, 1986.

# Annexe

# ANNEXE A

# Opérateurs de requête dans QBF

## A.1 Définition des opérateurs

Etant donné que les données sont représentées dans QBF sous forme de collections d'items, l'implémentation de QBF considère des opérateurs définissant les manipulations possibles sur cette structure de données. Nous y ajoutons un opérateur *Buffer* ayant la capacité de stockage, nécessaire pour les diverses techniques d'évaluation que nous avons présentées dans ce document. Les opérateurs suivants sont implémentés dans QBF :

**L'opérateur** Selection

- Collection(s) d'entrée : un ensemble d'items
- Paramètre(s) : un *prédicat* logique $p$ spécifiant la condition de sélection.
- Sortie : un ensemble d'items satisfaisant $p$
- Algorithme(s) : la sélection (filtrer les items dans l'ordre de leur arrivée)

**L'opérateur** Join

- Collection(s) d'entrée : deux ensembles d'items
- Paramètre(s) : (i) un *prédicat* logique $p$ spécifiant la condition de jointure de deux items et (ii) une *fonction* $f$ spécifiant la construction d'un item du résultat à partir de deux items en entrée[1].
- Sortie : un ensemble d'items dont chacun est construit par l'application de la fonction $f$ sur deux items de deux entrées respectives satisfaisant $p$.

[1]En cas d'absence du paramètre $p$, l'opérateur est considéré comme le *produit cartésien*. En cas d'absence du paramètre $f$, l'item du résultat est le résultat de la concaténation de deux items en entrée.

- Algorithme(s) : (i) la jointure par boucle imbriquée, (ii) la jointure par tri fusion, (iii) la jointure par hachage, (iv) la jointure par hachage symétrique.

**L'opérateur** Union

- Collection(s) d'entrée : n ensembles d'items
- Paramètre(s) : aucun
- Sortie : un ensemble d'items contenant tous les items des entrées
- Algorithme(s) : (i) l'union séquentielle (selon l'ordre des entrées), (ii) l'union "sélective" (selon la préférence des entrées ou les propriétés des flux des données arrivées)

**L'opérateur** Intersection

- Collection(s) d'entrée : deux ensembles d'items
- Paramètre(s) : aucun
- Sortie : un ensemble d'items communs à deux ensembles en entrée
- Algorithme(s) : (i) l'intersection par boucle imbriquée, (ii) l'intersection par tri.

**L'opérateur** Difference

- Collection(s) d'entrée : deux ensembles d'items.
- Paramètre(s) : aucun
- Sortie : un ensemble d'items complémentaire de la deuxième ensemble dans le premier.
- Algorithme(s) : (i) la différence par boucle imbriquée, (ii) la différence par tri.

**L'opérateur** GroupBy

- Collection(s) d'entrée : un ensemble d'items.
- Paramètre(s) : une *fonction* de regroupement.
- Sortie : un ensemble d'items dont chacun est un ensemble d'items ayant la mêm valeur de l'application de la *fonction* de regroupement
- Algorithme(s) : groupement par boucle imbriquée.

**L'opérateur** Distinct

- Collection(s) d'entrée : un ensemble d'items.
- Paramètre(s) : aucun
- Sortie : un ensemble d'items non dupliqués.

- Algorithme(s) :

**L'opérateur** Sort

- Collection(s) d'entrée : un ensemble d'items.
- Paramètre(s) : une *fonction de tri* $f$.
- Sortie : un ensemble d'item ordonné selon $f$.
- Algorithme(s) :

**L'opérateur** Mapper

- Collection(s) d'entrée : un ensemble d'items.
- Paramètre(s) : une *fonction* $f$
- Sortie : un ensemble d'items dont chacun est le résultat de l'application de la fonction $f$ sur chaque item en entrée.
- Algorithme(s) : mapper

**L'opérateur** Buffer

- Collection(s) d'entrée : un ensemble d'items.
- Paramètre(s) : un nombre entier $n$ spécifiant la taille du tampon utilisé par cet opérateur.
- Sortie : un ensemble d'items.
- Algorithme(s) : buffer

La table A.1 résume les opérateurs de requête dans QBF.

| **Opérateur** | **Type** | **Fonction** |
|---|---|---|
| Selection | unaire | restriction |
| Jointure | binaire | jointure de deux ensembles d'items |
| Union | n-aire | collection de n entrées |
| Intersection | binaire | intersection de deux ensemble |
| Difference | binaire | différence de deux ensemble |
| Group | unaire | groupement des items |
| Distinct | binaire | élimination des duplication |
| Sort | unaire | tri |
| Mapper | unaire | |
| Buffer | unaire | stockage temporaire des données |

Table A.1: Liste des opérateurs de requête dans QBF

# ANNEXE B

# Opérateurs de requête dans MediaGRID

## B.1 Représentation intermédiaire de données XML

L'algèbre XML que nous avons utilisée dans ce travail est une extension de l'algèbre relationnelle. En considérant une représentation intermédaire de données *à la relationnelle*, la plupart des opérateurs sont définis pour cette structure nommé *Env* qui peut être représentée comme suit :

env[tuple [label$_1$[t$_1$1], ... label$_n$[t$_1n$]],
...,
tuple [label$_1$[t$_k$1], ... label$_n$[t$_k n$]]]

Intuitivement, *Env* est une collection de n-uplets (*tuple*). Chaque n-uplet est, quant à lui, une collection des variables dont le nom est $label_i$ et la valeur est un arbre $t_{ik}$ représenté un élément XML. La structure *Env* peut être considérée comme une collection de n-uplets constituant des éléments de structure d'arbre. Excepté les deux opérateurs *Source* et *Return*, tous les autres opérateurs sont définis pour *Env*.

## B.2 Définition des opérateurs

Les opérateurs suivants sont considérés dans l'évaluateur de MediaGRID :

**L'opérateur** Source

- Collection(s) d'entrée : aucune
- Paramètre(s) : le nom de la source, un filtre d'entrée spécifiant l'ensemble des variables associées à des expressions de chemin.
- Sortie : un ensemble *Env*

- Algorithme(s) : (i) une implémentation basée sur DOM (bloquant) et (ii) une implémentation basée sur SAX (non-bloquant).

**L'opérateur** Return

- Collection d'entrées : un ensemble *Env*
- Paramètre(s) : un filtre de sortie spécifiant la structure XML du résultat
- Sortie : un document XML
- Algorithme(s) : (i) une implémentation basée sur DOM (bloquant) et (ii) une implémentation basée sur SAX (non-bloquant).

**L'opérateur** Selection

- Collection d'entrées : un ensemble *Env*
- Paramètre(s) : un prédicat logique $p$ spécifiant la condition de sélection.
- Sortie : un ensemble *Env* dont les éléments satisfaisant $p$.
- Algorithme(s) : la sélection (filtrer les items dans l'ordre de leur arrivée)

**L'opérateur** Projection

- Collection d'entrées : un ensemble *Env*
- Paramètre(s) : un ensemble de variables spécifiant les éléments du résultat (i.e. les éléments à retenir).
- Sortie : un ensemble *Env* des n-uplets constitués par des variables données.
- Algorithme(s) : la projection.

**L'opérateur** Join

- Collection d'entrées : deux ensembles *Env*
- Paramètre(s) : un *prédicat* logique $p$ spécifiant la condition de jointure de deux items.
- Sortie : un ensemble *Env* dont chacun élément est le résultat de la concaténation de deux éléments de deux entrées respectives satisfaisant $p$.
- Sortie : un ensemble de *Env*
- Algorithme(s) : (i) la jointure par boucle imbriquée, (ii) la jointure par hachage symétrique et (iii) une implémentation de POSHJ comme présenté dans la section 7.2.

**L'opérateur** Union

- Collection d'entrées : deux ensembles *Env*
- Paramètre(s) : aucun
- Sortie : un ensemble *Env*
- Algorithme(s) : (i) union séquentielle (selon l'ordre des entrées), (ii) union "sélective" (selon la préférence des entrées ou les propriétés des flux des données arrivées).

**L'opérateur** Intersection

- Collection d'entrées : deux ensembles *Env*
- Paramètre(s) : aucun.
- Sortie : un ensemble *Env*
- Algorithme(s) : intersection par boucle imbriquée

**L'opérateur** Difference

- Collection d'entrées : deux ensembles *Env*
- Paramètre(s) : aucun
- Sortie : un ensemble *Env*
- Algorithme(s) : différence par boucle imbriquée

**L'opérateur** GroupBy

- Collection d'entrées : un ensemble *Env*
- Paramètre(s) : un ensemble des variables à regrouper
- Sortie : un ensemble *Env*
- Algorithme(s) :

**L'opérateur** Sort

- Collection d'entrées : un ensemble *Env*
- Paramètre(s) : la variable sur laquelle trier
- Sortie : un ensemble *Env*
- Algorithme(s) :

| **Opérateur** | **Type** | **Fonction** |
|---|---|---|
| Source | unaire | lecture de données XML |
| Return | unaire | construction du document XML de résultat |
| Select | unaire | restriction |
| Project | unaire | projection |
| Join | binaire | jointure de deux entrées |
| Union | binaire | union |
| Intersection | binaire | intersection |
| Difference | binaire | différence |
| GroupBy | unaire | groupement |
| Map | unaire | |
| Sort | unaire | ordonnancement |

Table B.1: Liste des opérateurs de requête dans MediaGRID

## Une approche pour la construction d'évaluateurs adaptables de requêtes

**Résumé** :

Cette thèse présente une approche pour la construction d'évaluateurs de requêtes adaptés aux besoins des applications et/ou à l'environnement d'exécution. L'approche proposée distingue trois types d'adaptation : statique (à la construction de l'évaluateur), personnalisée (avant l'exécution de requête ayant des contraintes spécifiques) et dynamique (pendant l'exécution de la requête). Nous avons présenté une analyse systématique des différentes dimensions d'un évaluateur.

La principale contribution de cette thèse est un canevas d'évaluation de requêtes nommé QBF (***Q**uery **B**roker **F**ramework*). QBF présente les fonctionnalités d'un évaluateur de requêtes de manière abstraite et séparée dans le but de favoriser leur réutilisation et leur adaptation. En s'appuyant sur QBF, nous avons proposé les mécanismes pour assurer les trois types d'adaptation, à savoir l'instanciation de QBF, l'évaluation personnalisée et l'évaluation interactive de requêtes.

Une implémentation de QBF ainsi que de quelques instances (i.e. évaluateurs) ont été réalisées. Elles nous ont permis de montrer la faisabilité de notre approche et le bénéfice de la séparation et de l'abstraction des fonctionnalités proposées dans QBF. Nous avons également mené une évaluation expérimentale des mécanismes implémentés (i.e. opérateurs, surveillances) afin de mesurer le surcoût de l'adaptation dans QBF et d'aider les programmeurs à choisir des éléments (ou composants) appropriés pour construire leurs évaluateurs.

**Mots-clés :**

Optimisation et exécution de requête, Evaluation personnalisée, Evaluation interactive, Canevas logiciel adaptable.

## An approach for building adaptable query evaluators

**Summary:**

This dissertation presents an approach for building query evaluators adapted to application requirements and the execution environment. The proposed approach considers three types of adaptation: static (at building time), personalized (when query execution starts) and dynamic (during query execution). We presented an analysis of different dimensions of a query evaluator.

The main contribution of the work is a query evaluation framework named QBF (**Q**uery **B**roker **F**ramework). QBF presents query functionalities in an abstracting and separating manner so as to promote their reusability and adaptability. Based on QBF, we proposed mechanisms for ensuring three types of adaptation: instantiation of QBF, personalized and interactive query evaluation.

In order to validate our approach, we implemented QBF and instantiated it for building some evaluators. Through this experience, we show the advantage of the separation of query processors functionalities proposed within QBF. We also conducted some experimental evaluations of implemented mechanisms (i.e. operators, monitoring) so as to measure adaptation overheads in QBF and to assist programmers to choose appropriated components for build their query processors.

**Keywords**:

Query Processing, Personalized Query Evaluation, Interactive Query Evaluation, Adaptable Framework.

Laboratoire LSR (Logiciels, Systèmes, Réseaux) – IMAG
BP 72 – 38402 Saint Martin d'Hères Cedex, France