# Le middleware

## INTRODUCTION

Le *middleware* est ce logiciel du milieu qui assure les dialogues entre clients et serveurs souvent hétérogènes. Aucune traduction française du mot *middleware* n'étant reconnue, nous proposons et utilisons **médiateur**, le terme ayant l'avantage de souligner le rôle d'intermédiaire et de négociateur généralement joué par le *middleware.*

Quels sont plus précisément les objectifs d'un médiateur ? C'est ce que nous précisons dans la section 2. Au-delà du transport des requêtes et des réponses, le médiateur assure le départ des interfaces des serveurs vers les clients. Son rôle d'intermédiaire le conduit à négocier les connexions et à convertir les types de données échangées. Fiabilité, sécurité et performance sont aussi des objectifs promis par les médiateurs.

Compte tenu de l'ampleur de tous ces objectifs, il existe différents types de médiateurs. En première approche, il est naturel de distinguer les médiateurs de manipulation de données, les médiateurs transactionnels, et les médiateurs d'invocation d'objets. Ce chapitre se consacre essentiellement aux médiateurs de manipulation de données. Les médiateurs transactionnels seront étudiés au chapitre 5. Les médiateurs d'invocation d'objets seront développés aux chapitres 9 et 10. Nous distinguons trois niveaux de médiateurs permettant la manipulation de données: transporteur de requêtes, accès multibases, et coordinateur de systèmes distribués. Le dernier niveau va bien au-delà de la manipulation de données, car il permet la coordination d'applications entre un client et de multiples serveurs.

Ce chapitre est surtout orienté vers la présentation de solutions technologiques, incluant les standards et les produits. En effet, les concepts fondamentaux des médiateurs, tels que le dialogue par messages ou par appel de procédures à distance, ont été introduits au chapitre précédent. Nous analysons dans les sections 3 et 4 du présent chapitre les propositions de standards en matière d'interface applicative SQL (CLI) et de protocole de transfert de requêtes SQL {RDA). Les Standards CLI de I'X/Open et RDA de l'IS0 sont particulièrement intéressants, car à la base des grands produits du marché. La section 5 est consacrée à l'étude des principaux produits de transport de requêtes: produits indépendants tels ODBC de Microsoft, SequeLink de Techgnosis, DAL de Apple, et produits propriétaires d'un constructeur dé SGBD. Les architectures multibases DRDA d'IBM, IDAPI de Borland, EDA/SQL de Information Builders permettent de traiter des requêtes portant sur des tables réparties dans plusieurs bases de données. Bien que limitées, elles définissent un cadre proche des solutions de type bases de données réparties que nous approfondirons au chapitre 7. Nous étudions ces approches dans la section 6.

Pour démontrer le besoin de coordination d'application par un couplage fort allant bien au-delà du transfert de requêtes bases de données, nous introduisons l'architecture DCE de l'OSF. Cette architecture de coordination généralisée, qui devrait être la base de systèmes distribués de nombreux constructeurs, est construite autour du RPC et permet en particulier une gestion de fichiers distribués sécurisée. Elle est étudiée dans la section 7.

Pour conclure ce chapitre, les architectures basées sur l'invocation d'objets distribués tels Corba de 1'0 MG et OLE de Microsoft sont introduites brièvement: elles seront étudiées en détails aux chapitres 9 et 10. L'évolution des médiateurs vers des logiciels de plus en plus complexes et intelligents, capables par exemple d'optimisation multibases, est enfin évoquée.

## OBJECTIFS ET TYPOLOGIE DES MEDIATEURS

### 2.1) Objectifs

Nous avons vu au chapitre 2 qu'un médiateur assurait le lien depuis un client vers un serveur, en visant la transparence aux réseaux, SGBD, et programmes. Les objectifs d'un médiateur sont plus précisément développés figure 1. Bien sûr, tous les médiateurs ne satisfont pas tous ces objectifs. D'un autre côté, ces objectifs sont ouverts, si bien que certains médiateurs peuvent réaliser des fonctions très complexes, comme par exemple les jointures multibases, ou l'assemblage d'objets à partir de tuples composants, ou le désassemblage.

 Transport de requêtes et réponses

 Simplification de la vision utilisateur

* développement d'API transparente
* intégration uniforme aux langages

 Harmonisation des types de données

 Performance

* gestion de caches clients et serveurs
* parcours des résultats complexes (ensembles, objets longs)

 Fiabilité

* gestion de transactions intégrée

Figure 20 - Objectifs d'un médiateur.

Le transport des requêtes et réponse depuis le client vers le serveur est la fonction de base à assurer. Pour cela, un médiateur s'appuiera souvent sur un protocole permettant de réaliser le *Remote Procedure Gall (RPC)* comme vu au chapitre 2. La simplification de la vision utilisateur conduit à développer des interfaces applicatives (API) transparente, c'est-à-dire à déporter sur le client les primitives d'accès que l'on trouve sur le serveur. Elle nécessite aussi de rendre invisibles les problèmes de conversion de types de données qui doivent être pris en charge par le médiateur, de sorte à offrir une intégration uniforme aux langages.

Les problèmes de performance sont aussi essentiels: un médiateur doit transmettre les requêtes aussi efficacement que possible. Pour éviter des transferts inutiles, certains médiateurs proposent la gestion de caches clients et serveurs, pour les résultats et parfois pour les requêtes. Les caches de résultats sont particulièrement nécessaires pour les objets complexes (ensembles, listes, longues chaînes de bits ou d'octets). Ceux-ci ne peuvent être transférés globalement; des techniques de **défilement** *(scrolling)* doivent être mises en place pour les transférer par parties successives. La gestion de caches de requêtes afin d'éviter des transferts inutiles nécessite en général la compilation de la requête afin d'être capable de comparer les requêtes. Ceci ne peut être réalisé que dans les systèmes qui intègrent sur le client un noyau de SGBD.

Enfin, les problèmes de fiabilité sont multiples. la fiabilité des communications doit être assurée par des techniques de ré-émission en cas de non réponse, mais aussi de détection de double émission à l'aide par exemple d'un numéro de séquence associé aux requêtes. La fiabilité des mises à jour nécessite l'intégration de techniques de gestion de transactions, avec validation *(commit)* en cas de succès, et reprise *(rollback)* en cas d'échec. Nous étudierons les protocoles et techniques de base dans le cadre des systèmes de données réparties, au chapitre 5.

### 2.2) Typologie des médiateurs

Au chapitre 2, nous avons déjà distingué les médiateurs selon qu'ils sont ou non attachés à un SGBD. Nous allons maintenant les distinguer selon les fonctionnalités offertes. Puis, nous étudierons les médiateurs de chacune des catégories en restant pour l'essentiel dans le monde du client-serveur de données.

#### 2.2.1 Les interfaces applicatives et les transpol1eurs de requêtes

Une interface applicative *(Application Programming Interface API)* est un ensemble de fonctions standard pour accéder à un service local ou distant. Leur standardisation a été particulièrement poussée par l'initiative CAE *(Common Application Environment)* de I'X/Open. L'accès à des bases de données relationnelles distantes est un exemple d'interface applicative normalisée sous le nom CLI par le SOL Access Group et I'X/Open.

Au-delà des interfaces applicatives, les transporteurs de requêtes mettent en oeuvre des protocoles de niveau application et session pour expédier des requêtes à des services distants et récupérer les réponses. Ils s'appuient sur des protocoles de transports standard. Ils développent très souvent des protocoles spécifiques qui peuvent être basés sur l'appel de procédures à distance.

#### 2.2.2 Les architectures multibases

Ces architectures intègrent à la fois des interfaces applicatives et des transporteurs de requêtes afin de rendre plus ou moins transparente l'hétérogénéité des systèmes voire la distribution des données. Elles offrent un même langage (en général SQL) pour accéder à des bases de données distantes et hétérogènes. Elles cherchent aussi à résoudre les problèmes de gestion de transactions multisites.

#### 2.2.3 Les architectures pour applications distribuées

Au-delà des bases de données, les architectures pour applications distribuées cherchent à définir un cadre général pour exécuter des services distribués. Elles s'intéressent bien sûr à définir une gestion de données uniformes (qui peut être basée sur des fichiers distribués) mais aussi à intégrer des services de dénomination globaux, de synchronisation entre processus distants, de gestion de sécurité et de transactions distribuées. Elles sont au cœur des systèmes d'exploitation de demain.

## Interfaces Applicatives : Le Standard CLI

### 3.1) Place dans les systèmes ouverts

Un système ouvert est un système dont les interfaces obéissent à des standards internationaux établis au sein de structures accessibles à tous [Cray93]. De nombreux groupes proposent des standards, dont l'ISO qui est l'Institut International de Standardisation, L'ANSI qui est sa branche Nord Américaine, le CCITT pour les télécommunications, l'IEEE pour les systèmes d'exploitation en particulier, etc. Le mouvement des systèmes ouverts a été bien développé autour de Unix. Plus particulièrement, l'IEEE a proposé le standard POSIX d'interface de système d'exploitation à partir de Unix. Unix System V respecte les standards ANSI Cet POSIX.

Une organisation à but non lucratif appelée I'X/Open a vu le jour en 1984 à l'initiative des constructeurs et d'utilisateurs. Cette organisation coordonne le développement de standards d'interfaces applicatives dans le cadre de l'initiative *Common Application Environment* (CAE). Le but de CAE est d'assurer la portabilité des logiciels applicatifs en

atteignant trois objectifs :

* interface utilisateur commune ;
* interface de programmation commune ;
* modèle commun d'interconnexion.

Des standards ont été définis dans le cadre du CAE pour les interfaces avec le système d'exploitation (POSIX), les langages de programmation (C, Fortran, Cobol, ADA), la gestion de données (SOL), la gestion des réseaux (Interface transport), la gestion du fenêtrages (X Windows). Ces standards sont certifiés par I'X/Open qui délivre le label de conformité «XPG».

Dans le domaine des médiateurs, I'X/Open a repris à son compte les travaux d'un groupe de constructeurs indépendants, le SQL Access Group (SAG). En conjonction avec le SAG, I'X/Open a publié un rapport intermédiaire *(draft ou snapshot)* proposant une interface applicative standard pour un médiateur de données [X/Open92]. Cette API est dénommée CLI *(Client Interface).* Elle est basée sur le langage SQL et permet une standardisation des appels depuis un client vers un serveur SQL depuis un langage quelconque, par exemple C ou Cobol. Le langage SQL considéré est en principe celui spécifié par I'X/Open *(X/Open* SQL), qui est conforme à celui de l'ANSI.

CLI est une API pour serveur SQL. A ce titre, elle ne s'oppose pas aux standards d'intégration de SQL dans les langages de programmation *(embedded SOL)* basés sur des curseurs pour parcourir les tables résultats avec l'opération FETCH. CLI est une interface de plus bas niveau, qui peut être le langage cible d'un pré-processeur de langage de programmation intégré à SQL. Il s'agit donc d'un complément indispensable au standards de l'ISO, qui vise une certaine compatibilité au niveau du binaire pour l'application, notamment dans le domaine du client-serveur. CLI doit donc permettre de développer des applications portables, à condition d'utiliser un SQL portable.

### 3.2) Types de données, contextes et connexions

CLI gère des types de données de base: chaînes de caractère de taille fixe ou variable (CHAR(L)), entier court (SMALLINT), entier long (INTERGER) et pointeur (POINTEUR). Les paramètres peuvent être passés par valeurs ou par pointeurs pour les langages supportant les pointeurs (C par exemple).

CLI gère des structures de données afin de conserver les variables de contexte d'un environnement base de données, d'une connexion permettant l'accès à un serveur, et d'un ordre SOL. Une application peut avoir plusieurs connexions actives à la fois. Les contextes sont spécifiques à chaque implémentation et référencés par des **poignées** *(handle).* Ils doivent être alloués par des fonctions spéciales commençant par *Alloc* avant utilisation, et désalloués par des fonctions symétriques commençant par *Free.*

La procédure de connexion à un serveur s'effectue par appel de la fonction *Connect().* La procédure de déconnexion s'effectue par la commande *Disconnect().* Les paramètres essentiels de *Connect* sont une poignée désignant le contexte de connexion (hdbc), un nom de server (SeNer), un nom d'utilisateur (UID), une chaîne d'authentification (AuthStr), si bien que la signature de la fonction est la suivante :

**CONNECT** (hdbc, Servcr, urD, AuthStr) **RETURNS** (SMAI"LIN'I') .

La fonction (comme toute fonction CLI) retourne un code réponse qui est un entier court. Les codes réponses sont spécifiés dans la proposition de norme. La commande de déconnexion est plus simple; il suffit de donner en paramètre la poignée désignant le contexte de connexion; d'où la signature :

**DISCONNECT** (hdbc) **RETURNS** (SMAIJLINrr) .

En résumé, la figure 2 montre les fonctions de base à exécuter pour réaliser la connexion puis la déconnexion à un serveur. L'envoi de requêtes et la réception des résultats sont détaillés ci-dessous. Plusieurs connexions peuvent être associées à un même environnement base de données.

AllocEnv()

AllocConnect()

Connect()

AllocStmt()

*Envoi de requêtes*

*Réception résultats*

FreeStmt()

Disconnect()

FreeConnect()

FreeEnv()

Figure 21 - Enchaînement des appels pour connexion et déconnexion.

### 3.3) Envoi des requêtes

Il existe deux manières de spécifier et exécuter un ordre SQL.

La première, plutôt adaptée au SQL dynamique, consiste à demander l'exécution immédiate d'un ordre SQL au serveur, bien sûr après établissement d'une connexion comme vu ci-dessus. Après établissement d'un contexte d'ordre référencé par une poignée hstmt, l'application peut appeler la fonction de signature :

**EXECDlRECT** (hstmt, SqlStr) **RETURNS** (SMAl,LINT) .

Celle-ci permet simplement d'envoyer la chaîne de caractère SqlStr, qui en principe contient un ordre SOL, au serveur.

La seconde permet de demander au serveur la préparation d'un ordre SOL avec d'éventuels paramètres; cet ordre peut être exécuté plusieurs fois avec des paramètres différents. Avant chaque exécution, les paramètres doivent être instanciés. L'instanciation de paramètres s'effectue soit par spécification de valeurs (fonction Se *tParamValue),* soit par liaison avec une variable en mémoire (fonction *BindParam()).* Au-delà du chargement des paramètres, l'application devra donc appeler les deux fonctions :

**PREPARE** (hstmt, SqlStr) **RETURNS** (SMALLINT)

**EXECUTE** (hstmt) **RETURNS** ( SMALLINT) .

Après préparation d'une requête ou exécution directe, l'application peut appeler l'interface pour analyser la structure du résultat, en particulier connaître le nombre de colonnes et obtenir un descripteur (nom, type, etc.) des colonnes ou d'une colonne d'un tuple résultats. CLI offre ainsi des fonctions *NumResultCols(), DescribeCol(), et ColAttribute()* pour que l'application puisse se documenter sur la structure des tuples résultats. La norme ne précise pas si ces fonctions doivent être réalisées par le client (ce qui impliquerait une analyse des requêtes à ce niveau) ou le serveur.

### 3.4) Transfert des résultats

Les résultats d'un ordre SELECT sont obtenus par un mécanisme de curseur associé à un ordre SQL exécuté. Un nom peut être donné au curseur, mais ceci est sans importance pour la fonction qui ne précise qu'un contexte d'ordre SQL précédemment exécuté dénoté hstmt, comme suit :

**FETCH** ( hstmt ) **RETURNS** ( SMALLINT ) .

L'ordre FETCH permet d'obtenir le tuple suivant du résultat. Auparavant, certaines colonnes peuvent être liées à des variables de l'application par la fonction *BindCol()* : elles sont alors chargées dans les variables par l'ordre FELTCH(). Les autres peuvent être accédées par la fonction *GetCol().*

Dans le cas de commandes de mise à jour (INSERT, DELETE, UPDATE), il n'y a pas de résultats transférés à l'application. Cependant, celle-ci peut demander le nombre de lignes affectées par la commande à l'aide de la fonction *RowCount().* Une gestion de transaction est aussi prévue dans le cas de mise à jour: l'application peut demander à valider ou défaire la tran­saction en cours par la fonction Transact().

### 3.5) Bilan et conclusion

En résumé, l'émission d'une requête et l'exploitation d'un résultat s'effectuera selon le schéma représenté figure 22.

Exécution multiple ?

Non

Oui

BindParam()

ou SetParamValue()

ExecDirect()

Prepare()

BindParam()

ou SetParamValue()

Execute()

DELETE

INSERT

UPDATE

SELECT

Type d'ordre ?

NumResultCols()

DescribeCol()

BindCol()

Fetch()

RowCount()

Transact()

Oui

D'autres lignes ?

Figure 22 - émission de requêtes et exploitation des résultats.

Les fonctions de base de CLI sont résumées figure 23. Des fonctions de traitement d'erreurs (*Error()* et *Cancel())* sont aussi définies. La proposition de I'X/Open précise également l'intégraton des fonctions en C et en Cobol. Les conversions de types de données sont particulièrement spécifiées.

* Connect() : Ouverture d'une connexion à un serveur
* Prepare() : Préparation d'un ordre pour exécution ultérieure
* Execute() : Exécution d'un ordre préparé
* ExecDirect() : Exécution directe d'un ordre
* SetCursorName() : Positionnement d'un nom de curseur
* DescribeAtt() : Obtention du descripteur d'une colonne résultat
* Fetch() : Lecture du tuple suivant du résultat
* RowCount() : Nombre de lignes affectées par un ordre SOL
* Transact() : Validation ou annulation d'une transaction
* Cancel() : Annulation d'un ordre SOL en cours d'exécution
* Error() : Retour des informations d'erreurs des contextes

Figure 23 -Principale fonctions de CLI.

En conclusion, CLI est donc un standard fondamental du client-serveur, dont une implémentation très populaire a été proposée par Microsoft sous le nom ODBC *(Open DataBase Connection).* CLI a cependant des faiblesses (dont certaines sont levées par ODBC), notamment une forte limitation des types de données reconnus, l'absence de support pour les objets longs, le non support de dialogues asynchrones permettant à l'appplication de ne pas être bloquée lors de l'envoi d'une requête, et un support de transactions très limité.

## Les Protocoles d'échange : le standard RDA

RDA *(Remote Data Access)* est un protocole de communication pour accéder depuis un client à des bases de données distantes. C'est un standard ISO depuis 1993, aussi accepté par l'IEC *(International Electronical Commission)* et l'ANSI. Il est divisé en deux parties : une partie générique [ANSI93a] et une partie spécifique au langage SQL [ANSI93b]. La première partie spécifie un cadre général pour le modèle client-serveur, les services nécessaires, et le protocole pour se connecter à une base quelconque. La deuxième partie définit précisément les requêtes possibles pour une base de données SQL. Une troisième partie a été proposée dans le cadre de bases réseaux à partir d'un langage de navigation appelé NDL (Network Data Language), mais elle n'a guère connu de développement.

RDA est un protocole d'application construit au-dessus des couches présentation et ses­sion de l'architecture OSI de l'ISO. Le protocole générique permet à un client de se connecter à un serveur, puis d'ouvrir une ressource de données. Plusieurs ressources peuvent être imbriquées. Une gestion de transactions atomiques est supportée. RDA s'appuie sur le mode session conversationnelle TP de l'OSI et sur la forme canonique ASN.1 de représentation des données de l'ISO. Il suppose un service d'association programme à programme de type OSI ACSE *(Association Control Service Element)* basé sur trois commandes de demande d'association *(Associa te),* de relâchement d'association *(ARelease)* et d'abandon d'association *(Abort).* Plus précisément, le standard RDA fournit un service d'interface a un élément de communication qui convertit les requêtes en appels aux services ACSE et TP .

Le service d'interface se compose d'éléments pour contrôler les associations, pour transférer les requêtes et les paramètres du client au serveur, pour transférer les résultats en sens inverse, et pour gérer les transactions. Le contrôle des associations gère l'établissement de connexions entre client et serveur et la connexion aux bases du serveur. Les requêtes bases de données sont envoyées comme des chaînes de caractères avec paramètres. Les paramètres, données résultats et exceptions sont décrits et représentés à l'aide du standard de codage ASN.1. La gestion de transactions permet une validation en une ou deux étapes. Dans ce dernier cas, le protocole suppose l'implémentation sous-jacente de la norme ISO TP de gestion de transaction (voir chapitre V).

Cinq types de services RDA sont offerts aux clients :

* Les services de gestion de dialogues *(Dialogue Management Services),* pour débuter et terminer les dialogues ;
* Les services de gestion de transaction *(Transaction Management Services),* pour débuter et terminer les transactions ;
* Les services de contrôle *(Control Services)* pour rapporter sur l'état ou arrêter des opérations en cours ;
* Les services de gestion de ressources *(Resource Handling Services)* pour permettre ou interdire l'accès aux ressources ;
* Les services du langage bases de données *(Database Language Services)* pour accéder et modifier les ressources.

La spécialisation SQL de RDA définit l'ensemble des messages nécessaires au transport de requêtes SQL. La figure 5 résume l'ensemble des types de messages du standard. Pour chaque type, il existe en général une version requête, une version réponse avec succès et une version réponse avec problèmes, ce qui augmente le nombre de messages. Une version TCP/IP de RDA est proposée par I'X/Open. Les messages sont en principe ceux permettant le transport des requêtes générées par l'interface CLI étudiée ci-dessus et des réponses associées.

* Initialize : Initialisation d'une requête
* Terminate : Terminaison d'une requête
* BeginTransaction : Début de transaction
* Commit: Validation de transaction,
* Rollback : Reprise e transaction
* Cancel : Annulation de requête
* Status : Demande d'état d'une requête
* Open : Ouverture d'une ressource
* Close : Fermeture d'une ressource
* ExecuteDBL : Exécution d'une requête BD
* DefineDBL : Préparation d'une requête BD
* InvokeDBL : Invocation d'une requête préparée
* DropDBL : Suppression d'une requête BD

Figure 24 : Principaux types de messages normalisés par RDA.

## Interfaces et Protocoles: les PRODUITS

Dans cette section, nous étudions les produits qui permettent d'accéder à des bases de données distantes en utilisant SQL. Ces produits fournissent une interface voisine de CLI, mais supportent des protocoles de communication variés.

### 5.1) ODBC de Microsoft

ODBC est une interface de programmation définie au départ pour le langage C permettant l'interopérabilité avec les serveurs de base de données. Il existe maintenant deux versions de ODBC. Dans cette section, nous introduisons la nouvelle version ODBC 2.0, conformé­ment à la spécification de Microsoft [Microsoft94]. ODBC définit une librairie de fonctions proches de CLI qui constitue l'interface pour se connecter à un SGBD, exécuter des ordres SOL, et retrouver des résultats. Le SOL utilisé est celui du SOL Access Group retenu pour l'environnement commun des applications (CAE). En particulier, ODBC standardise les types de données échangés et les codes erreurs.

Les composants de l'architecture ODBC sont représentés figure 25. Au-delà de l'application portable (en principe sans recompilation), il s'agit du gestionnaire de pilotes, du pilote, et d'une source de données. Le gestionnaire de pilotes est une librairie dynamique (DLL) chargée de superviser les pilotes. A partir d'un nom de source de données, le pilote est retrouvé et chargé si nécessaire. Les appels de l'application sont dirigés vers le bon pilote et les paramètres validés.

**CLIENT**

Application

**SGBD**

Gestionnaire

de pilotes

Adaptateur

Pilote Traitant

Pilote Transparent

FAP

FAP

Fichiers Dbase

Réseau

Figure 6 - Architecture de ODBC.

Le pilote est aussi une DLL qui dépend de la source de données. Il implémente les fonctions ODBC et interagit avec la source de données. Au-delà de l'établissement des connections, de la soumission des requêtes, et de la gestion des résultats, le pilote effectue les conversions de format nécessaires et standardise les codes erreurs.

Une source de données est composée d'un SGBD et, dans le cas où le SGBD est distant, des protocoles permettant d'interagir avec lui. Ces protocoles de type FAP *(Format and Access Protocols)* sont a priori quelconques: c'est au pilote de réaliser les fonctions qu'ils ne font pas. L'approche ODBC est donc de s'intégrer au-dessus des protocoles d'accès au SGBD, ce qui permet une plus grande ouverture et portabilité. On distingue les **pilotes trai­tant** *(single-tier)* et les pilotes transparents *(multiple-tier)* : les premiers traitent les ordres ODBC et SOL en complétant les fonctions de la source, alors que les seconds ne font que transmettre les ordres au serveur. Un pilote de fichiers sera en général un pilote traitant afin d'assurer le niveau minimal de support de SOL.

ODBC propose différents niveaux de conformité, aussi bien pour l'interface applicative (API) que pour SOL. Concernant l'interface applicative, trois niveaux de conformité ODBC sont définis.

1. **Le niveau noyau *(Core API).*** Il correspond à l'interface CLI de I'X/Open. Il comporte les fonctions d'allocation et libération de poignées d'environnements, de connexions, et d'ordres, de connexion à une source de données, de préparation d'ordre SOL, d'instan­ciation de paramètres, d'exécution d'ordres SOL préparés ou directement, de récupéra­tion de résultats, de validation et reprise de transactions, et de gestion d'erreurs.

2. **Le niveau étendu 1** ***(Level 1 API).*** Par rapport au niveau précédent, il ajoute le support d'objet long (BLOB) en tant que type de colonne ou de paramètre, la gestion d'informa­tions de la métabase (schéma des tables et colonnes, statistiques), et des fonctions per­mettant de documenter le pilote et la source de données (types de données, fonctions scalaires et fonctions ODBC supportés).

3. **Le niveau étendu 3** ***(Level 3 API).*** Il permet des fonctionnalités plus avancées tels que le support des tableaux de paramètres et de résultats pour demander l'exécution de lots de requêtes préparées, l'utilisation de curseurs déplaçables en avant et en arrière *(scrollable),* la récupération de plus d'informations de la métabase (droits, clés et procédures) et une meilleure documentation sur les pilotes et sources de données disponibles.

Indépendamment du niveau de conformité de l'API, ODBC propose trois niveaux de conformité SQL. Chaque niveau correspond à une grammaire définie comme suit.

1. **Le niveau grammaire minimum** *(Minimum SQL Grammar).* Seuls les ordres SQL de base CREATE TABLE, DROP TABLE, SELECT, INSERT, UPDATE et DELETE avec des expressions de calculs simples sans agrégat sont supportés. Les types de données connus sont réduits aux caractères (CHAR, VARCHAR, LONG VARCHAR).

2. **Le niveau grammaire noyau** *(Core SOL Grammar).* Les ordres de définition et modification de schéma CREATE VIEW, DROP VIEW, CREATE INDEX, DROP INDEX, ALTER TABLE, d'attribution et retrait de droits GRANT et REVOKE sont supportés. L'ordre SELECT est complètement supporté, avec les agrégats. Les types de données DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT et DOUBLE PRECISION sont gérés en plus des caractères du niveau minimum.

3. **Le niveau grammaire étendue** *(Extended SQL Grammar).* Les unions et jointures externes, les appels de procédures stockées, les lots de requêtes SQL, et les ordres de mise à jour positionnés *via* curseurs sont acceptés. Les types de données sont étendus, notamment pour les objets longs et le temps; BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINAR)/, TIME, DATE, TIMESTAMP sont notamment supportés.

Du point de vue gestion de transactions, ODBC maintient une transaction pour chaque connexion active. Une transaction est implicitement ouverte en début de connexion. Une option permet de valider chaque ordre SOL après exécution; sinon, les transactions doivent être explicitement terminées (validées ou annulées).

Les pilotes ODBC sont de plus en plus performants. En particulier, la gestion des résultats ne nécessite pas un échange par tuple. Elle s'effectue en général avec des curseurs qui peuvent être bloqués; dans le cas bloqué, l'utilisateur peut demander qu'à chaque ordre FETCH, plusieurs lignes soient transférées. Les curseurs peuvent être statiques ou dynamiques : un curseur statique ne permet pas de voir les modifications apportées par les autres transactions, alors qu'un curseur dynamique le permet. Les résultats associés aux curseurs peuvent aussi être gérés comme des ensembles de clés; l'accès à la ligne est alors effectué lors du FETCH. Les attributs longs peuvent être échangés par morceaux. Tout ceci permet de nombreuses combinaisons et de multiples optimisation pour les résultats volumineux.

En résumé, ODBC présente de nombreux avantages: c'est une solution sur le site client qui respecte le standard CLI ; ODBC est intégré à Windows et à tous les produits Microsoft (Access, Visual C++, Visual Basic) ainsi qu'à la plupart des produits des autres constructeurs ; ODBC assure un niveau standard de SOL que l'on peut choisir; ODBC en version étendue supporte efficacement les curseurs bloqués, les objets longs, et les procédures stockées. Cependant, ODBC s'empile souvent au-dessus des médiateurs des constructeurs de SGBD, ce qui dégrade les performances (on perdrait typiquement 20 %). De plus, bien que possible, le dialogue asynchrone entre client et serveur reste difficile.

### 5.2) DAL/DAM de Apple

DAL *(Data Access Language)* et son gestionnaire DAM *(Data Access Manager)* est un médiateur intégré au Système 7 du MacIntosh [Meadow91 ]. DAL est un langage complet intégrant un langage de définition de données et un langage de manipulation de type SOL, mais aussi un langage de contrôle permettant la programmation avec des ordres DECLARE, GOTO, SET, IF, WHILE, FOR EACH, etc., et un langage d'édition avec des ordres PRINT, PRINTROW, PRINTINFO, etc. DAL permet de composer et appeler des procédures et des fonctions. Il intègre aussi des commandes de gestion de transactions COMMIT et ROLLBACK. La solution proposée par Apple est donc plutôt en dehors des normes des systèmes ouverts: elle ne respecte pas l'interface CLI et DAL est assez éloigné de SOL.

L'architecture supportant DAL est représentée figure 26. Côté serveur, elle nécessite un serveur du langage, et un adaptateur au SGBD pour traduire les requêtes DAL en requêtes SOL. Il faut aussi LIn adaptateur au réseau. Côté client, l'application s'adresse au DAM. Les requêtes peuvent être construites en mémoire ou passées à partir de textes contenus dans des documents. Ainsi, le DAM fournit deux types d'interface, une de haut niveau permettant d'envoyer des enregistrements requêtes depuis des documents au serveur, une de bas niveau permettant d'envoyer des requêtes depuis la mémoire. Dans les deux cas, il faut lancer une session avec le serveur par appel de la fonction DBlnit, et la fermer par DBEnd.

L'envoi d'une requête s'effectue par DBSend, l'exécution par DBExec, et la lecture du code réponse par DBState. DBGetltem permet de lire les résultats et leurs types. Les commandes du langage DAL qui ne nécessitent pas d'appels au client sont traitées par l'extension DAL. L'architecture est en principe ouverte, et l'utilisateur peut développer ses propres extensions.

En résumé, l'approche DAL/DAM semble attrayante, un point fort étant que DAL est un langage complet pour accéder aux serveurs. Malheureusement, cette approche n'obéit pas au standard. Elle nécessite de plus un serveur DAL au-dessus du SGBD. Elle est aussi redondante par rapport aux serveurs SQL qui permettent le support de procédures stockées. Ces faiblesses expliquent sans doute le faible succès du médiateur général DAL/DAM proposé par Apple.

### 5.4) Les solutions propriétaires

En général, les constructeurs de SGBD fournissent deux types de médiateurs: l'un pour permettre le dialogue depuis un outil client avec leur serveur, l'autre pour assurer l'intégration de bases de données hétérogènes par des protocoles de serveur à serveur. Les interfaces applicatives (API) sont généralement ouvertes, alors que les protocoles sont spécifiques. RDA est considéré pour le futur. Au-delà des médiateurs, des solutions bases de données réparties intégrées sont possibles, comme nous le verrons au chapitre 7.

#### 5.4.1 Le cas d'Oracle

Oracle propose plusieurs niveaux d'interface client au dessus d'un protocole client-serveur dénommé SQL\*Net [Holt94]. Ce protocole est invoqué directement par l'interface OCI *(Oracle Common Interface).* OCI offre une structure de données fondamentales (le curseur) pour appeler Oracle à partir d'une quinzaine de fonctions proches de celles de CLI de I'X/Open, telles que OSOL, OPARSE , OEXEC, OEXN, OBIND, OFECTH, OFEN, OCLOSE, etc. Ces fonctions permettent respectivement de définir une requête SOL, de lancer son analyse syntaxique, de faire exécuter une ou plusieurs requêtes, de lier des paramètres à une requêtes, de lire un ou plusieurs tuples résultats, d'ouvrir ou de fermer une base. Depuis la version 2 de SOL \*Net, l'interface OCI permet depuis tout poste client disposant de SOL \*Net d'appeler tout serveur Oracle par un protocole de communication quelconque. Les précompilateurs des langages avec SOL intégré *(Embedded SOL)* de type PRO\*C, PRO\*Cobol, etc., génèrent d'ailleurs des appels à OCI.

Une API type ODBC a été développée au-dessus de OCI en conjonction avec Microsoft et est disponible. IDAPI est aussi supportée par Oracle. Oracle Glue est une interface de niveau supérieur qui permet d'envoyer des ordres SQL dynamiques depuis des documents.

L'effort d'Oracle porte aussi sur les protocoles et interfaces serveur-serveur. Des passerelles vers des serveurs non Oracle sont ainsi disponibles. On distingue les passerelles transparentes *(Transparentes Gateways),* basées sur la transmission d'ordre SOL et l'adaptation au SGBD distant, et les passerelles procédurales *(Procedural Gateways)* permettant l'appel de procédures à distance (RPC) depuis des procédures stockées. Des passerelles transparentes sont notamment disponibles pour IMS, DB2, SOL/DS, RMS, RDB, TurboImage, et les fichiers V SAM. Une boite à outil de développement est proposée pour construire des passerelles transparentes. Les passerelles procédurales permettent de réaliser le RPC de programmes externes depuis le langage de développement de procédures d'Oracle appelé PL/SOL. Les types de données PL/SOL sont passés et convertis vers le langage cible, les paramètres peuvent être filtrés, les transactions sont coordonnées. Une passerelle procédurale est disponible pour APPC, ce qui permet d'invoquer des transactions CICS ou IMS/TM. Une boîte à outil de développement est aussi proposée pour construire des passerelles procédurales.

En résumé, Oracle offre des médiateurs de plus en plus riches, qui permettent d'intégrer des données en provenance de sources quelconques dans les applicatifs développés avec les outils Oracle.

#### 5.4.2 Le cas de Sybase

Sybase offre une interface client constituée par une bibliothèque de fonctions appelée DBLIBRARY (en abrégé, DB-LIB). Il s'agit d'un ensemble de fonctions et macros pour initialiser et terminer une connexion, envoyer un programme SOL étendu, demander son exécution, traiter les résultats et les erreurs. Les programmes SOL étendu sont des commandes TRAN-SACT-SQL, le langage d'écriture de procédures de Sybase. En plus du SOL de base, TRAN-SACT-SQL ajoute un langage de contrôle de flux, avec variables (DECLARE), tests (IF... ELSE...), boucles (WHILE), procédures, traitement d'erreurs, etc. TRANSACT-SQL supporte aussi des types de données étendus (monnaie, texte, image, types utilisateurs). La DB-LIB supporte en plus la validation en deux étapes et un mode balayage *(browse)* qui permet d'effectuer des mises à jour de résultats de requêtes en multi-usagers. En résumé, TRANSACT-SQL et la DB-LIB fournissent un canal d'interconnexion très riche entre le client et le serveur, bien qu'en dehors des standards actuels. Pour les applications voulant se conformer au standard, il existe bien sûr un pilote ODBC pour Sybase.

Sybase est un des premiers systèmes qui a ouvert son serveur en rendant accessible l'interface d'appel côté serveur. Cette interface appelée ODS (Open Data Services) est symétrique à la DB-LIB côté serveur. Elle permet de récupérer les ordres de connexion, de recherche, d'exécution, d'exécution de procédures, de déconnexion, etc. L'intérêt essentiel est de réaliser des passerelles avec d'autres serveurs ou plus généralement générateurs de données (des capteurs par exemple) qui se comportent alors exactement comme des serveurs Sybase.

En utilisant la technologie ODS, Sybase a développé (ou repris) des passerelles vers de nombreux autres SGBD. En particulier, la passerelle MIDI permet de récupérer des données depuis DB2. Il est aussi possible de récupérer des données issues de transactions CICS ou IMS/TM.

En résumé, comme Oracle, Sybase offre des médiateurs de plus en plus riches, qui permettent d'intégrer des données en provenance de sources quelconques. La DB-LIB permet de récupérer toutes ces données et de les traiter dans tout outil interfacé.

## Architectures Distribuées : DCE de l'OSF

DCE est un standard d'architecture distribuée proposée autour de POSIX afin de faciliter le développement de grandes applications client-serveur autour de réseaux locaux, nationaux, et internationaux [Rosenberry92]. DCE utilise le RPC pour faire communiquer client et serveur. Au-delà d'un mode de dialogue client-serveur, DCE propose des standards de services pour la mise en place d'application C/S .

* Service de sécurité (login et authentification);
* Service de répertoires des ressources (annuaire);
* Gestion d'un temps global synchronisé global;
* Gestion de fichiers distribués.

Au-delà, DCE propose un standard pour la gestion et l'appel de serveurs multifilières *(multi-threads).*

### 7.1) La cellule d'administration

La cellule est l'unité d'administration et d'opération pour un groupe d'utilisateurs dans un réseau DCE. Une cellule comporte les services minimums de gestion derépertoires, de sécurité, de gestion du temps synchronisé. Pour répertorier les applications disponibles sur les serveurs, DCE s'appuie sur le standard X500 de l'ISO

Une cellule possède un nom global de structure /.../C=<initiales pays)/O=<organisation>/ OU=<unité>, par exemple /../FR/UVSQ/PRISM pour le laboratoire PRISM de l'Université de Versailles UVSQ. A l'intérieur de la cellule, tout élément DCE (serveur, fichier, etc.) a un nom local désigné par une chaîne de structure /.:/X1/X2/…/Xi. Les noms sont répertoriés dans les répertoires globaux (noms de cellules) ou locaux

La figure 27 illustre l'organisation d'une cellule composée de trois calculateurs Une application serveur doit tout d'abord s'enregistrer auprès du service annuaire (gestion de répertoire). Elle peut ensuite être utilisée par un client qui doit tout d'abord s'authentifier, puis obtenir l'adresse du serveur. Enfin, il peut appeler les fonctions du serveur par des RPC.



Figure 27 - Organisation d'une cellule DCE.

### 6.2) Le RPC DCE

Le RPC est donc un élément essentiel définit par DCE. Il repose sur le langage de définition d'interface IDL DCE, qui est une extension des déclarations de variables et fonctions en C. Une définition IDL est similaire à une définition de fonctions C avec quelques attributs supplémentaires tels *in/out* pour préciser le sens de passage des paramètres. A partir d'une définition d'interface exprimée en IDL, un compilateur génère deux souches *(stub)* et un fichier d'en-tête comme schématisé figure 28. Compilée avec le client, la souche client récupère les appels et les transmet au serveur. La souche serveur compilée avec le serveur récupère les appels et les transmet aux implémentations des fonctions. L'en-tête permet d'uniformiser les déclarations de types pour le client et le serveur et est avantageusement utilisée des deux côtés.

Fichier de définition d'interface

Souche client

Fichier source

En-tête pour code client et serveur

Souche serveur

Fichier source

Compilateur

IDL

Figure 28 – Génération des souches client et serveur.

Le fonctionnement du RPC DCE n'est pas différent de celui définit en général au chapitre 2. Il est rappelé figure 29.



Figure 29 - Fonctionnement du RPC DCE.

### 6.3) Le système de fichiers distribués (DFS)

En plus des fonctionnalités décrites ci-dessus, DCE spécifie un système de fichiers distribués (DFS) gérant des ensembles de fichiers connus globalement dans le réseau à l'aide du système de nomination et de répertoires présenté ci-dessus. DFS retrouve les ensembles de fichiers à partir du nom et rend possible le déplacement transparent pour l'utilisateur des fichiers d'un nœud à un autre. Il gère des sauvegardes automatiques, protège les fichiers par un système étendu de permissions *(read, write, execute, control, insert, delete),* et optimise les accès client-serveur par des caches sur les clients. DFS offre les commandes standard POSIX sur les fichiers distribués, telles *cd, mkdir, Is, fopen, fclose,* etc.

### 6.4) Bilan de DCE

DCE est une architecture intéressante de plus en plus implémentée, qui facilite la gestion de grands outils et intègre des facilités de communication client-serveur généralisées. Cette architecture est une bonne base pour faciliter la mise en place de grandes bases de données partagées par de nombreux systèmes. Pour cela, DCE doit être complémentée avec des protocoles et interfaces de manipulation de données type CLI et RDA.

## Les Architectures d'invocation d'objets

Dans cette section, nous introduisons pour mémoire les médiateurs basés sur l'échange de requêtes entre objets. Nous étudierons en détail ces approches dans des chapitres ultérieurs. Elles ont pour objectifs de permettre l'assemblage d'objets hétérogènes sur des réseaux. Dans de tels contextes, les objets sont écrits dans des langages différents et fonctionnent sur des plates-formes hétérogènes. Ils doivent pouvoir dialoguer les uns avec les autres, jouant alternativement le rôle de client ou de serveur.

### 7.1) CORBA de l'OMG

L'OMG propose une architecture distribuée basée sur **un échangeur de requêtes entre objets** *(Object Request Broker ORB)* et des services standardisés, comme illustré figure 30. Parmi les services sont distingués les **services objets** et les **utilitaires communs**. Les premiers fournissent les fonctions de base pour utiliser et implémenter des objets utilisateurs au niveau des clients et des serveurs. Ce sont par exemple la gestion d'événements, de noms, de transactions. Les seconds sont des assemblages de services directement utilisés par l'utilisateur final, tel le courrier électronique ou la gestion de documents composites.



Figure 30 - Vue d'ensemble de l'architecture de gestion d'objets de 1'OMG.

### 7.2) OLE 2 de Microsoft

Au-delà d'un médiateur, OLE 2 est une architecture distribuée d'intégration d'objets pour constituer des documents composites, c'est-à-dire des documents composés de plusieurs objets hétérogènes liés ou imbriqués. OLE 2 se pose en concurrent de Corba. En effet, cette architecture est bâtie sur un modèle objet limité où l'héritage est remplacé par la délégation, mais où les objets ont plusieurs interfaces, ceci afin d'assurer la compatibilité au niveau des binaires et de garantir les possibilités d'évolution. La distribution est basée sur une généralisation de l'appel de procédure à distance, comme nous le verrons au chapitre consacré à OLE 2 et à la gestion de documents composites.

## CONCLUSION: vers des médiateurs intelligents

Dans ce chapitre, nous avons tout d'abord approfondi les objectifs des médiateurs: simplification d'interface, négociation de connexions, transport de requêtes et réponses, homogénéisation des types de données, performance, fiabilité. Les médiateurs de manipulation de données ont plus particulièrement été étudiés, en particulier les standards d'interface (API CLI) et de transport (RDA). Différents transporteurs de requêtes tels ODBC, SequeLink, DAL et quelques architectures multibases telles DRDA, IDAPI et EDA/SOL ont été analysés. Afin de montrer les possibilités d'intégration des médiateurs de manipulation de données dans des architectures plus générales, nous avons présenté DCE de l'OSF. DCE offre des outils de base pour la coordination d'applications distribuées tels le RPC, les services d'annuaires, de temps distribué et de sécurité.

Intégrer à des architectures types DCE la gestion de transactions distribuées peut être une voie d'évolution vers des médiateurs distribués génériques. Une autre voie d'évolution des médiateurs est l'orienté objet. Standardiser l'envoi de requêtes à des objets implémentés dans un langage quelconque est l'un des objectifs de 1'OMG. Dans une certaine mesure, OLE définit aussi un standard d'invocation d'objets. Les services manipulation de données et gestion de transactions sont en bonne voie d'intégration dans ces architectures.

Finalement issus d'une simplification des services offerts par la couche communication des système de bases de données réparties, les médiateurs pourraient dans le futur récupérer une part des fonctions qu'ils ont perdues. Il apparaît déjà sur le marché des produits capables d'effectuer des jointures entre tables provenant de bases de données hétérogènes, telle *DataJoiner* d'IBM. La capacité à transporter des objets structurés, et non plus des tuples plats, est aussi étudiée par exemple dans des projets tels Pegasus [Rafi91 ] ou IRO-DB []. En résumé, les lignes d'évolution des médiateurs sont vers l'intégration de fonctionnalités multibases, transactionnels et objets, donc vers des médiateurs plus intelligents.

# techniques sgbd relationnels

## INTRODUCTION

Nés au début des années 60 pour le programme Apollo de conquête de la lune, les SGBD ont déjà une longue histoire. Aujourd'hui basés sur le modèle relationnel, ils deviennent serveurs dans des architectures client-serveur. C'est à ce titre que nous les étudions ici.

Les serveurs relationnels sont basés sur SOL, langage assertionnel non procédural ; c'est-à-dire que SOL permet de demander au serveur les données que l'on veut par des assertions constituées de formules logiques, et non pas d'écrire des procédures navigant dans la base. Ce langage est basé sur le modèle relationnel inventé par E. Codd à IBM au début des années 1970 [Codd70], que nous avons brièvement rappelé au chapitre Il. Les premiers SGBD relationnels ont été construits à partir de 1974 par le projet System R à IBM [Astrahan76] et le projet Ingres à l'Université de Californie Berkeley [Stonebraker76]. Les premiers systèmes ont été commercialisés au début des années 80 : ils avaient pour nom SQUDS, Ingres, Oracle, Informix. Sybase a rejoint ce club plutôt fermé en 1986, en basant son architecture sur le client-serveur. A partir de là, tous les SGBD relationnels sont devenus client-serveur.

Au-delà du client-serveur, les SGBD relationnels ont connu un succès croissant grâce à la normalisation. Dès 1986, une première version de SOL a été normalisée. On l'appelle aujourd'hui SQL 1-86. En 1989, une extension pour un meilleur support de l'intégrité a été acceptée. Cette version est désignée sous le nom SOL 1-89. Depuis 1992 une version étendue à trois niveaux (entrée, intermédiaire et complet) constitue la norme SQL2. Une version encore plus étendue, avec les objets et les traitements, est prévue pour 1997-1998. Nous avons décrit les fonctions essentielles de ces différentes versions de SOL au chapitre 2.

Dans ce chapitre, nous présentons les fonctions essentielles d'un serveur relationnel, vu de l'intérieur. Une architecture type est tout d'abord introduite, puis chaque composant de l'architecture est analysé. L'objectif est de donner une vue générale des algorithmes et méthodes implémentés dans un serveur relationnel. Ces algorithmes et méthodes sont aujourd'hui bien connus. Ils n'ont cessé d'être améliorés depuis le début des années 80.

Le chapitre est plus précisément organisé comme suit. Après la présente introduction, l'architecture type d'un serveur relationnel est introduite, d'un point de vue fonctionnel puis opérationnel. La section 3 est centrée sur l'analyse des requêtes et la définition d'un format interne couramment admis dans les SGBD. La section 4 traite de la gestion des schémas stockés dans la métabase. Les trois sections qui suivent se concentrent sur les problèmes de contrôle qui incluent le support des vues externes, la gestion de l'intégrité et le contrôle des droits d'accès. La section 8 examine un problème clé: l'optimisation des requêtes. Nous étudions les différentes étapes de l'optimisation. La section qui suit analyse les techniques d'exécution des opérateurs relationnels, en particulier la sélection et la jointure. Enfin, avant la conclusion, la section 10 aborde les mécanismes plus nouveaux et très importants en client-serveur qui permettent de supporter des traitements au sein des serveurs et de rendre les bases actives: procédures stockées, déclencheurs et règles.

## Architecture type d'un serveur relationnel

Les architectures client-serveur mettent en jeu des serveurs capables de gérer de grands volumes d'informations. L'importance du standard SQL a fait évolué les grands SGBD vers des architectures voisines.

### 2.1) Architecture fonctionnelle

Tous les grands serveurs accomplissent aujourd'hui des fonctions similaires essentiellement définies dans la norme SQL2. La réception des requêtes s'effectue depuis le client par les outils de connections qui mettent en oeuvre les protocoles de dialogue client-serveur et permettent de transmettre les requêtes au serveur sous forme de chaînes de caractères. Les fonctions essentielles d'un serveur sont résumées figure 31.

Une première couche assure la compilation des requêtes en **plan d'exécution** optimisée.

**Notion 1 : Plan d'exécution *(Execution plan)***

Programme optimisé composé d'opérations d'accès à la base de données de bas niveau permettant de calculer la réponse à une question.

La compilation nécessite tout d'abord une analyse syntaxique transformant la chaîne de caractères constituant la requête en format interne. Dans un deuxième temps, le SGBD effectue la génération des procédures de contrôles nécessaires à la vérification des droits d'accès et des contraintes d'intégrité des données dans le cas de mise à jour. De plus, si la requête porte sur une vue virtuelle définie par rapport aux tables réellement stockées dans la base, le compilateur doit la traduire en une requête généralement plus complexe portant sur ces tables stockées. Enfin, lorsque la requête a été correctement enrichie pour les contrôles, elle est passée à **l'optimiseur** qui génère alors le plan d'exécution optimisé.

**Notion2 : Optimiseur(Optimizer)**

Composant du SGBD chargé d'élaborer le meilleur plan d'exécution possible pour calculer la réponse à une question.

Le plan d'exécution est finalement soit directement exécuté (dans le cas de requêtes dynamiques saisies en conversationnel ou construites par programme), soit stocké pour de multiples exécutions ultérieures (dans le cas de requêtes statiques imbriquées dans des programmes) .

Une seconde couche assure l'exécution des plans optimisés. C'est à ce niveau que l'on trouve les algorithmes d'exécution des opérateurs d'accès référencés par le plan. Ceux-ci incluent les méthodes d'accès aux fichiers découpés en pages. Les entrées-sorties pages sont effectuées soient dans des fichiers gérés par le système opératoire, soit directement sur des disques virtuels. Dans tout les cas, la gestion des accès concurrents afin de prévenir les risques de pertes d'opérations et d'introduction d'incohérences entre transactions doit être intégrée. De plus, les mises à jour doivent être fiabilisées par des mécanismes de journalisation afin d'assurer les propriétés d'atomicité et de durabilité des effets des transactions.



Figure 31 - Architecture fonctionnelle type d'un SGBD.

### 2.2) Architecture opérationnelle

Ces dernières années, les serveurs ont évolué vers une notion de processus serveurs capables de supporter des requêtes multiples d'utilisateurs différents. On parle de serveurs multifilières.

**Notion 3: Serveur multifilières *(muJti-threadedserver)***

Serveur capable de traiter plusieurs requêtes en quasi-simultanéité au sein d'un même processus multiplexé.

Le multifilières se caractérise par le fait qu'un seul processus exécutant le code du SGBD est capable d'exécuter en quasi simultanéité plusieurs requêtes. Ceci nécessite un support multitâches au sein d'un même processus serveur, adapté au SGBD, en général sans protection mémoire entre les contextes des différentes requêtes.

La figure 32 illustre l'architecture typique d'un SGBD multiserveurs multifilières. Plusieurs processus SGBD sont activés lors du lancement du serveur, chacun d'eux pouvant être par exemple dédié à une ou plusieurs bases. Un processus répartiteur reçoit les requêtes des clients et les distribue aux différents serveurs, par exemple selon la base concernée. Un processus serveur traite les requêtes de ses clients simultanément dans un même environnement mémoire. Là, il génère des plans qui sont ensuite exécutés, par le même processus ou par un autre. Lors de l'exécution, les accès à la base sont effectués sur la mémoire cache. En cas d'absence d'une page demandée, celle-ci est montée en mémoire par un processus d'entrée-sortie asynchrone. Les pages modifiées sont reportées sur disques seulement à la validation de transaction (au *commit).* Plusieurs processus d'entrée-sortie peuvent travailler en parallèle, par exemple sur des disques différents.



Figure 32 - Architecture opérationnelle type d'un SGBD.

Afin d'assurer l'atomicité des transactions, il est nécessaire de gérer des journaux d'images avant et après mise à jour. Ceux-ci sont compactés et gérés souvent en différentiel (seules les données modifiées sont journalisées). Les tampons contenant les journaux sont écrits dans une mémoire cache séparée (appelée ici journal) et vidés sur disques par un processus d'archivage lors des validations de transactions.

Certains serveurs offrent un découpage en processus encore plus fins et plus paramétrables. Par exemple, le processus serveur multifilières peut lui-même être divisé en processus virtuels, chacun correspondant à des fonctions spécifiques telles que entrée-sortie et sélection, jointure et tri, gestion du réseau. Chaque processus virtuel pourra alors être implémenté par un ou plusieurs processus réels, selon la configuration du système. Le problème qui se pose alors est de partager efficacement les données constituant le contexte d'une requête en mémoire commune. Ce type d'architecture se prête bien aux machines parallèles où les multiples processus peuvent être assignés à des processeurs différents communiquant par mémoire commune.

## L'ANALYSE DES REQUETES

Le langage SQL comporte quatre opérations essentielles de manipulation de données :

* SELECT qui permet de rechercher les données satisfaisant un critère ;
* INSERT qui provoque l'insertion de nouveaux tuples dans une table ;
* UPDATE qui permet la mise à jour des tuples sélectionnés par un critère ;
* DELETE qui réalise la suppression des tuples satisfaisant un critère.

La figure 33 illustre une base de données typique avec une requête SELECT permettant de rechercher les noms et prénoms des buveurs qui habitent Mâcon et qui ont bu un Mâcon depuis le 1er janvier 1990.

**VINS** (NV : Int, Cru: CharVar, Millésime: Int, Degré: Fixed 5.2)

**BUVEURS** (NB : Int, Nom: CharVar, Prénom: Char(12) , Ville: CharVar)

**BOIRE** (NB : Int, NV : Int, Date: Date, Quantité: Int)

**SELECT** NOM, PRENOM

**FROM** BUVEURS B, BOIRE A, VINS V

**WHERE** B.VILLE = "MACON" **AND** B.NB = A.NB **AND** A.NV = V.NV **AND** A.DATE ≥ 01-01-90 **AND** V.CRU = "MACON"

Figure 33 - Exemple de base de données et de requête.

La première tâche du serveur est l'analyse des requêtes SOL. Celle-ci est dirigée par un automate supportant la grammaire du langage. Les actions de l'automate permettent de générer un format interne de la requête. Un arbre syntaxique résultant de l'analyse constitue souvent le résultat d'une première étape d'analyse. Une deuxième étape permet la construction d'un format interne plus facilement transformable.

Le format interne des requêtes retenu est généralement basé sur les arbres relationnels.

**Notion 4 : Arbre relationnel (Relational tree)**

Arbre représentant une question, dont les feuilles correspondent aux tables de la base, les nœuds internes aux opérations appliquées aux flots de tuples représentés par les arcs, et la racine au résultat de la question.

Un arbre relationnel est une représentation d'une expression de l'algèbre relationnelle sous forme d'un arbre. Une expression est obtenue par applications successives d'opérations de base aux relations. On distingue les **opérations unaires**, qui s'appliquent à une relation et produisent une relation en résultat. Ce sont la **restriction** et la **projection**.

**Notion 5 : Restriction (Restriction)**

Opération consistant à sélectionner les tuples d'une relation répondant à un critère du type <Attribut> <Comparateur> <Valeur>.

**Notion 6 : Projection (Projection)**

Opération consistant à éliminer les colonnes d'une relation en ne conservant que celles citées en argument et à supprimer les tuples en doubles.

**Les opérations binaires** combinent deux relations pour en produire une troisième. Les opérations ensemblistes sont les opérations traditionnelles de la théorie des ensembles appliquées aux relations: **union, intersection,** et **différence**. Les opérations de **jointure** et **produit cartésien** (jointure dégénérée sans critère) permettent de combiner dans une même table tous les tuples d'une relation avec tous les tuples de l'autre sous condition qu'un critère de composition soit vérifié.

**Notion 7 : Jointure (Join)**

Opération consistant à concaténer les paires de tuples t1 d'une table R1 et t2 d'une table R2 satisfaisant un critère du type t1.A e t2.B, où A et B sont des attributs (ou groupes d'attributs) respectivement de R1 et R2.

Le critère est souvent l'égalité des attributs de même nom: on parle alors de jointure naturelle. Ce critère peut être plus complexe et conduire à comparer des attributs de la première relation avec des attributs de la seconde via un comparateur de type :::;, 2, <, >, etc. ; on parle alors de **théta-jointure**.

Dans un arbre relationnel, les opérations de l'algèbre correspondent aux nœuds. Elles sont représentées par des symboles caractéristiques, comme indiqué figure 34. L'arbre représente les flux de données entre les nœuds, depuis les relations de base, jusqu'à la relation résultat. La figure 35 illustre un arbre relationnel résultant de l'analyse de la question donnée figure 33.



Figure 5 - Exemple d'arbre relationnel.

## LA GESTION DES SCHEMAS

Afin d'analyser les requêtes et de mémoriser la structure des données des bases, le SGBD doit mémoriser les schémas logiques et physiques des tables. La mémorisation s'effectue en général dans une base de données particulière, appelée *métabase.* Ce concept a déjà été défini au chapitre 2. Chaque système a en général son format spécifique de métabase. SQL2 propose cependant un schéma normalisé.

La structure d'une métabase est plutôt hiérarchique et comporte en particulier les entités représentées figure 6. Une représentation relationnelle est obtenue par attribution d'une clé à chaque tuple (numéro) et report dans les tables afin d'assurer les liens hiérarchiques sous forme de contraintes référentielles. La figure 36 présente un noyau de schéma de métabase. Une véritable métabase est beaucoup plus riche: elle comporte au moins une vingtaine de relations permettant de décrire finement le schéma et le contenu de la base (statistiques par exemple).

SCHEMA

CONTRAINTES

TABLES

DOMAINES

CLES

COLONNES

**(a) les entités principales**.

SCHEMAS (CATALOG, NOMB, Créateur,Caractère-Set, ...)

TABLES (CATALOG, NOMB, NOMR, Type, ...)

DOMAINS (ÇATALOG, NOMB,NOMD, Type, Défaut, Contrainte, ...)

COLUMNS (CATALOG, NOMB, NOMR, NOMA, Pos, Type, ...)

TYPES (CATALOG, NOMB, NOM, MaxL, Precision, ...)

CONSTRAINTS (CATALOG, NOMB, NO MC, TypeC, NomR, ...)

USERS (NOM, ...)

**(b) les tables principales.**

Figure 36 - Schéma de la métabase.

La métabase est gérée par le SGBD, comme toute base de données. Elle est accédée par le compilateur de requêtes. Les parties les plus fréquemment accédées restent dans un cache en mémoire. Son optimisation est essentielle pour un fonctionnement efficace du serveur.

## LE SUPPORT DES VUES EXTERNES

### 5.1) Objectifs

Les *vues* permettent de définir des tables virtuelles correspondant aux besoins des programmes d'application en termes de données. Dans le monde du client-serveur, les vues sont un élément essentiel d'optimisation de performance. Par exemple, la définition d'une vue jointure de deux tables, ou résultant d'un calcul d'agrégat, évite au client les risques de lire les tuples des tables et de faire le calcul de jointure ou d'agrégat sur le client: seules les données résultant du calcul de la vue sont exportées sur le site client. On laisse ainsi faire au serveur les calculs de jointure, agrégat, etc., qu'il sait en principe bien faire.

De manière plus générale, les vues permettent d'assurer une meilleure indépendance logique des programmes aux données. En effet, le programme restera invariant aux modifications de schémas de la base s'il accède *via* une vue qui l'isole de la base. Lors des modifications, l'administrateur modifiera seulement la requête définissant la vue, sans modifier celle-ci. Les vues ont aussi un rôle de sécurité: l'utilisateur ne peut accéder qu'aux données des vues auxquelles il a le droit d'accès; ainsi, les données en dehors de la vue sont protégées.

En pratique, les vues sont des relations virtuelles définies par une question. Cette définition est stockée dans la métabase (table REGLE de la figure 6). Les vues sont interrogées comme des relations normales. Idéalement, elles devraient pouvoir être mises à jour comme des relations normales.

### 5.2) Définition des vues

La syntaxe de la commande de création de vue en SQL est donnée figure 7 avec un exemple de définition des gros buveurs comme des buveurs ayant bu au moins une fois en quantité supérieure ou égale à 100. La clause WITH CHECK OPTION demande la vérification du critère (existence d'un tuple dans boire en quantité supérieure à 100) lors de l'insertion d'un tuple dans la vue, donc d'un buveur. Une telle clause permet d'imposer des contraintes lors des mises à jour ou insertion.

**CREATE VIEW** <nom de table> [ «nom de colonne>+)]

**AS** <spécification de question>

[**WITH CHECK OPTION**]

**CREATE** VIEW GROS-BUVEURS (NB, NOM, PRENOM)

**AS SELECT** NB, NOM, PRENOM

**FROM** BUVEURS B, BOIRE A

**WHERE** B.NB = A.NB **AND** A.QTE ~ 100

Figure 37 - Définition de vues.

### 5.3) Interrogation au travers de vues

L'interrogation au travers d'une vue s'effectue comme pour une table normale. Le rôle du serveur est alors d'enrichir la question avec la définition de la vue qu'il retrouve dans la métabase. Celle-ci est généralement stockée sous forme d'un arbre relationnel. L'arbre résultant de l'analyse de la question est simplement connecté à celui définissant la vue. L'ensemble constitue un arbre qui représente la question enrichie qui est alors passée à l'optimiseur. La figure 38 illustre ce mécanisme appelé **modification de question** pour la recherche des gros-buveurs de nom Martin.

**Notion 8 : Modification dequestlon *(Qu modification)***

Mécanisme consistant à modifier une question en substituant des sous-questions aux relations de base ou en ajoutant des sous-questions.

La modification de question est une technique inventée dans le projet Ingres à Berkeley [Stonebraker75]. Elle permet de remplacer les vues par leurs définitions lors des recherches ou de rajouter des prédicats afin de vérifier des propriétés avant d'exécuter une requête. Cette technique peut aussi être utilisée pour les mises à jour afin de vérifier par exemple les contraintes d'intégrité.

### 5.4) Mises à jour au travers de vues

Le problème est de traduire une mise à jour (ou insertion) portant sur une vue en mise à jour sur les relations de la base. La traduction doit bien sûr préserver la vue; c'est-à-dire que si l'on calcule la vue et on lui applique la mise à jour, on doit obtenir le même résultat que si l'on applique les mises à jour à la base et l'on calcule la vue. L'équation *u(v(B))* = *v(u'(B))* doit donc être vérifiée, en notant B la base, *v* le calcul de vue, *u* la mise à jour sur la vue et *u'* les mises à jour correspondantes sur la base. Malheureusement le problème du calcul de *u'* n'a pas toujours une solution unique [Abiteboul91 ].

En pratique, la plupart des systèmes résolvent le problème en restreignant les possibilités de mises à jour à des vues monotables : seuls les attributs d'une table de la base doivent apparaître dans la vue. En imposant de plus que la clé de la table de la base soit visible, il est possible de définir une stratégie de mise à jour simple. Celle-ci consiste à modifier les seuls tuples référencés par la mise à jour dans la table visible de la base. La définition d'une vue peut référencer d'autres tables qui permettent de vérifier que les tuples insérés ou modifiés appartiennent bien à la vue, ceci dans le cas de présence de la clause CHECK OPTION. En résumé, la mise à jour au travers de vue est un problème complexe, bien étudié en théorie, mais dont les solutions pratiques sont réduites, Certains SGBD permettent des mises à jour de vues multitables en laissant l'administrateur définir la stratégie de report.

### 5.5) Les vues concrètes

Une vue est en principe une fenêtre dynamique sur une base de données, dont une partie est instantiée lors d'une question. La concrétisation de la vue par le serveur peut être plus avantageuse si celle-ci est souvent utilisée et si les tables sources sont peu modifiées. Ainsi, certains serveurs supportent des **vues concrètes**.

**Notion 9 : Vue concrète *(Concret view)***

Table calculée à partir des tables de la base par une question et matérialisée sur disques par le SGBD.

Une vue concrète est calculée dès sa définition et mise à jour chaque fois qu'une transaction modifie la base sous-jacente. La mise à jour s'effectue en différentiel, c'est-à-dire que seuls les tuples modifiés sont pris en compte pour calculer les modifications à apporter à la vue. Ceci est simple lors des insertions, beaucoup plus complexe lors des mises à jour de type suppression ou modification, Pour permettre de répercuter suppressions et mises à jour, il faut en général garder -au moins partiellement -les chaînes de dérivation qui ont permis de calculer les tuples de la vue concrète,

## LA GESTION DE L'INTEGRITE

### 6.1) Objectifs et principes

Afin de représenter fidèlement le monde réel, les données doivent obéir à des règles implicites et parfois explicites. Ces règles permettent de contrôler les tuples lors de leur insertion dans des tables de la base. Elles sont appelées **contraintes d'intégrité**. Une base de données qui satisfait les contraintes d'intégrité est dite cohérente [Simon84].

Les principaux types de contraintes d'intégrité supportées par les serveurs relationnels sont énumérés ci-dessous.

* **Domaine de valeurs d'un attribut**. Il s'agit de contrôler les valeurs insérées dans une colonne d'une table. Une contrainte de domaine précise généralement un type de données (entier, réel, chaîne de caractère, date, etc,), une plage de valeurs voire une liste de valeurs possibles, éventuellement l'impossibilité d'une valeur nulle.
* **Unicité de clé**. Ce type de contrainte spécifie une ou plusieurs colonnes d'une table dont la valeur doit permettre de déterminer à tout instant un tuple unique. Deux valeurs idenditiques pour des tuples différents ne sont donc pas permises.
* **Contrainte référentielle.** Une telle contrainte précise qu'un ou plusieurs attributs d'une table doivent apparaître comme clé dans une autre table. A chaque insertion, le système vérifiera que la valeur existe comme valeur de clé dans la table référencée. A chaque suppression dans la table référencée, le système vérifiera que la valeur ne figure pas dans la colonne de la table référençante. Si elle figure, le système pourra refuser l'insertion, supprimer les tuples référençant (cascade), ou mettre à nul leur valeur de la colonne correspondante.
* **Assertion multirelations.** Il s'agit de contraintes spécifiant des assertions logiques entre tuples de tables différentes qui doivent rester vérifiées. Ces assertions peuvent faire inter. venir des calculs arithmétiques et des agrégats. Ce sont les contraintes les plus compliquées à gérer. Par exemple, pour gérer un stock de produits avec les achats et les ventes, une contrainte peut spécifier que la quantité en stock doit rester égale à la somme des quantités achetées moins la somme des quantités vendues, ceci pour chaque produit.

### 6.2) Définition des contraintes

Le langage SQL depuis sa version 89 spécifie une syntaxe afin de définir les types de contraintes évoqués ci-dessus. Les contraintes sont spécifiées lors de la création des tables. En SQL2, il est possible d'ajouter des contraintes sur les tables existantes ou d'en supprimer.

### 6.3) Vérification des contraintes

La vérification des contraintes fait en général appel à des algorithmes spécialisés par type de contraintes [Gardarin86]. Les contraintes de domaines sont vérifiées à la volée lors du transfert des tuples insérés ou modifiés dans les tampons du système. Il suffit de tester que les champs correspondants sont du type indiqué e/ou appartiennent à la bonne plage de valeurs.

Les contraintes de type unicité de clé et les contraintes référentielles nécessitent habituellement l'accès à un index qui doit en principe exister sur la clé primaire de chaque relation. Pour l'unicité de clé, lors d'une insertion, le SGBD vérifie que la clé n'existe pas déjà dans l'index. Pour les clés étrangères, on vérifie leurs existences dans l'index de la relation réfé­rencée lors des insertions dans la relation référençante.

Les assertions multirelations sont plus difficiles à vérifier. Lors de l'insertion ou de la suppression de tuples, une méthode différentielle consistant à vérifier que les tuples supprimés ou insérés ne rendent pas la contrainte fausse est souvent applicable. Pour les calculs arithmétiques et les agrégats, les algorithmes demandent en général de mémoriser dans la base les valeurs des calculs ou agrégats (par exemple, les quantités en stock, vendues ou achetées). Quoiqu'il en soit, ce genre de contraintes est coûteux à vérifier. Peu de systèmes les supportent.

## LE CONTROLE DES DROITS D'ACCES

### 7.1) Objectifs et moyens

L'objectif du contrôle des droits est de protéger les données contre les accès non autorisés. La résistance au vol de données est une tâche plus difficile, qui nécessite des outils de cryptage en dehors du serveur, au niveau des clients.

Les SGBD relationnels permettent d'accorder et retirer des **autorisations** aux utilisateurs sur les objets de la base [Griffiths76].

**Notion 10: Autorisation *(Authorization)***

Droit d'opération sur un objet de la base (une table, une vue, etc.) accordé à un sujet généralement constitué par un utilisateur.

Les objets considérés sont, dans la plupart des systèmes, les relations implémentées ou des vues sur ces relations. Les vues permettent de spécifier des parties de tables incluant des attributs voir des tuples spécifiques. La sécurité peut ainsi être définie très finement.

### 7.2) Accords et retraits d'autorisations

Les utilisateurs sont enregistrés avec nom et mot de passe par un administrateur. Les autorisations de lecture, écriture et administration sont *a priori* accordées à tout utilisateur qui crée une relation. Les droits sont ensuite passés entre utilisateurs par la commande GRANT définie figure 10. La clause WITH GRANT OPTION permet de préciser que l'on autorise le passage du droit. T out droit peut être retiré par la clause REVOKE aussi définie figure 10. Afin d'éviter les conservations abusives de droits, tout droit retiré est enlevé à tous ceux à qui il a été donné, à moins qu'il n'ait été aussi donné par quelqu'un d'autre. Ceci complique grandement les mécanismes de passation et de retrait de droits, qui doivent garder la trace des transmissions effectuées.

**GRANT** <DROITS> **ON** <RELATION>

**TO** «USAGER»

[**WITH GRANT OPTION** ]

**REVOKE** <DROITS> **ON** <RELATION>

**FROM** «USAGER»

<DROITS > .: :.= **ALL** | [<OPERATION>]

<OPERATION> : := **SELECT** I **INSERT** I **DELETE** I **UPDATE** I **ALTER**

Figure 39 - Accord et retrait des autorisations.

Il faut noter que le droit de créer une table dans une base est un droit initial accordé par l'administrateur. Il peut être vu comme le droit d'insérer des tuples dans la table RELATION de la métabase. De manière générale, l'accord de droit sur la métabase permet d'accorder des droits de création, modification ou suppression de schémas, donc de tables.

### 7.3) Groupes, rôles et procédures

Les systèmes modernes gèrent non seulement des utilisateurs avec nom et mot de passe, mais aussi des groupes d'utilisateurs et des rôles.

**Notion 11: Groupe d'utilisateurs *(User group)***

Groupe d'utilisateurs désigné par un nom constituant un sujet pour le mécanisme d'autorisation.

**Notion 12: Rôle d'utilisateur (User role)**

Nom représentant une fonction dans l'entreprise, constituant un sujet pour le mécanisme d'autorisation, pouvant être assigné à un utilisateur ou à un groupe d'utilisateurs.

Un rôle correspond à un nom logique qui permet de désigner des utilisateurs et/ou des groupes d'utilisateurs par leur fonction. Ceci simplifie l'accord des droits.

Dans les architectures clients-serveurs, l'exécution de procédures stockées groupant plusieurs requêtes SQL dans une même fonction permet d'éviter des transferts inutiles de données sur le réseau. Les droits sont généralement prolongés aux procédures stockées, si bien que l'on peut passer ou retirer le droit d'exécuter une procédure stockée par les commandes GRANT et REVOKE.

## L'OPTIMISATION DES REQUETES

### 8.1) Etapes de l'optimisation

L'optimiseur reçoit les requêtes sous forme d'une représentation interne constituée par un arbre d'opérateurs. L'ordre des opérations dérive directement de l'ordre des critères dans la question de l'utilisateur et dans la définition de la vue. En général, les opérations sont associatives et commutatives avec quelques précautions que nous étudierons plus en détail ci-dessous.

Une première étape de l'optimisation permet d'obtenir une représentation canonique de la question. Elle consiste à grouper les opérations similaires. Les critères de restriction portant sur une même table sont généralement groupés et mis en forme normale conjonctive ( *et* de *ou)* ou disjonctive *(ou* de *et).* Après les restrictions, les jointures sont représentées, puis les unions/différences portant sur des tables différentes, puis les agrégats et tris, et enfin les restrictions après agrégats (clause HA VING de SQL) et la projection finale.

La seconde étape permet d'effectuer un premier ordonnancement des opérations. Elle part de la simple constatation que restrictions et projections sont des opérations qui réduisent le volume des données à traiter par élimination de lignes ou de colonnes inutiles. Les règles de restructuration algébriques permettent ainsi d'appliquer d'abord les restrictions puis les projections. En fait, des projections sont insérées après chaque opération en appliquant la règle simple que toute colonne inutile dans la suite doit être éliminée.

La dernière étape consiste à générer des plans d'exécution candidats en permutant dans la mesure du possible les opérations restantes (jointures, unions, agrégats) et en choisissant pour chaque opérateur un algorithme d'exécution. Les choix d'algorithmes peuvent être insérés dans l'arbre au moyen d'annotations sur les nœuds. Un coût d'exécution estimé est alors affecté à chaque plan généré. Celui de coût minimum est retenu.

A noter que les étapes 1 et 2 sont indépendantes des données, alors que l'étape 3 nécessite un **modèle de coût** intégrant structures et volumes des données [Selinger76]. Dans la suite, nous détaillons les principes des étapes 2 et 3.

**Notion 13** : **Modèle de coût *(Cost model)***

Ensemble de formules permettant d'estimer le coût d'un plan d'exécution dans un SGBD relationnel.

### 8.2) Restructuration algébrique

A partir d'un arbre algébrique, il est possible de générer un plan d'exécution en parcourant l'arbre, des feuilles vers la racine, et en choisissant pour chaque opérateur un algorithme d'exécution. Une opération peut être exécutée dès que ses opérandes sont disponibles ; ainsi si l'opération a n'utilise pas les résultats de l'opération *b,* a et *b* peuvent être exécutées en parallèle.

On cherche bien sûr à optimiser les temps de réponse, donc à minimiser le temps nécessaire à l'exécution du plan. Le problème est donc de générer un plan optimal. Pour cela, il faut optimiser simultanément :

* le nombre d'opérations d'entrées-sorties,
* le parallélisme entre les entrées-sorties,
* la taille des tampons nécessaires à l'exécution,
* le temps unité centrale nécessaire.

L'optimisation effectuée dépend essentiellement de l'ordre des opérations apparaissant dans l'arbre algébrique utilisé. Il est donc essentiel d'établir des règles permettant de générer, à partir d'un arbre initial, tous les arbres possibles afin de pouvoir ensuite choisir celui conduisant au meilleur plan. En fait, le nombre d'arbres étant très grand, on est amené à définir des heuristiques pour déterminer un arbre proche de l'optimum.

Ces règles ont été précisées pour la première fois dans System R, puis complétées par la suite. Elles dépendent évidemment des opérations relationnelles considérées. Nous considérons ici l'ensemble des opérations restriction, jointure, union, différence et énonçons ci-dessous huit règles classiques.

Un exercice pourra consister à représenter les autres règles.

* Règle 1 .Commutativité des jointures.
* Règle 2 : Associativité des jointures.
* Règle 3 : Regroupement des restrictions.
* Règle 4 : Quasi-commutativité des restrictions et projections.
* Règle 5 : Quasi-commutativité des restrictions et jointures.
* Règle 6 : Pseudo-commutativité des restrictions et unions/différences.
* Règle 7 : Quasi-commutativité des projections et jointures.
* Règle 8 : Pseudo-commutativité des projections et unions.

**L'optimisation** **algébrique** simple [Smith75] consiste à exécuter tout d'abord les opérations unaires (restrictions, projection) puis les opérations binaires.

**Notion 14 : Optimisation algébrique (Algebraic optimization)**

Heuristique d'optimisation consistant à effectuer d'abord les restrictions suivies des projections, puis les autres opérations dans un ordre quelconque.

L'optimisation algébrique ne fait pas appel à un modèle de coût pour déterminer le plan d'exécution choisi. Celui-ci est en moyenne acceptable. En effet, les opérations unaires sont des réducteurs de la taille des relations, alors qu'il n'en est pas ainsi de certaines opérations binaires qui ont tendance à accroître la taille des résultats par rapport aux relations arguments. Aussi, afin de ne considérer que les arbres à flux de données minimum et de réduire ainsi le nombre d'entrées-sorties à effectuer, on est conduit à descendre les restrictions et projections. De plus, quand deux projections successives portent sur une même relation, il est nécessaire de les regrouper afin d'éviter un double accès à la relation.

Les principes précédents conduisent à l'algorithme d'optimisation suivant :

* séparer les restrictions comportant plusieurs prédicats à l'aide de la règle 3,
* descendre les restrictions aussi bas que possible à l'aide des règles 4, 5, et 6,
* regrouper les restrictions successives portant sur une même relation,
* descendre les projections aussi bas que possible à l'aide des règles 7 et 8,
* regrouper les projections successives en conservant les attributs restants et éliminer d'éventuelles projections inutiles qui auraient pu apparaître (projection sur tous les attributs d'une relation).

En règle générale, une restriction suivie par une projection sont exécutées simultanément par un même opérateur de sélection (restriction + projection) afin d'éviter plusieurs passes sur une même relation. Il en est de même pour une jointure suivie par une projection. Cet arbre n'est pas forcément optimal pour au moins deux raisons: la descente des restrictions et projections n'est qu'une heuristique et l'ordre des opérations binaires n'a pas été optimisé.

### 8.3) Modèle de coût

La restructuration algébrique est insuffisante car elle n'ordonne pas les opérations binaires. Aussi, l'application d'une sélection initiale peut faire perdre un index qui serait utile pour exécuter une jointure. Afin d'aller au-delà, une solution pour choisir le meilleur plan d'exécution consiste à les générer tous, à estimer le coût d'exécution de chacun et à choisir celui de moindre coût. Malheureusement, une telle stratégie se heurte à deux difficultés.

Tout d'abord, le nombre de plans d'exécution possible est très grand. Cet inconvénient peut être pallié en éliminant *a priori* tous les plans qui font appel à l'opération de produit cartésien comme dans système R ; en effet, la commutation de jointures peut générer des produits cartésiens qui, en général, multiplient les tailles des relations à manipuler. On a donc tout intérêt à éviter les plans d'exécution contenant des produits cartésiens. Une autre possibilité encore plus restrictive est d'éliminer tous les plans qui n'effectuent pas les sélections dès que possible. Ces heuristiques éliminant souvent des plans intéressants, on préfère aujourd'hui mettre en oeuvre une stratégie aléatoire d'exploration de l'espace des plans, comme nous le verrons ci-dessous.

Ensuite, l'estimation du coût nécessite d'une part la connaissance de la taille des résultats intermédiaires de toutes les opérations considérées, d'autre part la prise en compte des chemins d'accès aux relations (index, placement, lien. ..) qui change directement ces coûts. Nous allons ci-dessous examiner les résultats concernant ces deux problèmes.

L'estimation de la taille des résultats est basée sur un modèle de distribution des valeurs des attributs dans chacune des relations, ainsi qu'éventuellement des corrélations entre les valeurs d'attributs. Le modèle doit être conservatif, c'est-à-dire que pour toute opération de l'algèbre relationnelle, il faut être capable, à partir des paramètres décrivant les relations arguments, de calculer les mêmes paramètres pour la relation résultat. Le modèle le plus simple est celui qui suppose l'uniformité de la distribution des valeurs et l'indépendance des attributs [Selinger79]. De telles hypothèses sont utilisées dans tous les optimiseurs de systèmes en l'absence d'information plus précise sur la distribution des valeurs d'attributs. Un tel modèle nécessite de connaître au minimum :

* le nombre de valeur d'un attribut A noté CARD(A),
* les valeurs minimum et maximum d'un attribut A notées respectivement MIN(A) et MAX(A),
* le nombre de valeurs distinctes de chaque attribut A noté NDIST(A),
* le nombre de tuples de chaque relation R noté GARD(R).

Le nombre de tuple d'une restriction cr(R) est alors calculée par la formule :

*GARD (cr(R)) = p(critère) \* GARD(R)*

où p(critère) désigne la probabilité que le critère soit vérifié, encore appelé **facteur de sélectivité** du prédicat de restriction.

**Notion 15 : Facteur de sélectivité *(SeJectivity factor)***

Coefficient associé à une opération sur une table représentant la proportion de tuples de la table satisfaisant la condition de sélection.

Avec une hypothèse de distribution uniforme des valeurs d'attributs, le facteur de sélectivité peut être calculé comme suit :

* p(A = valeur) = 1 / NDIST(A)
* p(A > valeur) = (MAX(A) -valeur) / (MAX(A) -MIN(A))
* p(A < valeur) = (valeur -MIN(A)) / (MAX(A) -MIN(A))
* p(A IN liste valeurs) = (1/NDIST(A)) \* GARD(liste valeurs)
* p(P et O) = p(P) \* p(Q)
* p(P ou O) = p(P) + p(Q) -p(P) \* p(Q)
* p(not P) = 1 -p(P)

Le nombre de tuples d'une projection sur un groupe d'attributs X est plus simplement donné par la formule :

*CARD(Π(R)) = (1-d) \* CARD(R)*

avec d = probabilité de doubles. La probabilité de doubles peut être estimée en fonction du nombre de valeurs distinctes des attributs composant X; une borne supérieure peut être obtenue par la formule :

*d = (1-NDIST(A1)/CARD(R)\*(1-NDIST(A2)/CARD(R)) ... \* (1-NDIST(A2)CARD(R)*

où A1 , A2, ...An désigne les attributs retenus dans la projection.

La taille d'une jointure est plus difficile à évaluer en général. On pourra utiliser la formule :

*CARD( R1 I><I R2) = p \* CARD(R1) \* CARD(R2)*

où p, compris entre 0 et 1, est appelé facteur de sélectivité de la jointure. En supposant une équijointure sur un critère R1.A = R2.B et une indépendance des attributs A et B, le nombre de tuples de R2 qui joignent avec un tuple de R1 donné est au plus CARD(R2)/NDIST(B). Pour tous les tuples de R1, on obtient donc CARD(R1 )\*CARD(R2) / NDIST(B) tuples dans la jointure. On en déduit la formule p = 1/NDIST(B). Mais en renversant l'ordre des relations, on aurait déduit la formule p = 11 NDIST(A). Ceci est dû au fait que certains tuples de R1 (et de R2) ne participent pas à la jointure, alors qu'ils ont été comptés dans chacune des formules. Une estimation plus raisonnable demanderait d'éliminer les tuples ne participant pas à la jointure. Il faudrait pour cela prendre en compte le nombre de valeurs effectives du domaine commun de A et B, nombre en général inconnu. On utilisera plutôt une borne supérieure de p donnée par la formule p = 11 MAX(NDIST(A), NDIST(B)).

Les estimations de taille des relations et attributs doivent être mémorisées au niveau de la métabase. Elles devraient théoriquement être mises à jour à chaque modification de la base. La plupart des systèmes ne mette à jour les statistiques que lors de l'exécution d'une commande spécifique (RUNSTAT) sur la base. Afin d'affiner les estimations, il est possible de mémoriser des histogrammes de distribution des valeurs d'attributs dans la métabase. Ces valeurs sont calculées par des requêtes comportant des agrégats. Les formules précédentes sont alors appliquées pour chaque fractile de l'histogramme qui permet d'estimer le nombre de valeurs distinctes (NOIST) ou non (CARO) dans une plage donnée.

### 8.4) Stratégie de recherche

Comme vu ci-dessus, le nombre de plans d'exécution possibles peut être très grand pour des questions complexes. Afin d'éviter de les explorer tous, les optimiseurs modernes sont construits comme des générateurs de plans couplés à une **stratégie de recherche** décou­lant des techniques d'optimisation de la recherche opérationnelle [Swami88].

**Notion 16: Stratég.e *derecherche(Searchstrategy)***

Procédure utilisée par l'optimiseur pour explorer l'espace des plans d'exécution afin de déterminer un plan de coût proche du minimum possible.

L'optimiseur prend comme paramètres des règles de transformation d'arbres relationnels. Il peut être extensible, en ce sens qu'il est possible d'ajouter de nouvelles règles. Le généra­teur de plan applique les règles. La stratégie d'évaluation est basée sur un modèle de coût et tend à réduire le nombre de plans explorés afin d'atteindre le but qui est un plan de coût proche du minimal possible. La figure 13 représente l'architecture typique d'un optimiseur extensible, qui explore l'espace des plans d'exécution.

Les différentes stratégies de recherche sont classées en une hiérarchie de spécialisation figure 14. On distingue les **stratégies énumératives** des **stratégies aléatoires.** Les stratégies énumératives énumèrent systématiquement des plans possibles. La stratégie exhaustive les énumère tous. La stratégie par augmentation les construits progressivement en partant par exemple de la projection finale et en introduisant progressivement les relations par exemple par ordre de taille croissante. Elle évitera en général les produits cartésiens et appliquera dès que possible restriction et projection.

****

Figure 40 - Architecture d'un optimiseur extensible.



Figure 41 - Principales stratégies de recherche.

Les **stratégies aléatoires** (Lanzelotte91] explorent aléatoirement l'espace des plans. L'amélioration itérative tire au hasard (par exemple en permutant l'ordre des relations) n plans et essaie de trouver pour chacun d'eux le plan optimum le plus proche (par exemple par descente des restrictions et projections, et choix des meilleurs index). L'optimum des plans localement optimum est alors retenu comme plan d'accès. La stratégie du recuit simulé procède à partir d'un plan que l'on tente d'optimiser en appliquant des transformations successives. Les transformations retenues améliorent le plan, exceptées quelques-unes introduites afin d'explorer un espace plus large avec une probabilité variant comme la température du système (en e-1/T, ou T est la température). La température est de fait une variable dont la valeur est très élevée au départ et qui diminue au fur et à mesure des transformations, de sorte à ne plus accepter de transformation détériorant le coût quand le système est gelé. Cette stratégie, bien connue en recherche opérationnelle, permet de converger vers un plan proche de l'optimum, en évitant les minimums locaux. On citera enfin les stratégies génétiques qui visent à fusionner deux plans pour en obtenir un troisième.

Un dernier problème que nous n'avons pas abordé jusque-Ià est la prise en compte des agrégats. Les optimisations possibles peuvent dépendre des fonctions de calcul: compte (COUNT), somme (SUM), moyenne (AVG), minimum (MIN), maximum (MAX), etc. En général, les calculs d'agrégats nécessitent des tris que l'on a parfois intérêt à effectuer avant l'exécution d'opérations pouvant bénéficier du fait que les données sont triées, par exemple les jointures. Peu de résultats généraux sont connus sur l'optimisation des agrégats.

## LES OPERATEURS RELATIONNELS

Les opérateurs relationnels sont généralement au centre du moteur du SGBD responsable de l'exécution des plans. En conséquence, leur optimisation est importante. Dans la suite, nous nous concentrons sur la sélection, la jointure et nous disons quelques mots sur les calculs d'agrégats. Les autres opérations peuvent être implémentées de manière similaire à la jointure.

### 9.1) Opérateur de sélection

Le placement des données dans les fichiers est effectué dans le but d'optimiser les sélections les plus fréquentes, et aussi les jointures. La présence d'index sur les attributs référencés dans le prédicat argument de la sélection change radicalement l'algorithme de sélection. Cependant, il existe des cas dans lesquels l'utilisation d'index est pénalisante, par exemple si la sélectivité du prédicat est mauvaise (plus de 75 % des tuples satisfont le critère). De plus, il peut toujours ne pas exister d'index. En résumé, on doit donc considérer deux algorithmes différents: la sélection par balayage séquentiel et par utilisation d'index.

#### 9.1.1 Sélection sans index

Le filtrage séquentiel demande que chaque tuple de la relation opérande soit comparé avec le critère. Si celui-ci est vrai, les attributs utiles sont conservés en résultat. La méthode applique donc à la fois restriction et projection. Le critère peut être compilé sous forme d'un automate afin d'accélérer le temps de parcours du tuple. L'automate peut être vu comme une table ayant en colonne les codes caractères et en ligne les états successifs. Il permet pour chaque caractère lu d'appliquer une action et de déterminer l'état suivant de l'automate. Des filtres matériels ont été réalisés selon ce principe afin d'effectuer la sélection en parallèle à l'entrée-sortie disque. Les SGBD effectuent plutôt le filtrage en mémoire.

#### 9.1.2 Sélection avec index

Un **index** associe les valeurs d'un attribut avec la liste des adresses de tuples ayant cette valeur. Il est généralement organisé comme un arbre-B et peut être plaçant, c'est-à-dire que les tuples sont placés dans les pages selon l'ordre croissant des valeurs de clés dans l'index.

Le principe d'évaluation d'un critère dont certains attributs sont indexés est le suivant. On choisit les index intéressants (par exemple, ceux correspondant à une sélectivité du sous-critère supérieure à 60 %). Pour chaque index choisi, on construit une liste d'adresses de tuples en mémoire. Puis, on procède aux intersections (critère ET) et aux unions (critère OU) des listes d'adresses. On obtient ainsi la liste des adresses de tuples qui satisfont aux sous-critères indexés. On peut ensuite accéder à ces tuples et vérifier pour chacun le reste du critère par un test en mémoire.

Certains systèmes retiennent seulement le meilleur index et vérifient tout le reste du critère. Un cas dégénéré est le cas de placement par hachage. Dans ce cas, seul le sous-critère référençant l'attribut de hachage est en général retenu et le reste est vérifié en mémoire.

### 9.2) Opérateur de jointure

Comme avec les sélections, il est possible de distinguer deux types d'algorithmes selon la présence d'index sur les attributs de jointure ou non. Dans la suite, nous développons les principes des algorithmes de jointure sans index puis avec index. Nous considérons l'équi-jointure de deux relations R1 et R2 avec un critère du type R1.A = R2.B, où A et B sont respectivement des attributs de R1 et R2. Nous supposons de plus que R1 a un nombre de tuples inférieur à celui de R2.

#### 9.2.1 Jointure sans index

En l'absence d'index, il existe trois algorithmes fondamentaux: les **boucles imbriquées**, le **tri-fusion**, et le **hachage** [Valduriez84].

**L'algorithme des boucles imbriquées** est le plus simple. Il consiste à lire séquentielle­ment la première relation et à comparer chaque tuple lu avec chaque tuple de la deuxième. Pour chaque tuple de R1, on est donc amené à lire chaque tuple de R2. L'algorithme est schématisé figure 16. Les entrées-sorties s'effectuant par page, en notant PAGE(R) le nombre de pages d'une relation R, on obtient un coût en entrées-sorties de :

 COUT = PAGE (RI) + PAGE (RI) \* PAGE (R2) .

**L'algorithme par tri-fusion** effectue le tri des deux relations sur l'attribut respectif de join­ture. Ensuite, la fusion des tuples ayant même valeur est effectuée. L'algorithme est sché­matisé figure 17. En supposant des tris en N Log N, le coût en entrées-sorties de l'algo­rithme est :

COUT = PAGE (RI) \* LOG (PAGE (RI) ) + PAGE (R2) \* LOG(PAGE(R2) ) + PAGE (RI) + PAGE (R2)

Les logarithmes (LOG) sont à base 2 pour des tris binaires. L'algorithme est en général beaucoup plus efficace que le précédent. Il dépend cependant complètement de l'efficacité du tri.

**L'algorithme par hachage** consiste à hacher la plus petite relation (R1) en appliquant une fonction de hachage h sur l'attribut de jointure A. Dans la mesure du possible, les paquets générés sont gardés en mémoire. De manière similaire, la deuxième relation R2 est hachée par la même fonction appliquée à l'attribut B. On est alors ramené à joindre les paquets de même rang par l'un des algorithmes précédents. En effet, pour pouvoir joindre, deux tuples doivent donner la même valeur lorsqu'on applique la fonction de hachage aux attributs de jointure, car ceux-ci doivent être égaux.

L'algorithme peut être amélioré par gestion d'un tableau de bits de dimension N (N aussi grand que possible) en mémoire [Babb79]. L'astuce consiste à hacher la première relation simultanément avec la fonction h et une autre fonction g distribuant uniformément les valeurs de A sur [1 ,N]. A chaque tuple haché de R1, le bit g(B) est mis à 1 dans le tableau de bits. Lorsqu'on hache la deuxième relation, on effectue pour chaque tuple de R2 un test sur le bit g(B) du tableau. S'il est à 0, on peut éliminer ce tuple qui n'a aucune chance de jointure (aucun tuple de R1 ne donne la même valeur par g). Dans le principe de l'algorithme par hachage seuls les paquets de même rang, donc de la diagonale sont joints.

Une variante de l'algorithme moins coûteuse en mémoire consiste à hacher seulement la première relation dans un fichier haché si possible en mémoire. Ensuite, on balaye la deuxième relation, et pour chaque tuple lu on tente d'accéder au fichier haché.

Le coût de l'algorithme par hachage dépend beaucoup de la taille mémoire disponible. Au mieux, il suffit de lire les 2 relations. Au pire, on est obligé de faire plusieurs passes de hachage de la première relation, le nombre de passes étant le nombre de paquets de hachage divisé par le nombre de paquets disponibles en mémoire. On peut borner ceci par PAGE(R1)/M, M étant le nombre de pages disponibles en mémoire. On obtient donc PAGE(R1 )\*PAGE(R1 )/M entrées-sorties pour hacher R1. Pour R2, on va accéder au fichier haché pour chaque tuple, d'où un coût maximum de CARO(R2) entrées-sorties, en négligeant l'effet de la mémoire. Le tableau de bits permet de diminuer CARO(R2) d'un facteur difficile à évaluer, noté f avec f < 1, qui dépend de la sélectivité de la jointure. On obtient donc au total :

COUT = PAGE(R1)\*PAGE(Rl)/M + f\*CARD(R2)

Ceci est difficilement comparable avec les formules précédentes, mais très bon si M est grand et la sélectivité de la jointure faible (f petit).

#### 9.2.2 Jointure avec index

Lorsque l'une des relations est indexée sur l'attribut de jointure (par exemple R1 sur A), il suffit de balayer la deuxième relation et d'accéder au fichier indexé pour chaque tuple. En cas de succès, on procède à la concaténation pour composer le tuple résultat. L'algorithme est similaire à la deuxième phase de l'algorithme par hachage en remplaçant le fichier haché par le fichier indexé. Le coût est de l'ordre de 3\*CARD(R2), en supposant 3 accès en moyenne pour trouver un article dans le fichier indexé. Si R2 est grande, le coût peut être prohibitif et on peut avoir intérêt à créer un index sur R2 pour se ramener au cas suivant.

Lorsque les deux relations sont indexées sur les attributs de jointure, il suffit de fusionner les deux index [Valduriez87]. Les couples d'adresses de tuples joignant sont alors directement obtenus. L'algorithme est peu coûteux en entrées-sorties (lecture des index et des tuples résultats).

### 9.3) Le calcul des agrégats

Les algorithmes permettant de calculer les agrégats sont basés soit sur le tri, soit sur une combinaison du hachage et du tri. Un tri sur l'attribut de partitionnement (argument du GROUP BY de SOL) permet de créer des monotonies. Pour chaque monotonie, on applique les fonctions d'agrégat (SUM, MIN, MAX, etc.) demandées. Un hachage préalable sur l'attribut de partitionnement permet de ramener le problème à un calcul d'agrégat dans chaque paquet de hachage. L'approche est donc ici similaire à celle de la jointure par hachage et tri.

## Le support des procédures, déclencheurs et règles

Les SGBD utilisés en client-serveur doivent être capables de supporter des traitements au niveau du serveur. Les **procédures stockées** sont déjà un support minimum. Une procédure stockée se compose de plusieurs requêtes SOL avec le code de calcul et de contrôle permettant de les enchaîner. Elle possède un nom et des paramètres d'entrée et peut retourner des paramètres de sortie comme toute procédure. Les procédures stockées sont connues au niveau de la métabase. Elles sont exécutées par les clients ayant reçu le droit d'exécution. Ceux-ci instancient les paramètres lors de la demande d'exécution. Pour optimiser l'exécution, les procédures stockées peuvent être compilées.

Au-delà des procédures stockées, les bases de données deviennent actives. Les bases de données actives sont centrées autour des notions d'événement, d'action et de condition. Un événement peut être une modification de données (INSERT, UPOATE, OELETE) ou une recherche (SELECT) sur une table. Ce peut aussi être un temps absolu ou relatif. Enfin, un événement peut être défini par l'application et déclenché par elle (opération SET EVENT). Une action est soit une requête base de données (UPOATE, INSERT, SELECT), une requête transactionnelle (ROLLBACK, COMMIT) ou l'appel d'une procédure stockée. Une condition est une qualification incluant des prédicats du type clause WHERE de SOL.

Les bases de données actives permettent de définir des phrases du type événement-action (EA) ou événement-condition-action (ECA) au niveau du schéma. Les premières (EA) sont appelées **déclencheurs** (en anglais, triggers) alors que les secondes (ECA) sont appelées **règles**. Comme nous avons introduit la notion de déclencheur au chapitre 2 nous introduirons seulement ici la notion de règle.

**Notion 17: Règle *(Rule)***

Action gardée par une condition, de la forme si <condition> alors <action>, l'action ne pouvant être exécutée que si la condition est vraie, la condition pouvant être évaluée seulement suite à un événement du type opération sur table ou temporel.

La notion de règle est plus générale que celle de déclencheur, ce dernier étant une règle dont la condition est toujours vraie. La signification d'un déclencheur EA est de lancer l'action A lorsque l'événement E se produit (après ou avant). Une règle ECA ne lance l'action A lorsque E se produit que si des tuples de la base satisfont la condition C. L'intérêt des règles par rapport aux déclencheurs et de pouvoir limiter le déclenchement aux seuls cas intéressants.

Le support de déclencheurs ou de règles dans les SGBD nécessite la présence d'un **moniteur d'événements** capable de détecter les événements et de lancer l'évaluation des conditions et l'exécution des actions [Stonebraker87].

**Notion 18** : **Moniteur d'évènements *(Event monitor)***

Module système activé en début et fin d'opérations sur les bases de données, et aussi périodiquement, capable de déterminer l'apparition d'un événement, d'évaluer des conditions, de mémoriser des états, et d'activer des actions.

Le moniteur d'événements peut évaluer un déclencheur ou une règle pour le compte de la transaction qui a concrétisé l'événement. Il peut aussi travailler pour le compte d'une transaction indépendante. La sémantique des déclencheurs ou règles est alors différente. Un moniteur d'événements doit aussi gérer une mémoire entre ses activations successives. Dans le cas d'un déclencheur, il faut passer l'ancienne valeur et la nouvelle valeur du tuple

modifié en paramètre à la procédure réalisant l'action. Dans le cas d'une règle, les tuples résultants de l'évaluation de la condition doivent aussi être passés, ce qui pose des problèmes plus complexes. Un moniteur d'événements doit détecter les problèmes de bouclage qui peuvent survenir en cas de mauvaise conception de déclencheurs ou de règles mutuellement récursifs, c'est-à-dire qui s'appellent mutuellement.

## CONCLUSION

Nous avons décrit dans ce chapitre un ensemble de techniques aujourd'hui bien maîtrisées. Les serveurs SOL intégrant ces techniques sont de plus en plus nombreux. Ils connaissent un vrai succès dans les entreprises. Les problèmes de performance demeurent un point critique, bien que de nombreux progrès aient été accomplis. Des bancs d'essais *(benchmarks)* standard permettent d'évaluer les performances en transactionnel (mises à jour ponctuelles) aussi bien qu'en décisionnel (requête de type calculs voire statistiques complexes). Ce sont respectivement TPC/A et TPC/B qui simulent des transactions sur une base de données bancaire, et TPC/C qui effectue des éditions de rapports en parallèle à des mises à jour sur une base de données de magasins gérant des produits [Gray91 ].

Aujourd'hui, les techniques décrites ci-dessus se prolongent aux extensions objets de SOL, qui permettent de transformer les SGBD relationnels en SGBD objet-relationnel. Les idées essentielles sont d'introduire les concepts de l'objet (encapsulation, héritage, polymorphisme) au niveau du modèle et du langage SQL. Un tuple d'une table devient alors un objet dont certaines données peuvent être cachées par des fonctions. Des sous-tables peuvent être dérivées de tables précédemment définies. Les pointeurs sont aussi réhabilités, de sorte à permettre le stockage d'objets complexes liés par des pointeurs. Les attributs peuvent aussi prendre des valeurs multiples. Ce type de SGBD devrait permettre de toucher un plus grand nombre d'applications et de modéliser plus naturellement des objets complexes.

Au-delà des techniques décrites dans ce chapitre, les serveurs supportent des mécanismes permettant d'assurer l'atomicité, la concurrence, l'isolation et la durabilité des mises à jour des transactions. Ces techniques seront plus spécifiquement décrites au niveau du chapitre traitant des bases de données réparties et moniteurs transactionnels.