Systèmes de Gestion de Bases de Données

Jean-Pierre CHEINEY

Philippe PICOUET

Jean-Marc SAGLIO

Septembre 1998

PREFACE

Cette nouvelle édition du cours de "Bases de Données" de 2e Année à l'E.N.S.T. est peu différente de la précédente :

Le Chapitre 2 a été allégé des considérations trop détaillées de programmation et je l'ai augmenté d'une présentation des concepts et notations utilisés par les modèles conceptuels de données, indépendants des modèles logiques caractérisant les générations successives de SGBD.

Des imperfections ou des points obscurs, relevés grâce à la lecture attentive des élèves, ont été corrigées ou explicités.

Les nombreux compléments et approfondissements nécessaires ne seront disponibl es dans le proche avenir que dans les "supports" du cours oral, ou dans le début d'édition sur "la toile" que l' internaute pourra trouver à l'URL

http://www.infres.enst.fr/~dombd

A l'heure où je me retrouve le seul de l'équipe BD fondée par Jean-Pier re Cheiney encore en fonction dans l'Ecole, je tiens à exprimer toute ma gratitude et mon respect pour ceux qui, avant moi, ont si bien enseigné cette discipline qui traverse "verticalement" l'informatique, des modèles et langages aux architectures des systèmes.

Jean-Marc SAGLIO

N.B. Deux modules sont prévus en 3e Année de l'E.N.S.T. pour aller plus loin que cet enseignement de base :

BDDO = http://www.infres.enst.fr/~potier/modules/bddo.html

BDAS = http://www.infres.enst.fr/~potier/modules/bdas.html

PREFACE A L'EDITION 1995

Ce polycopié correspond à l'enseignement dispensé à TELECOM Paris dans le cadre du bloc "Bases de Données" de la Dominante Informatique.

Il aborde essentiellement les objectifs, fonctions et aspects généraux des Systèmes de Gestion de Bases de Données (SGBD) d'hier et d'aujourd'hui. Plus particulièrement les modèles (hiérarchique, réseau et relationnel), les langages de définition et de manipulation de données, les règles et méthodes de conception des schémas relationnels logiques (tables, vues, contraintes d'intégrité) et physiques (fichiers et index).

Les aspects internes des SGBD relationnels (optimisation de questions, validation des contraintes d'intégrité, gestion de la concurrence et de la durabilité…) n'y sont présentés que très généralement. Ils font, avec les aspects distribués (SGBD distribués, répartis, fédérés ...) l'objet d'un module optionnel d'approfondissement "Bases de Données: Architectures et Systèmes" (BDAS).

De même les environnements et standards de développement d'applications sur SGBD sont présentés dans leur généralité. Mais les évolutions et les perspectives nouvelles ne sont abordées que dans le modules optionnel d'approfondissement "Bases de Données: Déduction et Orientation Objet" (BDDO).

Les chapitres 1 à 7 sont presque identiques à ceux de la version précédente de Novembre 1989, à quelques corrections, précisions et petites additions près. Les chapitres 8 et 9 sont entièrement nouveaux et la Bibliographie a été actualisée.

Nous remercions, pour leur patient travail de relecture critique et correction, Bruno Joachim et Nathalie Leruyet.

Nous dédions cette nouvelle édition au professeur Jean-Pierre Cheiney qui en a été l'initiateur et principal inspirateur et qui, malheureusement, nous a quitté brusquement le 15 Janvier de cette année.

Philippe PICOUET et Jean-Marc SAGLIO

Table des matières

CHAPITRE 1 : OBJECTIFS DES BASES DE DONNEES

Les Bases de Données occupent aujourd'hui une place de plus en plus importante dans les systèmes informatiques. Les Systèmes de Gestion de Bases de Données (SGBD) remplacent les anciennes organisations où les données, regroupées en fichiers, restaient liées à une application particulière. Ils assurent le partage, la cohérence, la sécurité d'informations qui, de plus en plus, constituent le cœur de l'entreprise. Des premiers modèles hiérarchique et réseau au modèle relationnel, du mainframe au micro, du centralisé au réparti, les ambitions des SGBD augmentent d'année en année. Mais à quoi peuvent bien ressembler ces logiciels de plusieurs dizaines de milliers de lignes de code qui gèrent avec une relative facilité quelques milliards d'octets d'information ?

I.2. LE PRINCIPE D'UN MODELE DE DONNEES

Dès 1965 apparaît l'idée de distinguer les données de leurs traitements. Cela exige une description préalable des données qui doit être fournie par les utilisateurs.

Pour bien comprendre cela, observons le travail que devait réaliser le programmeur et que le système doit maintenant prendre en charge : le programmeur devait transformer en structures informatiques une certaine vision de la réalité (par exemple une personne est stockée sous la forme d'un enregistrement composé d'un champ "numéro de sécurité sociale", d'un champ "nom", d'un champ "prénom" et d'un champ "date de naissance"). C'est à présent le SGBD qui va effectuer ce travail : il recevra en entrée une *description abstraite* des données. Disposant de règles déclarées par l'administrateur, il choisira la façon de les stocker et de gérer les accès.

Les outils offerts par le SGBD pour décrire les données qu'il aura à stocker repose sur un ensemble de règles et de concepts permettant de décrire le monde réel (en fait une toute petite partie du monde réel : celle sur laquelle portent les traitements). Ces règles et concepts constituent un *modèle de données*. Le modèle entité-association peut ici être donné à titre d'exemple.

I.1. POURQUOI DES BASES DE DONNEES ?

Comme c'est le cas pour de nombreuses innovations technologiques, une importante pression des besoins est à l'origine de l'émergence des Systèmes de Gestion de Bases de Données. Dans l'environnement informatique traditionnel des gros systèmes d'exploitation, le seul mode de gestion de données reste le gestionnaire de fichiers. Les données traitées par une application (gestion de la paie, des stocks, de la comptabilité, des locaux…) demeurent spécifiques à cette application.

L'organisation des données en fichiers telle qu'elle est traditionnellement conçue répond à des besoins précis : les services utilisent des informations le plus souvent distinctes pour des traitements différents. Pour cette raison, chaque équipe de l'entreprise dispose de ses propres fichiers de données où ne figurent que les informations la concernant. La forme sous laquelle cette information est stockée dépend de facteurs nombreux et variés : matériel disponible, bonne volonté et harmonisation des équipes de développement (choix du langage de programmation, choix de la structure de stockage…) ; surtout, les accès à l'information déterminent le plus souvent l'organisation choisie pour les données : le choix des informations regroupées dans un même fichier, le choix de l'organisation du fichier (séquentiel, aléatoire…).

Dans ce contexte, on constate l'apparition de nombreuses difficultés :

Redondance des données

La répétition en différents endroits de données identiques pose de difficiles problèmes lors des mises à jour ; chaque modification doit être répercutée sur toutes les occurrences de l'objet concerné. Par exemple, le changement de nom d'un employé (c'est fréquemment le cas lors des mariages) doit être effectué aussi bien au service du personnel qu'à la comptabilité, etc.

Difficulté d'accès aux données

L'accès aux données nécessite l'intervention d'un informaticien (nécessité de connaître la machine où résident les informations, les structures utilisées pour stocker ces informations, les chemins d'accès disponibles…). La seule manipulation possible d'informations stockées sur fichiers est une manipulation "programmée" ; les informations sont en effet décrites de façon très physique et ne permettent pas un niveau d'abstraction suffisant pour envisager une interrogation directe et interactive par un utilisateur final non spécialiste.

Problèmes de partage des données

Le problème est bien connu des concepteurs de systèmes d'exploitation : comment réagir à plusieurs ouvertures simultanées d'un même fichier. Des problèmes identiques se posent dès que plusieurs utilisateurs désirent effectuer des modifications sur les mêmes données. De surcroît, l'existence de plusieurs occurrences d'une même donnée en plusieurs endroits (plusieurs fichiers différents) provoque une complexité accrue si le système doit garantir la cohérence pour des utilisateurs manipulant la même information au même moment.

Risques d'incohérence

Des liens sémantiques existent très souvent entre les données : le poste d'un employé, figurant dans le fichier du service du personnel, n'est pas sans rapport avec le salaire qui lui est versé par la comptabilité. Pourtant la modification de l'un n'entraîne pas automatiquement celle de l'autre. De telles situations nécessitent souvent force courrier et coups de téléphone ; des incohérences apparaissent trop fréquemment du fait de la difficulté et du coût qu'implique le contrôle de "contraintes d'intégrité" liant les données entre elles.

Problèmes de sécurité

Garantir de la sécurité physique de l'information est une des tâches de base d'un système informatique. Qu'arrive-t-il si une panne interrompt un programme augmentant tous les salaires de 3% ? Combien de modifications ont été prises en compte ; où reprendre le travail ? La présence de la même information en plusieurs exemplaires n'est évidemment pas de nature à faciliter la tâche du SGBD et des programmes de reprise après panne. De même, la confidentialité de l'information est plus facile à assurer avec une seule occurrence de l'information à protéger.

Manque de portabilité des applications

Que se passe-t-il lorsque l'on change de système d'exploitation ou de support de stockage ? Dans la plupart des cas, les applications développées sont si fortement liées aux possibilités propres du système et du gestionnaire de fichiers qu'il est parfois aussi difficile de les actualiser que de les réécrire entièrement. Si la normalisation des langages de programmation est de plus en plus fréquente, les accès aux fichiers dépendent la plupart du temps de la machine ; le manque de séparation entre le code de l'interface et celui du traitement des données interdit de faire une simple modification du code.

Problèmes de maintenance

Enfin, quand la modification de la structure des données manipulées devient nécessaire, il faut assurer le passage des données depuis les anciennes vers les nouvelles structures choisies et presque toujours réécrire complètement les programmes d'application dont la plupart réalisent pourtant la même opération qu'auparavant.

Au total, les organisations traditionnelles des données sous la forme de fichiers liés aux applications présentent de sérieux inconvénients dès qu'un ensemble de programmes manipulent et partagent des informations qui ne sont pas complètement indépendantes. Le défaut de base des organisations traditionnelles

provient du fait que *les données et les traitements ne sont pas explicitement séparés*. L'ancienne vision du monde faisait en effet tourner les données autour des traitements (tout comme le soleil tournait autour de la terre !). Aujourd'hui, on sait que le centre du monde est constitué par l'information manipulée (les données dont elle est constituée) : le centre c'est le système d'information de l'entreprise !

Une bonne solution à ces problèmes passe par le regroupement et l'unicité des données, ainsi que par la centralisation des moyens de gestion de ces données. L'administration unique et centralisée des données à l'échelle de l'entreprise garantit la cohérence de l'information et permet l'*indépendance des programmes et des données* . Elle est mise en œuvre par un Système de Gestion de Bases de Données (SGBD).

L'ensemble de toutes les données peut être regroupé sous l'appellation de *base de données*. Cette base de données représente l'information de l'entreprise. Elle existe indépendamment de l'usage qu'en font les programmes d'application ou les utilisateurs finals. L'information doit donc pouvoir être décrite indépendamment de l'usage qui en est fait. Cette description nécessite un *modèle de données*, c'est à dire un ensemble de concepts et de règles assez généraux et riches de sémantique pour pouvoir donner de façon formelle un *schèma* de la structure permanente de cette information.

Le système logiciel qui met en œuvre le modèle de données (qui stocke les données, gère les accès, les droits des utilisateurs, et plus généralement traite au mieux l'ensemble des problèmes que nous venons d'évoquer) forme véritablement le SGBD.

I.2.1. Le modèle entité-association

Ce modèle de données correspond à une modélisation assez naturelle du monde réel. Le modèle se compose de trois concepts élémentaires : l'entité, l'association et les contraintes portant sur ces associations.

Une entité peut être :

- un objet du monde réel ;
- des données composées ;
- un ensemble d'entités.

Une association peut être :

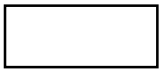
- un lien entre entités ;
- un ensemble d'associations.

Une contrainte peut être :

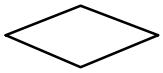
- une propriété précisant le nombre d'entités qu'une association peut unir. On parle alors de cardinalité d'association qui peut être de type 1-1, 1-N, ou N-M ;

- une propriété propre à l'association et sans rapport avec les entités rapprochées par ces associations.

Ce modèle est mis en œuvre par un langage de description. Ce langage est ici graphique :

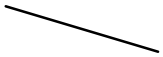


représente un ensemble d'entités (type d'entité)



représente un ensemble d'associations
(type d'association)

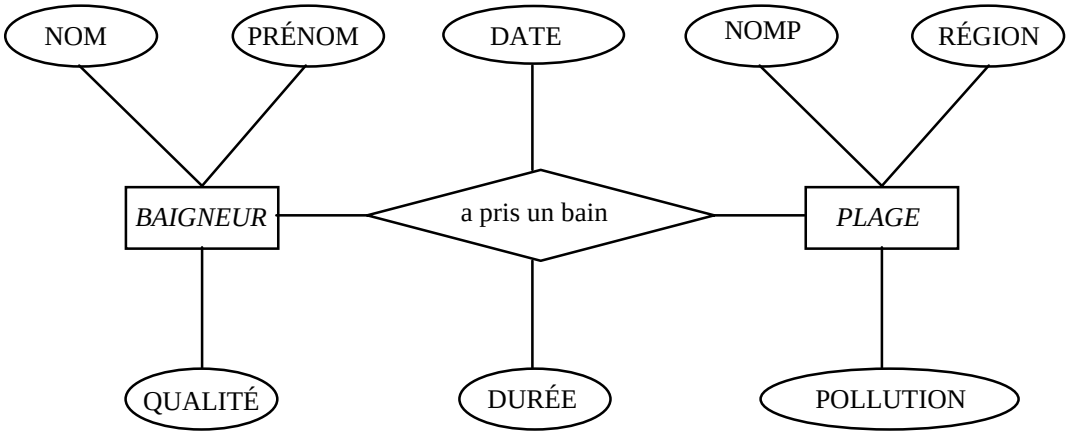
représente un attribut (type d'attribut)



représente une liaison entre attribut et entité
ou entre entité et association

Exprimons avec ce modèle la réalité suivante : des nageurs, dont les caractéristiques sont un nom, un prénom et une qualité, prennent des bains d'une certaine durée à une certaine date, sur certaines plages dont les caractéristiques sont le nom le la plage, la région et la pollution :

Le schéma entité/association est le suivant :



Les contraintes à définir (dont les cardinalités d'association) posent, par exemple, les problèmes suivants :

- un baigneur peut-il prendre plusieurs bains? sur des plages différentes?
- une ou plusieurs personnes peuvent-elles se baigner sur une plage très polluée ?
- autorise-t-on les mauvais nageurs à se baigner sur une plage des Landes ?

I.2.2. Modèles, langages et schémas

Plus généralement, nous dirons que le modèle de données est un mode de formalisation du monde réel. Il doit permettre la description et la représentation :

- des entités et des données qui les constituent ;
- des liens (association, relations, correspondances) qui les relient ;
- de certaines assertions (propriétés ou contraintes d'intégrité) que doivent vérifier les données de la base.

Définition : modèle de données

un modèle de données est l'ensemble des concepts et des règles de composition qui permettent de décrire les données.

Définition : langage de description

un langage de description est un langage supportant un modèle de données et permettant de définir ces données.

Définition : schéma

Un schéma est la description au moyen d'un langage déterminé d'un ensemble particulier de données.

Un Système de Gestion de Bases de Données est donc un système logiciel qui met en œuvre un modèle de données particulier et qui offre des outils de définition et de manipulation de données à des utilisateurs possédant des connaissances plus ou moins approfondies sur le modèle et sur le logiciel : administrateur, développeurs d'applications ou utilisateurs finals non spécialistes.

Les principaux modèles de données sur lesquels s'appuient les SGBD sont le modèle hiérarchique, le modèle réseau, et le modèle relationnel que nous examinerons ultérieurement.

I.3. PRECISIONS SUR LES NIVEAUX DE DESCRIPTION

La description des données peut s'effectuer à plusieurs niveaux d'abstraction qu'on appelle niveaux de description et de schémas.

Le premier niveau concerne la réalité informatique. C'est l'administrateur de la base de données qui effectue ce travail. Les objets décrits sont ici l'environnement matériel et système (organisation et partitionnement du disque, système d'exploitation…), les fichiers (nom, taille, organisation…), les articles (nom, longueur, placement…), les champs (nom, format, localisation…), les chemins d'accès (clé, lien, type…).

Le second permet la description conceptuelle. La réalité au niveau de toute l'entreprise y est décrite. L'administrateur ou l'utilisateur donne *une vue canonique des données* : c'est de ce schéma, unique, que seront dérivées *différentes vues externes*. Les objets décrits restent

ici abstraits : entité (nom, attributs…), association (nom, cardinalité…), contrainte (objet, assertion…), etc.

Enfin le niveau externe permet de décrire plusieurs visions différentes d'une même réalité (le schéma conceptuel); chacune de ces visions correspondra à une application ou à un groupe particulier d'utilisateurs (service du personnel, service de comptabilité, comité d'entreprise…). Le niveau d'abstraction de la description est ici le même que celui du niveau conceptuel.

Des règles de correspondance sont utilisées pour transformer les données d'un niveau de schéma dans un autre. Ces règles sont en général fixées par l'administrateur et permettent de libérer l'utilisateur des contraintes liées à "l'informatique profonde".

Historiquement, c'est en 1971, qu'une recommandation du DBTG-CODASYL (DB Task Group) isolait un niveau interne (spécialisé dans le stockage et l'accès aux informations) et un niveau externe, proche de l'utilisateur, qui gère les rapports avec les programmes d'application. La recommandation de trois niveaux (interne, conceptuel et externe) date de 1979.

Dans cette introduction, nous ne nous préoccupons que peu du niveau interne (voir cependant le chapitre 7 qui présente différentes possibilités d'implantation physique des données dans un SGBD), mais nous pouvons profiter de ce paragraphe pour en dire quelques mots.

Précisons en effet qu'un SGBD gère des milliards d'octets de données et que, pour ce faire, il n'utilise pas les structures de données offertes par les langages de programmation (tels l'article en Pascal ou la structure en C). Il n'utilise pas non plus, en général, les fichiers offerts par les systèmes d'exploitation pour stocker les informations sur disque ; il préfère utiliser ses propres techniques de stockage. Un SGBD réserve une partie du disque et de la mémoire, y manipule des pages mémoire (chargement, déchargement, lecture, écriture) grâce aux primitives de bas niveau offertes par le système d'exploitation. Sur ces pages mémoire, l'information est vue comme une suite d'octets dont l'organisation est connue du système[1] qui sait ainsi lire les informations stockées.

Dans l'avenir nous n'aborderons plus les détails de structure de cette couche interne, mais il faut garder à l'esprit qu'un SGBD, à son plus bas niveau, ne fait que lire ou écrire des pages mémoire. Il a de telles quantités de données à manipuler qu'il doit mettre en œuvre le maximum de techniques pour diminuer le nombre d'entrées/sorties nécessaires. Ces méthodes, que nous nous bornerons à présenter rapidement dans ces chapitres, s'appellent optimisation de requêtes, méthodes de stockage, etc.

---

[1]nous désignons, ici et dans les pages qui suivent, par "système" le SGBD qui, l'utilisant ou non, masque le système de gestion de fichiers - SGF - du sytème d'exploitation - OS.

I.4. LES OBJECTIFS DES SGBD

De façon plus formalisée, voici reprise et complétée la liste des objectifs des SGBD.

I.4.1. Offrir différents niveaux d'abstraction

Niveau physique :

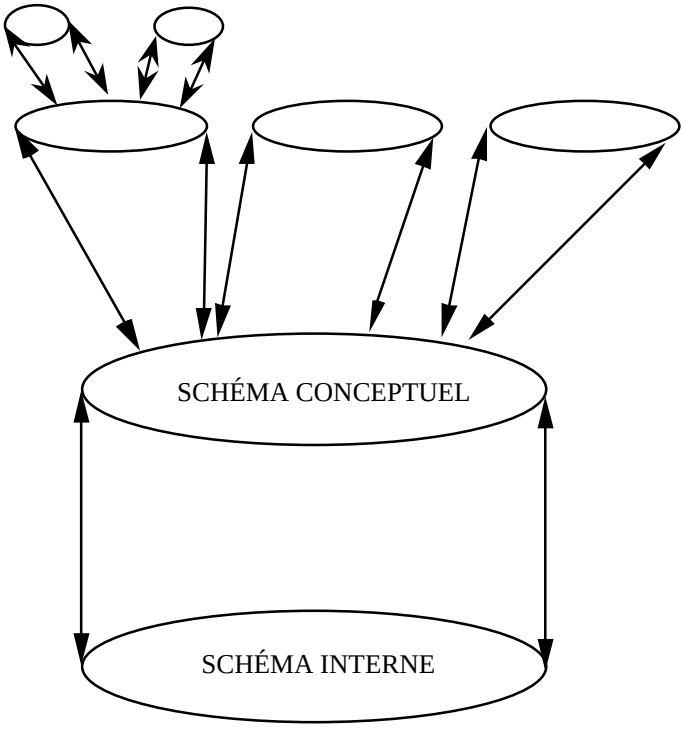
Ce niveau appelé aussi niveau interne, gère le stockage et l'accès aux données. Il n'y a qu'un seul niveau physique par SGBD.

Niveau conceptuel :

C'est à ce niveau, également appelé le niveau logique, que l'on parle de modèle (conceptuel) de données. Ce modèle décrit l'ensemble des données de l'entreprise. La description conceptuelle d'une base de données - BD - est unique.

Niveau vue :

C'est le plus haut niveau d'abstraction de la base de données. Il est aussi appelé niveau externe et est propre à un utilisateur ou à un groupe d'utilisateurs et ne lui présente qu'une vue partielle de la réalité : celle qui intéresse son service. Il y a bien entendu plusieurs vues d'une même BD.



I.4.2. Assurer l'indépendance physique des données

Le but est ici de permettre à l'utilisateur du niveau conceptuel d'ignorer la structure du niveau physique. Cela nécessite une transformation entre niveau logique et physique, mais présente deux avantages considérables :

- les programmes d'applications sont plus simples à écrire, du fait de ne pas avoir à manipuler des entités complexes (structures d'enregistrement, méthodes d'accès…) ;
- dans le cas d'une modification des caractéristiques du niveau physique, les applications n'ont pas à être modifiées.

I.4.3. Assurer l'indépendance logique des données

Le but est ici de permettre à l'utilisateur du niveau vue (typiquement les programmeurs d'application ou les utilisateurs finals) d'ignorer la structure du niveau conceptuel. Cela nécessite une transformation entre niveau externe et conceptuel, mais offre là encore deux avantages de poids :

- les programmes d'applications du niveau externe n'ont pas à avoir la vision globale de toute l'entreprise. Ils agissent à partir des vues ;
- les applications du niveau externe, en cas d'une modification du schéma au niveau conceptuel, ne sont à réécrire que si cette modification entraîne celle de la vue, ce qui est rarement le cas.

I.4.4. Contrôler la redondance des données

Un des objectifs de base des SGBD, la suppression de la redondance des données, vise à garantir la cohérence de l'information et à simplifier les mises à jour. Cependant, la redondance est parfois nécessaire pour garantir la fiabilité et les performances, ou pour la répartition des données.

Les données sont réparties quand plusieurs SGBD situés sur des sites distincts reliés par un réseau partagent les informations dont ils disposent. Posséder certaines informations sur tous les sites est alors parfois indispensable (par exemple la description des données présentes sur les différents sites pour savoir où, comment, et dans quel ordre aller les chercher en cas de besoin).

En conséquence, la redondance anarchique des données doit être éliminée et la redondance existante doit être contrôlée en propageant la mise à jour d'une donnée redondante.

I.4.5. Permettre à tout type d'utilisateur de manipuler des données

Le but est d'offrir aux différents types d'utilisateurs des moyens d'accès à la base adaptés à leurs besoins et à leurs connaissances. Nous devrons ainsi distinguer :

- un ou plusieurs administrateur de la base qui doivent pouvoir décrire les données aux niveaux physique (administrateur BD et ingénieur système) et conceptuel (administrateur BD et concepteur) ;
- un ou plusieurs développeur d'applications qui écrivent, à partir du niveau conceptuel ou des niveaux externes, des programmes d'application pour eux-mêmes ou pour les utilisateurs finals ;
- un ou plusieurs utilisateur final ont besoin d'un langage simple (si possible proche du langage naturel) pour manipuler les données de manière interactive ou à partir de programmes d'application.

I.4.6. Assurer l'intégrité des données

L'intégrité logique de l'information est souvent vérifiée par les programmes d'applications dans les organisations traditionnelles à bases de fichiers. Dans une approche base de données, elle fait partie de la description de la réalité conceptuelle du système d'information. La vérification de l'intégrité est une composante du modèle de données et une tâche du SGBD qui le met en œuvre. L'intégrité sémantique correspond à des règles explicitant des contraintes du monde réel.
Nous pouvons donner comme exemple :

- il est interdit et impossible de se baigner sur une plage très polluée ;
- un baigneur ne peut pas se baigner sur une plage non recensée ;
- un baigneur ne peut pas se baigner à Binic en mars.

Toute requête de mise à jour des données (insertion, modification ou suppression) ne respectant pas l'ensemble des contraintes d'intégrité doit être rejetée par le SGBD.

I.4.7. Assurer le partage des données

Composant transactionnel d'un système informatique, un SGBD doit permettre le partage des données entre différents utilisateurs et applications. Un utilisateur n'a pas à se demander si quelqu'un d'autre travaille sur les mêmes informations au même moment et doit pouvoir accéder aux données en consultation ou en mise à jour comme s'il était seul : le système doit gérer les conflits, en refusant ou en retardant éventuellement un ou plusieurs accès.

I.4.8. Assurer la sécurité des données

Cela consiste à protéger les données contre les pannes et à refuser les accès aux personnes non autorisées. Le système doit présenter un mécanisme de vérification des droits d'accès aux objets de la base. Il doit garantir des reprises après panne en restaurant la base de données dans le dernier état cohérent avant la panne. La fiabilité est traditionnellement sur les gros systèmes mise en œuvre des techniques très sophistiquées. La qualité de

l'implantation de ces techniques a d'importantes conséquences sur les performances en cas d'utilisation intensive.

I.4.9. Optimiser l'accès aux données

En permettant aux utilisateurs d'ignorer les structures physiques et les chemins d'accès à l'information, le SGBD prend à sa charge un lourd travail d'optimisation. En utilisant les meilleurs chemins d'accès, mais aussi le parallélisme ou des algorithmes de recherche sophistiqués, il permettra de minimiser le volume des données accédées et le temps d'exécution des questions.

I.5. CONCLUSIONS

Cette introduction nous a permis de présenter les motivations et les buts des Systèmes de Gestion de Bases de Données. Placés ici sur un pied d'égalité, les différents objectifs sont atteints à des degrés divers : les aspects transactionnels (concurrence, sécurité physique,…) sont bien implantés et assurent aujourd'hui une efficacité très importante aux systèmes utilisés en production et destinés à écouler un nombre important de transactions par seconde ; les indépendances physique - conceptuel / interne - et logique - externe / conceptuel - sont bien assurées par les systèmes relationnels, mais mal ou pas du tout dans les produits s'appuyant sur les modèles réseau ou hiérarchique ; la détermination par les systèmes des meilleurs algorithmes et chemins d'accès aux données ont fait faire des progrès considérables aux méthodes d'accès des systèmes informatiques modernes ; la vérification automatique des contraintes d'intégrité générales est, en revanche, encore peu implantée dans les produits.

I.6. REFERENCES

[ANSI 75] ANSI/X3/SPARC Study Group On Data Management Systems *"Interim Report"* ACM SIGMOD bulletin 7, N°2, 1975.

[Chen 76] CHEN P. P. : *"The Entity-Relationship Model-Toward a Unified View of Data"*, ACM Transactions on Data Base Systems, V1, N°11, March 1976.

CHAPITRE 2 : MODELES DE BASES DE DONNEES ET GENERATIONS DE SGBD

Dans le premier chapitre, nous avons précisé les caractéristiques que devait présenter un modèle de données afin de décrire - "modéliser" - correctement le **monde réel** : permettre la représentation et la description des entités et de leurs données constitutives, des liens qui les relient et de certaines assertions appelées contraintes d'intégrité. Ces composants d'un modèle *conceptuel* de données doivent autoriser une description de la réalité qui soit le plus indépendante possible des capacités de définition *logique* et de stockage *physique* des données par le **système logiciel** appelé SGBD .

Il faut donc distinguer entre modèles conceptuels, a priori indépendants des SGBD, et modèles logiques, caractéristiques d'une génération de SGBD. Ces modèles logiques sont eux-mêmes dépendants d'organisations physiques (placement et méthodes d'accès) comme, par exemple, les graphes en arbres ou en réseaux, seules structures offertes par la première génération de SGBD, celle des **modèles hiérarchiques et réseau**.

Ces SGBD obligent à "voir" - c'est à dire à traduire - une base de données comme un ensemble d'enregistrements, reliés les uns aux autres par des ensembles de pointeurs. Cette organisation fige les relations existant entre les différents enregistrements de la base et impose un type de manipulation, la *navigation*, qui consiste à suivre, d'enregistrement en enregistrement, les chaînes de pointeurs. En fait, il serait préférable de parler de *modèles d'accès* (physiques) plutôt que de modèles de données (logiques) pour ces anciens SGBD. En effet l'organisation physique des données impose la méthode d'accès et, au delà, la nature des langages de manipulation.

Parmi ces SGBD, encore largement utilisés, on peut citer IMS, produit hiérarchique de IBM, très répandu dans les applications de production, et, pour les produits "réseau", IDS2 chez BULL, IDMS de CULLINET, TOTAL de CINCOM, SOCRATE . Ces derniers ont le plus souvent appliqué les recommandations du "comité CODASYL" de l' ANSI publiées dans la décennie 70.

II.1. LE MODELE HIERARCHIQUE

Longtemps considéré comme le seul modèle permettant aux SGBD d'atteindre les performances exigées en production, le modèle hiérarchique possède effectivement la capacité de traiter rapidement des informations organisées sous la forme d'une hiérarchie stricte et interdit tout autre type de représentation, limitant ainsi considérablement la puissance d'expression du modèle.

II.1.1. Les objets du modèle

Les concepts de base du modèle sont le *champ*, plus petite unité de données possédant un nom, et l'*article* [2]

Définition : article

suite de champs, portant un nom et constituant l'unité d'échange entre la base de données et les applications. Les articles sont reliés entre eux par des liens hiérarchiques : à un article père correspondent N articles fils.

La notion de *type* d'article qui désigne le schéma d'un article (description du contenant) se distingue ici de celle d'*occurrences* d'article qui représentent les différentes valeurs stockées de la base.

Définition : arbre d'articles

collection d'articles reliés par des associations père-enfants organisées sous forme d'une hiérarchie.

Une base de données hiérarchique est une base constituée d'une forêt d'articles.

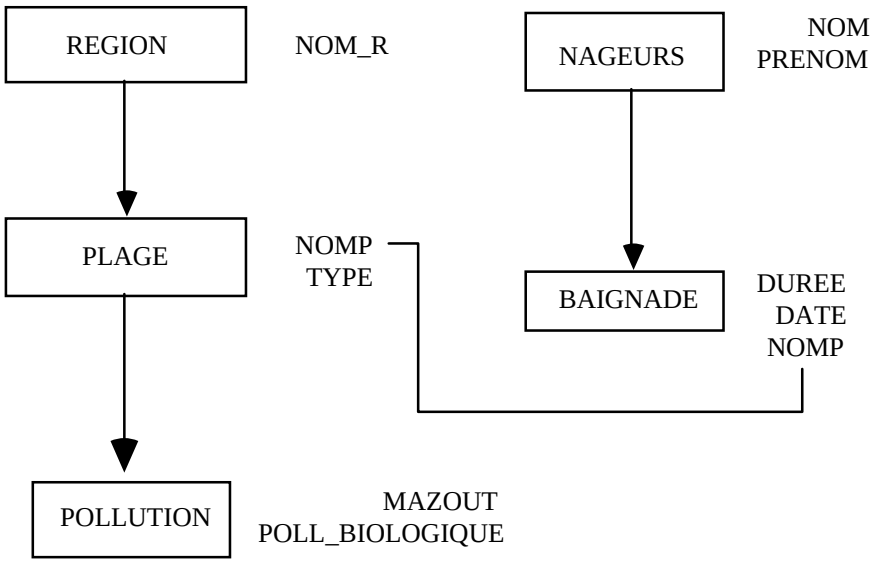
On peut formuler quelques remarques sur la structure hiérarchique :

- il y a un seul type article racine ;
- la racine peut avoir un nombre quelconque de types d'article enfant ;
- chaque type d'article enfant de la racine peut avoir un nombre quelconque de types d'article enfant, et ainsi de suite ;
- à une occurrence d'un type d'article donné, peuvent correspondre 0, 1 ou N occurrences de chaque type d'article enfant ;
- une occurrence d'article enfant ne peut exister sans l'occurrence d'un article père. Détruire une occurrence d'article père, détruit par conséquent également toutes les occurrences de ses enfants.

---

[2] la littérature de l'époque employait plutôt le terme de "segments" de données, nous l'avons remplacé ici par "article" pour simplifier la compréhension et la comparaison avec les modèles ultérieurs.

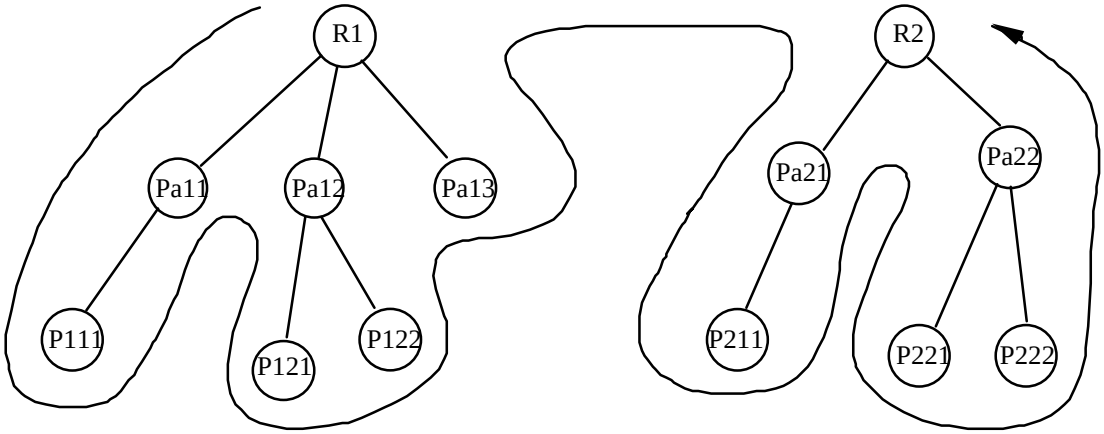
Exemple : représentation hiérarchique d'une base de baigneurs



Dans cet exemple, POLLUTION représente différentes mesures de pollution. Pour exprimer qu'une baignade a eu lieu sur une plage et a été le fait d'un nageur, il a fallu dupliquer l'attribut "nom de plage" (NOMP). En fait, dans certains systèmes hiérarchiques, cette duplication est simplement logique (pas de duplication physique) : la valeur de NOMP n'est présente que dans une des deux hiérarchies, l'autre comportant un simple pointeur sur la valeur.

II.1.2. Langage de manipulation de données

Le principe de la manipulation d'une base hiérarchique (nous prenons ici l'exemple de DL1 de IBM) consiste à parcourir en profondeur d'une structure arborescente : on part de la racine et on visite successivement tous les fils depuis le fils gauche jusqu'au droit.
Dans notre exemple, les Ri sont des occurrences de régions, les Paii des occurrences de plage, les Piii des occurrences de mesures de pollution.



DL1 utilise des curseurs permettant de mémoriser différentes positions sur la base de données. A un accès en recherche, est associée une *qualification de chemin* constituée de différentes *qualifications d'articles*, connectées par des ET ou des OU logiques, et formant des critères élémentaires < nom_de_champ comparateur valeur>. Ces qualifications sont déclarées préalablement et invoquées ensuite par les programmes qui utilisent des opérations GET UNIQUE (GU, permet de rechercher la première occurrence satisfaisant la qualification), des opérations GET NEXT (GN, recherche l'occurrence suivante), des GET NEXT WITHIN PARENT (GNP, idem GN, mais uniquement parmi des descendants du parent courant), etc.

II.1.3. Actualité du modèle hierarchique

Si l'on considère la représentation traditionnelle des occurences d'articles sous forme de "fiches", on s'aperçoit que toute solution au problème du placement physique d'un ensemble de fiches de différents types est un partitionnement de cet ensemble suivant une hiérarchie choisie sur ces types. Un exemple intéressant est celui des fiches de type "Management Information Base" (MIB) associées à chaque noeud d'un réseau gérable par le "Simple Network Management Protocol" (SNMP). Le langage associé a de nombreux points de ressemblance avec les langages de parcours des bases de données hierarchiques tels que DL1.

II.2. LE MODELE RESEAU

Ce modèle[3] a été introduit en 1961 par BACHMAN. Il propose une utilisation de structures de listes afin de relier sémantiquement des entités. Il permet ainsi une meilleure représentation de la réalité que le modèle hierarchique. Un des intérêts du modèle fut l'existence d'une proposition de normalisation émise par le groupe DBTG (Data Base Task Group) du Comité CODASYL (COnference on DAta SYstem Language). On parle ainsi souvent de modèle CODASYL pour les systèmes réseaux qui suivent ces recommandations. Deux niveaux de recommandations ont été émis : l'un, en 1971, conseillait la séparation d'un niveau de schéma externe et d'un niveau de schéma interne/conceptuel ; le second, en 1978, préconisait les trois niveaux de schéma - interne, conceptuel, externe-. Seul le premier niveau de recommandations a été réellement suivi par les produits. En 1978, il était déjà tard pour voir les produits prendre en compte ces recommandations : l'heure du relationnel avait déjà sonné pour les investissements à long terme ; quasiment aucun SGBD réseau réellement nouveau n'a été conçu depuis cette date. Mais à l'heure où les SGBD relationnels sont à leur tour défiés par les nouveaux SGBD orientés objets, il est encore

---

[3] son nom ne caractérise que la structure commune aux graphes de placement physique des données dans les SGBD de cette génération, et n'implique nullement une distribution des données sur un réseau informatique!

intéressant de comprendre ce que les SGBD réseau avaient apporté de plus novateur : l'expression des requêtes sur les données dans une *logique de chemin* (c.à.d. de *navigation*).

II.2.1. Description des objets et des associations

Les objets modélisés dans la base de données sont déclarés comme des *articles*.

Définition : article

- un *champ* est la plus petite unité de données possédant un nom;
- un *agrégat* est une collection de champs rangés consécutivement et possédant un nom ;
- un *article* est une collection d'agrégats et de champs rangés côte à côte dans la base; il constitue l'unité d'échange entre la base et les applications.

Définition : ensemble

- l'association entre un article appelé propriétaire et N articles membres de même type constitue un ensemble (SET).

Le type ensemble ainsi constitué porte un nom ; un type ensemble permet d'associer un type d'article propriétaire à un type d'article membre.

Il ne reste plus qu'à définir une base de données réseau :

Définition : base de données réseau

Une base de données réseau est composée d'articles reliés entre eux par des ensembles.

La description de la structure des données d'une base réseau utilise un langage de description du schéma des données. Cette déclaration constitue la création de la structure vide de la base, en décrivant les articles et les associations entre articles formant des ensembles.

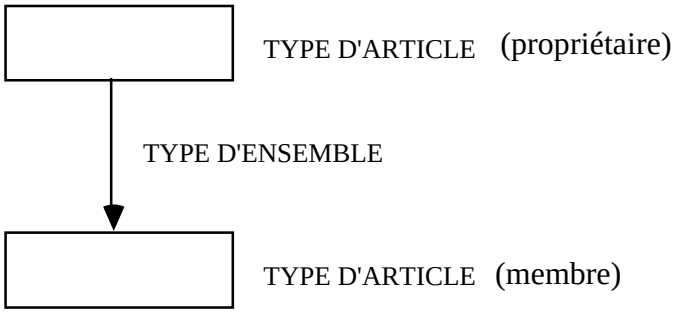
Le schéma d'une base de données réseau se représente par un *graphe des types*, parfois appelé *diagramme de BACHMAN*.

Définition : graphe des types

une base de données réseau se représente à l'aide d'un graphe des types où :

- les sommets représentent les types d'articles ;
- les arcs représentent les types d'ensembles.

Chaque sommet est valué par le nom du type d'article associé ; chaque arc est valué par le nom du type d'ensemble qu'il représente ; chaque arc est orienté du type propriétaire vers le type membre.

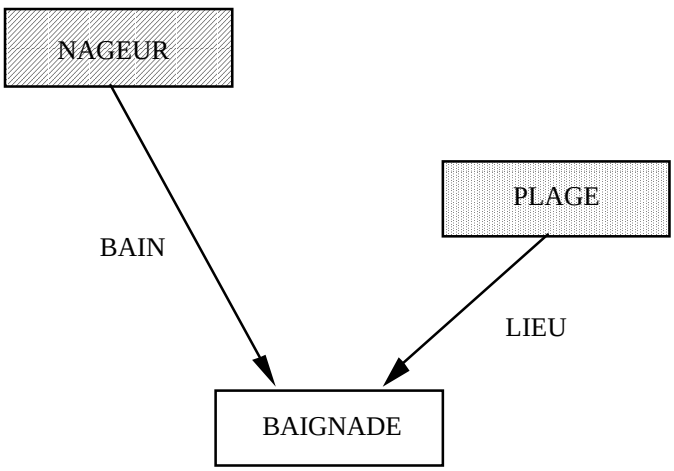

TYPE D'ARTICLE (propriétaire)
TYPE D'ENSEMBLE
TYPE D'ARTICLE (membre)

Exemple :

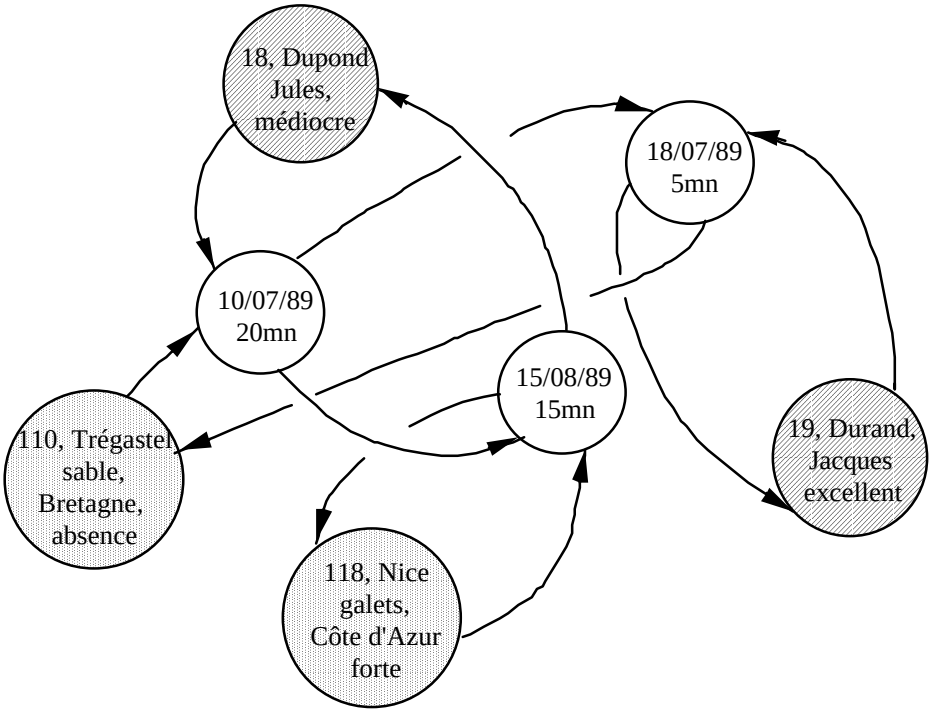
déclaration d'un schéma réseau d'une base de baigneurs, dans le langage de définition de données CODASYL

```
RECORD NAME IS NAGEUR
    02 NN TYPE IS FIXED BIN 3
    02 NOM TYPE IS CHAR 15
    02 PRENOM TYPE IS CHAR 15
    02 QUALITE TYPE IS CHAR 10

RECORD NAME IS PLAGE
    02 NP TYPE IS FIXED BIN 3
    02 NOMP TYPE IS CHAR 20
    02 TYPE TYPE IS CHAR 10
    02 REGION TYPE IS CHAR 20
    02 POLLUTION TYPE IS CHAR 10

RECORD NAME IS BAIGNADE
    02 DATE TYPE IS CHAR 6
    02 DURÉE TYPE IS DEC 3

SET NAME IS BAIN
    OWNER IS NAGEUR
    MEMBER IS BAIGNADE

SET NAME IS LIEU
    OWNER IS PLAGE
    MEMBER IS BAIGNADE
```

Le graphe des types correspondant à cette déclaration est le suivant :



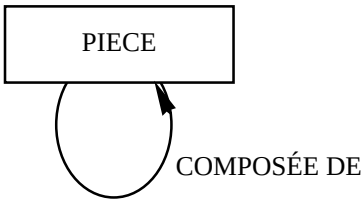
Pour bien voir la notion d'ensemble, on peut montrer une représentation du graphe au niveau des occurrences : ici, nous avons représenté deux occurrences d'ensemble de type BAIN (Dupond qui prend deux bains de 20 et 15 minutes, et Durand qui prend un bain de 5 minutes), deux occurrences d'ensemble de type LIEU (à Trégastel sont pris les bains de 20 minutes et de 5 minutes, à Nice est pris le bain de 15 minutes). Un ensemble de pointeurs existe pour chaque occurrence d'ensemble et lie les membres à leur propriétaire.



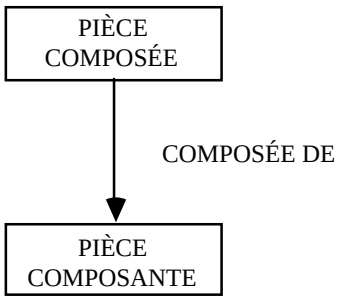
Cette approche impose deux limitations : (i) un type d'article ne peut être à la fois propriétaire et membre dans un même ensemble ; (ii) une occurrence d'article ne peut appartenir à plusieurs occurrences du même ensemble (pour des raisons physiques).

Remarque 1:

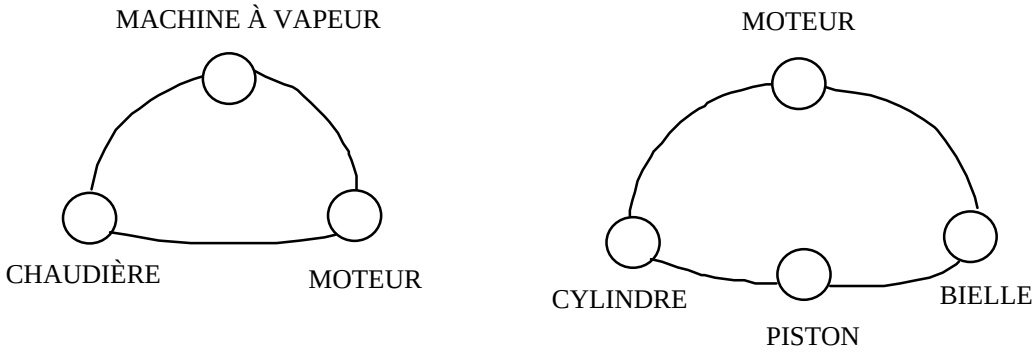
Une association ne peut relier que deux types d'articles différents. La configuration suivante est, par exemple, impossible :



La solution est de créer deux types d'article : un type "pièce composante", un type "pièce composée »



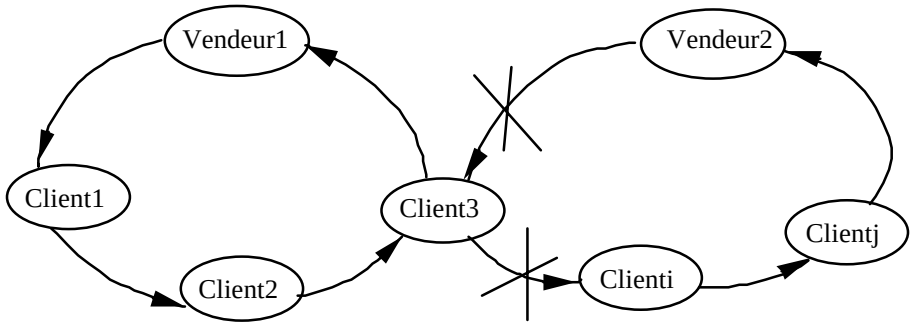
Cependant, dans certains cas, une pièce est à la fois pièce composée et pièce composante ; par exemple :



Faire figurer le moteur en pièce composée et en pièce composante constitue une assez mauvaise solution : cela créer une redondance.

Remarque 2 :

Le modèle impose d'autre part qu'une occurrence d'article n'appartienne qu'à une seule occurrence d'un même type d'ensemble. Ainsi la configuration suivante est impossible :



En effet chaque article est déclaré appartenir à un type d'ensemble (comme propriétaire ou comme membre) ; pour ce type d'ensemble, l'article se voit adjoindre un ensemble de pointeurs lui permettant de se positionner dans ce type d'ensemble. Il ne peut donc appartenir

qu'à une seule occurrence de ce type d'ensemble car un seul jeu de pointeurs est prévu par type d'ensemble d'appartenance.

II.2.2. Types d'informations d'un schéma CODASYL

Comme nous l'avons dit en introduction, le modèle réseau ne sépare pas clairement la description conceptuelle et la description interne du schéma. En fait, en plus de la structure déjà décrite par les clauses RECORD NAME, SET NAME, OWNER, MEMBER, le schéma CODASYL contraint l'administrateur à préciser l'organisation physique - fichiers AREAS - et la description des chemins d'accès - clés, références - qui seront utilisés.
Les modes de placement d'une base CODASYL sont assez particuliers et ne seront pas précisés ici Le chapitre 7 du polycopié présente des méthodes de placement générales et des développements sur le hachage.

II.2.3. Schéma et sous-schéma CODASYL

La déclaration d'un schéma, qui crée la structure vide, est donc préalable à toute action et manipulation d'articles de la base de données. Très statique, cette caractéristique du modèle impose que toute modification du schéma ne s'effectue que sur une base vide : ainsi une évolution du schéma entraîne un déchargement , puis un rechargement complet de la base.
On s'affranchit cependant en partie de cette dépendance en déclarant un sous-schéma qui, en CODASYL, correspond à la notion de schéma externe. Un sous-schéma est le sous-ensemble du schéma vu par un programme d'application. Le sous-schéma permet de redéfinir l'ordre, les noms, les caractéristiques des articles. Recomposer une vision différente de la structure de la base n'est cependant pas possible (comme ce sera possible en relationnel, voir le chapitre 6).

Ordonnancement des articles dans un ensemble

La déclaration de chaque ensemble, précise l'ordre dans lequel les articles membres insérés dans une occurrence d'ensemble. Ainsi la clause

ORDER IS PERMANENT
INSERTION IS {FIRST | LAST | PRIOR | SORTED BY DEFINED KEY}
[KEY IS nom-de-donnée]

accompagne la déclaration d'un type d'ensemble, pour préciser la position d'insertion des occurrences d'articles membres dans une occurrence d'ensemble de ce type. L'insertion s'effectue en première position (FIRST), en dernière position (LAST) ou par rapport au pointeur courant (PRIOR ou NEXT). Dans le cas où l'on désire qu'un ordre soit maintenu sur l'ensemble (SORTED BY DEFINED KEY), on ajoute la clause KEY IS nom-de-donnée.

Point d'entrée dans une occurrence d'ensemble

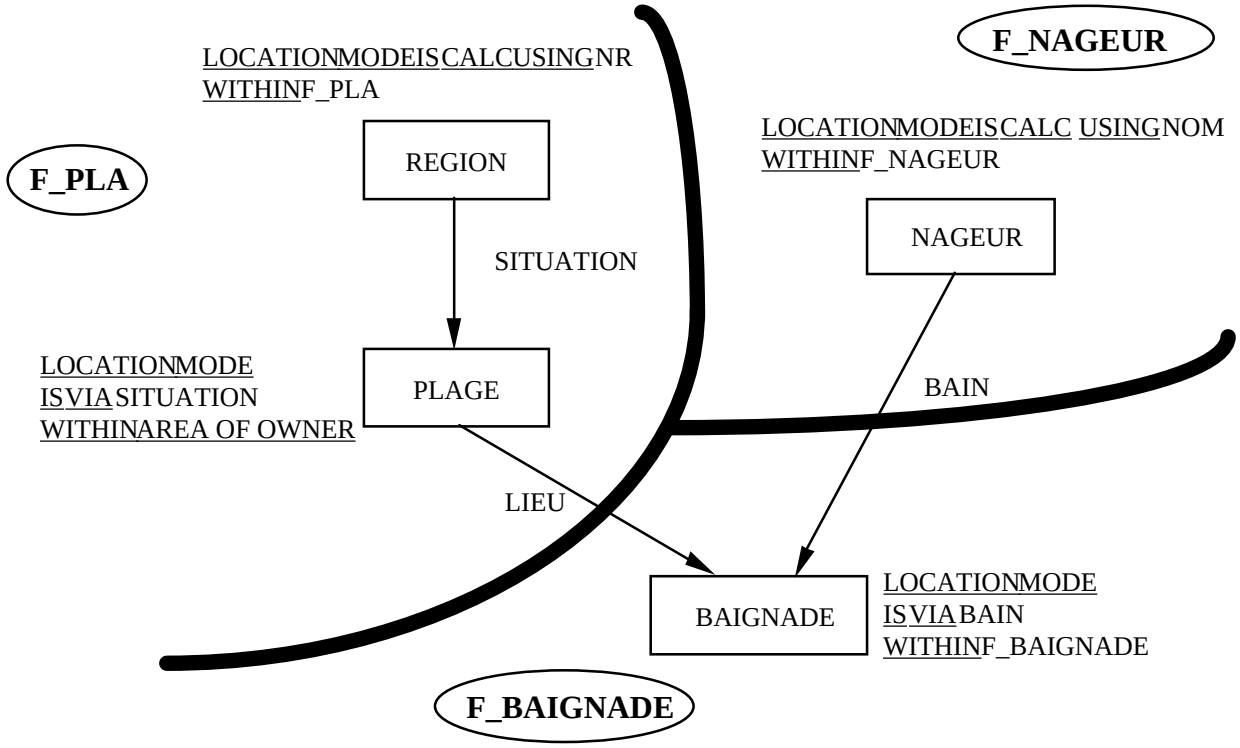
Pour chaque type d'ensemble, on précise dans le schéma comment seront accédés les articles de l'ensemble. Il existe deux possibilités :

- en fournissant la DBK du propriétaire de l'occurrence d'ensemble ; la clé sera alors un paramètre des programmes qui effectueront les accès. La clause du schéma est :
  SET SELECTION IS THRU nom-d'ensemble OWNER IDENTIFIED BY CALC KEY
- par application, ce qui signifie que le propriétaire pur un ensemble aura été repéré préalablement à une intervention sur un des membres. Dans ce cas, les accès aux membres seront toujours effectués par la navigation (voir § II.2.4) exprimée dans le programme d'application. La clause est dans ce cas :
  SET SELECTION IS TRHU nom-d'ensemble OWNER IDENTIFIED BY APPLICATION

Exemple de schéma de placement

On donne ici un exemple de déclaration du schéma de placement d'une base de baigneurs à quatre types d'articles (REGION, PLAGE, BAIGNADE, NAGEUR) et trois types d'ensembles (LOCALISATION, BAIN, LIEU).

Plutôt que de donner une écriture déclarative du schéma, nous illustrons l'organisation dans trois fichiers différents : F_PLA, F_NAGEUR et F_BAIGNADE. Les articles de type RÉGION sont placés par hachage sur un numéro de région (NR) dans un fichier F_PLA (la clause LOCATION MODE IS CALC USING NR WITHIN F_PLA est explicitée lors de la déclaration du type RÉGION). Les articles de type PLAGE sont placés par proximité dans ce même fichier (LOCATION MODE IS VIA LOCALISATION WITHIN AREA OF OWNER). Les articles de type NAGEUR sont stockés dans un fichier NAGEUR par un hachage sur le nom de nageur NOM (clause LOCATION MODE IS CALC USING NOM WITHIN NAGEUR associée à la déclaration du type NAGEUR). Enfin, les articles de type BAIGNADE sont placés dans un fichier F_BAIGNADE qui leur est propre, mais avec une organisation par homothétie qui les range en fonction des valeurs des nageurs ayant effectué ces baignades (clause LOCATION MODE IS VIA BAIN WITHIN F_BAIGNADE). Cette organisation favorise évidemment les questions qui évoquent, par exemple, les baignades de « Dupond», au détriment des questions recherchant les baignades ayant eu lieu sur la plage de «Quiberon».



II.2.4. La navigation CODASYL

La nature du langage de manipulation de données dans un système réseau est largement déterminée par la structure physique des informations. Pour accéder à une information, il faut préciser le chemin utilisé pour atteindre cette information. Le langage indique un chemin de pointeurs à suivre. On parle ainsi de *langage navigationnel*, l'idée étant de naviguer dans la base de données par une succession de déplacements. A un instant donné, on occupe une position précise dans la base.

Ce type de manipulation impose un accès par programmes. Une utilisation interactive n'est guère envisageable avec un SGBD réseau : elle s'avérerait en effet beaucoup trop lourde. Ce n'est qu'avec l'apparition de modèle relationnel que les langages de manipulation de données deviendront réellement interactifs.

Une fois un schéma et un sous-schéma définis, les programmes d'application peuvent invoquer le SGBD à l'aide de verbes-clés du langage de manipulation. Ces verbes sont en général inclus dans un programme COBOL (ou PL1, Pascal, C…). Il sont de quatre types :

- la recherche d'articles : FIND ;
- les échanges d'articles entre programmes d'application et SGBD : GET et STORE ;
- les mises à jour : ERASE, MODIFY, CONNECT, DISCONNECT ;
- le contrôle : READY, FINISH.

La recherche d'article (FIND) qui consiste à positionner le curseur courant sur un article particulier, se distingue des échanges (GET et STORE) qui utilisent un tampon pour charger ou récupérer un article. La notion de curseur est donc très importante dans ce modèle.

Définition : curseur

pointeur courant contenant la clé-base-de-données (DBK) du dernier article manipulé dans une collection d'articles, qui permet à un programme de se déplacer dans la base.

Afin de faciliter les déplacements, un programme qui s'exécute dispose de plusieurs curseurs (en fait de plusieurs positions courantes) dans des ensembles différents:

- un curseur dans chaque type de fichier référencé ;
- un curseur dans chaque type d'ensemble référencé ;
- un curseur dans chaque type d'article référencé ;
- un curseur courant sur le dernier article lu.

Un FIND particulier précise la position des curseurs. Cependant les échanges d'articles s'effectuent uniquement sur l'article pointé par le curseur courant.

L'accès à un fichier (positionnement du curseur de fichier) s'obtient par la clause :

FIND { FIRST | LAST | NEXT | PRIOR | I ème } nom-d'article WITHIN nom-de-fichier

De la même façon, l'accès par un ensemble sera demandé par :

FIND { FIRST | LAST | NEXT | PRIOR | I ème } nom--d'article WITHIN nom-d'ensemble

Se positionner sur le propriétaire de l'occurrence d'ensemble courante est également possible :

FIND OWNER WITHIN nom-d'ensemble

Enfin, une recherche sur clé est possible en connaissant la Clé-Base-de-Données :

FIND nom-d'article DB-KEY IS paramètre

Exemple

Quelles sont les plages où se sont baignés d'excellents nageurs pendant le mois d'août 89 ?

```
READY F_baignade, F_nageur
FIND FIRST nageur WITHIN F_nageur
PERFORM UNTIL fin-de-fichier
    GET nageur
    IF qualité IN nageur = 'excellent'
        FIND FIRST baignade WITHIN bain
        PERFORM UNTIL fin-de l'ensemble
                GET baignade
                IF date BETWEEN 01/01/89 AND 31/08/89
                        FIND OWNER WITHIN lieu
                        GET plage
                        PRINT NOMP
                END_IF
                FIND NEXT baignade
        END_PERFORM
    END_IF
    FIND NEXT nageur WITHIN F_nageur
END_PERFORM
FINISH F_plage, F_baignade, F_nageur.
```

Remarquez que différentes méthodes permettent de répondre à la question posée ; on aurait très bien pu rentrer dans le fichier F_plage par les plages, et aller voir, pour chacune des baignades de chacune de ces plages, s'il ne s'agissait pas d'un excellent nageur.

La formulation de la question n'est pas plus compliquée ; le lecteur réfléchira cependant au coût potentiel de cette seconde exécution… Il s'agit là d'un problème inhérent aux langages navigationnels, où toute la responsabilité des performances pèse sur le programmeur. Dans les systèmes relationnels, cette responsabilité sera, nous le verrons, confiée au système.

II.3. LES MODELES DE CONCEPTION INDEPENDANTS DES SGBD

Le modèle "Entité-Association" ("Entity-Relationship" ou ER), présenté au Chapitre 1, est contemporain des premiers SGBD décrits ci-dessus. C'est un modèle de conception [4], généralement utilisé en mode graphique, indépendant des possibilités (logiques et physiques) des SGBD. Pour cette raison il reste très populaire, d'autant que les schémas de données qu'il permet de construire peuvent être traduits presque automatiquement dans les

[4] qualifié aussi de "sémantique"...

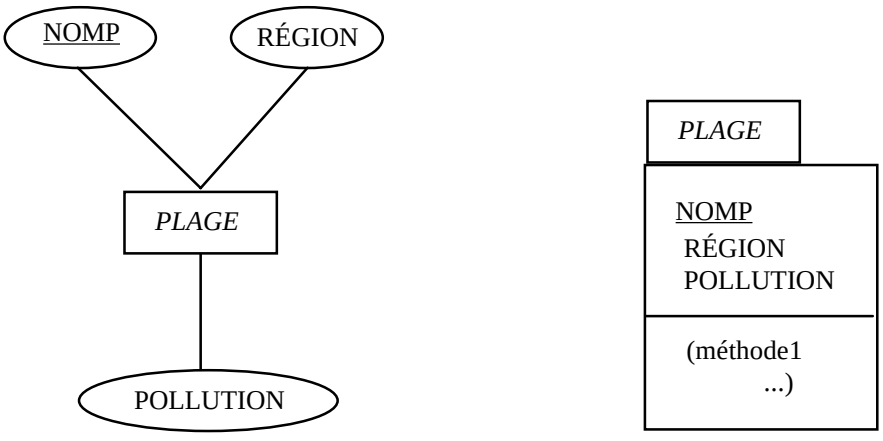
modèles logiques des SGBD des différentes générations : hiérarchique, réseau, relationnel et même orienté objet.

Il a été au fil du temps "enrichi" (Enhanced => EER) ou "complété" (ERC). Les nouveaux modèles de conception orientés objet (OO) ont toutefois vocation à le remplacer, surtout depuis leur récente[5] unification avec l' "Unified Modeling Language" (UML).

Le foisonnement des variantes de l'ER et des modèles OO (OMT, Booch, OOSE...) ont entrainé une certaine confusion chez ceux-là mêmes qui appréciaient la clarté des schémas graphiques produits par l'ER ou ses successeurs. Aussi tenterons-nous, en quelques courts paragraphes, de montrer que pour l'essentiel (les concepts) ils présentent, au delà des différences de notation, une progression continue et sans perte des acquis principaux.

II.3.1. Entités et individus ou classes d'objets

L'*entité* , comme ensemble d'*individus* de même type, est un concept auquel on peut facilement substituer celui de *classe* d'*objets* ayant les mêmes types (ou "structures") de données (et les mêmes "méthodes" pour les manipuler). Les individus " instancient" les entités comme les objets instancient les classes. Les individus comme les objets doivent avoir un nom commun (celui de l'entité ou classe) et *un identifiant unique et invariant* de leur naissance à leur mort, que le concept physique de clef sur un article ne suggère qu'imparfaitement (on peut changer une clef pas un identifiant). Ayant le même type ils sont de surcroît décrits par les mêmes variables nommées : *caractéristiques* ou *attributs* (synonymes). Ces variables peuvent être de structures simples ou complexes : n-uplets (variables composites par ex. une Adresse) ou ensembles (variables multivaluées par ex. un Prénom). Les formes graphiques représentant entités ou classes sont différentes mais traduisibles l'une dans l'autre comme illustré ci-dessous par l'exemple de l'entité PLAGE :



N.B. l'attribut souligné est un identifiant externe, il appartient au SGBD de créer le véritable identifiant invariant (appelé pour cette raison identifiant interne ou "surrogate")

[5] l'Object Management Group (OMG) en a accepté en 1997 la version 1.0

II.3.2. Liens, rôles et cardinalités

Entre les objets (ou les entités) peuvent exister des liens nommés porteurs d'une sémantique commune : le rôle. Ils peuvent lier des objets de classes différentes (comme BAIGNEUR et PLAGE, pour exprimer le rôle AIMER ou le rôle SE_BAIGNER ou un autre rôle) ou de même classe (comme BAIGNEUR, pour exprimer un rôle comme PARENT_DE ou comme ENSEIGNE_A). Dans le modèle ER ces liens sont représentés par des associations (il est interdit de relier directement des entités entre elles), même si elles n'ont pas d'attributs propres. Dans le modèle UML, si aucun attribut n'est associable au lien, celui-ci se représente par une simple arc entre les classes qu'il lie, sinon une classe association est ajoutée. Exemple :

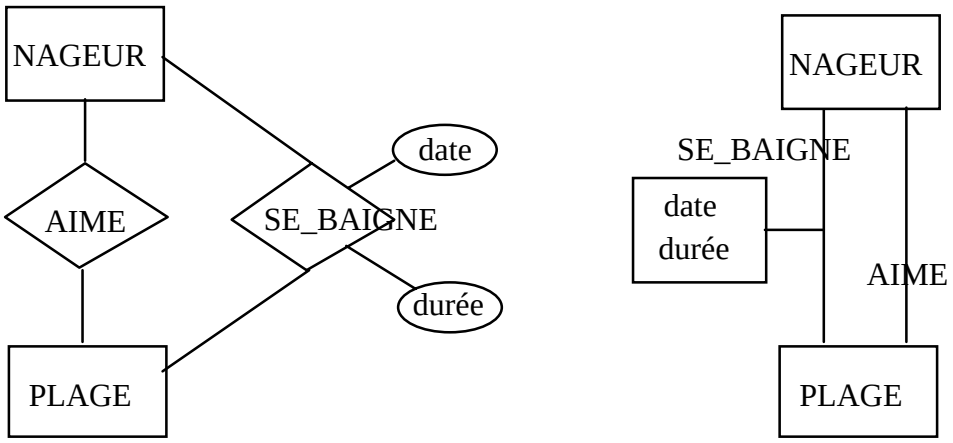
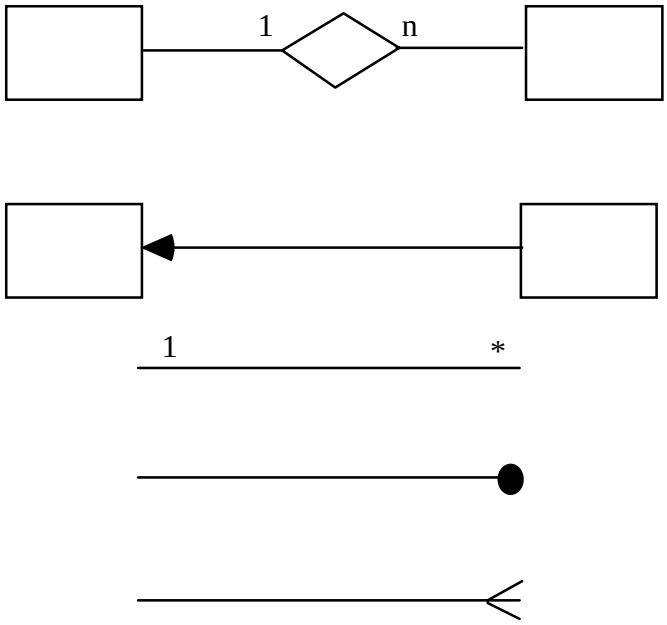
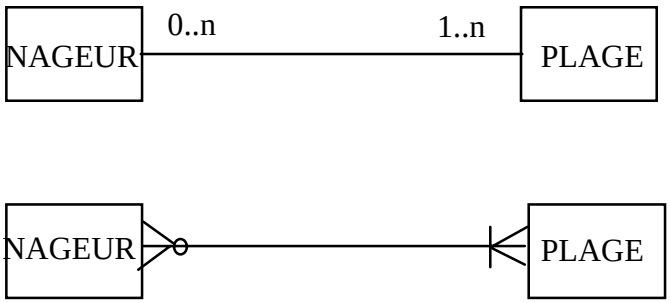


Schéma ER Schéma UML

Les cardinalités de ces liens ou associations spécifient, pour chaque extrémité d'un lien, c.à.d. pour chaque classe liée, le nombre d'objets pouvant être reliés à un même objet de la classe à l'autre extrémité. Les différentes notations ci-dessous sont équivalentes :

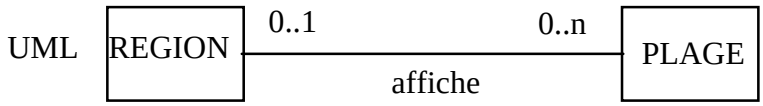


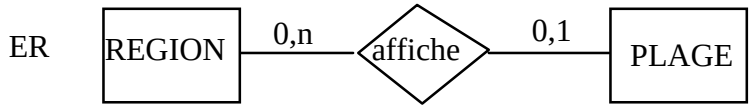
On peut être plus précis en spécifiant des cardinalités minimum et maximum, c.à.d. au lieu de 1 l'une des paires (0,1) ou (1,1), au lieu de n l'une des paires (0,n) ou (1,n), on peut aussi afficher cette valeur n maximale, par exemple (0,3) ou (1,10)... Les dernières notations de la figure précédente pour l'association - de plusieurs à plusieurs, "n à n" - NAGEUR-aime-PLAGE pourraient être utilisées comme ci-dessous :



(dans la 2e notation le rond est un zéro et la barre verticale un 1)

Attention: dans le modèle ER classique, quand les cardinalités (min,max) sont indiquées, elles le sont généralement à l'inverse de toutes les notations précédentes. En effet les liens directs entre entités étant impossibles et l'association étant toujours considérée comme une classe intermédiaire instanciable, cette inversion s'explique pourvu que l'on passe bien par toutes les étapes de la traduction, ce que nous illustrons ci-dessous pour l'association - de 1 à n - REGIONaffichePLAGE :

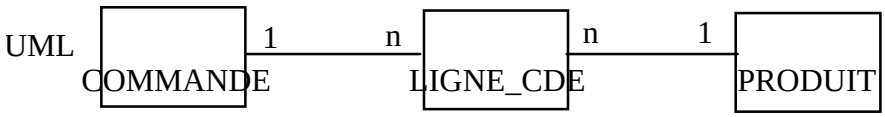


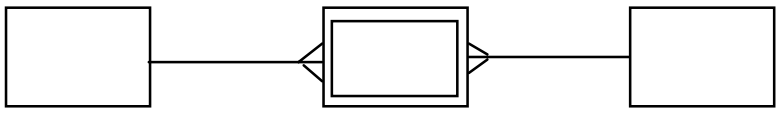


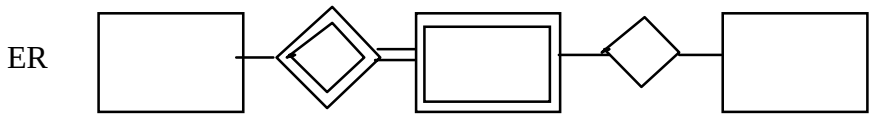
(la raison pour laquelle la "classe association" est notée par un double rectangle sera expliquée plus loin)

II.3.3. Associations identifiantes et/ou entités faibles

Les entités ou les classes s'opposent aux associations ou aux liens en ce qu'elles sont définissables en elles-mêmes. De même les identifiants de leurs individus ou instances sont autonomes. Or il existe de nombreux cas dans les univers à modéliser où certaines entités ne sont définies que *relativement* à d'autres, et les identifiants de leurs individus composés d'un identifiant d' *"individu parent"* suivi d'un identifiant propre en tant qu' *"individu enfant"*. Dans ce cas on dit que l'entité correspondante est "faible" et que l'association parent-enfant est "identifiante". Pensez par exemple aux différents comptes bancaires d'une même client d'une banque, aux différentes personnes à la charge d'un même assuré social, aux différents joueurs d'une même équipe de foot, aux différents bureaux dans un même couloir, etc... La suppression de l'individu parent entraine celle de tous les individus enfants liés (suppression en "cascade"). Les différentes notations en usage sont illustrées par l'exemple classique ci-dessous :





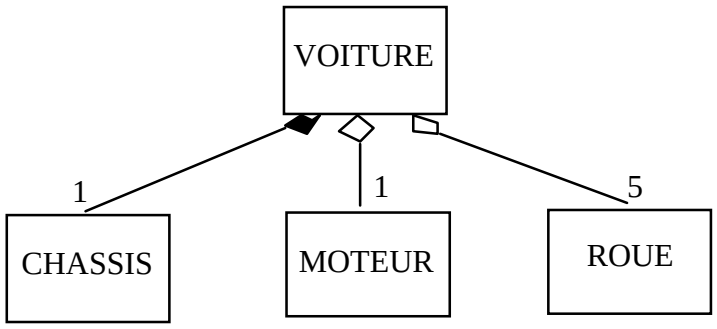


Dans cet exemple les COMMANDE et PRODUIT sont des entités "fortes" par opposition à LIGNE_CDE (rectangle doublé) qui est une entité "faible, dont le lien (trait continu) ou l'association (lozange doublé) identifiant à gauche se distingue du lien (trait discontinu) ou de l'association (lozange simple) non identifiant à droite.

II.3.4. Agrégations

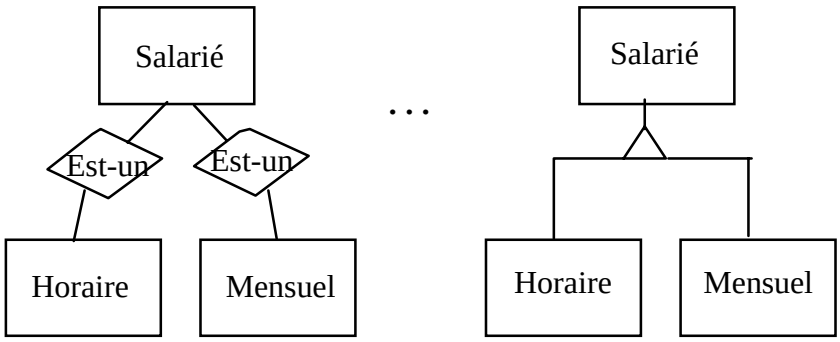
Les agrégations sont des associations ou liens particuliers ayant la sémantique " composé-composant".

La suppression d'un "composé" peut ou non entrainer celle des "composants" ("pièces détachées" irrécupérables ou récupérables!). Réciproquement la suppression d'un "composant" peut ou non entrainer la suppression du "composé" (une voiture sans roue ne change pas d'identité, une voiture avec un nouveau chassis est une autre voiture...). La notation ER classique était insuffisante pour rendre compte de ces surcharges sémantiques. La notation UML en rend compte assez précisement comme illustré ci-dessous :



II.3.5. Généralisation et spécialisation

C'est le couple de concepts le plus important pour enrichir le modèle ER. C'est un des concepts fondamentaux de l'approche orientée objet. Entre une classe, par exemple d'Employés, et une classe plus générale, par exemple de Personnes, le lien ou l'association est du type "EST-UN" (IS-A in english) : un Employé est une Personne. Même si les individus de la classe spécialisée (Employés) peuvent avoir une identification locale à cette sous-classe, il est entendu qu'ils peuvent d'abord être identifié en tant qu'individus de la super-classe (Personnes). Plus généralement un individu d'une sous-classe "hérite" de tous les attributs de la super-classe. Le lien réciproque de la généralisation est la spécialisation. Selon que les sous-classes d'une même classe forment ou non une partition de cette classe, on dit que la spécialisation est complète ou non. Différentes classes se généralisant dans une classe par l'opération d'union sont appelées des "catégories" pour cette classe. Les notations varient un peu entre les modèles ERC, .., OMT, UML. Le principal changement est apparu entre l'ER classique qui dessinait autant d'associations que de liens "Est-un" et les modèles suivants qui ont cré un nouveau symbole pour un noeud arborescent liant une super-classe à plusieurs sous-classes.



ER classique **UML**

II.3.6. Autres concepts

Il est très vite apparu nécessaire d'ajouter des *contraintes* à une association, comme, par exemple pour une association parent-enfant, d'ordonner les individus liés, ou des contraintes entre associations, comme, par exemple une contrainte d'exclusion pour le couple d'associations " défendu_par" et "accusé_par", ou, pour les associations de type "Est-un", une contrainte de partitionnement éventuel. Le fait que toute association peut ou non créer une *association inverse* (affirmer qu'un père "connait" ses enfants, ce n'est pas automatiquement affirmer que chaque enfant "connait" son père...) est aussi une surcharge que les modèles ajoutent souvent. D'autres concepts, comme celui de "qualificatif" (dans le modèle UML) ou celui d' "assertion" (dans l'ERC) ont aussi été ajoutés pour représenter d'autres contraintes que celles évoquées ci-dessus. Tous les modèles actuellement utilisés buttent sur l'infinie richesse du langage naturel pour l'expression des contraintes...

II.4. CONCLUSIONS

Les modèles pour produire des schémas de bases de données ont suivi deux histoires parallèles: les modèles de définition *conceptuelle* , qui cherchent la plus grande richesse sémantique plus que la facilité d'implémentation sur un SGBD, ont suivi une voie propre depuis l'ER de Chen; les modèles d'implémentation *logique*, liés aux structures de stockage et aux langages de manipulation des SGBD, ont connu une histoire faite de continuités et de ruptures (environ tous les 10 ans) du modèle hierarchique au modèle objet, fortement contrainte par les phénomènes de diffusion et de parc installé. A l'heure de la "convergence relationnel-objet" (et du SQL3) on se plait à espérer une "convergence logique-conceptuel".

II.5. REFERENCES

[Bachman 73] C. W. BACHMAN : *"The Programmer as Navigator"*, Communications of the ACM, Vol 16, N°11, Novembre. 1973.

[Chen 76] P. P. CHEN : *"The Entity-Relationship Model - Towards a Unified View of Data"*, ACM, Vol 1, N°1, Mars. 1976.

[CODASYL 71] CODASYL DATA BASE TASK GROUP : *"Report"*, Avril 1971, ACM, New-York.

[CODASYL 78] CODASYL Programming Language Committee *"COBOL Journal of Development"*, 1978.

[Parent et S. 85] C. PARENT, S. SPACCAPIETRA : *"An algebra for a general Entity-Relationship model"*, IEEE, Transactions on Software Engineering, July 1985.

[RFC 1066 et 1213] *"Management Information Base for Network Management of TCP/IP-based internets"*.

[Rumbaugh et al. 91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY et W. LORENSON : *"Object-Oriented Modeling and Design"*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[UML 97] HTTP://WWW.OMG.ORG/DOCS/DOCLIST-97.HTML *"UML1.1 (OMG)"*, 1997.

CHAPITRE 3 : LE MODELE RELATIONNEL

Début des années 70, le modèle relationnel fait son apparition [Codd 70]. La recherche se passionne : impossible de nier les p rogrès apportés concernant la représentation et la manipulation des données par les systèmes. Dix ans passent, les spécialistes déchantent : ce top-model engendre en définitive des systèmes commerciaux bien moins performants que leurs concurrents fondés sur les modèles réseau ou hiérarchique. Deux ans plus tard et voilà que les produits relationnels peuvent prétendre relayer les "vieux" systèmes. Leurs apports sont fondamentaux : les nouvelles fonctionnalités permettent un confort d'utilisation sans précédent. Les systèmes commerciaux s'emparent des concepts de ce nouveau modèle. Celui-ci, désormais, s'impose.

Mais quels sont ces concepts, leurs avantages, leurs limites ? Certains sont déjà bien répandus. Ce sont les notions de base que nous détaillerons dans la première partie : domaine, relation, attribut ; puis la manipulation ensembliste des relations par les opérateurs de l'algèbre relationnelle (deuxième partie) sur lesquels sont construits des langages non procéduraux comme SQL (chapitre 4) ; l'importa nte base théorique du modèle enfin fournit des méthodes pour la conception des bases de données (chapitre 5). D'autres concepts sont toutefois moins connus qui constituent pourtant un progrès essentiel : les vues relationnelles permettent à chaque utilisateur de personnaliser sa vision des données et les contraintes d'intégrité complètent la description de l'information (chapitre 6). Les chapitres suivants présentent chacun des ressorts qui font du modèle relationnel la référence obligée en matière de gestion de bases de données.

III.1. LE MODELE RELATIONNEL

Le modèle relationnel, contrairement à ceux présentés dans le chapitre précédent, ne manipule pas des structures de données figées, mais des valeurs : aucun chemin d'accès n'est préalablement défini (on ne parlera désormais plus de déplacements ou de langages navigationnels), toute manipulation des données est désormais possible. L'important bagage théorique, et la vision tabulaire des informations, qui est agréable à l'utilisateur, assurent le succès du modèle relationnel.

III.1.1. Une première approche du relationnel

Nous présentons dans ce paragraphe un moyen simple de visualiser la structure de base du modèle relationnel : la relation. Nous décrivons ci-dessous une extension de la base des nageurs sous la forme de tableaux de colonnes de valeurs typées et nommées.

La relation des nageurs :

NAGEURS	NN	NOM	PRENOM	QUALITE
	100	Plouf	Jean	Mauvaise
	110	Coule	Paul	Excellente
	120	Brasse	Jean	Bonne

La relation des plages :

PLAGES	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absente
	119	Nice	galets	Côte d'Azur	forte
	107	Oleron	sable	Atlantique	moyenne
	118	Binic	sable	Bretagne	forte

La relation des baignades :

BAIGNADES	NN	NP	DATE	DUREE
	110	118	14/07/89	2
	110	118	15/07/89	10
	120	119	12/07/89	120

III.1.2. La construction du modèle relationnel

Nous allons dans ce paragraphe introduire les notions utilisées dans la construction théorique du modèle relationnel: domaine, relation, tuple, attribut, schéma.

La notion de domaine

Définition : domaine

Un domaine est un ensemble de valeurs (distinctes).

Cette définition correspond à l'ensemble des valeurs que peut prendre une certaine manifestation du monde réel. Elle s'apparente souvent à un type (entier, réel), mais peut également être hétéroclite (date). Dans l'exemple précédent, un domaine est l'ensemble des valeurs d'une colonne d'un tableau : {'Mauvaise', 'Excellente', 'Bonne'} est un domaine.

Exemples de domaines :

- l'ensemble des entiers est un domaine ;
- {3, 5.7, 124} est un domaine ;
- [-3, 4] ∪ {5, 7} est un domaine ;
- ('Jean', 'Paul', 'Louis', 'Arthur') est un domaine ;
- (14/07/89, 15/07/89, 15/07/89) est un domaine ;
- Tout ensemble de valeurs est un domaine.

Produit cartésien de plusieurs domaines :

Définition : produit cartésien

Le produit cartésien d'un ensemble de domaines $D_1, D_2, \ldots, D_n$, noté $D_1 x D_2 x \ldots x D_n$, est l'ensemble de n-uplets (ou tuples) $<v_1, v_2, \ldots, v_n>$ tels que $v_i \in D_i$.

Exemple :

Réalisons le produit cartésien des domaines suivants, (Plouf, Coule, Brasse), (Jean, Paul).

Plouf	Jean
Plouf	Paul
Coule	Jean
Coule	Paul
Brasse	Jean
Brasse	Paul

Comme on le voit, si un domaine représente l'ensemble des valeurs possibles d'une manifestation du monde réel (exemples : les cheveux sont blonds, noirs, bruns ou blancs ; les ages sont compris entre 0 et 130, etc.), le monde réel "possible" est constitué par le produit cartésien des domaines et le monde réel "réel" est forcément un sous ensemble du produit cartésien.

La notion de relation

Définition : relation

une relation est un sous-ensemble du produit cartésien d'une liste de domaines.

Exemple :

la relation NAGEURS est un sous-ensemble du produit cartésien des domaines suivants : (100, 110, 120), (Plouf, Coule, Brasse), (Jean, Paul), (Mauvaise, Bonne, Excellente).

La notion de n-uplet

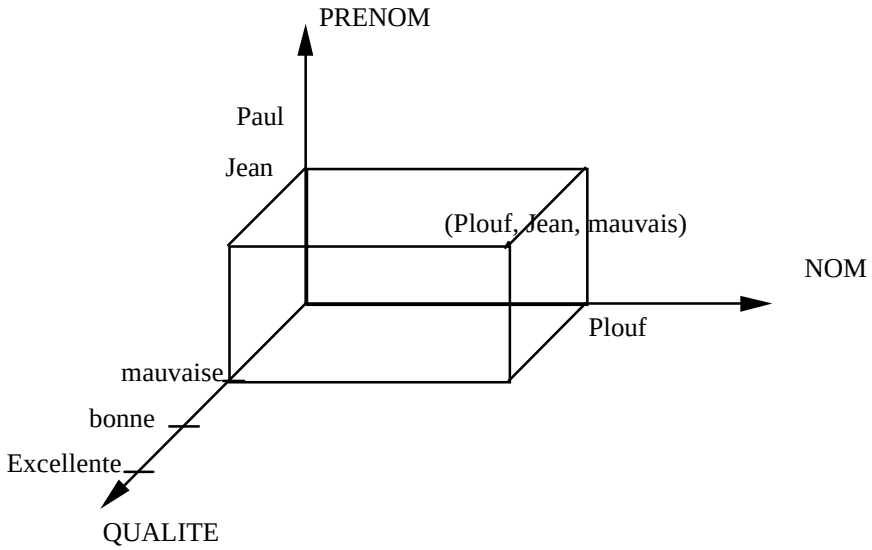
Définition : n-uplet ou tuple

un n-uplet (élément) correspond à une ligne d'une relation.

Une autre représentation des relations

Une autre façon de considérer une relation à N domaines D_1, D_2,... D_N est de représenter un espace à N dimensions. Dans cet espace chaque domaine correspond à l'une des dimension, chaque *n-uplet* ou *tuple* correspond à un point de l'espace.

Exemple



La notion d'attribut

Définition : attribut

Un attribut est le nom donné à une colonne d'un tableau représentant une relation.

Exemple :

les attributs de la relation PLAGES sont NP, NOMP, REGION, TYPE et POLLUTION.

La notion de schéma

Définition : schéma d'une relation

Le schéma d'une relation est composé de son nom suivi du nom de ses attributs et de leurs domaine :

$R(A_1 \subset D_1, A_2 \subset D_2,\ldots, A_N \subset D_N)$.

Lorsque le choix des domaines est évident, on simplifie l'écriture de la façon suivante :

$R(A_1, A_2,\ldots, A_N)$.

Exemple :

PLAGES (NP, NOMP, TYPE, REGION, POLLUTION).

Définition : schéma d'une base de données relationnelle

Le schéma d'une base de données relationnelle est l'ensemble des schémas des relations qui la composent..

Une base de données relationnelle est constituée par l'ensemble des tuples de toutes les relations définies dans le schéma de la base.

III.1.3. Une manipulation ensembliste des données

Nous avons étudié au chapitre précédent les premiers modèles de SGBD. Que ce soit le segment dans le modèle hiérarchique ou l'article dans le modèle réseau, la manipulation des informations s'effectue enregistrement par enregistrement.

Dans un système relationnel, les informations ne sont pas forcément repérées individuellement ; on sait appliquer le même traitement à un ensemble d'enregistrements caractérisés, non par la liste des identifiants individuels, mais par le critère que vérifie chacun des enregistrements qui composent l'ensemble. On dit que la manipulation est ensembliste.

En particulier, pour rechercher des tuples, il suffit de préciser un critère de sélection ; le système déterminera l'ensemble des tuples satisfaisant ce critère et rendra un résultat. Les tuples de ce résultat, extraits de relations de la base, constituent eux-même une relation qu'il sera ainsi aisé de conserver, si nécessaire, pour le plus grand confort de l'utilisateur.

Par exemple, pour connaître les dates et durées des bains pris par Paul Coule ainsi que les noms des plages, on obtient le résultat suivant, qui est une relation RESULTAT à trois attributs NOMP, DATE, DUREE :

RESULTAT	NOMP	DATE	DUREE
	Binic	14/07/89	2
	Binic	15/07/89	10

Typiquement, on peut classer les requêtes en deux grandes catégories : la mise à jour de tuples dans une relation et la recherche de tuples vérifiant une certaine condition. Eventuellement, ces deux types de requêtes peuvent être combinés.

La manipulation ensembliste est très utile en mise à jour. Elle permet, par exemple, de modifier directement la qualité de tous les nageurs ayant pris un bain sur la plage de Binic le 14 juillet 1989, sans avoir à déterminer préalablement la liste des nageurs qui présentent cette caractéristique. Cette puissance d'expression explique largement le succès du relationnel.

Les insertions/suppressions sont réalisées à l'aide de relations temporaires internes et d'opérateurs ensemblistes d'union et de différence (détails au paragraphe suivant). La combinaison de ces opérateurs et des opérateurs relationnels de sélection sur des critères de recherche facilite sensiblement les opérations de modification. C'est le cas en particulier des modifications calculées, c'est-à-dire des modifications portant sur un ensemble de tuples résultant eux-mêmes d'une sélection.

Exemples

- insertion ou suppression de Paul Brasse, mauvais nageur, qui a pris un bain de 2 minutes le 14/07/1989 sur la plage de sable très polluée de Binic, en Bretagne.
- recherche des noms des nageurs ayant pris des bains de plu s d'une minute en Bretagne.
- suppression de tous les nageurs ayant pris, en février, des bains de plus de 2 minutes en Bretagne (hydrocution ?).

Pour manipuler ces relations, nous avons besoin d'un langage adapté dont la particularité est de savoir manipuler aisément ces tableaux de données. Ce langage constitue l'algèbre relationnelle.

III.2. L'ALGEBRE RELATIONNELLE

L'algèbre relationnelle est le langage interne d'un SGBD relationnel. Elle se compose d'opérateurs de manipulation des relations. Ces opérateurs sont regroupés en deux familles : les opérateurs ensemblistes et les opérateurs relationnels. Chacune de ces familles contient quatre opérateurs.

III.2.1. Les opérateurs ensemblistes

L'union

Noté R $\cup$ S, où R et S représentent deux relations de même schéma, cet opérateur permet de réaliser l'insertion de nouveaux tuples dans une relation. Par exemple, ajouter à la relation permanente RP...

RP	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absence
	119	Nice	galets	Côte d'Azur	forte
	107	Oleron	sable	Atlantique	moyenne
	118	Binic	sable	Bretagne	forte

... la relation de travail RT...

RT	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absence
	115	Deauville	sable	Normandie	forte
	117	Trouville	sable	Normandie	forte

... on obtient la nouvelle relation RESULTAT = RP $\cup$ RT...

RESULTAT	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absence
	119	Nice	galets	Côte d'Azur	forte
	107	Oleron	sable	Atlantique	moyenne
	118	Binic	sable	Bretagne	forte
	115	Deauville	sable	Normandie	forte
	117	Trouville	sable	Normandie	forte

Remarquons que la manipulation est réellement ensembliste : les éléments qui pourraient être dupliquées (insertion de la plage de Trégastel qui figure déjà dans la relation permanente) ne le sont pas.

L'intersection

Opérateur noté R $\cap$ S, où R et S représentent deux relations de même schéma, qui permet de retrouver les tuples identiques dans deux relations .

La différence

Opérateur noté R - S, où R et S représentent deux relations de même schéma, qui permet de réaliser la suppression de tuples dans une relation. Par exemple, retrancher à la relation permanente RP...

RP	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absence
	119	Nice	galets	Côte d'Azur	forte
	107	Oleron	sable	Atlantique	moyenne
	118	Binic	sable	Bretagne	forte
	115	Deauville	sable	Normandie	forte
	117	Trouville	sable	Normandie	forte

... La relation de travail RT...

RT	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absence
	107	Oleron	sable	Atlantique	moyenne

... Pour obtenir la relation RESULTAT = RP - RS…

RESULTAT	NP	NOMP	TYPE	REGION	POLLUTION
	119	Nice	galets	Côte d'Azur	forte
	118	Binic	sable	Bretagne	forte
	115	Deauville	sable	Normandie	forte
	117	Trouville	sable	Normandie	forte

Le produit cartésien

Si R1 et R2 sont des relations de schémas respectifs ($att_{1,1}$, $att_{1,2}$,...,$att_{1,N}$) et ($att_{2,1}$, $att_{2,2}$,..., $att_{2,M}$) contenant respectivement T_1 et T_2 tuples, alors la relation $R = R_1 X R_2$ résultant du produit cartésien des deux relations a pour schéma ($att_{1,1}$, $att_{1,2}$,...,$att_{1,N}$, $att_{2,1}$, $att_{2,2}$,..., $att_{2,M}$) et contient $T_1 * T_2$ tuples obtenus par concaténation des T_1 tuples de R_1 et des T_2 tuples de R_2.

III.2.2. Les opérateurs relationnels

La restriction

Noté $\sigma_E(R)$ où E exprime un prédicat sur une relation R, cet opérateur permet de ne conserver, dans une relation, que les tuples dont les attributs vérifient une certaine condition. Par exemple, la relation permanente RP...

RP	NP	NOMP	TYPE	REGION	POLLUTION
	110	Trégastel	sable	Bretagne	absence
	119	Nice	galets	Côte d'Azur	forte
	107	Oleron	sable	Atlantique	moyenne
	118	Binic	sable	Bretagne	forte
	115	Deauville	sable	Normandie	forte
	117	Trouville	sable	Normandie	forte

… réduite aux plages à forte pollution (le prédicat E se note POLLUTION = 'forte') donne la relation RESULTAT…

RESULTAT	NP	NOMP	TYPE	REGION	POLLUTION
	119	Nice	galets	Côte d'Azur	forte
	118	Binic	sable	Bretagne	forte
	115	Deauville	sable	Normandie	forte
	117	Trouville	sable	Normandie	forte

La projection

Noté $\bullet_{A1,...,AN}(R)$ où $A_1,...,A_N$ représentent des attributs particuliers de la relation R, cet opérateur permet de ne conserver que certains attributs d'une relation. Par exemple, les différentes régions et leurs niveaux de pollution sont obtenus par projection de la relation PLAGES sur les attributs REGION et POLLUTION...

RESULTAT	REGION	POLLUTION
	Bretagne	absence
	Côte d'Azur	forte
	Atlantique	moyenne
	Bretagne	forte
	Normandie	forte

Notons l'aspect ensembliste de cette opération qui ne duplique pas le tuple ('Normandie', 'forte') qui figurait en double dans la relation intermédiaire résultat d'une projection brutale de la relation initiale.

La jointure

Définition :

La jointure de deux relations R et S selon une qualification multi-attributs Q est l'ensemble des tuples du produit cartésien RxS satisfaisant la qualification Q.

Notation

Il existe plusieurs notations possibles de cet opérateur. Les plus répandues sont :

- JOIN (R, S/Q) ;
- R $|x|_Q$ S

équi-jointure

La qualification Q consiste simplement en l'égalité entre différents attributs des deux relations. C'est le cas le plus courant de jointure.

Exemple :

Pour connaître les bains pris par les différents nageurs, il suffit d'effectuer l'équi-jointure entre les deux relations NAGEURS et BAIGNADES sur chacun de leur attribut NN

RESULTAT	NOM	PRENOM	QUALITE	NN	NN	NP	DATE	DUREE
	Coule	Paul	Excellente	110	110	118	14/07/89	2
	Coule	Paul	Excellente	110	110	118	15/07/89	10
	Brasse	Jean	Bonne	120	120	119	12/07/89	120

dans le cas d'une équi-jointure, on peut éliminer la colonne en double ; on parle alors d'équi-jointure naturelle.

Exemple d'inéqui-jointure :

Quels sont les nageurs ayant pris des bains sur une plage de numéro NP supérieur à leur propre numéro d'identification NN ?

RESULTAT	NOM	PRENOM	QUALITE	NN	NP	NN	DATE	DUREE
	Coule	Paul	Excellente	110	118	110	14/07/89	2
	Coule	Paul	Excellente	110	118	110	15/07/89	10

Autre exemple de jointure :

Quels sont les nageurs ayant pris des bains de durée égale à leur numéro ?

RESULTAT	NOM	PRENOM	QUALITE	NN	DUREE	NN	NP	DATE
	Brasse	Jean	Bonne	120	120	120	119	12/07/89

La division

Noté R ÷ S où R et S désignent deux relations, cet opérateur, plus complexe, permet de connaître toutes les valeurs d'un domaine qui sont en correspondance avec toutes les valeurs d'un autre domaine. Par exemple, existe-t-il des nageurs qui ont pris des bains à la fois le 14/07/89 et le 15/07/89 ? Pour le savoir, il suffit d'effectuer la division de la relation Baignade projetée sur (NN, DATE) (notée RES1) par la colonne DATE restreinte aux 14/07/89 et au 15/07/89 :

BAIGNADES	NN	NP	DATE	DUREE
	110	118	14/07/89	2
	110	118	15/07/89	10
	120	119	12/07/89	120

RES1	NN	DATE
	110	14/07/89
	110	15/07/89
	120	12/07/89

DATES	DATE
	14/07/89
	15/07/89

RESF	NN
	110

Les tuples du résultat RESF (NN), concaténés à tout tuple de DATES (DATE) donne un tuple de RES1 (NN, DATE). Ainsi, l'ensemble des tuples résultats du produit cartésien de RESF et DATES sont *dans* RES1.

III.2.3. Opérateurs de base et opérateurs dérivés

Cinq de ces huit opérateurs forment les opérateurs de base (ce sont l'union, la différence, le produit cartésien, la restriction et la projection) tandis que les trois autres, appelés opérateurs dérivés, s'obtiennent plus ou moins facilement par combinaison des opérateurs de base :

$R \cap S = R - (R - S)$

$R \,|x|\, S\ (Ai, Bj) = \sigma_{i\theta j}\ (R \times S)$

$R \quad S = \pi_{i1,\ldots,i(r-s)}\ (R) - \pi_{i1,\ldots,i(r-s)}\ (\ (\pi_{i1,\ldots,i(r-s)}\ (R) \times S) - R)$

Les cinq opérateurs de base permettent de répondre à toutes les questions que l'on peut poser avec la logique du premier ordre (c'est à dire sans les fonctions) : on dit que l'algèbre relationnelle est complète.

En réalité, nous n'utiliserons dans nos requêtes que les opérateurs les plus maniables : ce sont l'union et la différence pour l'insertion et la suppression de tuples dans la base et la restriction, la projection et la jointure pour la recherche sélective de tuples.

III.2.4. Des exemples de requêtes

Exemple 1 :

Donner les noms des plages fortement polluées.

Cette requête comprend deux opérations,

TEMP ♦ Rest (PLAGES, POLLUTION = 'forte')

RESU ♦ Proj (TEMP, NOMP)

Ce qui se note encore :

RESU ♦ Proj (Rest(PLAGE, POLLUTION = 'forte'), NOMP)

Exemple 2 :

Donner les noms des personnes ayant pris un bain en Bretagne.

T1 ♦ Rest (PLAGE, REGION = 'Bretagne')

T2 ♦ Join (T1, BAIGNADE, T1.NP = BAIGNADE.NP)

T3 ♦ Join (T2, NAGEURS, T2.NN = NAGEURS.NN)

RESU ♦ Proj (T3, NOM)

Exemple 3 :

Insertion de la plage de sable "Fort bloqué", en Bretagne, faiblement polluée avec le numéro 120.

PLAGES ♦ PLAGES ∪(120, 'Fort bloqué', 'sable', 'Bretagne', 'faible').

Exemple 4 :

Suppression des nageurs de qualité médiocre.

NAGEURS ♦ NAGEURS - Rest(NAGEURS, QUALITE = 'médiocre')

III.2.5. Les arbres algébriques et l'optimisation

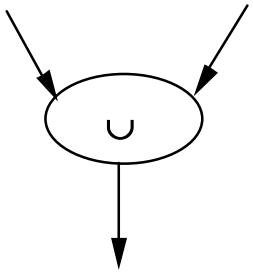
Nous venons de voir comment décrire une requête au moyen des opérateurs de l'algèbre relationnelle. Toutefois, l'écriture de cette combinaison d'opérateurs est malaisée à déchiffrer. Il existe en contrepartie une description graphique plus lisible : l'arbre algébrique.

Un arbre algébrique est un arbre dont :

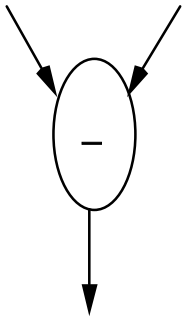
- les feuilles sont les relations de base ;
- les nœuds sont les opérateurs ;
- la racine est le résultat ;
- les liens représentent les flux de données.

Ainsi, des nœuds différents représentent les cinq principaux opérateurs relationnels : l'union, la différence, la restriction, la projection et la jointure.

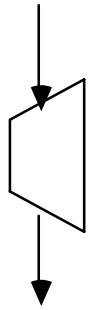
L'union :



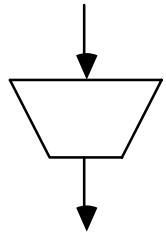
La différence :



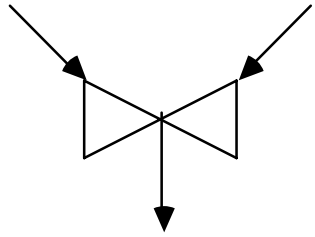
La restriction :



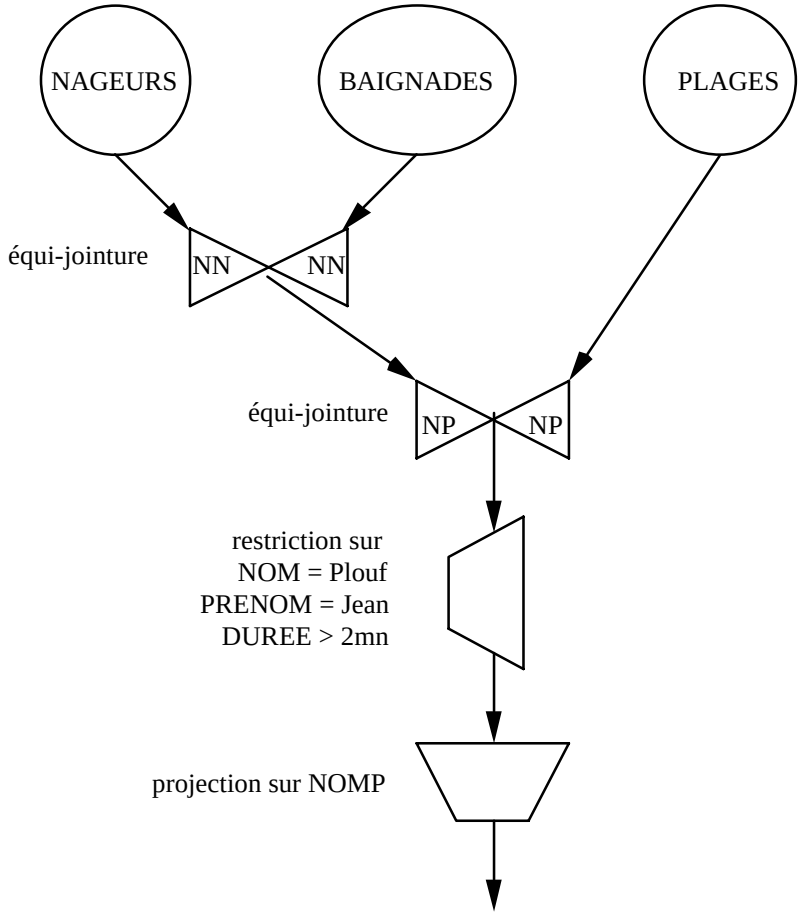
La projection :



La jointure :



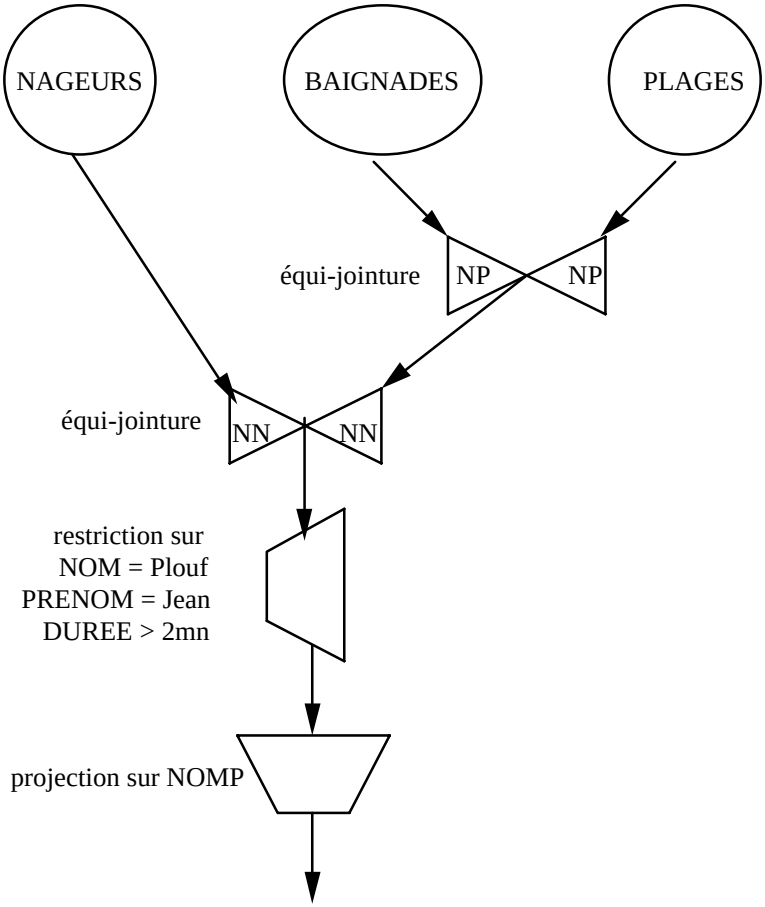
Nous pouvons dès lors facilement représenter la requête :
"Sur quelle plage (NOMP) Jean Plouf a-t-il pris des bains de plus de deux minutes ? "
Remarquons que cette requête peut s'exécuter de diverses façons :



Première méthode

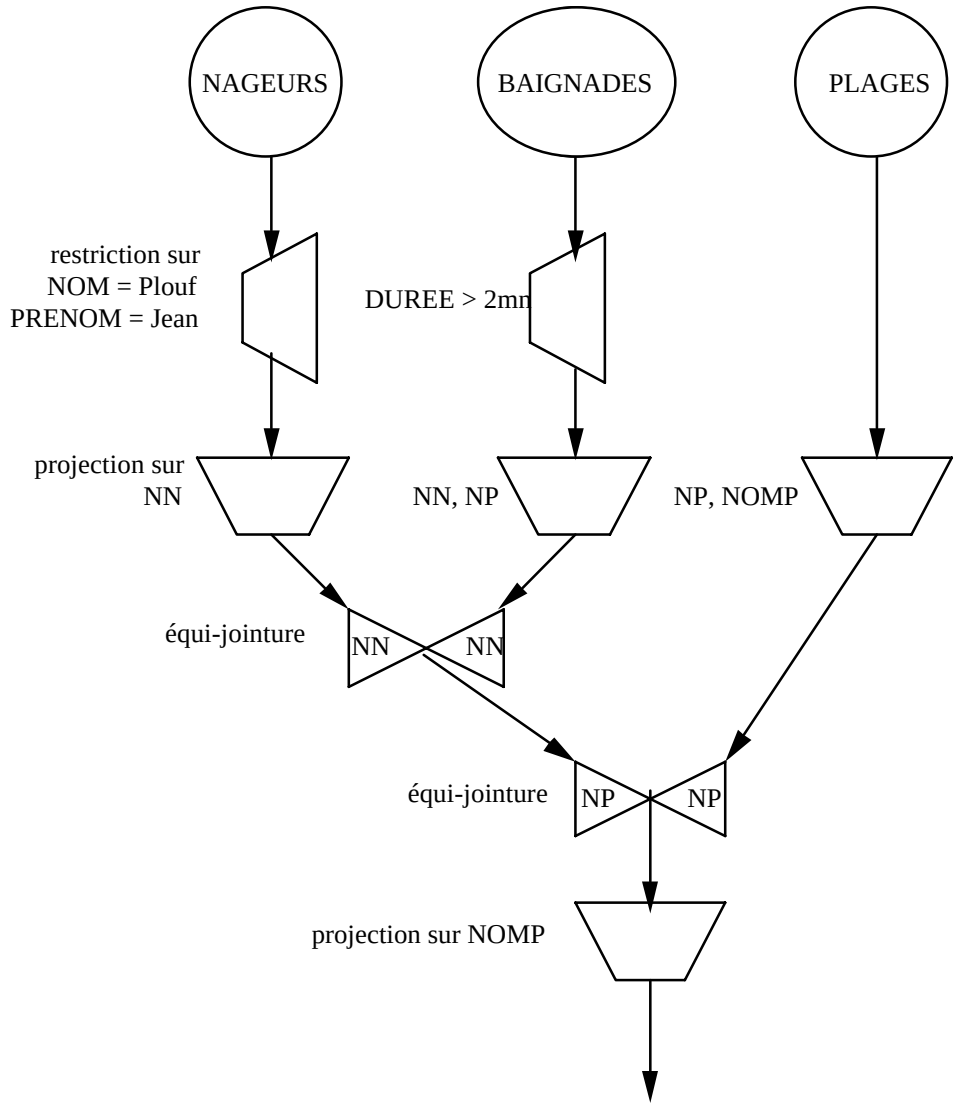
Une première méthode d'exécution de requête peut minimiser le nombre d'opérations à effectuer. C'est ce que nous faisons ici en exécutant les jointures dans un premier temps, puis en repoussant à la fin la restriction et la projection qui permettent d'obtenir le résultat désiré. On minimise ainsi le nombre d'opérations à effectuer, mais on réalise des jointures sur des relations très volumineuses. ce qui augmente considérablement les tailles des relations intermédiaires et donc le coût global d'exécution de la requête.

Deuxième méthode



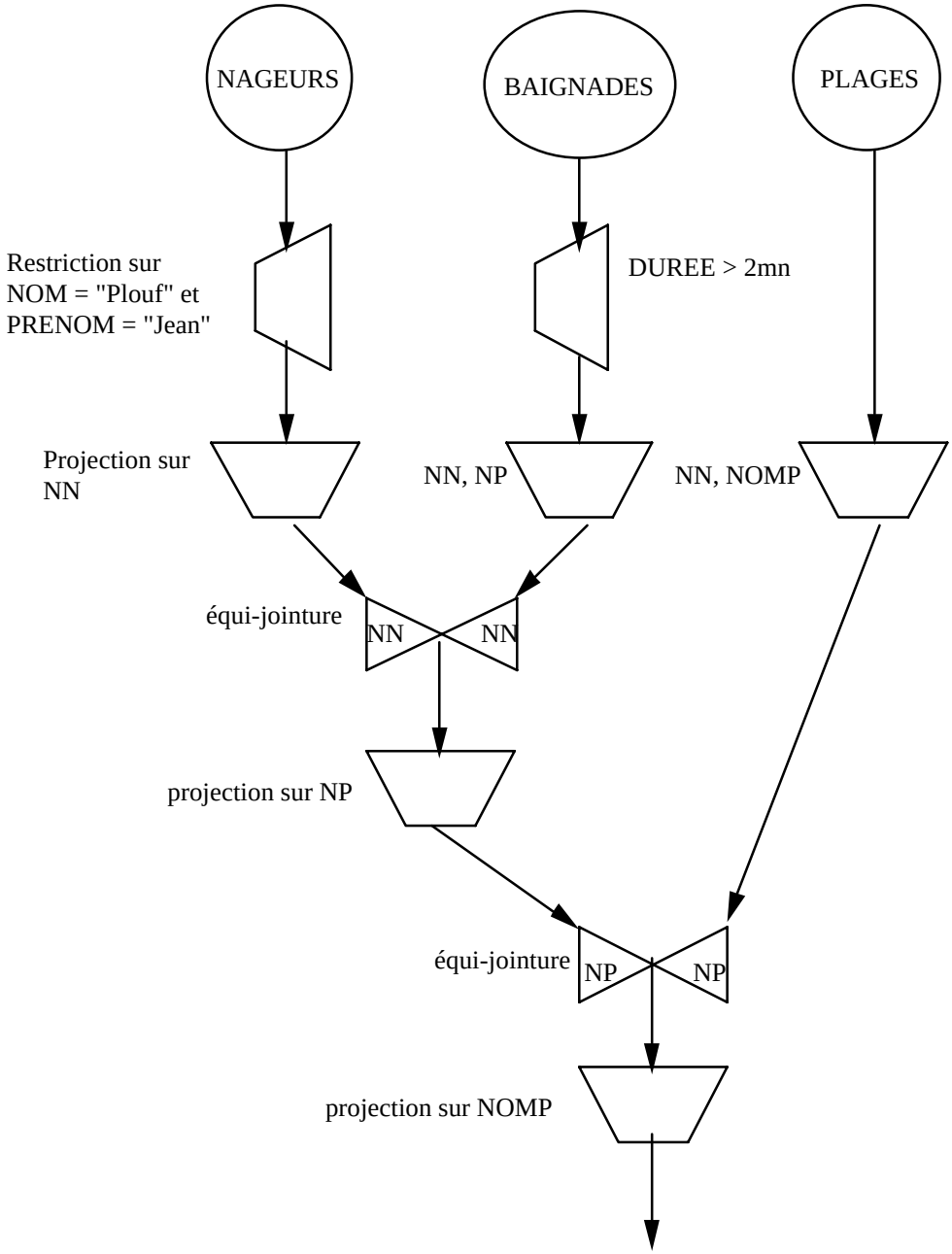
Quelle que soit l'heuristique choisie pour exécuter la requête. Déterminer dans quel ordre l'exécution des jointures entraîne un coût minimum est intéressant. Suivant la taille des relations mises en jeu, la différence de coût Correspondant à une inversion dans l'ordre des jointures peut être considérable.

Troisième méthode



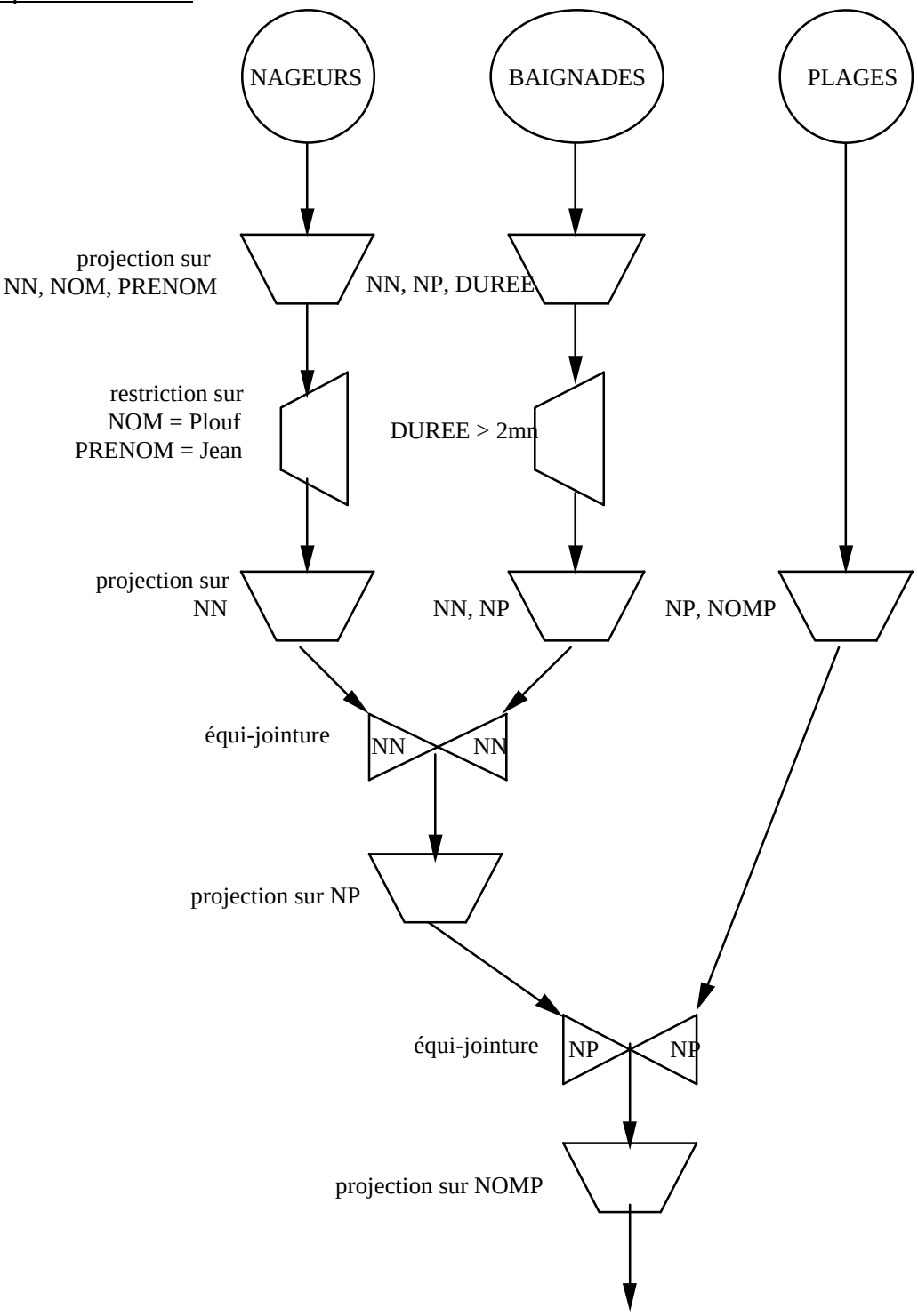
Toutefois, compte tenu du coût très important des jointures et de l'importance de la taille des relations mises en jeu, il vaut mieux rajouter des étapes intermédiaires qui diminuent la taille des relations à l'entrée des jointures. Le coût supplémentaire est compensé par une baisse sensible du coût des jointures. C'est ce que nous faisons ici : les opérations de jointure agissent sur des relations réduites au maximum par des restrictions et des projections.

Quatrième méthode



Nous pouvons affiner cette méthode en éliminant après chaque jointure les attributs devenus inutiles pour l'obtention du résultat final.

Cinquième méthode



Enfin, nous pouvons également être amenés à rechercher les critères de variation du coût entre la méthode décrite ci-dessous et la précédente. Nous rajoutons ici une première étape de projection avant la première restriction.

A partir de cet exemple, nous constatons qu'à une même requête correspond de multiples plans d'exécution. Le choix d'un plan par rapport à un autre a de nombreuses implications quant à la durée d'exécution de la requête.

Enfin, si la vérification que l'ordre des opérateurs permet de réaliser la question initialement posée, choisir parmi les nombreuses solutions qui n'affectent pas le résultat, mais seulement la rapidité d'exécution de la requête est plus délicat .

En effet, se préoccuper activement du coût d'exécution d'une requête est nécessaire : un SGBD est jugé sur sa rapidité.

Une heuristique simple consiste à restreindre le plus vite possible la taille des relations intermédiaires pour diminuer le coût . Pour ce faire, nous utilisons les propriétés des différents opérateurs :

- associativité des jointures ;
- commutativité restriction/jointure ;
- commutativité restriction/projection (si le critère de restriction porte sur les attributs projetés);
- commutativité projection/jointure (si le critère de jointure porte sur les attr ibuts projetés).

Une méthode relativement aisée à mettre en œuvre consiste à disposer en entrée des jointures de relations les plus petites possibles, ce qui revient à faire précéder les jointures, opérations les plus coûteuses, du maximum de restrictions et de projections possibles sans altérer le résultat final.

De façon générale ; il est indispensable de disposer :

- de relations initiales et intermédiaires les plus petites possibles ;
- de techniques de placement adaptées qui permettent de ne pas rapatr ier toute une relation pour isoler les informations intéressantes (voir chapitre 7) ;
- d'algorithmes performants implantant les opérateurs relationnels.

III.3. CONCLUSIONS

Les SGBD existant à ce jour mettent en œuvre des techniques performantes pour la manipulation des données.

Par exemple, tous les SGBD offrent aujourd'hui un langage assertionnel de requêtes, SQL. Un langage assertionnel permet de poser la requête sous la forme d'une ou de plusieurs conditions (assertions) à remplir sans se préoccuper de l'ordre des différentes opérations à effectuer au niveau du système. C'est au SGBD d'analyser la condition et de la décomposer en une requête de l'algèbre relationnelle.

L'optimiseur de requêtes est un composant très important d'un SGBD relationnel. L'étape fondamentale est la construction d'un arbre algébrique optimisé, en déterminant le meilleur ordre d'exécution des diverses opérations. Cependant, les SGBD utilisent d'autres critères (contraintes d'intégrité, taille des relations candidates, méthodes de placement…) soit pour aider à la construction de l'arbre optimisé, soit pour réfuter logiquement une requête et pour choisir l'algorithme adéquat à la réalisation d'une opération particulière (une même

opération dispose couramment de différents algorithmes dont les performances varient suivant les caractéristiques des relations mises en jeu).
L'opération de jointure ou la prise en compte des temps de transit de données dans des systèmes répartis sont un bon exemple de la multiplicité des algorithmes existants.
Les améliorations sur le marché suivent de près les progrès de la recherche et les produits évoluent vite : les premières versions de nombreux produits offraient des algorithmes peu performants, mais la situation a bien changé depuis. Les SGBD demeurent toutefois des produits délicats à évaluer, comme c'est souvent le cas des produits informatiques. L'utilisation d'un algorithme ne suffit pas à garantir sa bonne utilisation. Notez en particulier l'importance essentielle des méthodes de stockage et des primitives de bas niveau offertes par le système d'exploitation pour l'efficacité des SGBD. Pour toutes ces raisons, le modèle relationnel, malgré sa "beauté conceptuelle", a mis beaucoup de temps à s'imposer sur le marché.

III.4. REFERENCES

[Astrahan 76] M. ASTRAHAN et al : *"System R : A Relational Approach to Database Management"*, ACM Transactions on Database Systems, Vol 1, N°2, 1976.

[CODD 70] E.F. CODD, *"A RELATIONAL MODEL OF DATA FOR LARGE SHARED DATA BANKS"*, COMMUNICATIONS ACM, V13, N6, JUIN 1970, PP 377-**387.**

CHAPITRE 4 : LES LANGAGES RELATIONNELS

Les opérateurs algébriques, qui effectuent la manipulation interne des relations dans un SGBD relationnel, constitue un premier langage de requêtes. Ils présentent cependant l'inconvénient majeur d'exprimer les manipulations à effectuer de façon procédurale. En effet, la spécification directe par l'utilisateur d'une séquence d'opérations relationnelles correspond au choix d'une exécution particulière ; plusieurs exécutions sont possibles pour une même question. L'utilisateur à qui l'on offre un langage algébrique a donc plusieurs choix pour formuler sa question : certains sont performants, d'autres induisent des temps de réponses impressionnants, car ils génèrent un volume considérable de données intermédiaires et de très nombreux accès.

Les concepteurs de systèmes ont donc cherché à proposer un langage de manipulation de données qui ne demande précise pas l'ordre des opérations. Un tel langage est donc non-procédural. Pour cela, la complétude de l'algèbre relationnelle est exploitée, c'est-à-dire qu'elle est équivalente à la logique des prédicats du premier ordre (sans fonctions). Il s'agit d'un résultat important que nous admettrons simplement ici. Une requête s'exprime donc comme une assertion sur la base de données. Le critère logique que doivent vérifier les tuples du résultat est spécifié ; cette assertion décrit une caractéristique du résultat, indépendamment de la façon de l'obtenir. C'est le système qui, à partir de l'expression d'une assertion, génère une exécution particulière, c'est-à-dire une certaine séquence d'opérateurs relationnels.

La première tentative pour construire un véritable langage de requêtes, transformable en séquence d'opérations relationnelles, date de 76 : le langage SEQUEL du laboratoire IBM de San José. C'est lui qui donnera plus tard naissance à SQL (Structured Query Language).

IV.1. LE STANDARD SQL

Hier norme de fait, SQL est devenu une norme de droit quand l'International Organization for Standardization l'a pris en compte en 86 [ISO 86]. SQL est arrivé à maturité. Sa version normalisée par l'ISO fige ses différents aspects : définition des concepts, du vocabulaire, du langage de définition de données (déclaration du schéma, des vues, du contrôle des autorisations) et du langage de manipulation sont désormais stabilisés. Les procédures d'intégration des appels SQL dans un langage hôte (de type Cobol, Pascal, C, Fortran…), regroupées sous le nom de "Embedded SQL", souvent noté ESQL, sont également définies. Presque tous les constructeurs proposent le langage SQL. Plus de 70 produits du marché le fournissent, sur des machines qui vont du PC au plus importants mainframes. Celui-ci étend même sa percée vers des produits micros qui, sans devenir de véritables SGBD relationnels complets, intègrent déjà la partie de SQL qui concerne la manipulation des données.

SQL est un facteur important de promotion du modèle relationnel. Il apporte une vision unifiée et une large communauté de concepts et de vocabulaire. Reposant sur une base théorique solide (la logique des prédicats), il concerne à la fois les administrateurs, les programmeurs d'applications, et les utilisateurs finals pour la définition et la manipulation des données.

IV.1.1. Eléments d'un langage pour les SGBD relationnels

La norme SQL comporte trois parties distinctes concernant :

- la manipulation de l'information ; le (DML : Data Manipulation Language) concerne l'ensemble des requêtes de recherche et de mise à jour des informations : selection, insertion, modification et suppression. Il est essentiellement destiné à l'utilisateur final ;
- la définition des structures de données ; le DDL (Data Definition Language) contient l'ensemble des commandes de création, de suppression, de modification des relations. Elle intègre la créations, la suppressions de chemins d'accès (index) et les commandes concernant la gestion des droits et des autorisations. Elle est largement destinée à l'administrateur ;
- l'accès aux informations stockées dans le SGBD depuis un langage de programmation ; l' Embedded SQL définit l'ensemble des interfaces, l'utilisation des ordres SQL dans un langage de programmation, ainsi que l'ensemble des codes d'erreur renvoyés par le SGBD au programme d'application défini. L' Embedded SQL est destiné au programmeur d'applications.

La liste des avantages et inconvénients de la norme SQL telle qu'elle se présente aujourd'hui peut être rapidement dressée : à son actif, citons sa simplicité, sa souplesse et

ses fonctionnalités étendues ; à son passif, un manque de rigueur (il autorise - et encourage - certaines formulations procédurales), une utilisation lourde (mot clés, pas d'utilisation de moyens modernes d'interface homme/machine), le fait qu'il constitue, comme toute norme, un frein à la créativité des concepteurs de nouveaux systèmes. Rappelons enfin qu'il s'agit d'un langage inventé par IBM, argument stratégique délicat à placer dans les qualités ou les défauts du langage.

La normalisation de SQL, dont la première version a été achevée en 86, conduit également à se poser la question "à qui peut servir la normalisation ?". En effet, si l'on examine les différentes parties de la norme, on constate que l'administrateur doit disposer d'outils moins primitifs et plus spécifiques pour gérer convenablement son SGBD ; l'utilisateur final a bien du mal à utiliser directement SQL en interactif : il utilise des interfaces spécifiques (menus, écrans…) plus conviviales ; enfin le programmeur d'applications est aujourd'hui largement sollicité par les Langages de quatrième Génération (L4G) qui, sans être normalisés, sont plus pratiques et procurent les mêmes services de base que du Embedded SQL. Certaines mauvaises langues en concluent donc que la normalisation ne sert qu'aux normalisateurs… L'apport de la normalisation du SQL est cependant indiscutable pour les utilisateurs désirant garantir la portabilité de leurs applications d'un système SQL sur un autre. C'est l'interface standard de tout véritable SGBD relationnel. Il précise, d'autre part, un niveau d'interface précis lorsqu'on envisage des systèmes répartis et hétérogènes.

Comme pour les opérateurs de l'algèbre relationnelle, nous illustrerons l'utilisation de SQL sur la base-jouet suivante :

PLAGE (NP, NOMP, TYPE, REGION, POLLUTION)

NAGEUR (NN, NOM, PRENOM, QUALITE)

BAIGNADE (NN, NP, DATE, DUREE)

où NP désigne le numéro de plage, NN le numéro de nageur.

IV.1.2. Expression des opérations relationnelles

SQL permet d'exprimer simplement les opérations relationnelles de base.

Le bloc SQL se compose du mot-clé SELECT, suivi des attributs qu'on désire voir figurer dans le résultat, du mot clé FROM, suivi de la liste des relations touchées par la question, enfin du mot clé WHERE, suivi d'une qualification. Deux blocs SQL peuvent être conjugués par l'un des opérateurs binaires de l'algèbre relationnelle : UNION, INTERSECT, MINUS.

```
SELECT <liste d'attributs projetés>
FROM <liste de relations>
WHERE <qualification>
```

Expression de projections

L'obtention de colonnes déterminées s'effectue simplement en omettant de spécifier une qualification (le prédicat absent derrière le WHERE est alors supposé 'vrai' pour tous les tuples).

```
SELECT NOMP, REGION
FROM PLAGE
```

permet d'obtenir l'ensemble des couples (nom_de_plage, région) présents dans la relation PLAGE. Cependant l'élimination des doubles n'est pas automatique : pour lister l'ensemble des régions distinctes on utilise le mot clé DISTINCT:

```
SELECT DISTINCT REGION
FROM PLAGE
```

Enfin, les mots clés ORDER BY, ASC, DESC permettent de compléter le langage et d'ordonner les résultats pour l'utilisateur. Ainsi, pour obtenir toutes les plages, triées par ordre alphabétique décroissant sur les pollutions et croissant sur les noms de plage, on écrit :

```
SELECT *
FROM PLAGE
ORDER BY POLLUTION DESC, NOMP ASC
```

L'utilisation du "*" permet de demander tous les attributs de la relation.

Expression des restrictions

La restriction consiste à sélectionner l'ensemble des tuples vérifiant le critère placé dans la clause WHERE. Ce critère de restriction est une combinaison de critères élémentaires, connectés à l'aide de AND et de OR et pouvant présenter des NOT. Le critère atomique est de la forme <Attribut θ valeur> ou <Attribut1 θ Attribut2>. Le comparateur θ appartient à l'ensemble {=, <>, >, •, <, •}.

Un bloc permettant d'exprimer la restriction est par exemple:

```
SELECT *
FROM NAGEUR
WHERE QUALITÉ = 'médiocre' OR PRÉNOM <> 'Jean'
```

La forme <Attribut1 θ Attribut2> exprime un critère de restriction s'il s'agit d'une comparaison entre la valeur de l'Attribut1 et la valeur de l'Attribut2 *du même tuple*. Le critère est évalué pour chaque tuple en comparant deux valeurs d'attributs différents du

même tuple. Par exemple, pour connaître les bains pris sur des plages de même identifiant (NP) que celui du baigneur (NN) - sémantiquement peu intéressant! - on écrirait :

```
SELECT *
FROM  BAIGNADE
WHERE NP = NN
```

La sélection mono-relation générale utilise un bloc SQL qui combine une restriction et une projection. Ainsi, pour obtenir les plages de sable de Bretagne, triées par ordre alphabétique décroissant sur les pollutions et croissant sur les noms de plage, on écrira :

```
SELECT NP, NOMP, POLLUTION
FROM PLAGE
WHERE TYPE = 'sable' OR REGION <> 'Bretagne'
ORDER BY POLLUTION DESC, NOMP ASC
```

Expression des jointures

L'expression de questions multi-relations en SQL , c'est-à-dire de questions mettant en œuvre, dans le résultat ou dans la qualification, des informations issues de plusieurs relations, nécessite d'exprimer l'opération de jointure.

SQL propose deux façons d'exprimer la jointure : l'une est véritablement assertionnelle, l'autre reste procédurale et utilise une imbrication de blocs de base (un bloc de base est un ensemble SELECT…FROM…WHERE…).

Exemple : *donner les noms des nageurs ayant pris un bain*

a) Expression assertionnelle :

```
SELECT NOM
FROM NAGEUR,  BAIGNADE
WHERE NAGEUR.NN =  BAIGNADE.NN
```

Pour distinguer deux attributs portant le même nom, ceux-ci sont préfixés par le nom de la relation d'origine

b) Expression procédurale

```
SELECT NOM
FROM NAGEUR
WHERE NN IN  (SELECT NN
```

FROM BAIGNADE)

Cette manière d'exprimer la jointure présente deux inconvénients :

- elle est procédurale et demande par conséquent au système d'exécuter une séquence d'opérations particulières. L'optimisation et l'ordonnancement ne sont plus mis en œuvre par le système ; une question mal posée peut, par conséquen t, entraîner des temps de réponse catastrophiques. Néanmoins certains bons systèmes acceptent une question ainsi formulée, mais la transformer en une assertion de façon à pouvoir ensuite utiliser leur optimiseur.
- elle ne permet pas d'exprimer une 'vraie' jointure : en effet seuls les attributs originaires d'une seule des deux relations peuvent être présents dans le résultat ; il s'agit en fait d'une semi-jointure

Définition : semi-jointure:

la semi-jointure de R1 par R2, notée R1 |X R2, est la jointure de R1 et de R2 projetée sur les attributs de R1.

La semi-jointure peut être considérée comme une généralisation de la restriction. Ceci explique en effet son coût relativement faible : l'opération n'exige qu'une seule passe sur chaque relation. La relation R2 est d'abord lue pour obtenir les valeurs de l'attribut de jointure ; puis la relation R1 est restreinte avec les valeurs de l'attribut de jointure apparaissant dans R2. Chacun des tuples de R1 et R2 n'est lu qu'une seule fois.

L'utilisation de blocs imbriqués dans SQL permet cependant d'exprimer simplement des prédicats quantifiés ($\forall$, $\exists$) grâce aux sous-questions (voir § IV.1.3).

Remarque: restriction et auto-jointure

La présence de prédicat de la forme <Attribut1 θ Attribut2> dans une qualification peut correspondre soit à un prédicat de jointure, soit à un prédicat de restriction. En fait, la question est de savoir si la comparaison porte sur la valeur des attributs 1 et 2 d'un même tuple ou de tuples différents. Si l'Attribut1 et l'Attribut2 sont préfixés par une variable relation identique (ou ne le sont pas dans les cas où il n'y a pas d'ambiguité sur la relation), il s'agit des valeurs du même tuple. C'est donc d'un critère de restriction. Si l'Attribut1 et l'Attribut2 sont préfixés par des variables relations différentes, il s'agit de la comparaison d'une valeur de l'Attribut1 avec toutes les valeurs de l'Attribut. C'est donc d'un critère de jointure.

Question :

```
SELECT NN
FROM NAGEUR
WHERE NOM = PRENOM
```

exprime la requête *'quels sont les nageurs dont le nom est identique au prénom ?'* (restriction);

en explicitant par une variable N1 le lien attributs- relation, la question devient :

```
SELECT N1.NN
FROM NAGEUR N1
WHERE N1.NOM = N1.PRENOM
```

alors qu'en écrivant:

```
SELECT N1.NN
FROM NAGEUR N1, NAGEUR N2
WHERE N1.NOM = N2. PRENOM AND N1.NN <> N2.NN
```

on exprimerait *'quels sont les nageur ayant pour nom le prénom d'un autre nageur ?'* (auto-jointure). Deux variables distinctes N1 et N2 sont déclarées cette fois-ci sur la relation NAGEUR. SQL, on le voit, oblige a plus de précision que la langue "naturelle".

IV.1.3. Sous-questions, prédicats quantifiés, composition de questions

L'utilisation de blocs imbriqués permet d'exprimer simplement et de façon très compréhensible des expressions quantifiées. Le bloc interne est alors une *sous-question.*

Définition : sous-question

Une sous question est une question SQL qui rend une seule colonne

Cette colonne est un ensemble de valeurs qui constitue l'argument d'un prédicat IN, ALL, ANY ou EXISTS du bloc externe (dans le cas du EXISTS la sous-question peut rendre un nombre quelconque de colonnes).

Illustrons ces possibilités sur des exemples :

a) Prédicat IN : *quels sont les nageurs qui se sont baignés (au moins une fois) sur une plage très polluée ?*

```
SELECT NN FROM BAIGNADE
WHERE  NP   IN    (SELECT NP FROM PLAGE
                   WHERE POLLUTION = 'élevée')
```

IN est le test de présence de la valeur d'un attribut dans un ensemble de valeur du même type. On l'utilise pour exprimer la semi-jointure.

b) Prédicat EXISTS : *quels sont les nageurs qui se sont [ne se sont pas] baignés en 89 ?*

```
SELECT *
FROM NAGEUR N
WHERE [NOT] EXISTS (SELECT *  FROM BAIGNADE
                    WHERE DATE BETWEEN '01-JAN-89' AND '31-DEC-89'
                    AND B.NN = N.NN )
```

c) Prédicat ALL : *quelle est la plus longue baignade ?*

```
SELECT *
FROM BAIGNADE
WHERE   DUREE  >=   ALL   (SELECT DISTINCT DUREE
                           FROM BAIGNADE)
```

d) Prédicat ANY : *quels sont les noms des nageurs qui se sont baignés plus longtemps que Dupond ? (qui se sont baignés au moins une fois plus longtemps qu'au moins une des baignades de Dupond)*

```
SELECT N1.NOM
FROM NAGEUR N1, BAIGNADE B1
WHERE N1.NN = B1.NN
AND   B1. DUREE  >  ANY   (SELECT DISTINCT DUREE
                           FROM BAIGNADE B2, NAGEUR N2
                           WHERE B2.NN = N2.NN
                           AND N2.NOM = 'Dupond')
```

Enfin, on peut composer logiquement - par union, intersection, différence, cf. § III.2.1 - deux questions SQL. Si, par exemple, on voulait comparer notre table de nageurs à une table d'élèves qui aurait le schéma ELEVE(NE, NOM, PRENOM, AGE) alors les expressions SQL suivantes sont possibles - et leur sens est identique à celui des expressions algébriques équivalentes :

```
SELECT NOM , PRENOM FROM ELEVE   UNION       SELECT NOM , PRENOM FROM NAGEUR
                                 INTERSECT
                                 MINUS
```

IV.1.4. Fonctions et groupements

La norme SQL présente un ensemble de fonctions prédéfinies : COUNT, SUM, AVG, MAX et MIN. Ces fonctions opèrent sur une collection de valeurs scalaires d'une colonne (éventuellement plusieurs pour COUNT) d'une relation.

COUNT	nombre de valeurs de la colonne ;
SUM	somme des valeurs de la colonne (argument de type numérique) ;
AVG	moyenne des valeurs de la colonne (argument de type numérique) ;
MAX	plus grande valeur de la colonne ;
MIN	plus petite valeur de la colonne.

Cette collection de valeurs scalaires d'une colonne est obtenue en effectuant un agrégat.

Définition : agrégat

Un agrégat est un partitionnement horizontal (les tuples sont répartis en plusieurs groupes) d'une relation selon des valeurs d'attributs, suivi d'un regroupement par une fonction de calcul (somme, moyenne, minimum, maximum, compte…)

Les fonctions s'appliquent sur un agrégat défini par les attributs figurant derrière un mot clé GROUP BY, et après application d'un éventuel critère figurant derrière un mot clé WHERE.

La clause HAVING permet d'appliquer un critère de restriction sur des groupes.

Exemple :

Donner, par numéro de plage, la durée moyenne et la somme des durées des baignades effectuées après le 1/1/88, lorsque les durées cumulées dépasse 200 minutes.

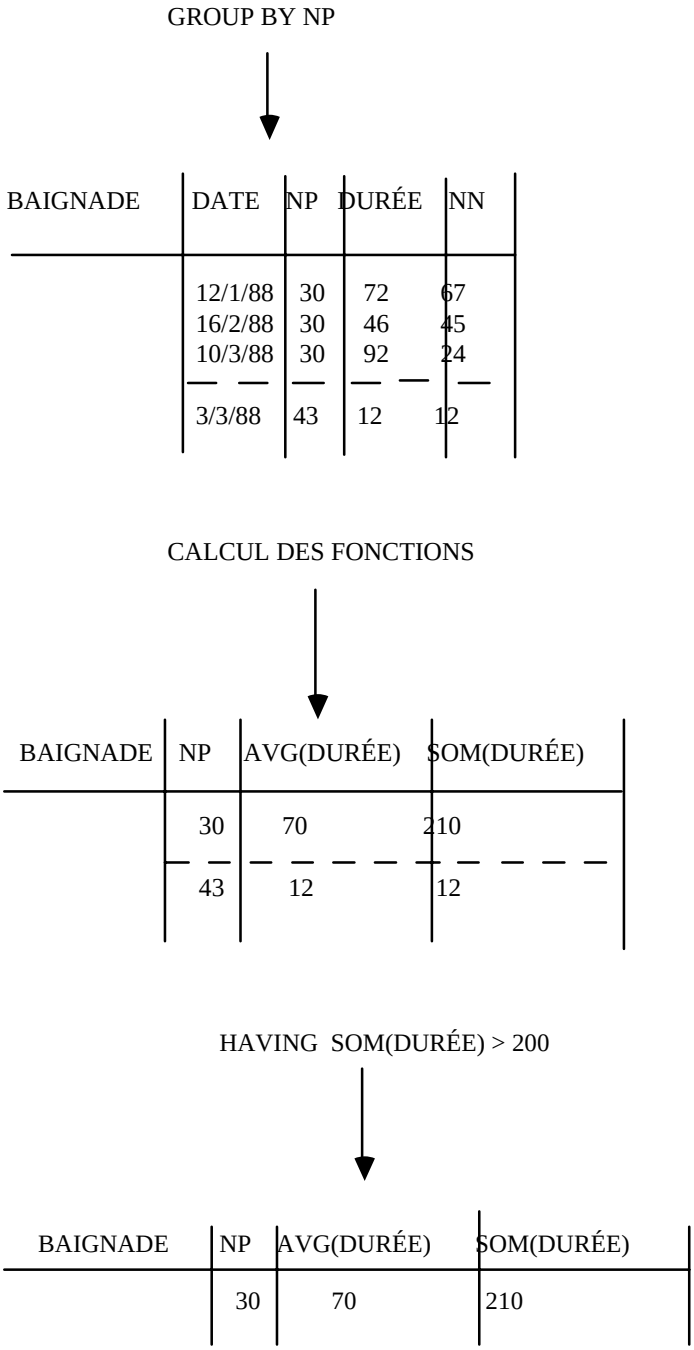
```
SELECT NP, AVG(DURÉE), SUM(DURÉE)
FROM BAIGNADE
WHERE DATE • '01-JAN-88'
GROUP BY NP
HAVING SUM(DURÉE) > 200
```

Le système exécute d'abord la restriction, puis effectue le regroupement, le calcul des fonctions et la restriction sur le résultat du calcul ; exemple exécution de la question précédente :

BAIGNADE	DATE	NP	DURÉE	NN
	12/1/88	30	72	67
	2/12/87	43	24	24
	16/2/88	30	46	45
	3/3/88	43	12	12
	10/3/88	30	92	24

WHERE DATE • 01/01/88

BAIGNADE	DATE	NP	DURÉE	NN
	12/1/88	30	72	67
	16/2/88	30	46	45
	3/3/88	43	12	12
	10/3/88	30	92	24



IV.1.5. Mises à jour

SQL propose trois clauses correspondant aux différents types de mises à jour possibles sur une base de données : l'insertion (clause INSERT), la modification (clause UPDATE) et la suppression (clause DELETE).

Insertion

Une insertion est l'ajout d'un ou plusieurs tuples dans une relation. On peut insérer, par la commande INSERT, des tuples explicitement mais un par un; le mot-clé VALUES précède les valeurs de ses attributs.

```
INSERT INTO NAGEUR
VALUES (145, 'BOUJUS', 'Jacques', 'excellent')
```

On peut également insérer directement un ensemble de tuples. Cet ensemble de taille quelconque doit être le résultat d'une requête ensembliste. La seule condition est que le schéma de la relation résultat (nombre et types des attributs) soit le même que celui de la relation dans laquelle on ajoute des valeurs. On parle dans ce cas d' *insertion calculée.*

La puissance d'expression d'un langage relationnel s'affirme ici clairement. Ainsi la requête :

```
INSERT INTO BAIGNADE
SELECT  '10', NP, DATE, DUREE
FROM BAIGNADE
WHERE NN = '20'
```

ajoute pour le nageur 10 les baignades effectuées par le nageur 20.

Mise à jour

L'aspect ensembliste du langage est ici encore particulièrement utile pour les applications. Accéder aux tuples un par un n'est pas nécessaire. Il suffit d'exprimer correctement la condition qui conduit à la modification.

Par exemple :

```
UPDATE PLAGE
SET POLLUTION = 'élevée'
WHERE REGION = 'Bretagne'
 AND   NP   IN  ( SELECT NP FROM BAIGNADE
                  GROUP BY NP, DATE
                  HAVING COUNT (NN) > 5000 )
```

permettra de prendre en compte le fait qu'une forte fréquentation des plages de Bretagne (les plages ayant vu plus de 5000 baignades en un jour !) entraîne une pollution élevée.

Suppression

Enfin, la suppression ensembliste s'exprime elle aussi très simplement. On supprime directement les mauvais nageurs considérés comme noyés au delà de 300 minutes dans l'eau.

```
DELETE FROM NAGEUR
WHERE QUALITE = 'exécrable'
```

```
AND   NN   IN  ( SELECT NN FROM BAIGNADE
                WHERE DURÉE > 300 )
```

ou, dans un exemple plus sérieux, on supprime les baignades de Dupond comme ceci :

```
DELETE FROM BAIGNADE
WHERE NN IN ( SELECT NN FROM NAGEUR
                WHERE NOM = 'Dupond' )
```

IV.1.6. Le langage de déclaration de données

Rappelons qu'une base de données est définie par un schéma. Ce schéma est composé d'un ensemble de relations. La création d'une nouvelle relation (vide) est réalisée par l'instruction :

```
CREATE TABLE nom_de_relation ( liste de noms_d'attributs )
```

Chaque nom d'attribut a un type (CHARACTER, NUMERIC, DECIMAL, INTEGER, SMALLINTEGER, FLOAT, REAL). Une contrainte NOT NULL ou UNIQUE peut, de surcroît, être imposé à un attribut . Déclarer une clé utilise simultanément ces deux contraintes.

Exemple :

```
CREATE TABLE PLAGE      (   NP INTEGER NOT NULL UNIQUE,
                            NOMP CHARACTER(30) NOT NULL,
                            REGION CHARACTER (30),
                            POLLUTION CHARACTER (20)  )
CREATE TABLE NAGEUR     (   NN INTEGER NOT NULL UNIQUE,
                            NOM CHARACTER(15) NOT NULL,
                            PRENOM CHARACTER(15),
                            QUALITÉ CHARACTER(20)  )
CREATE TABLE BAIGNADE   (   NN INTEGER NOT NULL,
                            NP INTEGER NOT NULL,
                            DATE CHARACTER(8) NOT NULL,
                            DURÉE INTEGER ,
                            UNIQUE (NN, NP, DATE)  )
```

Remarquez l'utilisation de la clause UNIQUE dans la création de BAIGNADE : il s'agit ici de déclarer une clé multi-attributs (il n'existe pas deux triplets identiques de valeurs NN, NP, DATE). Chaque attribut NN, NP et DATE est déclaré NOT NULL, puis l'unicité du triplet est déclarée.

La clause CREATE TABLE est beaucoup plus complète que nous ne l'indiquons ici : elle permet notamment de préciser d'autres contraintes d'intégrité, des droits ainsi qu'un certain nombre de contraintes concernant l'implantation physique de la relation.

IV.1.7. Intégration d'un langage de manipulation de données dans un langage de programmation

L'intégration de SQL dans un langage de programmation apparaît très vite indispensable lorsqu'on considère l'utilisation, par des programmes d'application, des informations stockées dans la BD relationnelle. Deux approches sont possibles : la première consiste à intégrer le langage de manipulation de données dans un langage de programmation quelconque, considéré comme un langage hôte. La seconde, plus ambitieuse, étend un langage de programmation donné avec des clauses spécifiques de la manipulation des données. Elle est restée jusqu'à présent du domaine de la recherche, mais le problème, bien réel, de la "cohabitation" entre un langage de programmation procédural, privilégiant un traitement unaire et un langage de manipulation ensembliste des données, redevient d'actualité avec les Bases de Données Orientées Objet. L'approche Objet tente, en effet, de réconcilier données et traitements.
La démarche dominante d'intégration de SQL est précisée jusque dans la norme sous le nom de *Embedded SQL*. Dans cette approche, les attributs des relations manipulées sont d'abord déclarés comme des variables du langage de programmation. Un programme hôte comporte des instructions standard du langage, une section de déclaration, un ensemble de définitions de curseurs et un ensemble de traitements SQL.
L'implantation d'un Embedded SQL est généralement effectuée à l'aide d'une technique de précompilation. Le précompilateur analyse des commandes SQL et insère des appels de sous-programmes du langage hôte à destination du SGBD (cela suppose l'existence d'une bibliothèque de primitives de bas niveau entre le langage hôte et le SGBD). Il recopie telles quelles les instructions du langage hôte. Le précompilateur génère en sortie, on obtient un programme en langage hôte qui est compilable par le compilateur de ce langage.

Nous donnons ici un exemple en C-Embedded SQL d'une mise à jour sur l'attribut POLLUTIONde la relation PLAGE :

```
#include <stdio.h>
EXEC SQL    INCLUDE SQLCA.H ;
main (argc,argv)
                    int argc ;
                    char *argv[]; {
```

```
                char *answer[1];
EXEC SQL        BEGIN DECLARE SECTION ;
                VARCHAR db_connect_string[32];
                VARCHAR xx[32];
                VARCHAR yy[16];
                VARCHAR zz[16];
EXEC SQL        END DECLARE SECTION ;
strcpy(db_connect_string, argv[1]);
EXEC SQL        CONNECT : db_connect_string;
EXEC SQL        DECLARE C1 CURSOR FOR
                SELECT nomp, pollution FROM  plage WHERE region = :yy
                FOR UPDATE OF pollution;
    while ( printf("\nRégion? (\0 pour fin): ") && scanf("%s", zz) &&
strcmp(zz,'\0') != 0) {
    EXEC SQL OPEN C1 ;
        while (sqlca.sqlcode = 0) {             /* la structure sqlca est dans SQLCA.H*/
        EXEC SQL   FETCH C1 INTO :xx, :zz;
        printf("\n%32s, qui était %16s, devient-elle Alarmante? (O|N)", xx, zz) ;
        scanf("%s", answer) ;
                if (answer == 'O')
                EXEC SQL        UPDATE PLAGE
                        SET pollution = 'Alarmante'
                        WHERE CURRENT OF C1 ;
        } /* end of while cursor open */
    EXEC SQL CLOSE C1 ;
    EXEC SQL    COMMIT WORK ;
    } /* end of while Région indiquée */
} /* end of main */
```

Commentaires

- un traitement SQL est annoncé par EXEC SQL et terminé par un caractère point-virgule :
- toutes les variables hôtes référencées dans un traitement SQL sont définies ; dans une "embedded declare section", délimitée par un BEGIN DECLARE SECTION et un END DECLARE SECTION ;
- le type des vari ables hôtes doit être compact avec celui des attributs qui les accueillent ;
- tout programme SQL doit inclure une variable hôte appelé SQLCODE. A chaque exécution d'un traitement SQL, l'indicateur numérique SQLCODE, renvoie le résultat de la requète SQL (OK, ′OK, fin curseur).

- un traitement SQL inclut des références aux variables hôtes. Celles-ci doivent être préfixées par un nom d'attribut pour les distinguer.

Des *curseurs.* sont utilisés pour traiter des ensembles de tuples dans le langage. Le mécanisme de curseur permet un traitement unaire (tuple à tuple) d'un ensemble de tuples fourni par le SGBD, par le langage hôte, en réponse à une requête SQL. Les curseurs peuvent être ouverts (OPEN C) , fermés (CLOSE C), positionnés en lecture sur un tuple (ordre FETCH) et utilisés comme position de référence (CURRENT OF C).

Remarquez que cette approche, à présent normalisée, est assez riche en possibilités ; elle est cependant relativement lourde à mettre en œuvre et l'on peut s'attendre dans les prochaines années à son abandon progressif au profit des langages de quatrième génération (L4G).

Langage de quatrième génération

Nous terminerons cette section par quelques mots sur les langages de quatrième génération. Concernant directement les développeurs d'applications, ces langages visent à faire gagner du temps et à baisser les coûts d'écriture des programmes. Ils utilisent largement les possibilités des SGBD relationnels à travers une manipulation essentiellement ensembliste des données. Ils ne constituent cependant pas simplement un langage de manipulation de données comme SQL, mais intègrent un ensemble complet de possibilités de traitement en fournissant des structures de contrôle d'un langage structuré de haut niveau (boucle, condition, parcours…). Ils s'efforcent, de surcroît, d'intégrer un ensemble d'outils (générateurs de rapports, de graphiques, d'applications…) souvent disponibles comme des logiciels à adjoindre au SGBD. Orientés vers la programmation visuelle, ils constituent parfois un véritable environnement de développement et sont aujourd'hui l'objet d'une forte demande.

Il ne faut cependant pas perdre de vue que ces langages modernes ne constituent qu'une couche de manipulation des informations stockées dans un SGBD. La qualité, les performances, la puissance d'expression résulte, en dernière analyse, essentiellement à la qualité du SGBD relationnel sous-jacent.

Signalons de surcroît qu'un L4G ne fournit aucun support à une gestion de transactions (voir chapitre 6), nécessaire à tout usage multi-utilisateurs : cette gestion de transactions doit alors être prise en compte au niveau de l'application. Les L4G ne sont pas l'objet d'une définition standard acceptée par tous : à ce jour, le choix d'un L4G particulier condamne l'utilisateur à une dépendance totale envers son fournisseur. Certains L4G génèrent cependant du SQL standard, élément favorable qui peut être pris en compte dans le choix.

IV.2. AUTRES LANGAGES RELATIONNELS

Deux autres langages méritent d'être mentionnés : QUEL et QBE.

IV.2.1. QUEL

Le langage QUEL fut le concurrent malheureux de SQL sur le terrain de la normalisation. Très proche, il constitue le langage natif du prototype INGRES de BERKELEY (INGRES, à présent propose les deux langages QUEL et SQL).
Sa syntaxe oblige à déclarer des variables sur les relations manipulées:

RANGE OF variable IS nom-de-relation

Plusieurs variables peuvent être associées à une relation. Les recherches élémentaires s'expriment par les clauses :

RETRIEVE (liste résultat)
WHERE qualification

Exemple :

donner le nom des nageurs qui se sont baignés à la fois à Deauville et à Bayonne en 1983.

```
RANGE OF P, P', N, B, B' IS PLAGE, PLAGE, NAGEUR, BAIGNADE, BAIGNADE
RETRIEVE N.NOM
WHERE N.NN = B.NN AND B.NP = P.NP AND B.DATE = '1983' AND P.NOM = 'Deauville'
AND N.NN = B'.NN AND B'.NP = P'.NP AND B'.DATE = '1983' AND P'.NOM = 'Bayonne'
```

Seules les recherches avec fonctions et agrégats s'éloignent de SQL par exemple :

```
RETRIEVE AVG (B.DURÉE WHERE B.DATE = '02-07-89')
```

permet de demander la durée moyenne des baignades du 2 juillet 1989.

IV.2.2. Query By Example (QBE)

La langage QBE est conçu comme un langage graphique. Développé par IBM à Yorktown Height, et aujourd'hui commercialisé par IBM. Il est basé sur une mise en œuvre visuelle des questions et des résultats ; il constitue à ce titre un précurseur des langages visuels d'aujourd'hui.
Lorsqu'un utilisateur désire utiliser les informations d'une relation, il en demande l'affichage du squelette, c'est-à-dire le schéma. C'est dans ce squelette qu'il spécifie ses requêtes. Nous illustrons très rapidement les possibilités de QBE par quelques questions simples.

a) *Quelles sont les durées des baignades du baigneur N°10 ?*

BAIGNADE	NN	NP	DATE	DURÉE
	10			P.

b) *Quelles sont les pollutions des plages où s'est baigné Durand ?*

BAIGNADE	NN	NP	DATE	DURÉE
	100	200		

NAGEUR	NN	NOM	PRENOM	QUALITÉ
	100	Durand		

PLAGE	NP	NOMP	TYPE	RÉGION	POLLUTION
	200				P.

on utilise ici la notion de variable exemple (une chaîne quelconque soulignée est un exemple) pour exprimer la jointure entre deux relations. Le "P." indique la colonne demandée en résultat.

c) *Lister les noms des plages de sable moins polluées que la plage n° 10.*

PLAGE	NP	NOMP	TYPE	RÉGION	POLLUTION
	10				Q
		P.	sable		< Q

On a exprimé ci-dessus une auto-jointure.

IV.3. CONCLUSIONS

Les langages relationnels d'aujourd'hui s'articulent tous autour du SQL. Sa standardisation par l'ISO - SQL1 ou SQL89, puis SQL2 ou SQL92 - est en effet incontournable, mais présente des inconvénients et des lourdeurs liées à l'absence, de prise en compte des outils moderne d'interface (menus, multi-fenêtrage, souris…). De surcroît, SQL, malgré son apparente simplicité, présente des subtilités qui le rendent difficilement utilisable par l'utilisateur final. Pour toutes ces raisons, l'ensemble des constructeurs proposent aujourd'hui des outils (easy SQL, générateur de SQL…) destinés à faciliter l'tilisation interactive de SQL.

IV.4. ANNEXE : SEMANTIQUE ET SYNTAXE DE SQL

a) Sémantique de SQL

Afin de bien comprendre la sémantique d'une requête SQL, il faut se rappeler qu'à chaque ligne d'une question correspond une opération. Les opérations sont interprétées dans l'ordre 1/ 2/ 3/ 4/ 5/ 6/.

SELECT <liste d'attributs>	5/ projection finale ;
FROM <liste de relations>	1/ produit cartésien de toutes les relations citées ;
WHERE <qualification>	2/ applications des restrictions sur les tuples (après le produits cartésien, tous les prédicats élémentaires de la qualification sont des prédicats de restriction) ;
GROUP BY <liste d'attribut>	3/ calcul des agrégats et des fonctions ;
HAVING <qualification>	4/ application d'une restriction sur les groupes résultant d'un calcul de fonction ;
ORDER BY <liste d'attribut>	6/ tri du résultat.

Attention : il s'agit de l'interprétation sémantique de la requête SQL ; il ne faut surtout pas en conclure qu'un SGBD relationnel exécute la requête de cette façon.

b) Règles syntaxiques en grammaire BNF

Query expressions

query-exp ::= query-term | query-exp [UNION | INTERSECT | MINUS query-term]

query-term ::= query-spec | (query-exp)

query-spec ::= SELECT [ALL | DISTINCT] selection table-exp

selection ::= scalar-exp-commalist | *

table-exp ::= from-clause
[where-clause]
[group-by-clause]
[having-clause]

from-clause ::= FROM table-ref-commalist

table-ref ::= table [range-variable]

where-clause ::= search-condition

group-by-clause ::= GROUP BY column-ref-commalist

having-clause ::= HAVING search-condition

Search conditions

search-condition ::= boolean-term | search-condition OR boolean-term

boolean-term ::= boolean-factor | boolean-term AND boolean-factor

boolean-factor ::= [NOT] boolean-primary

boolean-primary ::= predicate | (search condition)

predicate ::= comparison-predicate
| between-predicate
| like-predicate
| test-for-null
| in-predicate
| all-or-any-predicate
| existence-test

comparison-predicate ::= scalar-exp comparison {scalar-exp | subquery}

comparison ::= = | <> | < | > | • | •

between-predicate ::= scalar-exp [NOT] BETWEEN scalar-exp AND scalar-exp

like-predicate ::= column-ref [NOT] LIKE atom [ESCAPE atom]

test-for-null ::= column-ref IS [NOT] NULL

in-predicate ::= scalar-exp [NOT] IN
{ subquery | atom [, atom-commalist] }

all-or-any-predicate ::= scalar-exp comparison [ALL | ANY | SOME] subquery

existence-test ::= EXISTS subquery

subquery ::= (SELECT [ALL | DISTINCT] selection table-exp)

Scalar expressions

```
scalar-exp          ::=  term | scalar-exp { + | - } term
term                ::=  factor • term { * | / } factor
factor          ::= [ + | - ] primary
primary             ::=  atom
                         | column-ref
                         | function-ref
                         | ( scalar-exp )
atom                ::=  parameter-ref
                         | literal
                         | USER
parameter-ref       ::=  parameter [ [INDICATOR] parameter
function-ref        ::=  COUNT (*) • distinct-function-ref • all-function-ref
distinct-function-ref ::=  { AVG | MAX | MIN | SUM | COUNT } ( DISTINCT
column-ref )
all-function-ref    ::=  { AVG | MAX | MIN | SUM | COUNT } ( [ALL] scalar-
exp )
column-ref          ::=  [column-qualifier.] column
column-qualifier    ::=  table | range-variable
```

IV.5. REFERENCES

[Date 87] C.J. DATE : *"A Guide to the SQL Standard"* , Addison-Wesley, 1987.

[ISO 86] ISO ANSI, *"Database Language SQL"*, ISO/DIS 9075, Draft International Standard, 1986.

[ISO 88] ISO ANSI, *"Database Language SQL2"* , Working Draft, 1988.

CHAPITRE 5 : BIEN CONCEVOIR UNE BASE DE DONNEES

Nous étidions dans ce chapitre comment bien concevoir une BD. Pour ce faire, à partir d'un même ensemble de connaissances ayant trait aux plages, nous proposons deux choix d'organisation des informations sous forme relationnelle. Nous étudions les qualités et les défauts de ces différents choix avant de présenter les règles de "bonne" conception d'une BD relationnelle.

Premier choix :

BAINS (NN, NOM, PRENOM, QUALITE, DATE, DUREE, NP, NOMP, TYPE, REGION, POLLUTION)

Deuxième choix :

NAGEUR (NN, NOM, PRENOM, QUALITE)

PLAGE (NP, NOMP, TYPE, REGION, POLLUTION)

BAIGNADE (NN, NP, DATE, DUREE)

NN et NP sont des numéros permettant de distinguer respectivement les nageurs et les plages. NN est l'équivalent du numéro de sécurité sociale.

Nous constatons sur cet exemple l'existence de plusieurs façons de structurer un même ensemble d'informations. Si nous privilégions instinctivement l'une des deux solutions proposées, c'est qu'elle correspond davantage à notre perception du monde réel, dans laquelle nous distinguons naturellement certaines entités : les personnes, les plages, etc.
L'étape de conception est primordiale pour le bon fonctionnement d'un SGBD. Elle fait partie des quelques facteurs qui peuvent entraîner des incohérences dans les réponses et une diminution inacceptable des performances du système ; c'est pourquoi il est indispensable d'y attacher une attention toute particulière.
Une solution "instinctive" n'est pas suffisante pour concevoir le schéma d'une base importante. Il est donc nécessaire d'isoler les critères de décision et de formaliser des méthodes de conception des bases de données. Tel est l'objet de ce chapitre.
Les problèmes les plus courants rencontrés dans des bases de données mal conçues peuvent être regroupés selon les critères suivants :

Redondance des données

Certains choix de conception entraînent une répétition des données lors de leur insertion dans la base. Cette redondance est souvent la cause d'anomalies provenant de la complexité des insertions.

C'est, par exemple, le cas de la première organisation proposée : dès qu'une personne prend un nouveau bain, on doit non seulement répéter son numéro qui, par hypothèse,

suffit à le déterminer, mais aussi toutes les informations liées à ce numéro (son nom, son prénom, sa qualité). Au contraire, dans le deuxième choix, seul le numéro indispensable à la distinction d'un nageur est répété. La situation est identique pour les plages.

Incohérence en modification

La redondance de l'information entraîne également des risques en cas de modification d'une donnée répétée en différents endroits : on oublie fréquemment de modifier toutes ses occurrences (en général par simple ignorance des différentes places où figure l'information).

Par exemple, dans la première organisation proposée, si une personne change de nom (cas fréquent lors de mariages), il faut changer ce nom dans tous les tuples où apparaissent ses coordonnées. Dans la deuxième organisation, un seul tuple est modifié.

Anomalie d'insertion

Une mauvaise conception peut parfois empêcher l'insertion d'un tuple, faute de connaître la valeur de tous les attributs de la relation. Pour remédier à ce problème, certains SGBD implantent une valeur non typée qui signifie que la valeur d'un attribut d'un tuple est inconnue ou indéterminée. Cette valeur (appelée usuellement NULL) indique réellement une valeur inconnue et non une chaîne de caractères vide ou un entier égal à zéro (analogie avec un pointeur égal à NIL en Pascal).

Dans le premier schéma proposé, insérer une nouvelle plage où personne ne s'est jamais baigné est aussi impossible.

Anomalie de suppression

Enfin, une mauvaise conception peut entraîner, lors de la suppression d'une information, la suppression d'autres informations, sémantiquement distinctes, mais regroupées au sein d'un même schéma.

C'est ce qui se produit dans notre premier exemple, la suppression d'une plage entraîne automatiquement la suppression de tous les nageurs ne s'étant baignés que sur cette plage.

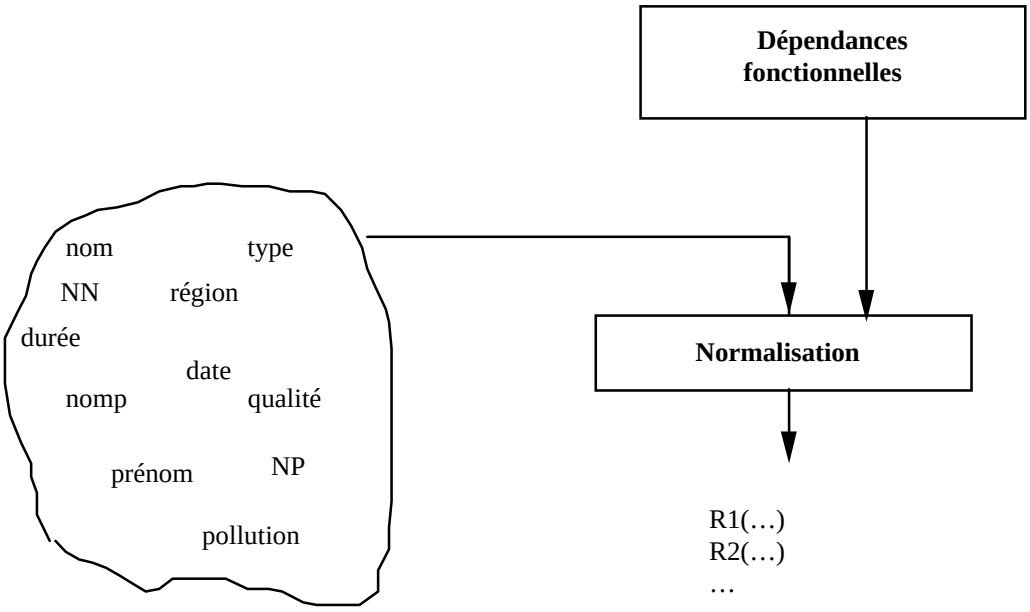
De nombreux travaux ont permis de mettre au point une théorie de conception d'une base de données relationnelle : la théorie de la normalisation, que nous allons maintenant développer.

V.1. LA THEORIE DE LA NORMALISATION

Cette théorie est basée sur les "dépendances fonctionnelles" (DF). Les dépendances fonctionnelles traduisent des contraintes sur les données (par exemple, on décide que deux individus différents peuvent avoir même nom et prénom mais jamais le même numéro NN). Ces contraintes sont représentatives d'une perception de la réalité et imposent des limites à la base.

Les dépendances fonctionnelles et des propriétés particulières, définissent une suite de formes normales (FN). Elles permettent de décomposer l'ensemble des informations en diverses relations. Chaque nouvelle forme normale marque une étape supplémentaire de progression vers des relations présentant de moins en moins de redondance. Ces étapes traduisent une compréhension de plus en plus fine de la réalité.

Chacune de ces formes normales peut être obtenue au moyen d'algorithmes de décomposition. Le point de départ de ces algorithmes est la relation universelle, c'est-à-dire la relation qui regroupe toutes les informations à stocker (dans notre exemple, le premier schéma représente cette relation universelle) ; le but est d'obtenir, en sortie, une représentation canonique des données présentant un minimum de redondance à l'intérieur de chaque relation et un maximum d'indépendance entre les différentes relations .



relation universelle

V.1.1. Les dépendances fonctionnelles

Définition : dépendance fonctionnelle

On dit qu'un attribut B dépend fonctionnellement d'un attribut A si, étant donné une valeur de A, il lui correspond une *unique* valeur de B (quel que soit l'instant t considéré).

Notation : $A \rightarrow B$

Exemple :

La dépendance fonctionnelle NN $\rightarrow$ NOM signifie qu'à un numéro est associé un nom unique (c'est, par exemple, le cas du numéro de sécurité sociale). Remarquons qu'une dépendance fonctionnelle n'est généralement pas symétrique, c'est-à-dire que NN $\rightarrow$ NOM n'interdit pas que deux personnes distinctes (correspondant à deux NN différents) portent le même nom.

Une dépendance fonctionnelle est une propriété définie sur tous les tuples d'une relation et pas seulement sur un tuple particulier. Elle traduit une certaine perception de la réalité (par exemple, deux personnes distinctes peuvent porter le même nom). Elle correspond à une contrainte qui doit être vérifiée en permanence.

Les dépendances fonctionnelles sont parties intégrantes du schéma d'une BD. Elles doivent donc théoriquement être déclarées par l'administrateur et contrôlées par le SGBD.

Exemple :

Nous définissons les propriétés vérifiées par notre base de baignades. Deux personnes distinctes peuvent, par exemple, porter le même nom, le même prénom, et avoir la même qualité de nage. Deux numéros de nageur différent les distinguent l'une de l'autre. Les dépendances fonctionnelles que nous venons de décrire sont donc

NN $\rightarrow$ NOM, NN $\rightarrow$ PRENOM, NN $\rightarrow$ QUALITE

Nous pouvons supposer également que deux plages distinctes ont toujours deux numéros différent ; ce qui implique :

NP $\rightarrow$ NOMP

NP $\rightarrow$ REGION

que la pollution et le type sont propres à une plage :

NP $\rightarrow$ POLLUTION

NP $\rightarrow$ TYPE

et que deux plages d'une même région ne peuvent porter le même nom :

(NOMP, REGION) $\rightarrow$ NP

Propriétés des dépendances fonctionnelles

Les dépendances fonctionnelles obéissent à certaines propriétés connues sous le nom d'axiomes d'Armstrong.

Réflexivité :

$Y \subset X \quad \Rightarrow \quad X \rightarrow Y$

Augmentation :

$X \rightarrow Y \quad \Rightarrow \quad XZ \rightarrow YZ$

Transitivité :

$X \rightarrow Y$ et $Y \rightarrow Z \quad \Rightarrow \quad X \rightarrow Z$

D'autres propriétés se déduisent de ces axiomes :

Union :

$X \rightarrow Y$ et $X \rightarrow Z \quad \Rightarrow \quad X \rightarrow YZ$

Pseudo-transitivité :

$X \rightarrow Y$ et $YW \rightarrow Z \quad \Rightarrow \quad XW \rightarrow Z$

Décomposition :

$X \rightarrow Y$ et $Z \subset Y \quad \Rightarrow \quad X \rightarrow Z$

L'intérêt de ces axiomes et des propriétés déduites est de pouvoir construire, à partir d'un premier ensemble de dépendances fonctionnelles, l'ensemble de toutes les dépendances fonctionnelles qu'elles génèrent. On parle alors de dépendance fonctionnelle élémentaire.

V.1.2. Dépendance fonctionnelle élémentaire

Définition : dépendance fonctionnelle élémentaire

Une dépendance fonctionnelle $X \rightarrow A$ est élémentaire si

- A n'est pas inclus dans X ;
- il n'existe pas X' inclus dans X tel que $X' \rightarrow A$.

Cette notion de dépendance fonctionnelle élémentaire est primordiale car elle permet de construire une sorte de famille génératrice minimale (appelée couverture minimale) des dépendances fonctionnelles utiles pour structurer la base.

Exemple :

Deux plages d'une même région ne peuvent pas porter le même nom (contrairement à deux plages de régions différentes) ; le degré de pollution d'une plage dépend exclusivement de la plage et non de la région. On a alors

$(NOMP, REGION) \rightarrow POLLUTION$

mais on n'a aucunement

NOMP → POLLUTION

ni

REGION → POLLUTION

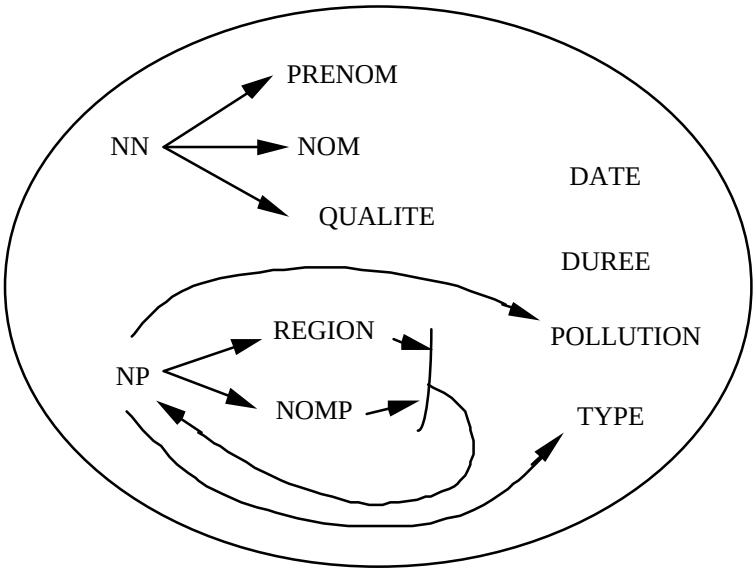
donc (NOMP, REGION) → POLLUTION est une dépendance fonctionnelle élémentaire.

V.1.3. Graphe des dépendances fonctionnelles

C'est une représentation graphique permettant de visualiser aisément toutes les dépendances fonctionnelles et d'isoler les principales (i.e. les DF élémentaires).

Exemple :

Toutes les dépendances fonctionnelles citées précédemment peuvent être représentées de la façon suivante.



V.1.4. Fermeture transitive

Définition : fermeture transitive

La fermeture transitive d'un ensemble de dépendances fonctionnelles est ce même ensemble enrichi de toutes les dépendances fonctionnelles déduites par transitivité.

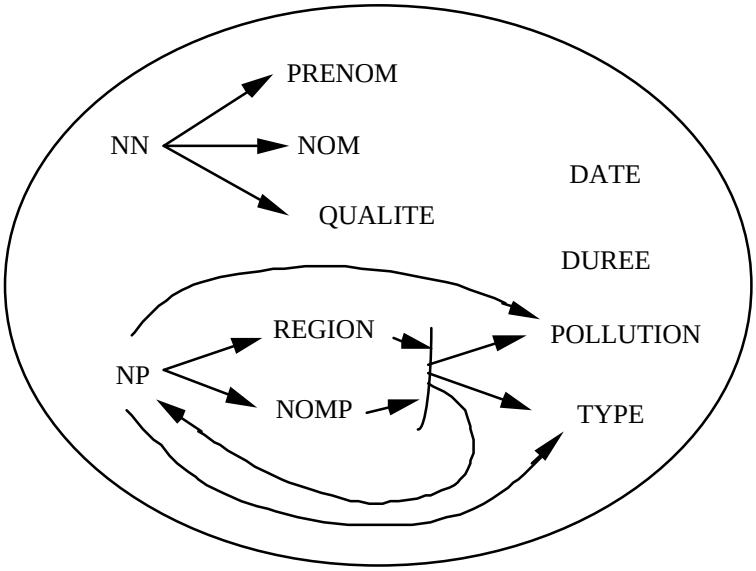
Exemple :

De l'exemple précédent , on déduit par transitivité deux nouvelles dépendances fonctionnelles :

(NOMP, REGION) → TYPE

(NOMP, REGION) → POLLUTION

qui enrichit le graphe de la façon suivante :



V.1.5. Couverture minimale

Définition : couverture minimale

La couverture minimale d'un ensemble de dépendances fonctionnelles est un sous ensemble minimum de dépendances fonctionnelles élémentaires permettant de générer toutes les autres.

Exemple :

L'ensemble des dépendances fonctionnelles suivant :

(NN → NOM, NN → PRENOM, NN → QUALITE, NP → NOMP, NP → REGION, (NOMP, REGION) → POLLUTION, (NOMP, REGION) → NP)

est une couverture minimale de l'ensemble des dépendances fonctionnelles.

Théorème :

Tout ensemble de dépendances fonctionnelles admet une couverture minimale, en général non unique.

Exemple :

L'ensemble des dépendances fonctionnelles suivant :

(NN → NOM, NN → PRENOM, NN → QUALITE, NP → NOMP, NP → REGION, NP → POLLUTION, (NOMP, REGION) → NP)

est une autre couverture minimale de l'ensemble des dépendances fonctionnelles.

La recherche de la couverture minimale d'un ensemble de dépendances fonctionnelles est un élément essentiel du processus de normalisation, dont le but est de décomposer une relation en plusieurs relations plus petites.

V.1.6. Clé d'une relation

Définition : clé d'une relation

Soit R(A_1, A_2, ..., A_N) un schéma de relation, soit F^+ l'ensemble des dépendances fonctionnelles associées à R, soit X un sous-ensemble de A_1, A_2, ..., A_N, on dit que X est une clé de R si et seulement si

- $X \rightarrow A_1, A_2, ..., A_N$;
- il n'existe pas de sous ensemble Y inclus dans X tel que $Y \rightarrow A_1, A_2, ..., A_N$.

Une clé d'une relation est un ensemble minimum d'attributs qui détermine tous les autres.

Exemple :

NN est clé de la relation PERSONNE (NN, NOM, PRENOM, QUALITE) ;

(NOMP, REGION) est clé de la relation (NP, NOMP, REGION, POLLUTION, TYPE).

Plusieurs clés peuvent être candidates pour une même relation.

Exemple :

NP et (NOMP, REGION) sont deux clés candidates à la relation (NP, NOMP, REGION, POLLUTION, TYPE).

Dans l'écriture des schémas de relations, on indique les clés en soulignant les attributs constitutifs.

Exemple :

PLAGE (NP, NOMP, REGION, POLLUTION, TYPE)

V.1.7. Décomposition des relations

La théorie de la normalisation repose sur un principe de décomposition des relations.

Définition : décomposition d'une relation

La décomposition d'un schéma de relation R(A_1, A_2, ..., A_N) est son remplacement par une collection de schémas de relations (R_1, R_2, ..., R_i) telle que :

SCHEMA(R) = SCHEMA(R_1) $\cup$ SCHEMA(R_2) $\cup$ ... $\cup$ SCHEMA (R_i).

Définition : décomposition sans perte

Une décomposition d'une relation R en N relations R_1, R_2, ..., R_N est sans perte si et seulement si, pour toute extension de R, on a :

$R = R_1 |x| R_2 |x| ... |x| R_N$

Théorème : décomposition sans perte

Une décomposition en deux relations est sans perte si l'attribut de jointure de la recomposition est clé d'une au moins des deux relations.

Définition : décomposition préservant les dépendances fonctionnelles

Une décomposition (R_1, R_2, ..., R_N) de R préserve les dépendances fonctionnelles si la fermeture des dépendances fonctionnelles de R est la même que celle de l'union des dépendances fonctionnelles des relations R_1, R_2, ..., R_N.

V.2. LES TROIS PREMIERES FORMES NORMALES

Nous définissons ici des règles de décomposition la relation universelle sans perdre d'information en utilisant les dépendances fonctionnelles. Le but est d'obtenir une représentation du monde réel qui minimise la redondance et les risques d'anomalies lors des mises à jour.

V.2.1. La première forme normale 1FN

Définition : première forme normale

Une relation est en première forme normale si tout attribut est atomique.

Conséquences :

- un attribut représente une donnée élémentaire du monde réel ;
- un attribut ne peut désigner, ni une donné e composée d'entités de nature quelconque, ni une liste de données de même nature.

La première forme normale correspond à un souci de simplicité au niveau du langage de manipulation. Elle a pour particularité d'éviter toute hiérarchie dans une relation en interdisant les domaines composés de plusieurs valeurs.

Exemple :

Nous pourrions imaginer une relation BAINS de schéma BAINS (NN, NP, DATE, DUREES) où DUREES serait la liste des durées des bains pris par le nageur NN sur la plage NP à la date DATE. A lieu de cela, la première forme normale nous oblige à décomposer cette relation en BAINS (NN, NP, DATE, DUREE) ce qui induira autant de tuples qu'il y a de baignade d'un même baigneur, sur la même plage, le même jour ; avec (nn, np, date, durée$_i$) pour le $i^{ème}$ bain de la journée.

V.2.2. La deuxième forme normale 2FN

Définition : deuxième forme normale

Une relation est en deuxième forme normale si et seulement si :

- elle est en première forme normale ;
- tout attribut n'appartenant pas à une clé ne dépend pas q ue d'une partie de cette clé.

Exemple :

Considérons la relation PLAGE de schéma suivant :

PLAGE (NOMP, REGION, TYPE, POLLUTION)

où la clé est (NOMP, REGION). Supposons que la pollution est bien dépendante de la plage mais que le type est, quant à lui, dépendant de la région. La deuxième forme normale nous impose de distinguer deux relations R1 et R2 de schémas respectifs :

R1 (NOMP, REGION, POLLUTION) ;

R2 (REGION, TYPE).

V.2.3. La troisième forme normale 3FN

L'objectif de cette troisième forme normale est l'élimination des redondances dûes aux dépendances fonctionnelles déduites par transitivité.

Définition : troisième forme normale

Une relation est en troisième forme normale si :

- elle est en deuxième forme normale ;
- tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé.

Théorème :

Toute relation R admet au moins une décomposition (R_1, R_2, ..., R_N) en troisième forme normale telle que :

- la décomposition conserve les dépendances fonctionnelles ;
- la décomposition est sans perte.

Exemple :

Considérons maintenant la relation PLAGE de schéma PLAGE (NP, REGION, TYPE, POLLUTION) où la clé est NP. Supposons maintenant comme dans l'exemple précédent que le type est dépendant de la région. La troisième forme normale nous impose de distinguer deux relations R1 et R2 de schémas respectifs :

R1 (NP, REGION, POLLUTION) ;

R2 (REGION, TYPE).

V.2.4. Algorithme de décomposition en troisième forme normale

Cet algorithme accepte en entrée la relation universelle à décomposer et les dépendances fonctionnelles de la relation.

Principe de l'algorithme :

1- à partir du graphe G des dépendances fonctionnelles, **CALCULER** une couverture minimale C ;

2- **EDITER** l'ensemble des attributs isolés dans une même relation (tous les attributs sont clés) ;

3- **RECHERCHER** le plus grand ensemble X d'attributs qui détermine d'autres attributs ;

4- **EDITER** la relation R(X, A_1, ..., A_N) qui est en troisième forme normale ;

5- **SUPPRIMER** toutes les dépendances fonctionnelles figurant dans R du graphe de couverture minimale C ;

6- **SUPPRIMER** les attributs isolés de C (c'est-à-dire les attributs non sources ou cibles de dépendances fonctionnelles) ;

7- **REPETER** l'opération de réduction du graphe C à partir de l'étape 3 jusqu'à ce que C soit vide.

Remarque : Cet algorithme fournit bien une décomposition en 3FN qui préserve les DF mais qui n'est pas nécessairement sans perte. Pour avoir une décomposition sans perte, il suffit d'ajouter à la décomposition finale une relation composée des *attributs formant la clé de la relation universelle* (ou ajouter ces attributs à la relation créée à l'étape 2, si elle existe).

V.2.5. Insuffisance de la troisième forme normale

Il s'agit ici de détecter les boucles pouvant exister dans le graphe des dépendances fonctionnelles. Supposons, par exemple une relation PLAGE indiquant le nom, la région et le canton où se situent les plages. Supposons en outre que deux plages d'une même région ne puissent pas porter le même nom et qu'il n'existe pas deux cantons de même nom. La relation se note :

PLAGE (NOMP, REGION, CANTON)

et possède les dépendances fonctionnelles suivantes :

(NOMP, REGION) → CANTON et CANTON → REGION

Bien que la relation soit en 3ème forme normale, il existe encore des redondances dues au fait qu'un attribut non clé détermine *une partie* de la clé. Afin d'éliminer ce type de redondance, BOYCE et CODD ont introduit une nouvelle forme normale :

Définition : Forme Normale de BOYCE-CODD (BCNF) :

Une relation est en BCNF si et seulement si les seules dépendances fonctionnelles élémentaires sont celles dans lesquelles une clé détermine un attribut. Pour l'exemple précédent une décomposition en BCNF serait :

PLAGE (NOMP,CANTON) et GEO(CANTON,REGION)

Théorème de décomposition en BCNF :

Toute relation admet au moins une décomposition en BCNF qui est sans perte ; cependant, une telle décomposition ne préserve généralement pas les dépendances fonctionnelles. Dans l'exemple précédent on ne préserve pas la première DF.

V.3. LA QUATRIEME FORME NORMALE

La notion de dépendance fonctionnelle nous a conduit à décomposer les relations en troisième forme normale et en forme normale de BOYCE CODD. Ceci est pourtant insuffisant pour éliminer les redondances et les anomalies de mises à jour. Considérons, par exemple, une relation PREFERENCES où nous indiquons, pour chaque nageur, ses différents domiciles et les plages qu'il a l'habitude de fréquenter :

PREFERENCES (NN, VILLE, NP)

Il n'existe aucune dépendance fonctionnelle entre les différents attributs, ce qui conduit à des situations du type :

PREFERENCES	NN	VILLE	NP
	1	Caen	115
	1	Caen	117
	2	Nice	119
	2	Nice	115
	2	Evreux	119
	2	Evreux	115

Dans cette relation, tout est clé : elle est en 3ème forme normale. Pourtant, il subsiste de nombreuses redondances. La relation n'est pas décomposable selon les critères que nous venons d'évoquer, ce qui nous conduit à introduire la notion de dépendance fonctionnelle multivaluée.

V.3.1. Les dépendances multivaluées

Définition : Dépendance Multivaluée

Soit R(A_1, A_2, …,A_n) un schéma de relation. Soient X et Y des sous-ensembles de A_1, A_2, …,A_n. On dit que X ->-> Y (X multi-détermine Y, ou il y a une dépendance multivaluée de Y sur X) si, étant données des valeurs de X, il y a un ensemble de valeurs de Y associées et cet ensemble est indépendant des autres attributs Z = R-X-Y de la relation R.

(X ->-> Y) $\Leftrightarrow$ ((xyz) et (xy'z') $\in$ R $\Rightarrow$ (xy'z) et (xyz') $\in$ R)

Propriété :

Les dépendances fonctionnelles sont des cas particuliers de dépendances multivaluées :

(X$\rightarrow$Y) $\Leftrightarrow$ ((xyz) et (xy'z') $\in$ R $\Rightarrow$ y = y')

De même que pour les dépendances fonctionnelles, il existe des axiomes d'inférence de dépendances multivaluées. Ce sont les suivants :

- Complémentation : (X ->-> Y) $\Rightarrow$ (X ->-> R - X - Y)
- Augmentation : (X ->-> Y) et (V $\subset$ W) $\Rightarrow$ (XW ->-> YV)
- Transitivité : (X ->-> Y) et (Y ->-> Z) $\Rightarrow$ (X ->-> Z-Y)

Dont on peut déduire d'autres règles telles l'union :

- Union : (X ->-> Y) et (Y ->-> Z) $\Rightarrow$ (X ->-> YZ)

A partir de ces axiomes, on introduit la notion de Dépendance Multivaluée Elémentaire :

Définition : Dépendance Multivaluée Elémentaire

Une dépendance multivaluée élémentaire X ->-> Y d'une relation R est une dépendance multivaluée telle que :

- Y n'est pas vide et est disjoint de X ;
- R ne contient pas une autre Dépendance Multivaluée du type X' ->-> Y' telle que X'⊂X et Y'⊂Y.

Ainsi, dans l'exemple précédent nous isolons les Dépendances Multivaluées Elémentaires suivantes :

NN ->-> VILLE et NN ->-> NP

V.3.2. Quatrième forme normale

Définition : quatrième forme normale

Une relation est en quatrième forme normale si et seulement si, lorsqu'il existe une dépendance multivaluée élémentaire, celle-ci est unique.

Propriété :

Toute relation a une décomposition en quatrième forme normale qui est sans perte. Cette décomposition n'est pas forcément unique.

Du fait qu'une dépendance fonctionnelle est un cas particulier de dépendance multivaluée, nous pouvons déduire qu'une relation en quatrième forme normale est en forme normale de BOYCE-CODD et donc en troisième forme normale.

Sur l'exemple que nous venons de voir (PREFERENCES), la clé est l'ensemble des trois attributs et il existe deux dépendances multivaluées : la relation n'est donc pas en quatrième forme normale. Une décomposition de cette relation en quatrième forme normale donne deux relations PREF1 (NN, VILLE) et PREF2 (NN, NP)

V.4. LA CINQUIEME FORME NORMALE

La quatrième forme normale n'est pas encore suffisamment poussée pour éliminer tous les risques de redondances et d'anomalies.

Exemple :

Nos précédents nageurs sont amateurs de fruits de mer et en consomment couramment quand ils vont sur leurs plages préférées. Voyons une extension de la relation CONSOMMATION (NAGEUR, CRUSTACES, VILLE) :

CONSOMMATION	NAGEUR	CRUSTACES	VILLE
	Plouf	Tourteau	Binic
	Plouf	Tourteau	Trouville
	Plouf	Langouste	Trouville
	Coule	Tourteau	Trouville

Nous constatons aisément que cette relation présente de la redondance. Cette redondance provient d'une contrainte d'intégrité implicite dans la conception du monde réel :

"Tout nageur ayant consommé un type de crustacés et ayant séjourné dans une ville les cultivant a consommé de ce type de crustacés dans cette ville."

Ainsi, le fait que Plouf ait mangé du tourteau est répété à plusieurs reprises, etc.

Toutefois, cette relation est en quatrième forme normale car il n'y a pas de dépendance multivaluée. Pour nous en assurer, nous pouvons considérer les projections B1, B2, B3 de la relation CONSOMMATION sur les différents couples d'attributs C1(NAGEUR, CRUSTACES), C2(NAGEUR, VILLE) et C3(CRUSTACES, VILLE), et constater que la jointure de deux de ces relations ne redonne jamais la relation CONSOMMATION. La relation n'est donc pas décomposable en deux autres relations.

V.4.1. Les dépendances de jointure

Le problème provient du fait que nous avons jusqu'alors toujours essayé de décomposer une relation en deux relations. Comme nous allons le montrer ci-dessous, certaines relations sont décomposables, non pas en deux, mais en trois relations. C'est le cas dans notre exemple en raison de la contrainte d'intégrité que nous rappelons ci-dessous :

"Tout nageur ayant consommé un type de crustacés et ayant séjourné dans une ville les cultivant a consommé de ce type de crustacés dans cette ville."

qui se note de façon plus formelle :

(nageur, crustacés) $\in$ C1 et (nageur, ville) $\in$ C2 et (crustacés, ville) $\in$ C3

$\Rightarrow$ (nageur, crustacés, ville) $\in$ CONSOMMATION

Définition : dépendance de jointure

Soit R(A_1, A_2, …, A_n) un schéma de relation et X_1, X_2, …, X_m des sous-ensembles de (A_1, A_2, …, A_n). On dit qu'il existe une dépendance de jointure *(X_1, X_2, …,X_m) si R est la jointure de ses projections sur X_1, X_2, …, X_m, c'est-à-dire si :

$R = \pi_{X_1(R)} |x| \pi_{X_2(R)} |x| \ldots |x| \pi_{X_m(R).}$

Par exemple, la relation CONSOMMATION obéit à la dépendance de jointure suivante :

* (NAGEUR CRUSTACES, NAGEUR VILLE, CRUSTACES VILLE)

Propriété :

Les dépendances multivaluées sont des cas particuliers de dépendances de jointure :

$(X \rightarrow Y) \Rightarrow ((xyz) \text{ et } (xy'z') \in R \Rightarrow y = y')$

V.4.2. Cinquième forme normale

La cinquième forme normale est une généralisation de la quatrième forme normale qui nécessite de prendre en compte les dépendances de jointure induites par la connaissance des clés d'une relation.

Définition : Cinquième Forme Normale

Une relation R est en cinquième forme normale si et seulement si toute dépendance de jointure est impliquée par des clés candidates de R.

V.5. CONCLUSIONS

Dans ce chapitre, nous avons présenté la théorie de la normalisation des schémas relationnels. Cette normalisation est très importante dans la pratique si l'on veut éviter de stocker des informations redondantes. On considère, en général, que la troisième forme normale est suffisante dans les cas courants.

Dépendant de la sémantique des données, le processus de normalisation reste à la charge de l'utilisateur du SGBD. Cette phase, délicate, est aujourd'hui largement facilitée par des outils d'aide à la conception que proposent différents constructeurs.

V.6. REFERENCES

[Codd 74] E.F. CODD, *"Recent Investigations in Relational Database Systems"*, IFIP Congrès 1974, North-Holland Ed., pp 1017-1021.

[Fagin 79] R. FAGIN, *"Normal Forms and Relational Database Operators"*, ACM SIGMOD 1979, Boston, Juin 1979, pp 153-160.

[Nicolas 78] J.M. NICOLAS, *"Mutual Dependencies and Some Results on Undecomposable Relations"*, 5th Very Large Data Bases, Berlin 1978, IEEE Ed., pp 360-367.

CHAPITRE 6 : LA PROTECTION DE L'INFORMATION

Comme nous l'avons évoqué dans le premier chapitre, un SGBD comprend trois niveaux d'abstraction différents. Le troisième chapitre est consacré au niveau conceptuel qui regroupe et gère l'ensemble des informations de l'entreprise. Le prochain s'intéresse aux méthodes de stockage du niveau interne. Nous nous plaçons ici au niveau externe, là où intervient l'utilisateur final dont on doit pouvoir contrôler les actions sur les données de la base.

La démarche consistant à centraliser les données induit un certain nombre de problèmes, tant pour l'utilisateur final d'un SGBD que pour les programmes chargés de contrôler la protection, et la cohérence des informations de la base.

La vision de l'entreprise qu'à l'utilisateur final est limitée à l'ensemble des informations liées à son travail (par exemple, le service du personnel ne se préoccupe pas de la gestion des stocks). Elle ne correspond donc pas au schéma global tel qu'il est décrit au niveau conceptuel. Le SGBD doit prendre en compte ce problème en recréant l'environnement de travail habituel de chaque utilisateur.

En outre, il est indispensable de limiter l'accès des utilisateurs aux informations de l'entreprise. De même que de nombreux systèmes d'exploitation offrent des possibilités de protection sur les fichiers par affectation de droits de lecture, d'écriture ou d'exécution, des moyens d'assurer la confidentialité des informations contenues dans la base doivent exister, dans les SGBD.

Toutefois, à la différence des systèmes d'exploitation, les objets sur lesquels ces droits doivent pouvoir être distribués sont plus variés. L'utilisateur d'un système d'exploitation possède un droit sur l'ensemble d'un fichier. L'exigence typique d'un utilisateur de SGBD est de demander des droits d'accès, non pas à une relation du niveau conceptuel, mais à un ensemble d'informations. Celles-ci sont issues d'un certain nombre de relations de ce niveau. La seule particularité de cet ensemble est sa conformité à un prédicat (cet ensemble de données répond à un critère de sélection).

En outre, la confidentialité n'est pas le seul facteur de contrôle des données d'un SGBD. Inroduisons maintenant la notion d'intégrité des données qui résulte de l'existence de liens sémantiques (appelés contraintes d'intégrité) entre les données de la base, lesquels doivent être vérifiés après chaque modification de la base.

VI.1. LES VUES EXTERNES

Dans les modèles à base de pointeurs (hiérarchique et réseau), sont limitées les possibilités d'accès de l'utilisateur à des sous-schémas stricts du schéma de la base. Un certain nombre

de champs du schéma initial peuvent être cachés ou renommés, mais la structure de l'information reste la même. Il est, par exemple, impossible à l'utilisateur final de définir son propre schéma provenant du "rapprochement" (ou jointure en relationnel) de deux schémas du niveau conceptuel. Ce qui signifie par exemple :

- que l'on peut connaître les noms, prénoms et niveaux de tous les nageurs (masquage du numéro NN et remplacement de QUALITE en NIVEAU) ;
- que l'on ne peut pas connaître les noms et prénoms des nageurs d'excellent niveau (car on ne peut pas isoler les enregistrements des excellents nageurs des autres) ;
- que l'on ne peut pas connaître uniquement les noms de tous les nageurs et les dates auxquelles ces nageurs se sont baignés (impossibilité de changer la structure des enregistrements visibles donc de visualiser un résultat du type jointure dans le modèle relationnel).

La solution idéale à ces nombreux problèmes est de recréer, pour chaque utilisateur, sa propre vision de l'entreprise (sans avoir bien entendu à répéter l'information au sein de la base) en laissant au SGBD le soin de gérer la confidentialité et l'intégrité des données.
Le mécanisme des vues des SGBD relationnels a l'ambition de résoudre un grand nombre de problèmes que les modèles précédents n'ont pas su traiter. Nous verrons successivement les objectifs de ce système et constaterons ensuite la facilité de leur mise en œuvre. Nous évoquerons enfin, les limites de ce mécanisme.

VI.1.1. Les objectifs : voir le monde comme il n'est pas !

Le mécanisme des vues permet de répondre aux besoins suivants :

- adaptation aux applications

  Les applications ne doivent pas dépendre du schéma adopté au niveau conceptuel. L'utilisateur manipule plus naturellement les données sous leur forme habituelle plutôt que de devoir les retrouver et les traiter au sein d'une quantité importante de données parasites.

- intégration des applications existantes

  Les données vues par l'utilisateur étant les mêmes que celles disponibles avant d'avoir accès à un SGBD, il doit pouvoir, grâce à de faibles modifications, aisément adapter ses anciennes applications.

- dynamique du schéma de la base

  Les modifications du schéma conceptuel, parfois nécessaires pour des raisons de performances ou de nouveaux besoins, ne doivent pas impliquer de changements des schémas du niveau externe si ces derniers n'évoluent pas. Si cette règle est respectée, ces modifications n'entraînent pas non plus la réécriture des applications du niveau externe.

- confidentialité et sécurité

  L'administrateur de la base définit les vues et les utilise pour définir précisémment les droits d'accès d'un utilisateur. Ces derniers peuvent ainsi dépendre de la valeur des informations.

- décentralisation de l'administration d'une BD

  Chaque utilisateur ou chaque équipe de développement évoluant au niveau externe doit être responsable des données qu'il gère avec ses propres droits. Il devient administrateur de sa vue et peut déléguer ses droits à un autre utilisateur.

- hétérogénéité des modèles

  Pouvoir présenter un accès homogène et unifié à un ensemble de bases de données de types différents est particulièrement intéressant . Le mécanisme de vue permet cette vision unique.

VI.1.2. Le relationnel simplifie les vues

Le modèle relationnel, grâce à la manipulation ensembliste des données, autorise la définition d'un ensemble particulier de tuples à l'aide d'un critère de recherche. Ces tuples peuvent être composés à partir de plusieurs relations de base. Le résultat de n'importe quelle requête est une relation au même titre que n'importe quelle relation effectivement implantée en machine ; les résultats d'une requête constituent ainsi une relation ordinaire qui peut servir de base à une nouvelle manipulation.
Définir une vue revient alors à déterminer le critère qui permet de rassembler les tuples désirés et de les présenter sous la forme souhaitée. Une telle opération se réalise aisément à l'aide de l'algèbre relationnelle, mais est encore simplifiée par l'utilisation des langages assertionnels des systèmes relationnels actuels.
Dans un système relationnel, la notion d'une vue est donc quasi identique à une interrogation de la base. Attention cependant : la création d'une vue *définit* le critère d'obtention de la vue désirée. Les tuples résultats ne sont pas réellement constitués ; le mécanisme reste entièrement virtuel. Nous allons maintenant en donner quelques exemples.

Exemple 1 :

Créer la vue des nageurs dont la qualité est bonne ou excellente.

```
CREATE VIEW BONS_NAGEURS AS
SELECT NN, NOM, PRENOM, QUALITE
FROM NAGEURS
WHERE QUALITE IN ('Excellente', 'Bonne');
```

Exemple 2 :

Des vues peuvent être définies à partir d'autres vues . Créer la vue des nageurs appartenant à BONS_NAGEURS et dont la qualité est excellente.

```
CREATE VIEW CHAMPIONS AS
SELECT NN, NOM, PRENOM
FROM BONS_NAGEURS
WHERE QUALITE = 'Excellente';
```

Exemple 3 :

On peut aussi créer la vue des mauvais nageurs :

```
CREATE VIEW MAUVAIS_NAGEURS AS
SELECT NOM, PRENOM
FROM NAGEURS
WHERE QUALITE = 'Mauvaise';
```

Exemple 4 :

Des vues peuvent également être crées à partir de critères plus complexes. Par exemple, créer la vue des exploits, c'est-à-dire des nageurs ayant pris des bains de plus de 10 heures. On souhaite conserver des informations sur le nom et le prénom du nageur, le nom de la plage où a eu lieu l'exploit, ainsi que la date et la durée du bain.

```
CREATE VIEW EXPLOITS AS
SELECT N.NN, N.NOM, N.PRENOM, N.QUALITE, P.NOMP, B.DATE, B.DUREE
FROM NAGEUR N, BAIGNADE B, PLAGE P
WHERE N.NN = B.NN
AND B.NP = P.NP
AND B.DUREE > 600mn ;
```

VI.2. MANIPULATION AU TRAVERS DES VUES

VI.2.1. Consultation au travers des vues

L'interrogation exprimée sur une vue est toujours possible. Une vue est une relation virtuelle. Chaque requête portant sur la vue est transformée en une requête portant sur les relations réelles du niveau conceptuel. Celles à partir desquelles la vue a été définie.

Exemple 1 :

Rechercher, dans la vue BONS_NAGEURS, les noms des excellents nageur se prénommant Paul.

```
SELECT NOM
FROM BONS_NAGEURS
WHERE QUALITE = 'Excellente'
AND PRENOM = 'Paul';
```

Cette requête, exprimée au niveau conceptuel, devient :

```
SELECT NOM
FROM NAGEURS
WHERE QUALITE IN ('Excellente', 'Bonne')
AND QUALITE = 'Excellente'
AND PRENOM = 'Paul';
```

Après optimisation elle devient :

```
SELECT NOM
FROM NAGEURS
WHERE QUALITE = 'Excellente'
AND PRENOM = 'Paul';
```

VI.2.2. Influence sur l'optimisation des requêtes

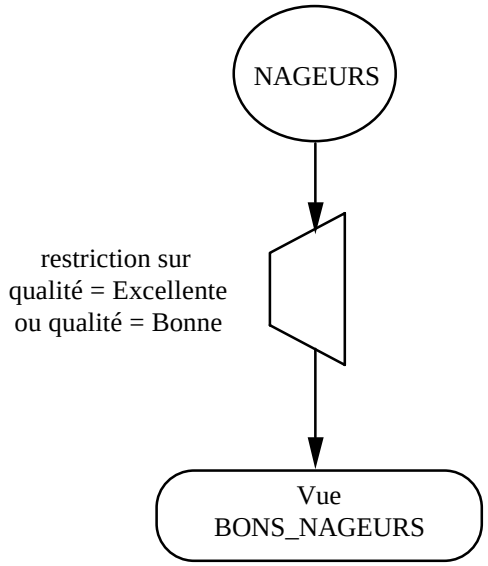
Nous avons déjà mis en évidence dans le modèle relationnel la simplicité du concept de vue et de sa mise en œuvre . Dans ce paragraphe, nous revenons sur cette simplicité en étudiant son influence sur l'optimisation des requêtes.

Les vues sont des relations fictives (ou virtuelle, ou dérivée). Les requêtes formulées sur ces vues sont transformées, en requêtes à appliquer à des relations de base réellement implantées en machine.

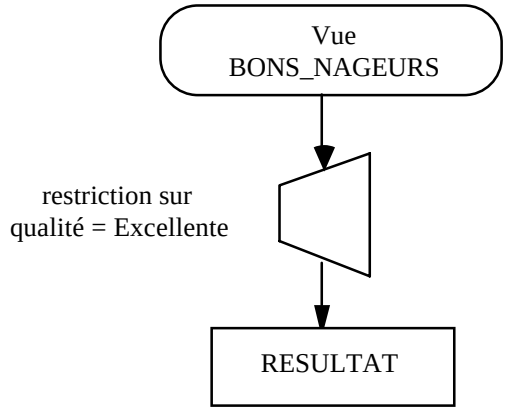
Comment le SGBD transforme t-il le niveau vue en niveau conceptuel ? Il concatène simplement la requête de l'utilisateur à la requête créant la vue et optimise l'ensemble.

Nous visualisons aisément ceci à l'aide des arbres algébriques.

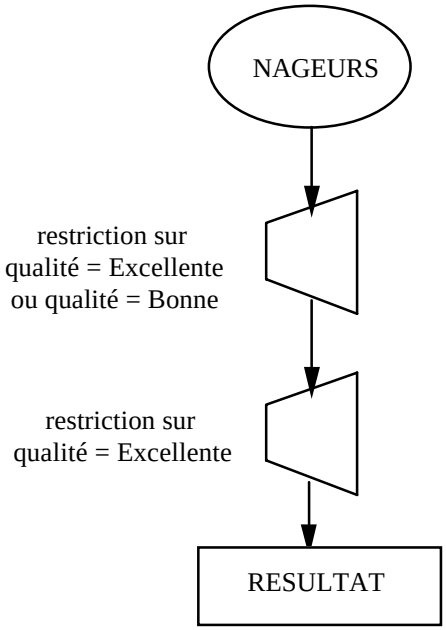
Créer la vue correspond à un arbre algébrique :



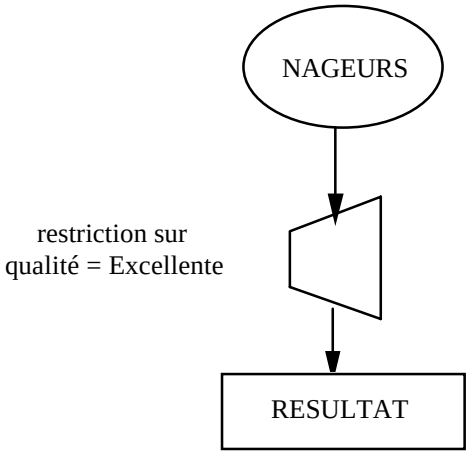
la requête sur la vue correspond également à un arbre algébrique :



La requête transformée finalement appliquée aux relations correspond à un arbre algébrique résultant de la concaténation des deux précédents.



L'optimiseur peut en conséquence modifier l'arbre pour accélérer son exécution.



Cet exemple d'optimisation, ici très simple, ne met en jeu que des restrictions. Cette étape est cependant importante et parfois complexe : mettre bout à bout deux arbres optimisés séparément ne donne pas nécessairement un arbre global optimal. Repasser par l'optimiseur pour déterminer l'arbre optimal (descente des opérateurs unaires, ordonnancement des jointures…) est indispensable.

VI.2.3. Mise à jour à travers les vues

De nouvelles difficultés surgissent lors de la mise à jour. Même si l'utilisateur possède des droits de modification sur la vue, les modifications qu'il souhaite apporter à la base demeurent indéfinies dans de nombreux cas. Voyons quelques exemples ;

Exemple 1 :

Peut-on insérer par la vue BONS_NAGEURS le tuple (73, 'Albert', 'COULE', 'Excellente') ? (on suppose que la clé NN=73 est libre). Cette insertion ne pose pas de problème si l'utilisateur possède un droit d'accès en insertion sur la vue. Tous les champs de la relation conceptuelle NAGEURS dont dépend la vue sont définis. L'insertion peut avoir lieu.

Exemple 2 :

Peut-on insérer à travers la vue CHAMPIONS le tuple (73, 'Albert', 'COULE') ? (on suppose que la clé NN=73 est libre). Ce cas est déjà plus complexe, mais cette vue correspond à une seule valeur de l'attribut manquant QUALITE. L'insertion du tuple devient celle du tuple (73, 'Albert', 'COULE', Excellente') dans la relation NAGEURS.

Exemple 3 :

Voyons maintenant un exemple de modifications. Si la modification au sein de la vue des BONS_NAGEURS du tuple (73, 'Albert', 'COULE', ' Excellente') en (73, Albert', 'COULE', 'Bonne') semble naturelle, transformer ce tuple en (73, 'Albert', 'COULE', 'Mauvaise') est moins évident : la modification qui fait disparaitre un tuple de la vue des BONS_NAGEURS est-elle autorisée? La question de savoir si ce droit doit être donnée ou non est intéressant. Un utilisateur a-t-il le droit de "faire disparaître" des tuples de sa vue ?

Exemple 4 :

Peut-on maintenant insérer par la vue MAUVAIS_NAGEURS le tuple ('Albert', 'COULE') ?

Comme dans l'exemple numéro 2, la valeur de l'attribut QUALITE de la relation NAGEURS. Mais, dans le cas présent, la valeur de la clé, l'attribut NN, doit également être complétée tout d'abord, Albert COULE peut déjà être recensé dans une autre catégorie de la base. L'insertion peut n'être en fait qu'une modification. Par exemple, Albert COULE peut être recensé en tant que bon nageur. Il s'agit alors d'une modification avec les risques que cela comporte. Nous les avons évoqués dans l'exemple précédent.

Albert COULE peut être une personne différente de toutes celles qui sont déjà recensées dans la base. Dans ce cas le traitement doit être l'ajout d'un tuple dans la relation NAGEURS sous un numéro différent.

Exemple 5 :

Enfin, pour insérer un tuple du type (73, 'Albert', 'COULE', NULL) depuis la vue BONS_NAGEURS on ne sait pas si le niveau d'Albert COULE est bon ou excellent. Doit-on accepter une telle insertion ? En effet, pour l'utilisateur de la relation NAGEURS, rien n'indique que Albert COULE est bon ou excellent nageur et non pas de niveau faible ou médiocre.

L'étude de ces exemples nous montre que le mécanisme des vues externes, apporte des avantages en confidentialité et en confort de travail considérables, mais pose des problèmes non résolus dans le domaine des mises à jour. L'utilisateur qui dispose des droits nécessaires, n'est pas certain de pouvoir modifier certaines données auxquelles il a pourtant accès.

Les produits actuels limitent les mises à jour à travers les vues de façon drastique : seules les vues mono-relation sans fonction dont la définition conserve la clé sont concernées. Toutes les autres ne peuvent être manipulées qu'en interrogation.

VI.3. CONFIDENTIALITE

La confidentialité assure la protection des données de la base contre les accès non autorisés.

Elle est généralement obtenue en définissant des autorisations diverses sur les objets susceptibles d'être manipulés par les usagers. Le choix de l'objet détermine la méthode de vérification des droits d'accès. Ces droits sont généralement de plusieurs sortes (lecture, modification et droit d'administration) car ils doivent répondre aux différentes exigences des utilisateurs.

Quelque soit la nature du SGBD, cette protection doit porter sur la plus petite unité d'information accessible par un utilisateur. Le modèle relationnel définie n'importe quel objet (relation, vue, ensemble de tuples, l'attribut d'une relation, d'une vue ou d'un tuple) par un simple critère permettant sa sélection. La définition des droitss sur tout objet, aussi petit soit il est théoriquement possible, mais l'encombrement occasionné par de telles informations nécessite une gestion trop gourmande en performances. Ainsi, la gestion de droits dans le système relationnels d'aujourd'hui est elle réduite aux seules relations et vues.

La plupart des Systèmes de Gestion de Bases de Données offrent, outre les droits habituels de lecture et de modification, des possibilités d'administration comparables aux services offerts par les systèmes d'exploitation. Un utilisateur peut ainsi transmettre ses propres

droits, droits d'administration inclus, à un autre utilisateur. Les SGBD compensent ainsi partiellement les inconvénients de la centralisation qu'ils imposent.

VI.3.1. Autorisations sur une relation ou sur une vue

Les droits accordés - ou autorisations - sur les relations ou vues sont de trois types :

- droit de consultation ;
- droit de modification (mises à jour, suppression, insertion) ;
- droit d'administration (définir les contraintes d'intégrité, attribuer les autorisations, détruire ou modifier le schéma de la relation).

Insistons plus particulièrement sur le droit d'administration qui est la seule contre-partie à la centralisation à outrance des données. Chaque utilisateur retrouve un certain pouvoir de décision : il a le droit de déléguer tout ou partie de ses droits sur les relations créées.
Forcer tout utilisateur à passer par une vue, conduit à considérer que l'utilisateur à des droits uniquement sur ce qu'il voit. Le mécanisme est très souple et permet de définir un droit en fonction d'un critère de sélection. Ainsi, le directeur d'un service ne peut, dans le cas d'une relation EMPLOYE lire que les salaires des membres de son propre service. De nombreuses limitations, dues à l'impossibilité des mises à jours générales à travers les vues, limitent ce mécanisme à des utilisations particulières.

VI.3.2. La cession de droits

Dans le standard SQL, le mécanisme d'attribution et de révocation des droits reste classique : les droits portent les relations de base. La requête permettant de céder ou de retirer tout ou partie de ses droits à un autre usager est la suivante :

GRANT | REVOKE <liste de droits> ON <nom de relation>
TO | FROM <listes d'usagers>
[WITH GRANT OPTION]

Les droits possibles sont les suivants :

SELECT
INSERT
UPDATE (nom d'attribut)
DELETE

Le créateur d'une relation dispose de tous les droits sur la relation créée. Il a le droit d'administration, c'est à dire qu'il peut détruire ou modifier le schéma de sa relation. Il peut donner des droits aux autres usagers par la clause GRANT ; si la clause WITH GRANT OPTION est spécifiée, les droits sont transmissibles en aval. Notons que l'instruction REVOKE retire également les droits transmis en aval sur la relation, c'est à dire qu'un utilisateur peut se voir retirer ses droits, non seulement par celui qui les lui a concédés,

mais aussi par un plus ancien donateur dont il ignorait l'existence... jusqu'au créateur lui-même.

VI.4. INTEGRITE SEMANTIQUE

VI.4.1. Les différents types de contraintes

On dresse ici une typologie possible des contraintes d'intégrité :

Domaine de variation

C'est la définition du domaine auquel appartient la valeur d'un attribut (entier, réel, texte, date…). Chaque donnée de la base doit vérifier cette contrainte qui peut correspondre à une simple concordance de type (c'est le cas de l'entier) ou à une condition plus complexe (c'est le cas du type date qui suppose des connaissances le nombre de jour dans un mois, les années bissextiles, etc.).

Plages de valeurs

C'est la restriction du domaine à un sous-ensemble d'un domaine. Par exemple, on peut imposer qu'un âge soit compris entre 0 et 150 ans, qu'aucun bain ne soit jamais pris entre novembre et mars…

Non nullité

La "nullité" ne correspond pas à un entier de valeur 0 ou à une chaîne de caractères vide, mais, pour tout type, à une valeur indéfinie ou non renseignée lors d'une mise à jour (NULL). Il peut être indispensable d'interdire la nullité à certains champs pour la permettre à d'autres. Par exemple, une clé de relation doit avoir une valeur non nulle, que ce soit un numéro ou une chaîne de caractère, puisqu'elle est le seul moyen de distinguer deux tuples ayant les mêmes valeurs d'attributs.

Unicité

L'unicité est également une notion importante, en particulier pour la définition des clés qui doivent être uniques.

Dépendances fonctionnelles

Les dépendances fonctionnelles sont des contraintes d'intégrité. Par exemple, un numéro de sécurité sociale détermine un tuple définissant un individu.

Dépendances référencielles

Elles indiquent que les valeurs d'un attribut (ou d'un groupe d'attributs) d'une relation R1 doivent déjà être définies dans un autre attribut (ou un autre groupe d'attributs). Par exemple, prendre un bain implique que NN et NP soient respectivement un individu et une plage et qu'ils figurent dans les tables appropriées. Un nageur peut exister sans nécessairement avoir pris de bain, mais un bain ne peut être pris que si le nageur existe.

Contraintes temporelles

Ces contraintes vérifient l'évolution des données par rapport au temps. Par exemple, le salaire affecté à un poste ne peut pas diminuer.

Contraintes avec agrégats

Ces contraintes ne portent pas sur la valeur d'un atome, mais sur une fonction concernant une collection de valeurs d'atomes. Par exemple, la durée moyenne des bains doit être supérieure à deux minutes.

Contraintes générales

Il existe également des contraintes qui n'appartiennent à aucune des catégories définies ci-dessus et sont plus générales. Par exemple : le nombre de bains pris par une personne ne peut pas décroître. Cette contrainte fait intervenir à la fois la notion de contrainte temporelle et de contrainte avec agrégats.

VI.4.2. Comment définir une contrainte d'intégrité ?

Comme nous pouvons le constater, les contraintes qui viennent d'être décrites sont mises en œuvre de différentes façons. Par exemple, les contraintes exprimant les dépendances fonctionnelles servent à la conception du schéma de la base : c'est le processus de conception qui assure automatiquement la vérification des contraintes de dépendance fonctionnelle.

Toutefois, toutes les contraintes ne sont pas liées à l'étape de conception de la base. Le langage de manipulation de données doit donc permettre la définition des contraintes d'intégrités. Ces contraintes peuvent alors être considérées comme des propriétés (assertions) indépendantes les unes des autres. Toute modification de la base de données doit les respecter. Ce formalisme est d'ailleurs le mieux adapté au modèle relationnel.

VI.4.3. Comment regrouper (classer) les contraintes ?

On peut regrouper les contraintes d'intégrité suivant différents critères : leur coût de validation, leurs caractéristiques individuelles (la contrainte s'applique tuple à tuple) ou ensembliste (la contrainte s'applique sur un ensemble de tuples), le fait qu'elles soient spécifiques d'une relation qu'elles en mettent plusieurs en jeu, etc.

VI.4.4. Contraintes d'intégrités et transactions bases de données

La notion de transaction sur une base de données est définie comme un ensemble d'actions qui font passer la base d'un état cohérent à un autre état cohérent (une BD cohérente est une BD où toutes les contraintes d'intégrité sont satisfaites). Trois fonctions d'un SGBD sont concernées : le contrôle de concurrence, la fiabilité et l'intégrité sémantique. Nous ne traitons dans ce paragraphe que le troisième aspect : le respect de règles sémantiques.

Le problème concerne l'instant de vérification des différentes contraintes : à quel moment doit-on vérifier les contraintes d'intégrité ? Quelles sont les contraintes vérifiables au gré des mises à jour de la base. Quelles sont celles qui sont vérifiées à la fin de la transaction ? Voyons quelques exemples.

Exemple 1 :

Considérons la relation NAGEURS (NN, NOM, PRENOM, QUALITE) et imposons la contrainte :

"Tous les nageurs dont le numéro est supérieur à 1000 sont d'excellent niveau."

Cette contrainte est vérifiable à chaque insertion de nouveau tuple ou à chaque modification de la relation NAGEURS.

Exemple 2 :

Considérons maintenant la même relation NAGEURS avec la contrainte d'intégrité suivante (contrainte avec agrégats) :

"Il doit y avoir autant de nageurs d'excellent niveau que de mauvaise qualité."

Supposons que l'on désire insérer dans la base un ensemble de nouveaux nageurs. A chaque insertion de nageur d'excellente ou de mauvaise qualité, la contrainte n'est pas vérifiée. Pourtant, cette même contrainte peut très bien être vérifiée lorsque touts les nouveaux nageurs ont été ajoutés.

Notifier au SGBD que l'ensemble des modifications effectuées précédemment doit être validé et que les contraintes d'intégrité peuvent désormais être vérifiées s'avère donc indispensable . L'ensemble des modifications validées marque une période appelée transaction[6].

VI.5. CONCLUSIONS

Garantir la cohérence de l'information est un apport très important des Systèmes de Gestion de Bases de Données. L'utilisateur est libéré d'une tâche parfois difficile. Le système vérifie lui même des contraintes et des droits associés aux données. Vérifier les contraintes et les droits d'accès dans les PA des anciens systèmes était peu cohérent.

VI.6. REFERENCES

[Ullman 80] J. ULLMAN :*"Principles of Databases Systems"* Computer Sciences Press, 1980.

[6] on développera ce concept au Chapitre 8 ci-après.

[Bancilhon 79] F. BANCILHON : *"Supporting View Updates in Relational Databases"* in Data Base Architectures, Bracchi Nijssen Editors, North Holland Publishing Company, 1979.

CHAPITRE 7 : L'ORGANISATION PHYSIQUE DES RELATIONS DANS UN SGBD

Le modèle relationnel permet une vision logique très simple des relations : des tables, manipulé par un langage facile à utiliser, qui ne spécifient aucune information physique. Une vision naïve de l'implantation physique des relations est possible : chaque relation est un fichier simplement séquentiel de données sur disque. Un tel mode de stockage conduit, à l'évidence, à des performances catastrophiques : l'organisation physique des relations sur disque ne peut se contenter d'une simple organisation séquentielle des informations. Les bases de données, utilisent naturellement des organisations physiques plus sophistiquées. Ces méthodes d'accès, efficaces, ont pour but de déterminer, à partir d'une clé de recherche, l'ensemble minimum des pages de disque à lire.

Les méthodes d'accès sont encore aujourd'hui l'un des éléments les plus mal connus et les plus difficilement maîtrisées par les utilisateurs des systèmes de gestion de bases de données (SGBD). Les raisons sont sans doute liées à la difficulté du problème, mais aussi, à notre avis, à la multiplicité des méthodes et des propositions particulières. L'utilisateur a beaucoup de mal à s'y retrouver et utilise de façon incomplète les possibilités proposées par les systèmes.

Dans un SGBD, le mode d'enregistrement des relations influe sur les performances. En effet, comme dans un livre, les informations sont écrites sur des pages (ici des pages de disque) regroupées selon un certain nombre de règles (les tuples sont, par exemple, triés sur les valeurs d'un attribut particulier ou bien sont regroupés dans une même page après application d'une fonction de hachage…). La connaissance par le SGBD, des règles adoptées de rangement des informations, permet de gagner un temps précieux en n'allant examiner que les pages utiles. Ce mode de rangement est appelé le *placement* ou *clustering*.

L'accès performant aux données est un problème traditionnel du développement des Systèmes d'Information. L'impératif de rapidité face à de gros volumes de données a entraîné, dès les années 60, la mise au point de méthodes évoluées. Elles n'ont fait que se perfectionner et se sophistiquer depuis. L'aboutissement de cette période est marquée par le livre monumental de D.E. Knuth [Knuth 73], qui dresse un panorama complet des méthodes désormais classiques de recherche et d'accès des données. Les organisations classiques de fichiers sont fondées sur les valeurs d'un champ particulier choisi généralement comme identifiant et appelé clé.

Les fichiers d'enregistrements sont des structures adressables : chaque enregistrement est identifié par une clé. Ils supportent deux techniques traditionnelles d'accès : le hachage et les arbres.

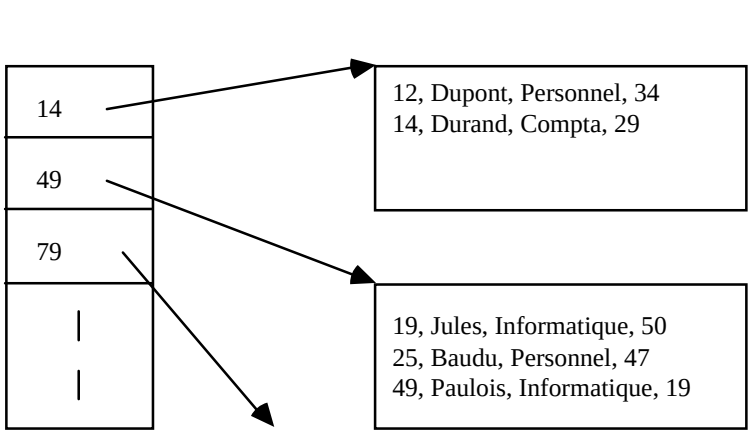
Dans le cas de fichiers statiques, un regroupement par hachage (une fonction de hachage, bien choisie, est appliquée à la clé et détermine la page où est placé l'article) permet de retrouver un enregistrement en un seul accès disque en théorie. L'arbre en requiert toujours plusieurs. Par contre, lorsque le fichier, comme c'est le cas dans des applications de plus en plus nombreuses, est dynamique, i.e. qu'il évolue dans le temps, alors l'arbre se comporte encore relativement bien, tandis que les performances du hachage peuvent devenir très mauvaises. Il peut même être nécessaire de re-hacher tous les enregistrements dans un nouveau fichier. De plus les arbres ont la propriété importante de conserver les articles triés, ce qui n'est pas le cas, en général, du hachage. Nous présenterons dans ce chapitre ces deux techniques de base, puis indiquerons le mode de fonctionnement et d'utilisation de structures d'accès indirect ou index secondaires.

VII.1. LES ARBRES EQUILIBRES

La recherche d'articles utilise ici une structure de données : l'arbre. Il s'agit d'un *index*. L'index constitue une table des matières du fichier :. il associe une clé d'article à l'adresse où est stocké cet article. L'index permet d'obtenir l'adresse de l'article : L'accès à l'article suppose au moins un accès à l'index. L'accès à l'index, comme pour l'article, nécessite un ou plusieurs accès disque. Ces structures de données sont, en général, trop importantes pour être conservées en mémoire centrale.

L'index peut être dense (nombre de clés égal au nombre d'articles dans le fichier) ou non dense (nombre de clés inférieur). Conserver l'ordre des clés permet de former des index non denses. Dans une clé k un index non dense renvoit à l'adresse de l'ensemble des articles présentant les valeurs de clés comprise entre la clé k et la clé suivante de l'index.

Par exemple, pour une relation EMPLOYE (N°, NOM, SERVICE, AGE) :

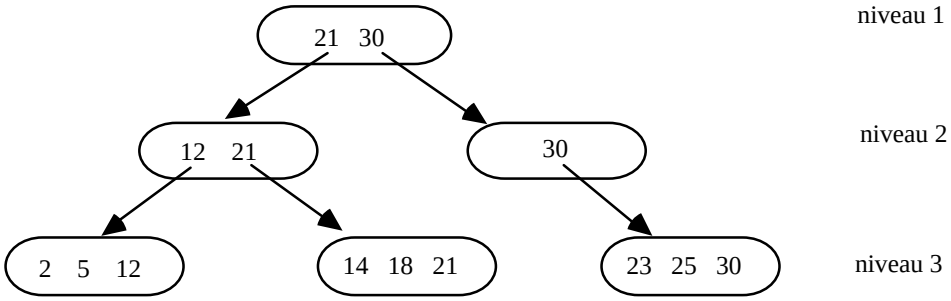


Les possibilités pratiques peuvent être visualisées sur le tableau suivant où sont indiquées quelques méthodes connues (ISAM, VSAM, UFAS, IS3) :

			Fichier	
			trié	non trié
Index	dense	trié	possible	IS3
		non trié		possible
	non dense	trié	ISAM VSAM UFAS	
		non trié		

L'index souvent stocké (au moins en partie) dans une mémoire secondaire,.est lui-même un fichier. Ce dernier peut être organisé en plusieurs niveaux et constituer un index hiérarchique.

Exemple d'un index à 3 niveaux :



Pour mieux caractériser la notion d'index hiérarchisé, et la rendre indépendante des particularités d'implantation, une structure d'arbre comportant un nombre variable de niveaux est introduite. Cette structure est appelée arbre équilibré ou arbres-B (B-Tree) [Bayer 72]. La définition formelle d'un arbre-B est la suivante : B=Balance

Définition : B-Tree

Un arbre balancé est un arbre tel que :

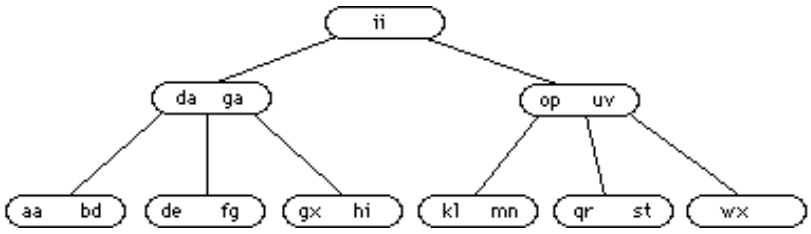
1) chaque chemin depuis la racine vers une feuille est de même longueur h (*hauteur* de l'arbre);
2) chaque nœud contient k clés, triées de gauche à droite, et k+1 pointeurs ;
3) un nœud est soit terminal, soit possède (k+1) fils tels que les clés du ième fils ($1 \leq i, i \leq k+1$) ont des valeurs comprises entre les valeurs des (i-1)ème et ième clés du père ;

4) on appelle *ordre* m d'un B-Tree le nombre de pointeurs utilisés dans un nœud à moitié plein : m = [k+1 / 2] (on le nomme parfois *éventail* - " fan-out " – de l'arbre).

L'utilisation d'une telle structure, a évidemment pour but d'obtenir le plus petit nombre possible de lectures de pages disque. Chaque nœud de l'arbre est stocké dans une page disque. Le nombre de niveaux de l'arbre constitue ainsi la borne maximum du nombre d'accès disque nécessaire pour un enregistrement quelconque.

Le principe de ces structures repose sur la comparaison de la valeur des clés. Les nœuds de l'arbre contiennent des clés triées et des pointeurs vers des nœuds fils. Lors de la recherche d'une clé k, cette dernière est d'abord comparée à chaque clé du nœud racine ; ce qui permet de choisir une branche vers un nœud de niveau inférieur où l'opération est renouvelée ; et cela jusqu'à trouver l'enregistrement de clé k ou atteindre les feuilles.

Les valeurs contenues dans un nœud partagent les valeurs de clé possibles en un nombre d'intervalles égal au nombre de branches.



L'arbre équilibré peut contenir uniquement les index ou l'ensemble index- articles. Dans le premier cas, les articles sont stockés dans un fichier séparé (méthode IS3 de IBM). Les articles ne sont pas triés. L'index est alors dense. Dans le second, les articles sont également stockés dans l'arbre ; les clés sont placées aux niveaux supérieurs et les articles dans les feuilles : on parle de B^+ *Tree* ; c'est l'organisation séquentielle indexée régulière (VSAM de IBM, UFAS de CII) ; les articles sont triés et l'index n'a pas à être dense.

L'insertion et la suppression d'une clé dans l'arbre sont des opérations relativement complexes et coûteuses. Il est souvent nécessaire de réorganiser l'arbre pour qu'il reste équilibré. Cependant une fois cette réorganisation (ré-équilibrage) effectuée, les performances restent inchangées.

Conserver l'ordre des articles permet sans utiliser l'index un accès séquentiel très rapide aux données. Cette possibilité est importante pour un grand nombre d'applications.

Les deux avantages des arbres B sont donc leur bon comportement pour des fichiers dynamiques et le stockage ordonné des articles qui permet deux types d'accès, sélectif et séquentiel. Par contre, l'utilisation d'une structure de données volumineuse qui nécessite plusieurs accès disque (typiquement 3 à 5 E/S pour l'accès à un article) rend les B-Trees peu attrayants lorsqu'accéder à des performances élevées sont recherchées.

VII.2. LE HACHAGE

Pour rechercher une clé k, nous avons jusqu'à présent considéré des méthodes comparant les valeurs de la clé k, donnée comme argument de la recherche, avec d'autres clés déjà enregistrées dans des structures de données (i.e. arbres). Une autre méthode, nommée hachage (ou hash-coding) permet d'éviter toutes ces comparaisons.

VII.2.1. Principe

Le principe de base du hachage est très simple : Un calcul arithmétique est effectué sur la clé K pour déterminer l'emplacement physique de l'article correspondant.

Cette méthode, comparée aux méthodes précédentes, est supérieure, à la fois du point de vue de l'espace mémoire (pas de structure de données supplémentaire) et de la vue vitesse (calcul arithmétique plus rapide qu'une succession de comparaisons). Malheureusement, trouver des fonctions qui déterminent des emplacements différents pour des clés différentes, en évitant de laisser trop d'adresses inutilisées n'est pas facile.

Les fonctions qui évitent de déterminer des valeurs en double sont étonnamment rares. KNUTH rapporte dans son livre " The Art of Computer Programming" le " birthday paradox" qui affirme que si 23 personnes ou plus se trouvent réunis dans une pièce, il y a une chance sur deux pour que deux d'entre elles aient le même jour et le même mois de naissance ! En d'autres termes, si nous voulons qu'une fonction aléatoire transforme et place 23 clés dans une table de 365 emplacements, la probabilité qu'il n'y ait pas deux clés au même emplacement est inférieure à 1/2 (0.49 exactement). Le " birthday paradox" nous indique qu'il y aura probablement des clés distinctes (ki • kj) dont le hachage donnera la même valeur h(ki) = h(kj). Un tel événement est appelé une collision. Plusieurs possibilités existent pour résoudre le problème des collisions.

Le hachage consiste donc à choisir la meilleure fonction h possible (c'est-à-dire qui minimise les collisions) et à sélectionner une méthode pour résoudre, le plus économiquement possible, les collisions.

Résoudre une collision consiste à déterminer une adresse à partir d'une information fournie. Cette information est un champ particulier de l'enregistrement : la clé. L'adresse obtenue est appelé le hash-code. La fonction utilisée peut être quelconque. Une bonne fonction est choisie en fonction de la distribution des clés. Elle doit satisfaire deux critères parfois contradictoires :

1) son calcul doit être très rapide ;
2) elle doit minimiser les collisions.

VII.2.2. Fonctions de hachage

Nous ne citerons ici que quelques fonctions courantes. La fonction qu'on peut considérer comme standard est la méthode par division. Elle est particulièrement simple, et utilise le reste de la division par M, M étant le nombre de valeurs que peut prendre la fonction h(k).

h(k) = k mod M

Certaines valeurs de M sont, dès la première approche, meilleures que d'autres. Par exemple, si M est un nombre pair, h(k) restera pair si k est pair, et impair si k est impair. Cela peut déterminer une mauvaise répartition dans de nombreux cas. Pour obtenir une bonne répartition, quelle que soit la distribution des clés, M doit être un nombre premier.

Le traitement suivant peut s'appliquer aux chaînes de caractères :

1) traiter la valeur de la clé comme une séquence de bits, formée par concaténation des bits de chaque champ de la clé
2) découper la séquence en tranches d'un certain nombre de bits
3) additionner les groupes de bits considérés comme des entiers
4) diviser la somme par le nombre premier M et utiliser le reste.

Si certaines parties de clé sont plus significatives que d'autres, on utilise la somme de ces parties ("pliage", certaines parties sont repliées sur d'autres pour être sommées). On peut également, dans le cas de distributions particulières (fréquence des caractères dans une langue par exemple) utiliser une table de conversion à n entrées (les n valeurs de clé) et M sorties ; si la table a une taille importante, la technique est cependant pénalisante.
D'autres fonctions pseudo-aléatoires plus ou moins complexes peuvent être utilisées (pour plus de détails voir [Knuth 73]).

VII.3.3. Résolution des collisions

La façon la plus simple de résoudre les collisions est sans doute de maintenir M listes, une par valeur possible du hash-code. Chaque liste est composée de couples article-pointeur et il existe M têtes de liste, numérotées de 1 à M. Après le calcul du hash-code sur la clé, la recherche se résume à un parcours séquentiel de la liste sélectionnée. On peut, en outre, gérer des listes triées.
Cette résolution par "chaînage" est souvent une idée simple et excellente. Les insertions et les recherches infructueuses sont rapides.
Un chaînage plus sophistiqué, dit chaînage imbriqué, est également réalisable. Il permet de mieux occuper l'espace (de nombreuses têtes de liste restent vides si augmenté est pour diminuer la taille moyenne des listes). Dans cette approche, chaque emplacement de la table a un champ pointeur qui indique l'emplacement de la table où se trouve l'article suivant de la liste. Lors d'une insertion, si la place est occupée, on insère à la première

place libre (par ordre croissant ou décroissant des adresses), puis les pointeurs sont mis à jour.

Dans le même esprit, la méthode d'*adressage ouvert* permet de ne plus utiliser de pointeur du tout. La politique d'insertion est simplement d'occuper la première place libre. Le gain de place est important, l'insertion très simple. La recherche, en revanche, se complique très sérieusement : si l'article de clé K n'est pas trouvé en h(K), on ne peut conclure à son absence dans le fichier avant d'avoir parcouru tous les paquets qui par débordement peuvent lee contenir. L'organisation étant cyclique (le dernier paquet débordant en priorité dans le premier paquet), cette organisation conduit dans le cas des recherches infructueuses à parcourir tout le fichier. En revanche, la méthode est assez bonne pour des recherches où la présence de la clé recherchée dans le fichier est sûre.

Pour conclure, nous indiquerons que le hachage a longtemps été considéré essentiellement comme une méthode de recherche en mémoire centrale (recherche dans des tables). En effet, dans ce cas, la répartition statique des valeurs de clé est généralement connue et permet de choisir la bonne dimension de la table et une bonne fonction de hachage. En revanche, si le nombre d'enregistrements croît et la répartition des clés est mal connue, les mauvaises performances dues aux collisions limitent la méthode. C'est le cas du hachage statique pour les fichiers. Récemment des méthodes dite de hachage dynamique lèvent ces inconvénients.

VII.2.4. Hachage et accès aux mémoires secondaires

Le hachage est une méthode d'accès sélective aux données d'un fichier. Comme pour le hachage en table, une partie de la clé primaire qui identifie chaque article pour déterminer l'adresse où sera placé l'article est utilisée. La recherche d'un article à partir de sa clé sera généralement effectuée en un seul accès, puisque l'adresse est déterminée par un simple calcul.

Plus précisément, le fichier est divisé en p cellules de taille fixe. Une fonction pseudo-aléatoire h, appelée fonction de hachage, attribue à la clé une cellule mémoire identifiée par h(c). La cellule est appelée *paquet* et peut contenir un certain nombre d'enregistrements. Un fichier aléatoire définit la distribution des articles dans des paquets dont l'adresse est calculée à l'aide d'une fonction de hachage appliquée à la clé. La taille des paquets correspond à un nombre entier de pages (un accès disque).

Un article de clé c est inséré dans le paquet h(c), appelé paquet primaire pour c, tant que le paquet n'est pas plein. La recherche de c démarre toujours par un accès au paquet h(c). Si le paquet h(c) est plein lorsque l'enregistrement de clé c doit être inséré, une collision se produit. Un algorithme de résolution de collision détermine alors une adresse de paquet de débordement m (m•h(c)), où sera rangé l'enregistrement en débordement.

Une méthode de résolution de collision simple place l'enregistrement en débordement dans un paquet non primaire (• h(c), ∀ c) qui sera chaîné au paquet primaire. Mais, dès que les paquets primaires débordent, les recherches nécessitent au minimum deux accès et les performances se dégradent rapidement. D'autres techniques de résolution, citées lors du hachage en mémoire centrale, peuvent alors être utilisées.

La difficulté reste le choix d'une "bonne" fonction de hachage. Elle doit distribuer uniformément les enregistrements dans les paquets, et minimise, ainsi les débordements, tout en conservant un bon taux d'occupation de la mémoire secondaire. Le choix d'une telle fonction, déterminant une distribution uniforme d'adresses à partir d'une distribution non uniforme de clés, n'est pas évident. Il nécessite, de surcroît, une bonne connaissance *à priori* de la répartition des valeurs de clés. C'est rarement le cas pour les fichiers.

La rapidité est l'avantage principale de la méthode d'accès aléatoire. Par contre, le taux d'occupation de la mémoire secondaire réellement utilisée peut rester assez éloigné de 1. Enfin, la taille d'un fichier doit être fixée à priori (choix du nombre de paquets primaires).

VII.3. LES INDEX SECONDAIRES

Les méthodes de placement présentées précédemment permettent de résoudre une large gamme de problèmes. Elles définissent un regroupement des tuples par rapport aux valeurs de certains attributs. Cependant, toute méthode de placement physique privilégie certains attributs. Le choix des attributs (et celui des fonctions de hachage qui leur sont appliquées) détermine la répartition physique des enregistrements les uns par rapport aux autres. Le choix des attributs précède donc nécessairement le chargement de tuples dans une relation. L'administrateur choisit précisément les privilégiés. Cependant, ce choix s'effectue, à un instant donné, en fonction des besoins instantanés de l'application. Du point de vue de la conception physique, ces besoins sont représentés par un ensemble de questions fréquentes. Or, rien ne garantit que cet ensemble n'évoluera pas dans le temps.

Il n'est pas toujours possible de changer l'organisation physique. En effet, l'interruption de service imposée par le déchargement/rechargement peut être inacceptable pour certaines applications. De surcroît, l'évolution des questions peut être trop rapide ou trop provisoire pour qu'une solution par réorganisation physique soit envisageable.

Une technique complémentaire au placement les index est donc nécessaire.

Les index constituent des chemins d'accès secondaires (c'est-à-dire permettant un accès direct sur les valeurs d'attributs non plaçants). Ces index sont des structures de données et sont utilisés comme des accélérateurs d'accès.

La définition de chemins d'accès secondaires complète le placement. La possibilité de construire ou de supprimer ces chemins "à la demande" assure une adaptation instantanée aux besoins des applications. Index et placement sont complémentaires. Le placement reste en effet optimal (seuls les blocs utiles sont accédés) et indépendent de la sélectivité des

attributs (on appelle ici sélectivité le rapport du nombre de valeurs distinctes d'un attribut au nombre de tuples de la relation). En revanche, les index, très sensibles à la sélectivité des attributs, deviennent rapidement inefficaces quand celle-ci est trop élevée. Pour une application donnée, une répartition judicieuse des attributs entre placement et index rend un maximum d'accès très efficace.

Un index est une structure destinée à localiser rapidement les tuples dont un attribut a une certaine valeur. Il associe la valeur de l'attribut et l'adresse des pages où se trouvent les tuples possédant cette valeur d'attribut. A partir d'une entrée sur la valeur d'un attribut spécifié dans la qualification d'une question, l'index fournit la liste des pages contenant les tuples qui peuvent vérifier le prédicat de recherche.

L'index peut ainsi être défini comme un ensemble de couples (valeur d'attribut de recherche, adresse). Une seconde possibilité constitue des couples (valeur d'attribut de recherche, valeur d'attribut plaçant). L'attribut plaçant est généralement la clé dans les organisations traditionnelles. La première solution a pour avantage un format plus court et un accès plus rapide aux pages de la relation qui sont alors recherchées par une adresse directe. Cependant, la manipulation des adresses de blocs nécessite la mise à jour de l'index à chaque migration de tuple d'une page vers une autre (ce qui se produit fréquemment lors des insertions). Cette mise à jour est nécessaire à chaque insertion ou suppression de tuple, mais également à chaque éclatement ou regroupement des pages de données. La seconde solution ne présente pas cet inconvénient. Elle utilise une sorte d'indirection : l'index permet de retrouver les clés plaçantes des tuples présentant la valeur cherchée. La clé plaçante est ensuite utilisée pour atteindre le tuple par la méthode d'accès employée pour le placement de la relation (hachage ou B-Tree par exemple).

Dans un SGBD, une solution très naturelle constitue une relation INDEX, de schéma (valeur d'attribut, adresse de page) pour stocker les informations de l'index. Cette méthode est largement utilisée par les produits du commerce, mais complique le processus d'insertion à cause des mises à jour de l'index.

VII.4. LE CHOIX DE QUELQUES SYSTEMES DE REFERENCE

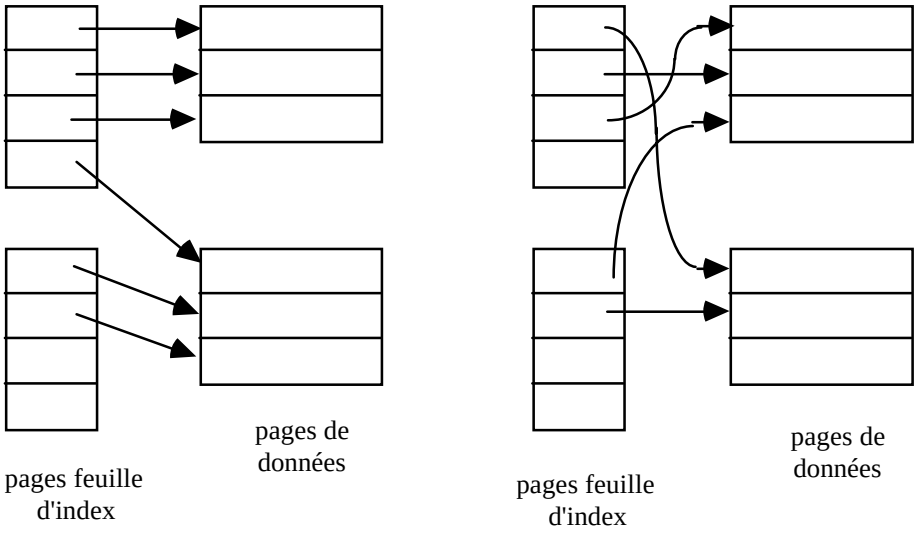
Nous examinons ici les structures qu'utilisent System R d'IBM et POSTGRES de l'Université de Berkeley. Les implantations confirment que les B-Trees sont la structure la plus fréquente. Cependant, dans POSTGRES, la méthode d'accès est paramétrable : le hachage peut être employé au même titre que les B-Trees. Deux approches sont donc présentées : l'une traditionnelle avec une méthode unique implantée (system R) ; l'autre, plus novatrice, avec une large gamme de possibilités (INGRES/POSTGRES).

VII.4.1. Le modèle d'accès et de stockage de System R

Dans System R, les tuples sont répartis dans des pages de disque et repérés par un identifiant (tuple identifier TID). Un TID, puisqu'il repère la page et la position qu'occupe un tuple, permet d'y accéder directement.

System R implante des chemins d'accès secondaires des index, est formés de couples dont le premier terme est la valeur de l'attribut indexé (c'est-à-dire sur les valeurs duquel on construit un index) et le second terme l'identifiant. Un index est un ensemble des couples (clé d'accès, TID). Il est stocké sous forme d'un B^+ Tree dont les pages sont les pages d'index, triées sur la clé d'accès. Il permet d'accéder aux tuples recherchés dans un ordre donné, c'est-à-dire sur les valeurs croissantes ou décroissantes de l'attribut indexé. L'obtention des TID des tuples recherchés n'exige pas d'accès aux pages de données.

System R permet d'organiser le placement d'une relation en fonction d'un index. L'efficacité d'un index dépend en effet du placement physique de la relation. Si la relation est placée en accord avec l'index, l'index est dit plaçant (clustering index). Dans le cas contraire il est dit non plaçant (unclustering index).



clustéré **non clustéré**

Les chemins d'accès dont dispose le système sont donc le parcours séquentiel, l'index plaçant et l'index non plaçant. Leur utilisation est pour chaque relation référencée dans un traitement décidée par l'optimiseur.

Définir des index multi-attributs plaçants (on a alors un placement multi-attributs) est possible. Les champs concernés sont enregistrés sous la forme d'un code qui conserve l'ordre numérique ou lexicographique. Une question du type "At1=valeur1 et valeur2<At2<valeur3" peut ainsi être accélérée par un index unique. D'autres méthodes implantent des index séparés (un par attribut) et un opérateur d'intersection des listes d'adresses que fournissent les différents index (utilisés dans le cas de questions qui

spécifient les valeurs de plusieurs attributs indexés). System R évite cette intersection et les structures de données multiples. Cependant, l'utilisation de l'index est impossible si le premier champ n'est pas spécifié dans la question car l'ordre n'est plus utilisable.
Les index sont de surcroît largement utilisés par les algorithmes de jointure implantés.
Ainsi, System R privilégie les accès sur valeur de clé (égalité ou inégalité) destinés à sélectionner un ensemble important de tuples triés sur le critère de recherche. Les accès sur valeurs de clé sont également efficaces si un index est construit sur la clé. Par contre, les chemins d'accès implantés ne sont guère efficaces pour des critères peu sélectifs de type 'exact match' (égalité d'un attribut et d'une valeur) sélectionnant un ensemble de tuples.

VII.4.2. POSTGRES

Le prototype POSTGRES [Stonebraker 86], de l'Université de Berkeley, est conçu comme un nouveau système de gestion de base de données, capable de dépasser le système INGRES, arrivé à maturité, qui admet difficilement l'introduction de nouvelles fonctions sans modifications importantes. L'objectif principal de POSTGRES est l'extensibilité. L'utilisateur doit pouvoir définir ses propres types de données, ses opérations, mais aussi ses méthodes d'accès. Le système doit supporter de nouveaux types de données (types abstraits). Les concepteurs de POSTGRES considèrent qu'il n'existe pas de méthode d'accès universelle et concluent qu'il faut en proposer plusieurs. Cette conception déjà présente dans INGRES, où l'on dispose à la fois de hachage et de B-Trees, est développée dans POSTGRES. Il est possible d'y définir diverses méthodes d'accès qui sont stockées dans la métabase. Les méthodes adaptées pour des types de données particuliers (objets spatiaux, cartes…)sont ainsi disponibles.
POSTGRES tente, pour des non spécialistes, de simplifier la définition de ces nouvelles méthodes grâce à des interfaces conviviales. Les nouvelles méthodes d'accès sont enregistrées dans une structure de données particulière appelée "gabarit". Cette structure documente les conditions d'efficacité de la méthode en fonction des opérateurs, des types de données et stocke des informations d'évaluation de coût. L'ensemble des méthodes est vu uniformément comme une collection d'abstractions génériques telles que "ouvrir", "fermer", "chercher(critère)" (premier, suivant, unique), "insérer", "supprimer".
Son concepteur a toutefois beaucoup de mal à interfacer correctement la nouvelle méthode, notamment en ce qui concerne la gestion de transactions. Même si l'on partage l'avis des concepteurs de INGRES/POSTGRES selon lequel il n'existe pas de méthode universelle et qu'il faut en proposer plusieurs, définies par l'utilisateur, des outils qui facilitent le travail de ce dernier restent à définir.
Des index secondaires, organisés en B-Trees sont également possibles. Enfin, POSTGRES permet à l'utilisateur de définir une collection d'index sur des types de données abstraits.

VII.5. CONCLUSIONS

Une bonne méthode d'accès doit avoir des propriétés précises : elle doit être applicable à tout fichier, quelle que soit sa dynamique de croissance ou de répartition, et conserver de bonnes performances, quel que soit le type d'accès. Or, les méthodes présentées sont loin d'être universelles. En effet :

1/ l'accès direct reste proche d'un accès disque pour les meilleures méthodes de hachage. Par contre, l'accès séquentiel trié est d'un coût prohibitif ;

2/ les B-Trees sont les méthodes qui supportent le mieux toutes les distributions (parce qu'ils se réorganisent en conséquence), mais au prix d'un coût moyen assez élevé.

Les inconvénients des méthodes traditionnelles (c'est-à-dire hachage et B-Trees) sont ainsi bien connus : le hachage est rapide, mais ne tolère que des distributions statiques des valeurs de clés (sauf pour les méthodes de hachage dynamique [Litwin 80]) et ne conserve pas l'ordre ; les B-trees acceptent les évolutions (fichiers dynamiques) et conservent l'ordre, mais sont plus lents.

Ce chapitre et les précédents permettent d'avoir maintenant une idée plus complète des étapes de l'évaluation d'une requête SQL que résume le schéma ci-dessous :

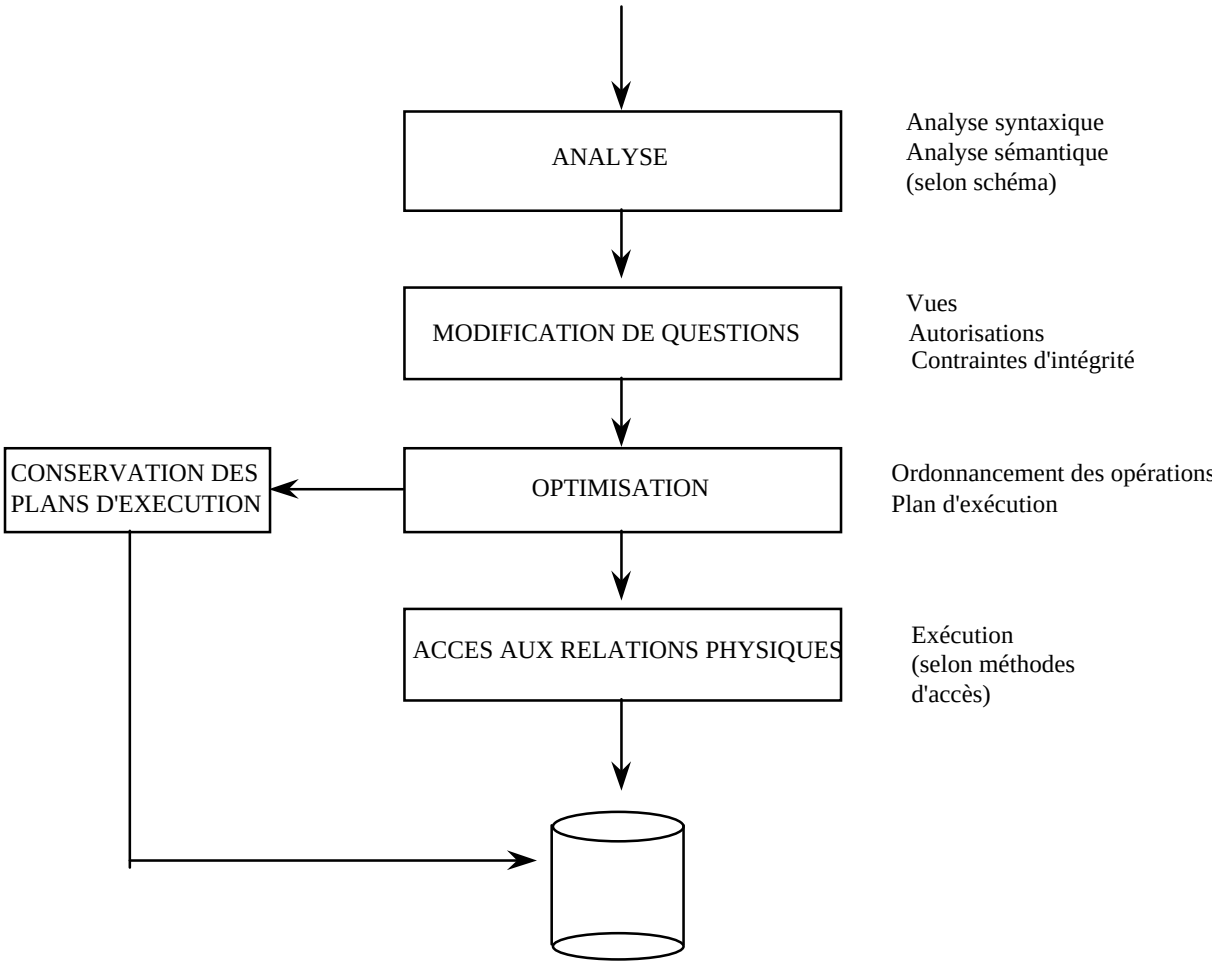


Figure 7.1. De la question SQL aux accès physiques.

La première étape analyse syntaxiquement et sémantiquement la requête

L'*analyse syntaxique* correspond à la vérification de la correction de la question par rapport au langage. Cette étape n'est guère différente de l'analyse syntaxique d'un langage de programmation quelconque et ne constitue pas un aspect spécifique des systèmes de bases de données. Nous ne la développerons donc pas ici.

L'*analyse sémantique* en revanche fait appel à des informations de la métabase qui constituent la source des vérifications qui doivent être effectuées : chaque objet évoqué dans la requête - relation, attribut, vue - doit être présent dans la métabase. Cette vérification se traduit par des *contrôles sur le schéma de la base*, contrôles qui ne sont en fait que des sélections sur les relations de la métabase stockant ce type d'informations[7]. La non existence d'objets évoqués dans la question conduit bien sûr à l'arrêt du traitement de la question et un message sera renvoyé à l'utilisateur.

[7]•Cette première étape conduit à rappeler le rôle particulier joué par les relations de la métabase. Il est en effet clair que ces relations présentent des particularités par rapport aux autres relations de la base. Il s'agit de relations (i) relativement petites, (ii) qui sont accédées fréquemment et (iii) dont le schéma n'est pas modifiable. Ces particularités conduisent à "bloquer" leur contenu en mémoire centrale afin d'en limiter le coût d'accès.

La deuxième étape reconstituee les vues, puis à vérifie les autorisations et les contraintes d'intégrités

On a vu, au chapitre 6, que les vues pouvaient être des transformations du texte SQL, des connexions arbre/sous-arbres algébriques ou des créations de relations temporaires. Le terme "modification" de la question résume ces transformations.

Les droits relatifs aux relations de la base effectivement invoquées peuvent ensuite être vérifiées. Le mécanisme vérificateur des contraintes d'intégrité (a priori et/ou a posteriori, par requête ou par transaction[8]) prend ensuite le contrôle.

La troisième étape - l'optimisation - recherche et génère le plan d'exécution optimum.

L'optimiseur composant clé d'un SGBD relationnel, transforme une requête SQL ou algébrique en un plan d'exécution physique optimal. Cette optimalité est un problème complexe qui n'est pas encore aujourd'hui totalement maîtrisé : à partir d'une question SQL, un nombre important de plans d'exécution différents ; le problème est donc soit de générer tous les plans possibles (optimisation exhaustive), soit seulement un sous-ensemble de ces plans (optimisation non exhaustive) peut être produit. L'optimiseur calcule pour chacun d'eux un coût et exécute le plan de moindre coût.

Deux problèmes principaux se présentent : (1) comment réduire l'espace des solutions et ne calculer les coûts que d'un nombre "raisonnable" de plans (ne pas passer trop de temps dans un composant du SGBD qui est sensé précisément gagner du temps est important) ; (2) disposer de modèles de coûts à la fois simples et réalistes.

La dernière étape est l'exécution proprement dite

Les coûts des plans d'exécution dépendent non seulement des méthodes d'accès disponibles, mais aussi du choix des algorithmes pour les opérations de l'algèbre relationnelle. L'opération de jointure est une opération clé pour l'efficacité et les temps de réponse d'un SGBD relationnel. Les performances des algorithmes de jointures disponibles constituent donc paramètre un important de l'optimisation. Le lecteur intéressé trouvera ci-après quelques références pour en savoir plus.

---

[8]cf. chapitre suivant.

VII.7. REFERENCES

Méthodes d'accès :

[Bayer 72] R. BAYER , Mc CREIGHT ; *"Organization and Maintenance of Large Ordered Indexes"*, Acta Informatica 1 (1972) .

[Knuth 73] D.E. KNUTH : *"The Art of Computer Programming"*, Vol 3, "Sorting and Searching", Addisson Wesley 1973.

[Litwin 80] W. LITWIN : *"Linear Hashing : A New Tool For Files and Tables Addressing"* , Proc. of the 6th Int. Conf. on Very Large Data Bases, Montréal, Sept. 1980.

[Stonebraker 86] M. STONEBRAKER, L. ROWE : *"The Design of POSTGRES"*, ACM SIGMOD, Int. Conf. on Management of Data, 1986.

Algorithmes de jointure :

[KITS83] KITSUREGAWA M. et al : "Application of Hash to Database Machines and its Architecture", New Generation Computing, N°1, 1983.

[VALD84a] VALDURIEZ P.,GARDARIN G.: " Join and Semijoin Algorithms for Multiprocessor Database Systems", ACM Transactions on Databases Systems, V9, N° 1, march 1984.

[DEWI85] DEWITT DJ, GERBER R. :"Multiprocessor Hash-Based Join Algorithms", Proc. of the 11th Int. Conf on VLDB , Stockholm, 1985.

CHAPITRE 8 : LA GESTION DES TRANSACTIONS

Dans les contextes multi-utilisateurs inhérents aux SGBD, développer des techniques performantes pour contrôler les accès partagés aux données de la base - en écriture comme en lecture - et pour assurer la persistance des écritures validées est indispensable.

Dans la première partie de ce chapitre, nous revenons d'abord sur la notion de transaction[9], nous présentons ensuite quelques exemples d'incohérences possibles en cas d'absence de contrôle de concurrence, puis nous exposons les différents mécanismes utilisés pour assurer le contrôle de concurrence dans un SGBD.

Dans la seconde nous voyons comment garantir l'intégrité physique des données validées en assurant une reprise correcte dans tous les cas de panne du système.

Enfin, nous évoquons rapidement, dans une troisième partie, les indicateurs standards utilisés pour la performance de la gestion des transactions et leur mesure sur bancs d'essai ("benchmarks").

VIII.1. LES PROPRIETES TRANSACTIONNELLES DES SGBD

Une *transaction* par définition est une séquence d'actions - c'est-à-dire de requêtes consultatives ou modificatives - sur les données de la BD, demandées par un même utilisateur et considérées par lui comme un tout indivisible. Cette séquence est à effectuer totalement ou pas du tout et, prenant la BD dans un état cohérent, elle doit la "rendre" dans un (nouvel) état cohérent. Ces propriétés d'***atomicité*** et de ***consistance*** distinguent les transactions de n'importe quelle séquence d'actions - éventuellement réduite à une seule - sur la BD.

Une transaction a donc nécessairement un auteur, une marque de début "begin of transaction" (BOT) et une marque de fin "end of transaction" (EOT). Sa fin peut être négative - "annulation", "abandon", " abort" ou "rollback" - ou positive - "validation", "confirmation", "engagement" ou "commit", le choix du mot étant en général indifférent - le langage SQL n'en a retenu que deux COMMIT et ROLLBACK[10], mais peut laisser deviner que les points de vue de l'utilisateur et du SGBD sont différents.

---

[9] évoquée à la fin du Chapitre 6

[10] En SQL la marque BOT est implicitement posée avant la première requête et immédiatement après toute marque EOT c'est-à-dire une validation - COMMIT - ou une annulation - ROLLBACK. De plus, la plupart des interpréteurs de SQL offrent un mode AUTOCOMMIT qui fait de toute requête SQL une transaction dont la validation est automatiquement demandée; il faut faire SET AUTOCOMMIT [ON|OFF] pour commuter d'un mode à l'autre.

Un **utilisateur**, au vu de considérations extérieures à la BD, décide finalement de confirmer/valider sa transaction, ce qui correspond toujours peu ou prou à un *engagement* signé et daté, ou au contraire de l'abandonner/annuler, ce qui correspond à son *droit de regret* . Le **SGBD**, pour sa part, peut ou non mener la séquence d'actions jusqu'au terme choisi par l'utilisateur. Il peut, soit en cours, soit au terme, être contraint de " *défaire* " ce qu'il a déjà fait pour le compte de cette transaction, quitte à *refaire* automatiquement ensuite. Une concurrence d'accès insoluble ou une panne peut en être la cause. Normalement, après une validation - COMMIT - de l'utilisateur, le SGBD doit garantir la propagation et la persistance des modifications demandées par la transaction. C'est la propriété de persistance ou ***durabilité*** des transactions.

VIII.2. LA GESTION DE LA CONCURRENCE

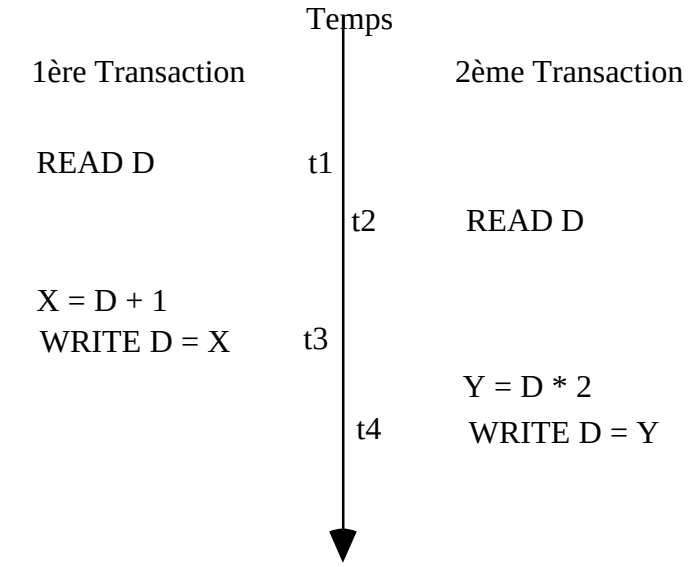
Comment le SGBD peut-il servir plusieurs utilisateurs ou, plutôt, plusieurs transactions simultanément?

Si ces transactions, a priori, ne concernent que des données logiquement différentes, le SGBD, qui dispose des ressources nécessaires et les gère correctement, peut "se couper en quatre" - ou plus! - pour les traiter. Il ordonnance de fait son travail pour donner l'apparence extérieure d'une exécution en parallèle. Par contre si les transactions sont logiquement en concurrence sur une ou plusieurs données communes (une ligne comptable, une place d'avion, etc...) le SGBD devra gérer une ou plusieurs files d'attente c'est à dire des exécutions en séries ordonnées. L'absence de contrôle de ces concurrences logiques conduirait à des interférences plus ou moins inacceptables.

VIII.2.1 Les interférences à éviter

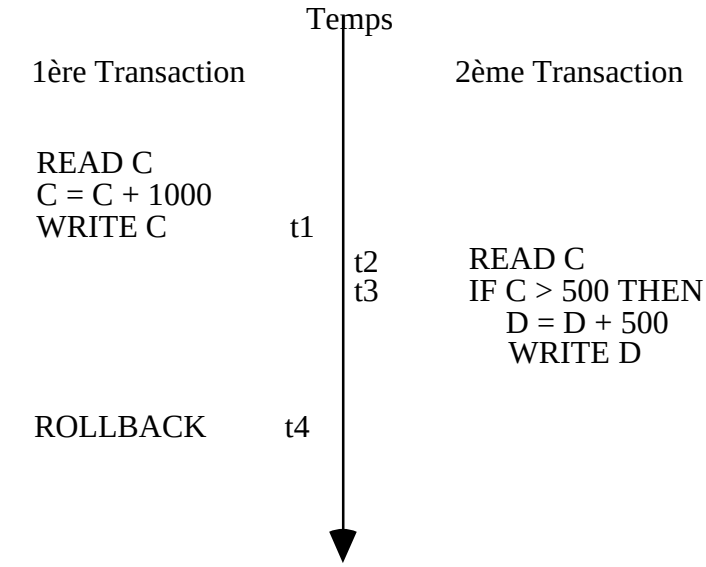
VIII.2.1.1 entre écrivains

- "lost update" (ou perte d'opération)



Deux transactions accèdent à la même donnée D pour d'abord la lire. Puis la 1ère fait X = D + 1 et met à jour D par recopie de X. De son côté, la 2ème fait Y = D * 2, puis, après la 1ère, met à jour D par recopie de Y. La mise à jour faite par la 1ère transaction est perdue, c'est le dernier qui écrit qui a raison! [11]

- "dirty write" (ou écriture inconsistante)



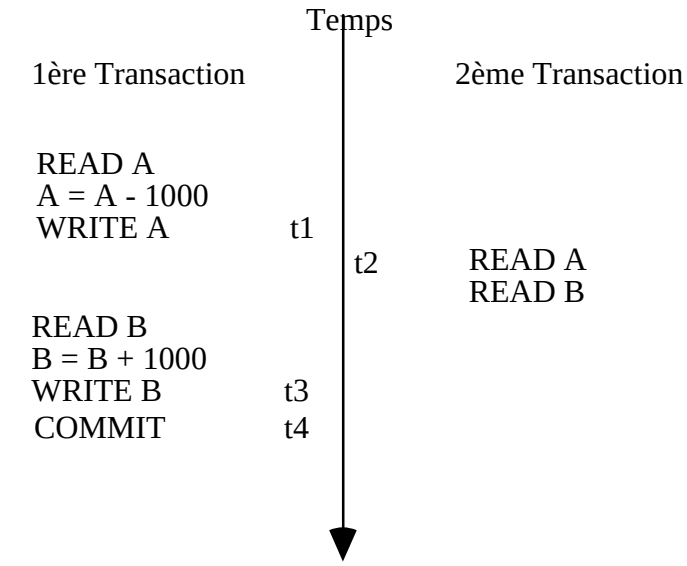
Une 1ère transaction modifie le crédit C d'un compte. Une 2ème lit C dans ce nouvel état, puis, constatant que la provision est suffisante, modifie le débit D du

[11] C'est très exactement ce qui se passe sous Unix quand deux utilisateurs éditent et modifient le même fichier de texte.

compte. Mais la 1ère fait un "rollback". La donnée D a désormais une valeur qui ne satisfait plus la contrainte d'intégrité D =< C c'est à dire que le solde est négatif!

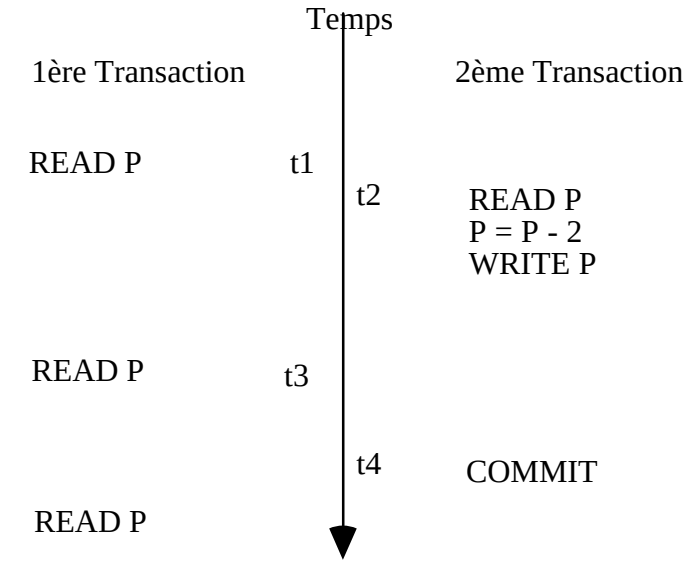
VIII.2.1.2 entre lecteurs et écrivains

- "dirty read" (ou lecture inconsistante)



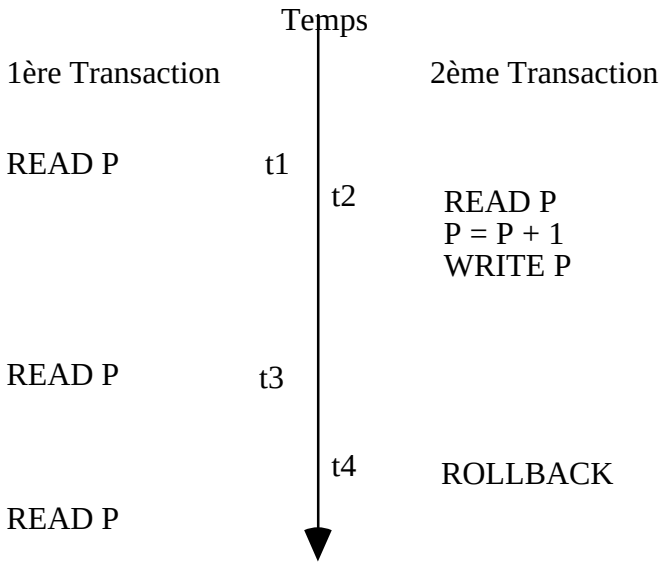
Une 1ère transaction, pour faire un virement entre deux comptes A et B, qui satisfait la contrainte d'intégrité "somme A+B invariante", commence par débiter A. Juste à ce moment une 2ème lit A puis B et trouve A+B diminué!

- "non-repeatable read" (ou lecture non reproductible)



Une 1ère transaction consulte les places libres dans un avion et laisse un temps de réflexion ("think time") à l'utilisateur. Une 2ème transaction, qui voit les mêmes places libres, valide deux réservations. La 1ère demande à nouveau les places libres: la liste est diminuée alors qu'elle n'a pas encore choisi ses réservations!

- "phantom read" (ou lecture fantôme)



Cas identique au précédent mais avec libération au lieu de réservation de place par la 2ème transaction. Il n'y a pas alors de conflit à proprement parler mais seulement interrogation sur la validité de ces "apparitions fantômes".

VIII.2.2 Les niveaux d'isolation

VIII.2.2.1 exécution sérialisable

Obliger les transactions concurrentes à s'exécuter en *succession totale* est assurément la plus forte isolation possible, mais aussi la plus pénalisante en temps de réponse car les temps morts et les actions sans concurrence logique, inclus dans ces transactions, sont inutilement mis en série. Il est donc intéressant de considérer la concurrence entre transactions au plus bas niveau possible, celui des actions qui les composent. Dans une même transaction certaines actions sont permutables si l'exécution des permutations donne le même résultat final. Certaines transactions sont compatibles si elles ne sont pas en concurrence logique. *Une transaction T1 est dite précédente logique d'une transaction T2 si une action de T1 non permutable avec une action de T2 se présente la première* . Si plusieurs transactions T1,T2, .., Tn doivent être considérées simultanément, le problème de précédence s'exprime par un graphe. Notons Ai1, Ai2,..,Aip les actions successives de la transaction Ti. Une exécution entrelacée sur le modèle:

A11,A21,A31,A12,A22,A23,A32,A13

est dite *sérialisable* si elle donne le même résultat qu'une exécution successive des transactions (que l'on pourrait reconstituer par permutation des actions compatibles ou permutables) respectant leur graphe de précédence. Si celui-ci est sans circuit, au moins une exécution sérialisable peut être trouvée. Elle assure, comme la succession, le meilleur niveau d'isolation possible, mais avec un temps de réponse global bien inférieur.

VIII.2.2.2 autres niveaux

Le standard SQL92 - alias SQL2 - prévoit une instruction SET TRANSACTION ISOLATION LEVEL <niveau> pour le niveau SERIALIZABLE défini ci-dessus, mais aussi pour trois autres niveaux d'isolation de plus en plus dégradés par rapport à celui-ci:

- REPEATABLE READ qui accepte les lectures fantômes (un nouveau venu dans une liste)
- READ COMMITTED qui accepte les lectures non reproductibles (on lit à tout moment tout ce qui est validé)
- READ UNCOMMITTED qui accepte les lectures inconsistantes (conséquences d'écritures non encore validées)

L'objectif de ces "relâchements" plus ou moins importants est évidemment d'augmenter le parallélisme apparent et le débit en transactions.

VIII.2.3 Les mécanismes utilisés

Dans le modèle physique les "tuples" sont des structures adressables d'octets. Un "gestionnaire de données"[12] les charge dans un espace de la mémoire principale ou les en décharge, par des opérations élémentaires de lecture/écriture de "pages" - ou "blocs" - de données, pouvant contenir plusieurs "tuples", depuis ou vers des fichiers sur disque. Ainsi l'adresse physique de stockage d'un tuple est de la forme <fichier>,<page>,<offset>. Le "granule" de concurrence le plus souvent gérée par les SGBD est la page ou la page et l'offset - c'est à dire le tuple.

Le gestionnaire de transaction enchaine deux fonctions: (1) il découpe les transactions en actions élémentaires au niveau physique où est gérée la concurrence, puis (2) il ordonnance ces actions. Cet ordonnancement nécessite le calcul et le maintien d'un graphe de précédences a priori et/ou d'un graphe d'attentes a posteriori, puis le choix des séquences d'exécution selon le niveau d'isolation demandé et enfin la supervision de l'exécution. Un pré-processeur alimente l'ordonnanceur en flux parallèles d'actions. Ce dernier alimente le gestionnaire de données action par action et rend compte au pré-processeur de l'état instantané de chaque transaction: encours, suspendue, rejetée, validée.

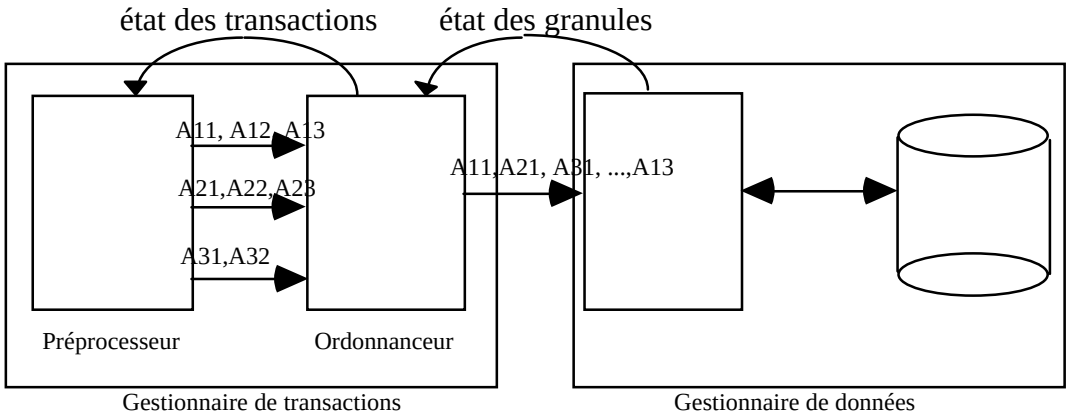


Figure 8.1 Principe de gestion des transactions concurrentes.

VIII.2.3.1 verrouillage à deux phases ("two phases locking[13]")

Dans un bon SGBD deux types de verrous peuvent être posés sur les granules de concurrence, pages ou tuples: le verrou exclusif - eXclusive lock - et le verrou partageable - Shared lock. Le premier autorise la transaction qui l'a posé sur un granule à continuer mais met en attente toute autre transaction qui voudrait ensuite lire ou écrire sur ce granule. Le second autorise par contre les lectures concurrentes. L'ordonnanceur a la

[12] il sera décomposé plus loin en gestionnaire de reprise sur panne et gestionnaire de cache, ce dernier est responsable des mouvements entre la mémoire principale et la mémoire secondaire - les disques.
[13] ne pas confondre avec le protocole de validation à deux phases - " two phases commit" - pour le transactionnel distribué, que l'on verra dans le module optionnel BDAS.

charge de transformer les requêtes READ et WRITE qu'il reçoit du préprocesseur en requêtes SREAD ou XREAD et XWRITE, puis d'ajouter des requêtes SRELEASE ou XRELEASE selon les protocoles suivants:

Verrouillage en mode X (exclusif):

Une transaction T qui veut accéder à un granule de données D pour *mise à jour* doit d'abord faire un XREAD D qui a pour effet: si D est non verrouillé - ni au niveau X ni même au niveau S - alors de le verrouiller au niveau X sinon de suspendre l'exécution de T jusqu'au déverrouillage de D par la transaction concurrente qui l'avait verrouillé à ce niveau.

Verrouillage en mode S (partagé) :

Une transaction T qui veut accéder à un granule de données D pour *lecture reproductible* doit d'abord faire un SREAD D qui a pour effet: si D est verrouillé au niveau X alors de suspendre l'exécution de T jusqu'au déverrouillage - XRELEASE - de D par la transaction concurrente qui l'avait verrouillé, sinon T est ajoutée à la liste des transactions qui ont verrouillé D au niveau S. T ne peut elle-même faire une mise à jour de D que par un XWRITE D qui tente de verrouiller au niveau X ce qui peut la suspendre jusqu' après le dernier SRELEASE de D des transactions concurrentes.

Restriction à deux phases

Le respect des deux protocoles précédents n'interdit pas des verrouillages et déverrouillages au fur et à mesure de la progression des transactions. Mais cela laisse la porte ouverte à de nombreux cas d'interblocage de toutes les transactions - appelés "verrous mortels" - ou de blocage éternel de quelques unes - appelés "famines". On démontre que si l'exécution de chaque transaction en deux phases seulement est imposée, une première pour tous les verrouillages - SREAD, XREAD ou XWRITE - une seconde pour tous les déverrouillages - SRELEASE, XRELEASE - sans qu'il soit possible de verrouiller à nouveau [14], *il existe alors au moins une exécution globale sérialisable* . De plus certains cas de verrou mortel ou de famine sont aussi implicitement résolus.

Détection des verrous mortels résiduels

Il n'en demeure pas moins des cas où le gestionnaire de concurrence ne peut que constater l'arrêt d'une exécution sur un verrou mortel. Il lui appartient alors de décider laquelle des

[14] en pratique la plupart des SGBD n'ont pas de S- ou XRELEASE explicites et c'est le COMMIT ou le ROLLBACK de fin de transaction qui implicitement relache les verrous posés par elle.

transactions suspendues doit être rejetée - rollback - et, éventuellement, rejouée, si la sémantique de la transaction rejetée le permet. Ces cas sont heureusement détectables - par exemple en gérant un graphe des attentes effectives entre transactions. Ce graphe est très semblable au graphe des précédences - et la règle du choix de la victime exprime le genre de "politesse" que l'on veut pratiquer entre transactions "premières nées" et "dernières nées"...

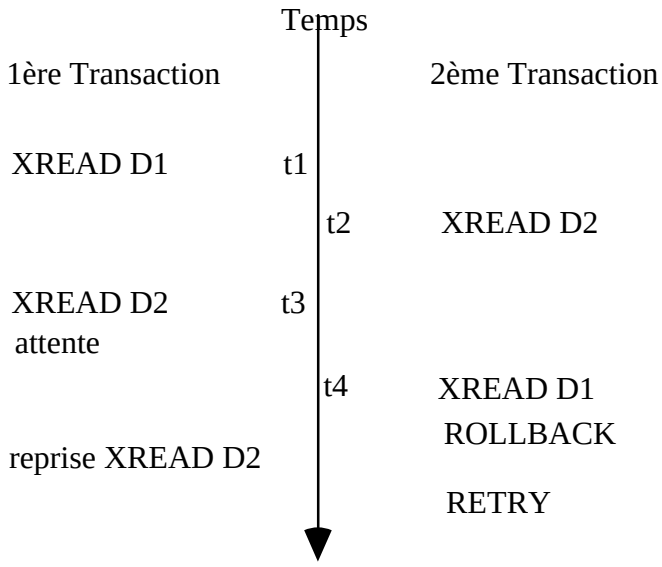


Figure 8.2 "s'effacer devant l'ancien"

VIII.2.3.2 autre mécanisme

Le mécanisme de gestion des concurrences par verrouillage à deux phases *détecte* a posteriori des attentes entre transactions. Des mécanismes de *prévention* basés sur un ordonnancement a priori des actions composantes des transactions peuvent être aussi envisagés.

L'*estampillage* est un mécanisme du type prévention. Les transactions T reçoivent de l'ordonnanceur à leur début - Begin Of Transaction - une estampille qui est un numéro d'ordre de présentation en entrée de l'ordonnanceur. Chaque granule G de données note l'estampille de la dernière transaction qui y a accédé. Un granule n'accepte un nouvel accès que s'il est demandé par une transaction "plus jeune" - c'est-à-dire de numéro d'estampille plus grand.

```
PROCEDURE LIRE (T, G, X)
DEBUT
    SI  Estampille(T) >= Estampille(G)
    ALORS
```

```
PROCEDURE ECRIRE (T, G, X)
DEBUT
SI  Estampille(T) >= Estampille(G)
        ALORS
```

```
        DEBUT                                   DEBUT
        X <-  lecture(G);                       écriture(G) <-  X;
        Estampille(G) := Estampille(T)              Estampille(G)        :=
Estampille(T)
        FIN                                         FIN
    SINON  rollback(T)                      SINON  rollback(T)
FIN                                     FIN
```

La transaction rejetée est reprise par l'ordonnanceur après avoir reçu de lui une nouvelle estampille.

Un tel algorithme d'ordonnancement total conduit souvent à de nombreuses reprises en cascades, parfois par excès de prudence. Aussi a-t-on conçu des algorithme d'ordonnancement partiel qui distinguent pour chaque granule un compteur pour les estampilles des transactions accédant en lecture et un compteur pour les estampilles des transactions accédant en écriture, ou mieux qui distinguent des versions et les couples d'estampilles correspondantes pour chaque granule. S'ils améliorent clairement le débit transactionnel leur coût en écritures et en place mémoire n'est toutefois pas négligeable.

VIII.3. LA GESTION DES REPRISES SUR PANNES

VIII.3.1 Les différents types de panne

Une panne est un évènement logique ou physique qui provoque une fin anormale de transaction - hors du cas normal d'abandon "conscient". On peut classer les pannes en:

VIII.3.1.1 Panne de transaction

du fait de la transaction elle-même (si elle demande une action illégale: une division par zéro ou une modification violant une contrainte d'intégrité) ou du fait du SGBD (si il ne peut résoudre un "dead-lock" dans la concurrence de deux transactions). Dans les deux cas, un rollback de la transaction peut être automatiquement déclenché et complètement effectué.

VIII.3.1.2 Panne de système

ou de mémoire principale, qui malheureusement est toujours "volatile": le système a en quelque sorte "perdu la mémoire" ou ne sait plus la relire correctement. Une des parties de la mémoire consacrée soit aux données, soit aux journaux, soit aux codes des gestionnaires eux-mêmes, est devenue illisible. Le système doit redémarrer "à chaud" à partir du dernier

point de reprise ("checkpoint") qui correspond à une écriture en mémoire secondaire de toutes les pages mises à jour en mémoire principale.

VIII.3.1.3 Panne de "media"

ou de mémoire secondaire: un volume de disque ou un fichier ou certains blocs d'un fichier ne sont plus accessibles ou ne sont plus corrects. S'ils ne concernent que la base de données, on peut les reconstituer à partir du dernier " backup" (sauvegarde) qui correspond à une écriture sur un autre disque, ou sur une bande, de tout ou partie de la base de données physique et des journaux[15]. C'est un redémarrage "à froid". S'ils concernent la base et tout ou partie des journaux depuis la dernière sauvegarde, la panne est dite "catastrophique" et seule une réparation manuelle par l'administrateur de la base est envisageable.

VIII.3.2 Les redondances nécessaires

VIII.3.2.1 Mémoires "sûres" et mémoires "fiables" (ou "à haute disponibilité")

Les mémoires sûres supposent que l'écriture physique d'une unité d'enregistrement ("page" ou "bloc") soit atomique, c'est-à-dire ou totalement et correctement exécutée ou pas exécutée du tout. Les mémoires fiables sont d'abord sûres et, de plus, se corrigent le plus souvent elles-mêmes des défaillances physiques, grâce à une redondance et des comparaisons systématiques.
Les mémoires secondaires fiables les plus répandues aujourd'hui utilisent la technologie RAID: Redundant Array of Inexpensive Disks.

VIII.3.2.2 Base de données, journaux, sauvegardes et archives

Les mémoires secondaires doivent être de fiabilité croissante: si la base de données physique est *non sûre,* les journaux doivent être *sûrs* et les sauvegardes doivent être *fiables* . Mais, bien sûr, augmenter la fiabilité de toutes les mémoires secondaires améliore la disponibilité (une reprise à froid nécessite - sauf dans le cas de dispositifs de mémoires en miroir - un arrêt d'exploitation). Dans tous les cas on minimise les risques en utilisant *au moins des disques différents pour la BD d'une part et les journaux et sauvegardes d'autre part* .. Les archives sont soit des sauvegardes anciennes conservées pour usage historique, soit des déchargement partiels des parties les moins souvent accédées de la base de données, soit enfin des déchargements partiels ou des duplications partielles des journaux quand ils deviennent trop longs.

---

[15] cf.supra

VIII.3.3 Les mécanismes utilisés

Ils sont tous basés sur le couple base de données et journaux. La base de données est toujours considérée sous son aspect physique. Parmi ses états successifs, deux sont des repères essentiels: ceux correspondant à une sauvegarde (duplication physique sur un autre volume de mémoire secondaire) et ceux correspondant à un point de reprise (recopie des pages mises à jour en mémoire principale vers la mémoire secondaire - " flush", faite si possible dans un temps mort transactionnel, c'est-à-dire sans transaction en cours). Le ou les journaux peuvent être conçus logiquement ou physiquement selon que l'on conserve la trace des transactions (pour défaire ou refaire) ou la trace des données modifiées (images avant ou après les transactions)[16].

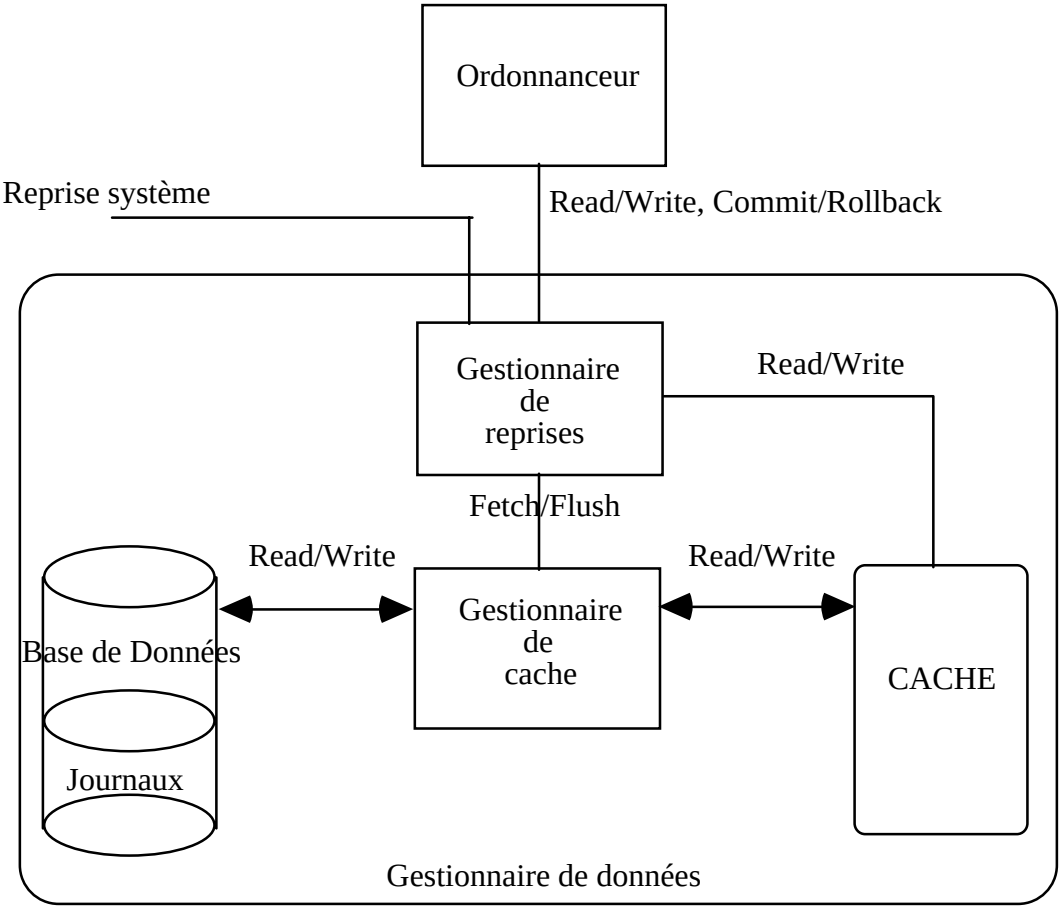


Figure 8.3 Principe de gestion des reprises après panne.

VIII.3.3.1 "do-redo-undo" (faire-refaire-defaire)

[16] "undo" or "redo" logs, "before-" or "after-images" journals.

Logiquement "faire" l'exécution d'une transaction, comme d'un flux de transactions, c'est lire et écrire des données dans le cache prévu à cet effet dans la mémoire principale puis écrire sur le ou les journaux en mémoire sûre secondaire. Quelque soient les stratégies de transfert des pages de données de la mémoire principale vers la mémoire secondaire (écriture laissée à l'initiative du gestionnaire de cache, écriture forcée au commit, écriture différée jusqu'au prochain point de reprise automatique - en général déclenchée par le constat d'un journal assez long) deux règles doivent être respectées:

> REFAIRE toujours possible: les IMAGES APRÈS mise à jour doivent être inscrites DANS LA MÉMOIRE SECONDAIRE (journal et éventuellement base de donnée) AVANT la fin de transaction EOT
> DÉFAIRE toujours possible: les IMAGES AVANT mis e à jour doivent être inscrites dans le journal correspondant AVANT que les IMAGES APRÈS soient transférées dans la base de données.

VIII.3.3.2 Stratégies

Elle sont logiquement quatre, mais les trois premières seulement sont pratiquement envisageables, selon qu'après une panne REFAIRE et DÉFAIRE sont nécessaires, REFAIRE seul ou DÉFAIRE seul est nécessaire, ni l'un ni l'autre ne sont nécessaires.

Considérons d'abord le cas où le dernier point de reprise correspond à un point mort hors transactions. Une panne non catastrophique de media est reprise à froid par rechargement de la dernière sauvegarde et déroulement du journal des IMAGES APRÈS - ou du REDO log - jusqu'au dernier point de reprise. Ensuite on est dans le même cas qu'une reprise à chaud qui correspondrait à une panne système intervenue après ce point. Il y a deux catégories de transactions à considérer alors, toutes ayant commencé après le point de reprise: celles qui avaient été validées avant, celles qui n'étaient pas encore validées. Comme en général on ne sait pas quelles pages mises à jour le gestionnaire de cache a ou n'a pas déjà transféré sur la base en mémoire secondaire, il faut d'abord DÉFAIRE les transactions non validées - en remontant le journal des IMAGES AVANT - jusqu'à leur début BOT, puis REFAIRE les transactions validées - en redescendant le journal des IMAGES APRÈS - jusqu'à leur fin EOT.

Considérons maintenant le cas où le dernier point de reprise avait eu lieu alors que des transactions étaient en cours: la méthode de reprise à chaud ne diffère que par l'étendue de la relecture des journaux car il faut remonter au début BOT de la plus ancienne transaction interrompue par la panne ou commise après le point de reprise mais avant la panne, ce qui pouvait être bien avant le point de reprise; la reprise à froid elle devra

cette fois-ci nécessairement être suivie d'une reprise à chaud. Ainsi, pour garantir un redémarrage depuis une base cohérente, quelqu'ait pu être la cause de l'arrêt, tous les SGBD font toujours une reprise à chaud après le "startup".

VIII.4. LES BANCS DE MESURE DES PERFORMANCES ("BENCHMARKS")

La question des performances transactionnelles a été déjà un peu évoquée ci-dessus en termes de temps de réponse et de "débit" de transactions. Définissons un peu plus précisément les grandeurs à mesurer:

- **N = nombre d'utilisateurs actifs simultanément présents:** dans une exploitation normale les utilisateurs ouvrent des sessions au cours desquelles ils effectuent des transactions successives. Du point de vue du SGBD, celles-ci sont seulement interrompues par les temps de lecture, de réflexion et de frappe de l'utilisateur (temps d'inactivité TI pour le système). Si les utilisateurs sont actifs, les sessions comme les transactions sont à tout instant N en parallèle.
- **TR = temps de réponse moyen par transaction:** supposons que la transaction est préparée par remplissage d'un écran qui est transmis en bloc au système[17], le cycle vécu par chaque utilisateur est un alternat: TI = utilisateur actif et système inactif, TR = utilisateur inactif et système actif. C'est ce deuxième qui constitue le temps réponse du système. La somme TI + TR représente un cycle transactionnel complet.
- **TPS = débit de transaction (par unité de temps):** si l'on considère le flux global de toutes les transactions, quelque soit l'utilisateur responsable, on peut en mesurer le débit à travers le système par unité de temps - généralement la seconde, d'où le nom Transaction(s)_Par_Seconde. Ce débit est d'autant plus grand que le "taux d'arrivée" est grand et le "temps de service" court.

Il est facile de vérifier que ces trois grandeurs sont liées par **la formule de Little**:

TPS = N / (TI + TR)

qui donne précisement la définition du débit d'un SGBD: nombre de transactions en cours divisé par temps de cycle moyen d'une transaction.

VIII.4.1 Mesures synthétiques et mesures analytiques

Le point de vue de l'*utilisateur* est toujours *externe* au système - "derrière" le terminal - et *synthétique:* il ne s'intéresse qu'à des performances globales - moyennes ou extrêmes (le "pire cas") - qui tiennent compte de tous les besoins: dispersion géographique des utilisateurs et des données, volume et distribution statistique des données, complexité du

[17] par système il faut entendre ici système de communication écran-machine et SGBD/OS/machine.

schéma et des requêtes, fiabilité des composants, etc... L'utilisateur doit prendre une décision d'achat et son cahier des charges spécifie en général une configuration pour une application type avec un volume minimum de données, des contraintes d'extrêmes sur les temps de réponse et la disponibilité. Il demande finalement des performances - et des rapports prix/performance - pour ce volume et pour plusieurs fois ce volume (dans le cas très probable où sa BD devra grandir avec le temps).

Le point de vue du *constructeur* est toujours *interne* au système et *analytique* du fonctionnement de chacun de ses composants. Pour lui, la performance globale n'est que le résultat de la performance combinée des sous-systèmes de gestion des mémoires, des reprises, d'ordonnancement des transactions, d'optimisation des plans d'exécution des requêtes et des "piles" de protocoles de communication - pour ne citer que les principaux. Pour les caractériser et pouvoir les prédire en exploitation il est amené à construire des bancs de test où, tous les autres sous-systèmes ayant des performances connues par ailleurs, un sous-système est soumis à un véritable plan expérimental. Il s'agira, pour lui, plus d'identifier les coefficients de formules de coûts que de mesurer les coûts les plus avantageux pour une charge de travail idéale.

Quelque soit le point de vue, les qualités attendues des bancs pour réaliser la mesure de ces performances - les "benchmarks" - sont les mêmes:

- *pertinence:* le benchmark doit bien spécifier le domaine et la signification de la performance qu'il entend mesurer; il doit être reproductible, étalonnable et le plus précis possible.
- *portabilité:* on doit pouvoir l'appliquer à une grande variété de configurations - pour comparer les résultats de plates-formes différentes sous un même SGBD, de SGBD différents sur une même plate-forme.
- *dimensionnabilité:* on doit pouvoir l'appliquer, pour une configuration donnée, à des tailles variables de BD et avec des quantités variables de ressources CPU et/ou disques, afin de tester les linéarités performance/volume BD à ressources constantes, performance/ressources à volume BD constant, volume BD/ressources à performance constante.
- *simplicité:* malgré les trois précédentes exigences, le benchmark doit rester simple à comprendre - au moins pour les "spectateurs" - et pas trop coûteux à implémenter pour les participants à la compétition !

VIII.4.2 Les benchmarks du Transaction Processing Council (TPC)

A fin des années 80, un "benchmark" pour les systèmes transactionnels était de plus en plus utilisé et ses résultats discutés tant par les utilisateurs et les journalistes spécialisés que par les constructeurs. Il était issu des propositions d'un article de Anon et al. de **1985**

dans DATAMATION pour la mesure du "transaction processing power" des systèmes et portait le nom de "débit-crédit" ou "**TP1**".

Devant la bataille des chiffres sans juge ni arbitre qui fit rage alors, la plupart des grands constructeurs - une cinquantaine à ce jour - ont préféré se réunir en "concile" pour stabiliser, dans des spécifications longuement discutées et précisées dans le détail, et finalement soumises à un vote formel en 1990, les définitions de deux benchmarks inspirés du TP1: les **TPC-A et B**. Ces benchmarks bien que ne représentant qu'une application très simple - un guichet automatique de banque ne faisant que des retraits ou des dépôts - ont été considérés comme des références obligées par presque tous les éditeurs de SGBD et ont eu une grande influence sur leur compétition. Les tps-A ou B, débits exprimés en transactions par seconde conformément aux spécifications des benchmarks TPC-A ou B, et les rapports prix/performance exprimés en milliers de US$ par tps-A ou B, ont fourni les chiffres de palmarès largement diffusés par la presse, bien que tout le monde s'accordait déjà pour trouver l'application " débit-crédit" comme très peu représentative des applications réelles des utilisateurs professionnels: l'accès exclusivement article par article ne donne aucun avantage aux SGBD relationnels par rapport aux SGBD réseaux ni à ces derniers par rapport aux SGF séquentiels indexés traditionnels. Ainsi ces benchmarks sont vite apparus plus dépendants d' aspects système généraux - multiplexage des terminaux pour le TPC-A, rapidité et/ou parallélisation des accès disques pour le TPC-B - que d'aspects proprement SGBD - optimisation et gestion transactionnelle des données.

C'est pourquoi le TPC a mis en chantier deux autres spécifications l'une pour le *"business transaction processing"* avec le **TPC-C** (1992), l'autre pour le *"decision support"* avec le **TPC-D** (1995). Toutes deux offrent un schéma et des requêtes plus complexes et plus représentatifs des applications réelles. Leur sémantique commune est inspirée des applications de gestion de clients, commandes et stocks. Les métriques en sont respectivement des tpm-C (transactions par minute pour le TPC-C) et des qph-D (requêtes - queries - par heure pour le TPC-D). D'autres spécifications ciblant plus particulièrement les gros serveurs BD et les configurations clients-serveurs sont en cours de discussion.

VIII.4.3 Le réglage ("tuning") des applications/SGBD réelles

Obtenir les meilleures performances d'une configuration complète - terminaux, système de communication, SGBD, OS, plate-forme mono ou multiprocesseurs, batterie de disques - est un *art* d'expert quand il s'agit des benchmarks de compétition, c'est un *métier* encore relativement complexe quand il ne s'agit "que" d'administrer des systèmes de bases de données réelles. Outre une excellente connaissance de la métabase - le "schéma des schémas" - et des outils de surveillance et contrôle en cours d'exploitation - "monitoring" - il faut à l'administrateur des règles et une stratégie pour les optimisations locales aux applications et globale au système - le fameux "débit". Ce qui nécessite un apprentissage

lourd, d'autant plus qu'il n'existe pas de modèles de simulation et de prévision qui soient à la fois universels et détaillés.
Les éditeurs de SGBD eux-mêmes, ou des sociétés indépendantes d'ingénierie du logiciel, proposent des interfaces dédiées aux administrateurs de SGBD qui varient de la simple manipulation visuelle des données de la métabase à de véritables systèmes-experts dont la base de modèles et de règles peut être très riche - et chère !

VIII.5. CONCLUSIONS

Le concept de transaction donne aux Systèmes de Bases de Données toute leur **dimension dynamique**: l'exigence de cohérence à la fin de chaque transaction - et de consistance dans la suite des transactions - conduit à l'exigence de services de gestion de la concurrence et de gestion des reprises non seulement fiables mais performants.
On peut théoriquement imaginer que ces services soient assurés par des sous-systèmes différents, comme ceux de contrôles des droits sur les données, d'optimisation des requêtes et de controle d'intégrité, en amont, ainsi que celui de gestion des caches de données, en aval.
Leur implantation sur des architectures distribuées pousse dans ce sens. Mais le besoin de performance pousse au contraire à intégrer le plus possible ces services entre eux et avec le gestionnaire de cache. Enormément d'efforts de conception d'algorithmes nouveaux, de modélisation et de mesures de performances sont encore en cours, tant du côte des chercheurs que du côté des constructeurs, pour de nouvelles intégrations sur de nouvelles architectures.

VIII.6. REFERENCES

[Gardarin 82] G. Gardarin, *Bases de Données: Les Systèmes et leurs Langages,* Eyrolles, 1982

[Date 83] C.J. Date, *An introduction to DataBase Systems - Volume 2* Addisson-Wesley, 1983

[Bernstein & al. 87] Ph.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency control and recovery in DataBase Systems,* Addison-Wesley, 1987

[Gray 93] Jim Gray and Andreas Reuter, *Transaction Processing: concepts and techniques,* Morgan Kofmann, 1993

[Gray 93] Jim Gray editor, *The Benchmarks HandBook for DataBases and Transaction Processing Systems - 2nd edition,* Morgan Kaufmann, 1993

CHAPITRE 9 : LES ENVIRONNEMENTS DE PROGRAMMATION D'APPLICATIONS SUR BD

Tout commence et tout fini par **l'utilisateur**. Trois fois dans les chapitres précédents le point de vue de l'utilisateur a été explicitement au départ des progrès demandés au SGBD:

- comme **concepteur et développeur**: ses exigences conduisent à des modèles et langages de richesse sémantique de plus en plus grande et indépendants de l'implémentation physique;
- comme **usager non informaticien** : la prise en compte de sa vision propre des données - liée à ses droits et responsabilités - conduit à la définition des vues externes, logiquement indépendantes du schéma du concepteur;
- comme **opérateur sur terminal**: son temps propre - de lecture, de réflexion, de frappe et/ou " cliqué" - définit le "temps réel" que devra respecter au mieux le système et lui fixe ainsi ses performances.

Dans ce chapitre nous rappellerons d'abord quelques règles de l'ergonomie des interfaces graphiques entre homme et machine pour réfléchir sur leur spécification, puis nous regarderons quels sont les bons outils pour programmer de telles interfaces, enfin nous indiquerons dans quels environnements ces outils peuvent être intégrés.

IX.1. LES INTERFACES GRAPHIQUES

Les interfaces homme-machine (IHM) ont, depuis l'essor du fameux Macintosh, pris l'aspect graphique et interactif qui permet aujourd'hui de retrouver des éléments graphiques communs à toutes les " graphical user interfaces" (GUI). Il ne faut pas croire pour autant qu'avant les terminaux, PC ou stations graphiques il n'y avait que les "bêtes" terminaux en mode asynchrone caractère par caractère du type ANSI vt100 ou émulateur "telnet".
Dès les premières grandes applications transactionnelles, on avait conçu un terminal [18] dit mode synchrone par pages transmises en un seul bloc. La séparation du temps de réflexion de l'utilisateur, sur une pleine page devant ses yeux, du temps d'attente de la réponse du système était alors totale. De nombreuses - et bonnes! - habitudes de présentation des écrans ont été prises depuis cette époque par les utilisateurs. Les applications modernes sur interfaces graphiques doivent en tenir compte.

---

[18] L'IBM 3270 en est le standard de fait, au même titre que le DEC, puis ANSI, vt100 pour les terminaux asynchrones.

IX.1.1. Vocabulaire descriptif

L'usage en a établi un lexique [19] presque commun à toutes les bonnes spécifications et manuels utilisateurs. Mais certains néophytes ou éditeurs, trop liés à un seul type de générateur d'interfaces, font encore souvent des confusions: ainsi entre les termes de "formulaire" - ou "forme" (d'affichage) - et celui de "fenêtre" on ne comprend pas toujours que le premier désigne l'objet à visualiser - qui peut être plus grand que l'écran! - et le deuxième le moyen de visualisation.

IX.1.2. L'ergonomie "navigationnelle"

Dans une situation normale de travail, la recherche et la manipulation des données obéissent à des enchainements logiques d'opérations: choix du contexte, choix d'une entité de référence ("maître"), liste d'entités liées ("détail"), calcul d'un agrégat ou choix de l'une d'elles qui devient "maître", liste d'entités liées, etc... On appelle cela une navigation à travers les données. Les applications sur ce modèle comportemental sont les plus simples d'apprentissage et donc les plus répandues.

IX.1.3. L'approche orientée objet

IX.1.3.1 Les objets graphiques standards ("widgets")

Les "WIndow gadGETS" - un bouton de sonnette par exemple - sont de véritables objets: accessibles seulement par leurs méthodes - enfoncer ou relâcher - leurs caractéristiques externes permanentes - position X,Y, graphisme - sont fixées à la conception, mais la variation de leur état interne, si elle est visible pour l'utilisateur par un écho de son action, peut envoyer à d'autres objets des messages préprogrammés - affichage de cet objet, par exemple.
On peut classer ces objets en inactifs et actifs, les premiers sont des informations ou des décorations invariantes à la fin de la conception, les seconds sont en quelque sorte les actionneurs ou/et capteurs offerts à l'utilisateur pour l'exploitation de l'application.

IX.1.3.2 objets de la base de données et objets affichables

[19] avec synonymes bien sûr comme par exemple "ascenseur" pour "barre de défilement verticale".

Quelle correspondance peut-on et doit-on faire entre les objets tels qu'ils sont inscrits dans le schéma "externe" de l'utilisateur dans la métabase et les objets tels qu'il peut et veut les visualiser?
Les options par défaut sont presque toujours:
FORME (ou FORMULAIRE) <-> résultat d'une requête sur une ou plusieurs vues du schéma externe;
BLOC (ou CHAMP COMPOSITE) <-> affichage d'un tuple ou d'une liste de tuples d'une vue de ce schéma;
CHAMP [ELEMENTAIRE] <-> affichage d'un attribut réel ou virtuel de cette vue, ou d'un résultat de calcul d'agrégat.

IX.1.3.3 objets de métiers standards et réutilisables

Outre les objets "naturellement" composites (un bon de commande et ses lignes, un équipement et ses sous-ensembles, un ménage et ses membres, etc...), chaque métier a l'habitude de classer, associer, hiérarchiser, relier des objets selon des schémas qui deviennent des standards de fait dans leurs domaines respectifs. Il est par conséquent normal que les interfaces des applications développées pour l'un d'eux finissent par ne différer que sur des styles, mais conservent les mêmes topologies et les mêmes enchaînements de formulaires. Les usagers des applications informatiques sont d'autant plus rapides au travail que la conception des interfaces applicatives dans un même métier renforce les habitudes de pensée de ce métier. Pour le meilleur comme pour le pire[20] ...

(insérer figure)

IX.1.4. Les styles

Le plus simple est le plus beau[21]: ne pas encombrer l'écran de plus de fenêtres qu'il n'est à chaque instant nécessaire, ne visualiser qu'un contexte à la fois, coordonner le défilement des objets dont les instances sont liées, etc...

---

[20] Pourquoi les tableaux de bord des automobiles ont-ils encore des indicateurs de vitesse gradués au delà de 150km/h?
[21] "make it simple", "small is beautifull"

IX.2. LES GENERATEURS D'APPLICATIONS SUR BD

On appelle ainsi les outils qui interprètent un langage de définition et de manipulation des données à travers écran-clavier-souris selon une interface standard, ou plus généralement elle-même définissable et paramétrable.
Le soucis de la productivité des développements a conduit au choix de méthodes de prototypage par une sorte de dessin des formes d'affichage avec les mêmes outils du terminal, c'est ce que l'on appelle la "programmation visuelle".
Mais il reste toujours du code à écrire dans des "scripts" associés aux objets graphiques. Les langages plus ou moins proches de l'embedded-SQL conçus dans ce but sont depuis l'origine appelés "langages de 4ème génération" (L4G, ou 4GL en anglais).
L'ensemble de ces dessins de formes et de scripts est finalement compilé, pour des raisons évidentes de performance, et stocké, soit dans un fichier, soit dans la BD elle-même pour permettre sa sauvegarde et son export éventuel simultanément avec les données de la base.

IX.2.1. La "programmation visuelle"

IX.2.1.1 Les interprêteurs de SQL ou de QBE

Quand l'administrateur d'un SGBD, ou le développeur d'application, veut interroger directement la BD, il utilise un interprêteur de SQL[22]. Celui-ci admet généralement l'entrée de texte de requête en mode ligne ou en mode plein écran et peut formater les sorties en tableau; des instructions supplémentaires au SQL standard permettent le contrôle du texte entré et de l'affichage du résultat. Mais on ne peut pas mettre une telle interface entre les mains d'un non-informaticien. Aussi les éditeurs de SGBD proposent-ils des interprêteurs de requêtes qui guident l'utilisateur par une succession de choix à faire sur un schéma graphique des relations (rectangles) et de leurs possibles jointures (arcs joignant les rectangles). Des menus permettent de compléter logiquement les prédicats WHERE de restriction du SELECT. L'utilisateur peut, avant ou après exécution de sa requête, visualiser le texte SQL automatiquement généré en tâche de fond.
Le langage QBE[23] avait, au contraire du SQL, été conçu pour être graphique dès l'origine, mais malgré sa puissance d'expression et les enrichissement picturaux qui sont proposés encore à ce jour par des chercheurs, il représente encore une voie peu suivie par les éditeurs de SGBD.

---

[22] comme on le fait toujours en Travaux Pratiques d'initiation au SQL.
[23] cf. ChapIV §2.2

IX.2.1.2 Les outils QBF et dérivés OO

Presque tous les SGBD offrent des générateurs de formulaires d'affichage de type " Query By Form" faits de champs interactifs qui acceptent des valeurs ou des prédicats restrictifs mono-table et des jointures entre tables, mais implicitement prédéfinies par le choix des tables affichées. C'est, bien sûr, sémantiquement inférieur au QBE, mais acceptable pour des applications simples et stables.

Enrichis avec la panoplie des objets graphiques et avec les mêmes principes de guidage que pour les aides à la formulation de requêtes SQL, on trouve, aujourd'hui, des dérivés "orientés objet" du QBF très ergonomiques.

IX.2.1.3 Les "langages visuels"

On appelle ainsi un ensemble de langages dont les éléments terminaux sont des objets graphiques dans le plan de l'écran et les expressions des assemblages construits par déplacements et " cliqués" de la souris, les identifiants d'objets ou d'attributs et leurs valeurs restant des éléments textuels ajoutés. Si certaines de leurs règles de construction sont reprises dans les outils QBF et dérivés, leur utilisation exclusive pour les GUI reste du domaine de la recherche: leur conception autant que la vérification de leur richesse sémantique prend toujours comme référence un des langages relationnels SQL ou QBE ou un modèle sémantique général comme l'Entité-Association.

IX.2.2. Les "langages de 4ème génération"

Ces langages, mi-visuels mi-textuels, le plus souvent interprétés, sont nés en même temps que les SGBD Relationnels. Ils ont pour principal objectif de permettre le *développement d'applications* sur terminaux semi-graphiques par maquettage-prototypage beaucoup plus *rapidement* que par programmation et compilation avec les langages classiques - COBOL, PL/1, Pascal, etc... - relégués au rang de "langages de 3ème génération" par les promoteurs de ces nouveaux outils. Leur simplicité apparente d'utilisation avait pour revers leur grande diversité de syntaxe et de capacité sémantique. Aujourd'hui l'existence et l'utilisation par ces outils de bibliothèques semi-graphiques ou graphiques standards et le renforcement de la norme SQL ont créé une convergence notable.

IX.2.2.1 Le prototypage des formes d'écran

Tous les L4G le proposent, dans le mode "dessinez-le vous même", à partir d'une collection de types d'objets graphiques visuellement manipulables. Ainsi une *application* apparaît d'abord comme un ensemble de *formulaires* d'affichages - ou "formes" ou "grilles" - chacune composée d'un *fond* - ou "décor" ou "papier peint" - éventuellement d'une *barre de menus* et de *blocs* - ou "zones" ou "entités" ou "champ composite" - positionnés sur ce fond. Un bloc est lui-même *de type grille* , s'il ne permet que l'affichage individu par individu, ou *de type liste* , s'il permet une vision

tabulaire de plusieurs individus à la fois avec déroulement vertical pour voir le reste de la liste. Le bloc est composé de *champs* élémentaires actifs ou non, c'est-à-dire d'état modifiable ou non à partir du clavier ou de la souris.

Tous ces objets sont nommés. Leurs variables publiques et leurs méthodes sont regroupés dans des structures référencées par les mêmes noms. S'agissant d'objets, la hiérarchie des types entraîne celle des héritages de variables et méthodes. Le couplage entre la valeur des variables et celle des attributs de données dans la base d'une part, entre les méthodes et les actions sur les données de la base ou sur d'autres objets de l'application d'autre part, est en général spécifié par un "script" attaché à chacun de ces objets.

IX.2.2.2 La spécification des enchainements dans l'application

Avec les L4G les plus récents les enchaînements sont le plus souvent spécifiés et programmés selon une logique "événementielle" décrite par des *règles de déclenchement d'action* - ou "triggers" applicatifs sur le modèle:

SI détection_changement_état_objet ALORS lancer_action

l'action lancée, pouvant être une modification de données dans la base ou l'appel à une méthode d'un objet graphique ou une séquence d'actions de l'un ou l'autre type. Elle peut en conséquence changer l'état d'un autre objet qui, ayant lui aussi un trigger attaché, relancera une nouvelle action et ainsi de suite... Comme il n'est pas exclu que la base de données elle-même soit pourvue de triggers détectant eux des changements d'état sur les données, on a, dans le cas le plus général un modèle d'enchaînements à deux boucles :

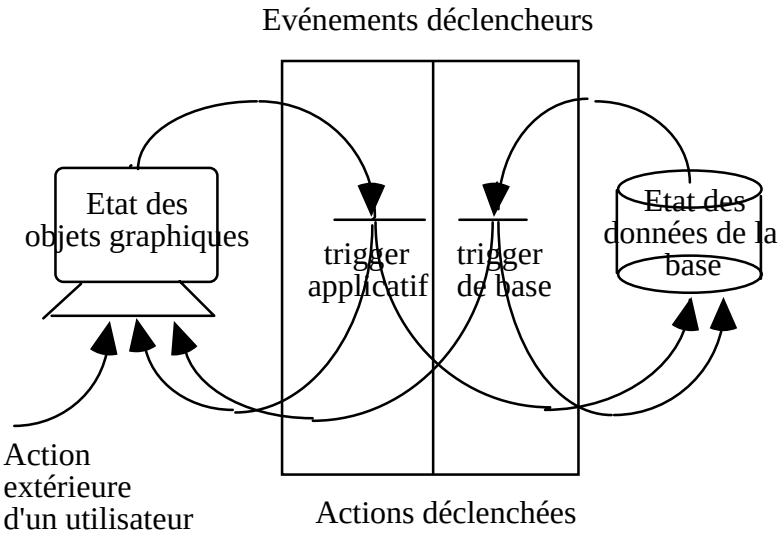


Figure 9.1: enchaînement "événementiel" des actions d'une application.

Une contrainte d'intégrité sémantique nouvelle apparaît: *l'utilisateur doit voir les objets graphiques dans un état cohérent avec celui des données de la base relevant de sa vue ou de ses vues propres* et le décalage temporel entre les changements d'état de part et d'autre doit être le plus réduit possible.

Un script attaché à un objet graphique - par exemple un bouton - a typiquement [24] l'allure suivante:

```
ON CLICK formeA.boutonB DO
    BEGIN
    INSERT INTO tableT(colonneC) VALUES (:saisieS);/* action sur la BD */
    formeA.popupP(messagetext = 'Valeur entrée dans la BD'); /* action sur
l'écran*/
    END
```

IX.2.2.3 Les procédures SQL+while[25]

Le programme de l'action déclenchée par un trigger applicatif doit nécessairement pouvoir inclure des instructions de manipulation de la base de données. Celles-ci s'exprimeraient aisément en C-Embedded SQL si le programme était codé en C. Les grands éditeurs de SGBD ont tous choisi de s'écarter le moins possible de ce stantard pour incorporer du SQL dans leur L4G, et même, si possible, de préfigurer le futur standard SQL3 qui ajoutera le WHILE au SQL. Ainsi le PL/SQL d'Oracle distingue soigneusement la manipulation des données de la base de la manipulation de l'interface. L'exemple ci-après en donne une illustration :

---

[24] Nous nous inspirons plus directement ici de la syntaxe de CA/ openROAD du L4G de CA/Ingres

[25] On désigne ainsi des expressions SQL augmentées de variables et de structures de boucle avec rupture conditionnelle

```
DECLARE                                          Returns the five highest paid
  CURSOR c1 IS                                        employees from the emp table,
    SELECT ename, sal FROM emp                        and stores the result in tmp,
    ORDER BY sal DESC;                                a table created for transfer.
       name  emp.ename%TYPE;
       salary              emp.sal%TYPE;          Output:    SELECT   * FROM
tmp
 BEGIN                                                ORDER BY earns  DESC;
  OPEN c1;
  FOR i IN 1..5 LOOP                                  EMPLOYEE   EARNS
    FETCH c1 INTO name, salary;                       -------------------------
    EXIT WHEN c1%NOTFOUND;                            KING          5000
    INSERT INTO tmp VALUES(name, salary);             SCOTT         3000
    COMMIT;                                           FORD          3000
   END LOOP;                                          JONES         2975
   CLOSE c1;                                          BLAKE         2850
 END;
```

IX.2.3. Les interfaces au standard SQL et la portabilité

IX.2.3.1 Une portabilité encore bien relative

Un développeur d'application ne peut pas ne pas avoir le soucis de la portabilité, par rapport à la variété des terminaux et stations semi-graphiques et graphiques, d'une part, et par rapport à la variété des SGBD, d'autre part. Il y a deux façons de le faire: soit il programme, par exemple en C, en utilisant une bibliothèque de procédures standards pour les entrées-sorties graphiques et l'Embedded SQL pour les entrées-sorties de données de la base, soit il programme en L4G et récupère, après mise au point complète et traduction, un programme intermédiaire en C incluant des appels de procédures de X11 Motif et des blocs codés en Embedded SQL. Mais deux limitations de portabilité subsistent:

- les spécifications des procédures de la bibliothèque semi-graphique curses(3v) ou de la bibliothèque graphique X11 Motif assurent la portabilité, et même, dans le monde Unix standard, l'indépendance physique par rapport à l'écran. Elles ne valent néanmoins pas pour le monde PC sous MS Windows, ni pour le monde Macintosh.
- la précompilation, qui traduit les blocs de code Embedded SQL en appels de procédures de définition ou de manipulation des données de la base, ne peut se

faire qu'en présence d'un SGBD et le code C obtenu reste ensuite lié par ces appels de procédures à ce SGBD particulier.

IX.2.3.2 Vers une ouverture plus grande

Selon que l'application regarde vers l'interface graphique ou vers la BD l'effort pour une plus grande ouverture n'est pas de même nature:

- le code de l'application s'exécute sur le PC ou la station donc avec l'unique window manager de ce système, le problème pour le développeur est donc seulement un problème classique de compilation multi-cibles à partir d'un code source commun. La spécification d'une bibliothèque source graphique commune relève donc du choix de l'éditeur de L4G et de son client pour le temps du développement [26] seulement.
- par contre une compilation d'Embedded SQL ne pouvant se faire qu'avec un SGBD donné, ce PC ou cette station ne peut plus, pendant l'exploitation [27], qu'accéder à ce SGBD. Il faudrait simultanément lancer l'exécution d'un autre code compilé - du même source, mais avec un autre SGBD - pour accéder à un autre SGBD depuis le même PC ou la même station. Ce besoin d'*interconnectivité* a été très tôt exprimé par les clients, aussi les éditeurs de L4G se sont-ils concertés dès après l'adoption du SQL1 pour spécifier un niveau commun d'appels de procédures pour manipuler les BD. Il a été adopté en 1991 par le consortium X/Open sous le nom de Call Level Interface (ou CLI) et est candidat à la normalisation par l'ISO[28].

On voit ainsi se préfigurer symétriquement deux bibliothèques programmatiques standards - ou Application Programmatic Interfaces (API) - une pour les entrées-sorties côté terminal, l'autre pour les entrées-sorties côté SGBD, qui, ensemble, donneront le maximum de portabilité et d'interconnectivité aux applications sur BD.

IX.3. LES ATELIERS DE GENIE LOGICIEL (AGL / CASE)

Les cibles pour le développement d'applications multi-utilisateurs sur BD apparaissent maintenant complètement:

- la métabase - Information Schema - du SGBD: c'est le réceptacle des définitions logiques et physiques des tables, vues, contraintes d'intégrité, droits, index, procédures et d'une façon générale de tout ce qui peut constituer le dictionnaire de définition et de manipulation cataloguée des données;

---

[26] Le "compil time"

[27] Le "run time"

[28] Sans plus attendre MicroSoft en a fait une implémentation sous le nom d'Open DataBase Connectivity (ODBC). Toute application utilisant ODBC et compilée sur PC sous MSWindows peut ouvrir successivement plusieurs connexions avec plusieurs SGBD.

- les PC ou stations et leurs bibliothèques graphiques : ils recevront les codes exécutables des programmes applicatifs, de préférence par téléchargement et configuration automatique;
- la base elle-même : avec nécessairement son "peuplement" initial de données, mais aussi, dans des "annexes" spécialement prévues à cet effet, les "images" des applications qui seront sauvegardées ou exportées en même temps que les données, ce qui, somme toute, peut rendre de grands services pour la sécurisation du système applicatif tout entier.

Quels outils rassembler pour une telle production logicielle et dans quel atelier les trouver ou les intégrer ?

IX.3.1. Outils détachés

Quel est la panoplie minimale d'outils ? En général, on souhaite au moins y trouver, dans l'ordre des productions citées ci-dessus :

- un outil d'*aide à la conception du schéma* logique des données, doté de plus ou moins d'expertise, éventuellement étendu pour l'aide à la conception du schéma physique;
- des *générateurs d'applications* pour interfaces graphiques mais aussi des *générateurs de rapports* imprimés;
- des *chargeurs de données* pour le peuplement initial de la BD à partir de fichier externes de formats variés;

auquels il faut ajouter les outils d'exploitation pour les administrateurs du SGBD et des BD:

- un *moniteur* des ressources statiquement ou dynamiquement allouées;
- des outils d'*export et d'import* .

IX.3.2. Ateliers intégrés

Que l'on suive ou non des méthodes générales de gestion des projets de développement logiciel - MERISE ou l'une des nouvelles méthodes Orientée Objet [29] - on peut vouloir se doter d'un environnement intégré pour rassembler de façon cohérente, avec une gestion minimale des versions des configurations - SCCS, SRCS ou mieux - toutes les pièces de la production d'une application sur BD, sans oublier l'aide en ligne, la documentation papier et les jeux et procédures de validation.

Trois stratégies sont alors possibles:

[29] Booch et OMT

- l'achat de l'*atelier d'un éditeur de SGBD* qui fait de l'intégration verticale[30], avec le confort d'une validation garantie par celui-ci pour toutes les étapes du cycle de production d'applications sur ce SGBD, et, également, avec les garanties de portabilité et d'ouverture offertes par cet éditeur, mais avec le danger d'une dépendance grandissante par rapport à cet éditeur (ses limites seront vos limites);
- l'achat d'un *atelier indépendant* par rapport aux différents SGBD comme par rapport aux différentes interfaces graphiques [31], mais aussi par rapport à l'intégration des autres outils détachés de production, comme les outils de "middleware" pour les architectures clients-serveurs;
- la constitution de son propre *atelier sur un standard ouvert* comme par exemple le "Portable Common Tool Environment" (PCTE) sous Unix, permettant l'intégration de tous les outils souhaités dont aucun n'imposera son environnement aux autres.

IX.4. CONCLUSIONS

Pouvoir développer les interfaces utilisateurs, et les programmes dont elles permettent le contrôle, en parallèle à la conception des schémas - conceptuels, externes et internes - d'une Base de Données est l'intérêt premier de ces logiciels généraux que l'on appelle Systèmes de Gestion de Bases de Données.

Si la séparation des programmes et des données a permis de fonder théoriquement les modèles, langages et architectures logicielles de ces SGBD, et même si cette séparation est encore un des fils directeurs des principales méthodes de développement d'application - y compris celles qui se disent orientées objet - il n'en reste pas moins que du point de vue de l'utilisateur terminal, les interfaces qui donnent une visualisation graphique des objets manipulés, les programmes qui, "derrière", manipulent ces objets en mémoire vive et le système qui gère le stockage persistant de ces objets en mémoire secondaire, doivent apparaître, en fonctionnement opérationnel, comme un seul et même système, avec le moins possible de "coutures" visibles.

Cette exigence conduit le double mouvement d'intégration verticale AGL/L4G/"moteur"BD et de définition de standards pour les interfaces entre les éléments de cette intégration. Ce qui libére les acheteurs en ouvrant la concurrence à des offres de technologies et d'architectures variées mais interopérables.

---

[30] Oracle en est l'exemple commercialement le plus "omniprésent".

[31] Uniface en est un bon exemple.

IX.5. REFERENCES

[AFNOR 86] AFNOR, *"Memorandum Ergonomie du Logiciel"*, 1986

[Batra 92] Dinesh BATRA and Ananth SRINIVASAN, *"A review and analysis of the usability od data management environments"*, International Journal of Man-Machines Studies (1992) **36**, 395-417

[Cartarci et al. 95] Tizinia CATARCI and M-F COSTABILE, *"Special Issue on Visual Query Systems"*, Journal of Visual Languages and Computing, (1995) 6-1, March 1995

X. BIBLIOGRAPHIE GENERALE

S. ABITEBOUL, R.HULL and V.VIANU	FUNDATIONS OF DATABASES, ADDISSON-WESLEY, 1995
M. ADIBA et C.DELOBEL	BASES DE DONNEES ET SYSTEMES RELATIONNELS, DUNOD, 1982
C.J. DATE	AN INTRODUCTION TO DATABASE SYSTEMS, ADDISSON-WESLEY, VOLUME 1, 4th Edition, 1987, VOLUME 2, 1983
C.J. DATE	A GUIDE TO THE SQL STANDARD ADDISSON-WESLEY, 1987
R. ELMASRI and S.B. NAVATHE	FUNDAMENTALS OF DATABASES SYSTEMS ADDISSON-WESLEY, World Student Series, 2nd Ed, 1994
G. GARDARIN	BD: LES SYSTEMES ET LEURS LANGAGES, EYROLLES, 3e Edition, 1985
G. GARDARIN	MAITRISER LES BD: MODELES ET LANGAGES, EYROLLES, 1993
J.MELTON and A.SIMON	UNDERSTANDING THE NEW SQL: A COMPLETE GUIDE, MORGAN KAUFMANN, 1993
J.D. ULLMAN	PRINCIPLES OF DATABASE SYSTEMS, COMPUTER SCIENCE PRESS, 1982
G. VOSSEN	DATA MODELS, DB LANGUAGES AND DBMS, ADDISON-WESLEY, 1991