



# L'OBJET-RELATIONNEL ET SQL3

## 1. INTRODUCTION

---

Le développement des SGBD objet s'est rapidement heurté à la nécessité de conserver la compatibilité avec l'existant. En effet, la fin des années 80 a vu le déploiement des SGBD relationnels et des applications client-serveur. Les utilisateurs n'étaient donc pas prêts à remettre en cause ces nouveaux systèmes dès le début des années 90.

Cependant, les tables relationnelles ne permettaient guère que la gestion de données alphanumériques. Avec l'avènement du Web au milieu de la décennie 90, la nécessité de supporter des données complexes au sein de la base de données s'est amplifiée. Ces données complexes peuvent être des documents textuels, des données géométriques ou géographiques, audiovisuelles ou soniques. En bref, il s'agit de supporter de manière intelligente des données multimédia (voir figure XIII.1).

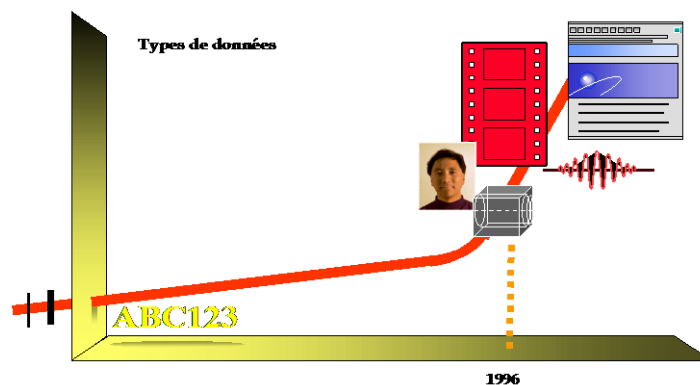


Figure XIII.1 — Le besoin de types de données multimédia

Ce chapitre est consacré à l'étude du modèle objet-relationnel et du langage de requêtes associé SQL3. SQL3 est une extension de SQL2, développée par le groupe de normalisation ANSI X3 H2 et internationalisée au niveau de l'ISO par le groupe ISO/IEC JTC1/SC21/WG3. Il s'agit de

la nouvelle version de SQL. SQL3 comporte beaucoup d'aspects nouveaux, l'essentiel étant sans doute son orientation vers une intégration douce de l'objet au relationnel. Ce chapitre traite en détail de cette intégration et survole les autres aspects de SQL3.

Après cette introduction, la section 2 motive l'intégration de l'objet au relationnel, en soulignant les forces et faiblesses du relationnel, qui justifient à la fois sa conservation et son extension. La section 3 définit les notions de base dérivées de l'objet et introduites pour étendre le relationnel. La section 4 propose une vue d'ensemble de SQL3 et du processus de normalisation. La section 5 détaille le support des objets en SQL3 avec de nombreux exemples. La section 6 résume les caractéristiques essentielles du langage de programmation de procédures et fonctions SQL/PSM, appoint essentiel à SQL pour assurer la complétude en tant que langage de programmation. Nous concluons ce chapitre en soulignant quelques points obscurs et l'intérêt d'une fusion entre OQL et SQL3.

## **2. POURQUOI INTEGRER L'OBJET AU RELATIONNEL ?**

---

Le modèle relationnel présente des points forts indiscutables, mais il a aussi des points faibles. L'objet répond à ces faiblesses. D'où l'intérêt d'une intégration douce.

### **2.1 Forces du modèle relationnel**

Le relationnel permet tout d'abord une modélisation des données adaptée à la gestion à l'aide de tables dont les colonnes prennent valeurs dans des domaines alphanumériques. Le concept de table dont les lignes constituent les enregistrements successifs est simple, bien adapté pour représenter par exemple des employés, des services, ou des paiements. Avec SQL2, les domaines se sont diversifiés. Les dates, les monnaies, le temps et les intervalles de temps sont parfaitement supportés.

Dans les années 80, les SGBD relationnels se sont centrés sur le transactionnel (*On Line Transaction Processing* — OLTP) et sont devenus très performants dans ce contexte. Les années 90 ont vu le développement du décisionnel (*On Line Analysis Processing* — OLAP). Le relationnel s'est montré capable d'intégrer la présentation multidimensionnelle des données nécessaire à l'OLAP. En effet, il est simple de coder un cube 3D en table, trois colonnes mémorisant les valeurs d'une dimension et la quatrième la grandeur analysée.

En bref, grâce au langage assertionnel puissant que constitue le standard universel SQL2, à sa bonne adaptation aux architectures client-serveur de données, à sa théorie bien assise aussi bien pour la conception des bases (normalisation), l'optimisation de requêtes (algèbre, réécriture, modèle de coûts) et à la gestion de transactions (concurrency, fiabilité) intégrée, le relationnel s'est imposé dans l'industrie au cours des années 80.

### **2.2 Faiblesses du modèle relationnel**

Le relationnel présente cependant des faiblesses qui justifient son extension vers l'objet. Tout d'abord, les règles de modélisation imposées pour structurer proprement les données s'avèrent trop restrictives.

L'**absence de pointeurs visibles** par l'utilisateur est très pénalisante. Dans les architectures modernes de machine où la mémoire à accès directe est importante, il devient intéressant de chaîner directement des données. Le parcours de références est rapide et permet d'éviter les jointures par valeurs du relationnel. Des enregistrements volumineux représentant de gros objets peuvent aussi être stockés une seule fois et pointés par plusieurs lignes de tables : le partage référentiel d'objets devient nécessaire. Par exemple, une image sera stockée une seule fois, mais pointée par deux tuples, un dans la table Employés, l'autre dans la table Clients. Passer par une clé d'identification (par exemple, le numéro de sécurité sociale) nécessite un accès par jointure, opération lourde et coûteuse. En clair, il est temps de réhabiliter les pointeurs sous la forme d'identifiants d'objets invariants au sein du relationnel.

Le **non-support de domaines composés** imposé par la première forme normale de Codd est inadapté aux objets complexes, par exemple aux documents structurés. L'introduction de champs binaires ou caractères longs (*Binary Large Object* — BLOB, *Character Large Object* — CLOB) et plus généralement de champs longs (*Large Object* — LOB) est insuffisante.

#### **Notion XIII.1 : Objet long (*Large Object*)**

Domaine de valeurs non structurées constitué par des chaînes de caractères (CLOB) ou de bits (BLOB) très longues (pouvant atteindre plusieurs giga-octets) permettant le stockage d'objets multimédia dans les colonnes de tables relationnelles.

Les objets longs présentent deux inconvénients majeurs : ils ne sont pas structurés et ne supportent pas les recherches associatives. La seule possibilité est de les lire séquentiellement. Ils sont donc particulièrement insuffisants pour le stockage intelligent de documents structurés, où l'on souhaite accéder par exemple à une section sans faire défiler tout le document. En clair, il faut pouvoir lever la contrainte d'atomicité des domaines, serait-ce que pour stocker des attributs composés (par exemple l'adresse composée d'une rue, d'un code postal et d'une ville) ou multivalués (par exemple les points d'une ligne).

La **non intégration des opérations** dans le modèle relationnel constitue un manque. Celui-ci résulte d'une approche traditionnelle où l'on sépare les traitements des données. Certes, les procédures stockées ont été introduites dans le relationnel pour les besoins des architectures client-serveur, mais celles-ci restent séparées des données. Elles ne sont pas intégrées dans le modèle. Il est par exemple impossible de spécifier des attributs cachés de l'utilisateur, accessibles seulement via des opérations manipulant ces attributs privés.

Tout ceci conduit à un mauvais support des applications non standard telles que la CAO, la CFAO, les BD géographiques et les BD techniques par les SGBD relationnels. En effet, ces applications gèrent des objets à structures complexes, composés d'éléments chaînés souvent représentés par des graphes. Le relationnel pur est finalement mal adapté. Il en va de même pour le support d'objets multimédia que l'on souhaite pouvoir rechercher par le contenu, par exemple des photos que l'on souhaite retrouver à partir d'un portrait robot.

## 2.3 L'apport des modèles objet

Les modèles objet sont issus des langages objets. Ils apportent des concepts nouveaux importants qui répondent aux manques du relationnel, comme l'identité d'objets, l'encapsulation, l'héritage et le polymorphisme.

L'**identité d'objet** permet l'introduction de pointeurs invariants pour chaîner les objets entre eux. Ces possibilité de chaînage rapide sont importantes pour les applications nécessitant le support de graphes d'objets. On pourra ainsi parcourir rapidement des suite d'associations, par navigation dans la base. Par exemple, passer d'un composé à ses composants puis à la description de ces composants nécessitera simplement deux parcours de pointeurs, et non plus des jointures longues et coûteuses.

L'**encapsulation des données** permet d'isoler certaines données dites privées par des opérations et de présenter des interfaces stables aux utilisateurs. Ceci facilite l'évolution des structures de données qui peuvent changer sans modifier les interfaces publiques seules visibles des utilisateurs. Au lieu d'offrir des données directement accessibles aux utilisateurs, le SGBD pourra offrir des services cachant ces données, beaucoup plus stables et complets. Ceux-ci pourront plus facilement rester invariants au fur et à mesure de l'évolution de la base de données.

L'**héritage d'opérations et de structures** facilite la réutilisation des types de données. Il permet plus généralement l'adaptation de composants à son application. Les composants pourront être des tables, mais aussi des types de données abstraits, c'est-à-dire un groupe de fonctions encapsulant des données cachées. Il sera d'ailleurs possible de définir des opérations abstraites, qui pourront, grâce au polymorphisme, porter le même nom mais s'appliquer sur des objets différents avec pour chaque type d'objet un code différent. Cela simplifie la vie du développeur d'applications qui n'a plus qu'à se soucier d'opérations abstraites sans regarder les détails d'implémentation sur chaque type d'objet.

## 2.4 Le support d'objets complexes

L'apport essentiel de l'objet-relationnel est sans doute le support d'objets complexes au sein du modèle. Ceci n'est pas inhérent à l'objet, mais plutôt hérité des premières extensions tendant à faire disparaître la première forme normale du relationnel. On parle alors de base de données en non première forme normale (NF<sup>2</sup>).

**Notion XIII.2 : Non première forme normale (*Non First Normal Form - NF<sup>2</sup>*)**

Forme normale tolérant des domaines multivalués.

Un cas particulier intéressant est le modèle relationnel imbriqué [Scholl86].

### Notion XIII.3 : Modèle relationnel imbriqué (*Nested Relational Model*)

Modèle relationnel étendu par le fait qu'un domaine peut lui-même être valué par des tables de même schéma.

Par exemple, des services employant des employés et enregistrant des dépenses pourront directement être représentés par une seule table externe imbriquant des tables internes :

```
SERVICES (N° INT, CHEF VARCHAR, ADRESSE VARCHAR, {EMPLOYES (NOM, AGE) }, {DEPENSES (NDEP INT, MONTANT INT, MOTIF VARCHAR) })
```

Employés correspond à une table imbriquée (ensemble de tuples) de schéma (Nom, Age) pour chaque service, et de même, Dépenses correspond à une table imbriquée pour chaque service. Notons que le modèle relationnel permet une imbrication à plusieurs niveaux, par exemple, Motif pourrait correspondre à une table pour chaque dépense.

N°	Chef	Adresse	Employés		Dépenses		
			Nom	Age	NDep	Montant	Motif
24	Paul	Versailles	Pierre	45	1	2600	Livres
			Marie	37	2	8700	Mission
					3	15400	Portable
25	Patrick	Paris	Eric	42	5	3000	Livres
			Julie	51	7	4000	Mission

Figure XIII.2 — Exemples de relations imbriquées

Le multimédia, particulièrement la géométrie, a nécessité d'introduire des attributs multivalués plus généraux. Comme l'objet, l'objet-relationnel offre des collections génériques prédéfinies telles que des listes, ensembles, tableaux qui peuvent être imbriqués pour représenter des objets très compliqués. Par exemple, on pourra définir une ligne comme une liste de points, chaque point étant défini par une abscisse et une ordonnée. Une région pourra être définie par une liste de lignes fermée. Ces possibilités sont essentielles pour la représentation de structures complexes. Celles-ci peuvent être cachées par des fonctions offertes aux utilisateurs afin d'en simplifier la vision. Ainsi, objets complexes et encapsulation seront souvent couplés.

## 3. LE MODELE OBJET-RELATIONNEL

Le modèle objet-relationnel est fondé sur l'idée d'extension du modèle relationnel avec les concepts essentiels de l'objet (voir figure XIII.3). Le cœur du modèle reste donc conforme au

relationnel, mais l'on ajoute les concepts clés de l'objet sous une forme particulière pour faciliter l'intégration des deux modèles.

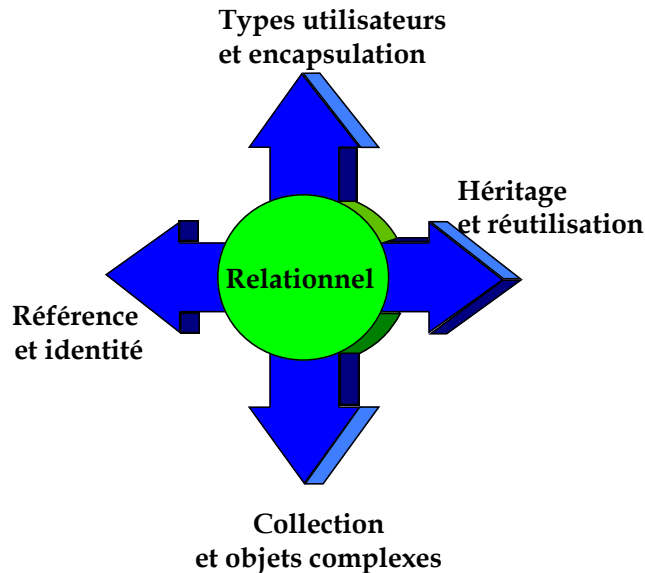


Figure XIII.3 — Les extensions apportées au relationnel

### 3.1 Les concepts additionnels essentiels

Les types utilisateur permettent la définition de types extensibles utilisables comme des domaines, supportant l'encapsulation.

**Notion XIII.4 : Type de données utilisateur (*User data type*)**

Type de données définissable par l'utilisateur composé d'une structure de données et d'opérations encapsulant cette structure.

Le système de type du SGBD devient donc extensible, et n'est plus limité aux types alphanumériques de base, comme avec le relationnel pur et SQL2. On pourra par exemple définir des types texte, image, point, ligne, etc. Les types de données définissables par l'utilisateur sont appelés **types abstraits (ADT)** en SQL3. La notion de type abstrait fait référence au fait que l'implémentation est spécifique de l'environnement : seule l'interface d'un type utilisateur est visible.

**Notion XIII.5 : Patron de collection (*Collection template*)**

Type de données générique permettant de supporter des attributs multivalués et de les organiser selon un type de collection.

Les SGBD objet-relationnels offrent différents types de collections, tels que le tableau dynamique, la liste, l'ensemble, la table, etc. Le patron LISTE(X) pourra par exemple être instancié pour définir des lignes sous forme de listes de points : LIGNE LISTE(POINT).

### **Notion XIII.6 : Référence d'objet (*Object Reference*)**

Type de données particulier permettant de mémoriser l'adresse invariante d'un objet ou d'un tuple.

Les références sont les identifiants d'objets (OID) dans le modèle objet-relationnel. Elles sont en théorie des adresses logiques invariantes qui permettent de chaîner directement les objets entre eux, sans passer par des valeurs nécessitant des jointures.

### **Notion XIII.7 : Héritage de type (*Type inheritance*)**

Forme d'héritage impliquant la possibilité de définir un sous-type d'un type existant, celui-ci héritant alors de la structure et des opérations du type de base.

L'héritage de type permet donc de définir un sous-type d'un type SQL ou d'un type utilisateur. Une table correspondant à un type sans opération, l'objet-relationnel permet aussi l'**héritage de table**.

### **Notion XIII.8 : Héritage de table (*Table inheritance*)**

Forme d'héritage impliquant la possibilité de définir une sous-table d'une table existante, celle-ci héritant alors de la structure et des éventuelles opérations de la table de base.

En résumé, le modèle objet-relationnel conserve donc les notions de base du relationnel (domaine, table, attribut, clé et contrainte référentielle) auxquelles il ajoute les concepts de type utilisateur avec structure éventuellement privée et opérations encapsulant, de collections supportant des attributs multivalués (structure, liste, tableau, ensemble, etc.), d'héritage sur relations et types, et d'identifiants d'objets permettant les références inter-tables. Les domaines peuvent dorénavant être des types utilisateurs. La figure XIII.4 illustre tous ces concepts. Deux vues sont possibles : la vue relationnelle dans laquelle les relations sont des relations entre objets (voir figure XIII.5), chaque colonne étant en fait valuée par un objet ; la vue objet (voir figure XIII.6) où une ligne d'une table peut correspondre à un objet simple (un tuple) ou complexe (un tuple avec des attributs complexes).

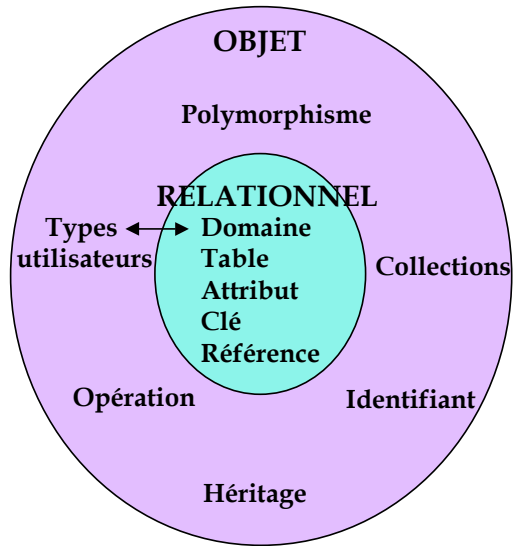


Figure XIII.4 — Les concepts de l'objet-relationnel













Accident	Rapport	Photo
134		
219		
037		

Figure XIII.5 — Une relation entre objets

Police	Nom	Adresse	Conducteurs		Accidents		
24	Paul	Paris	Conducteur	Age	134		
			Paul	45			
			Robert	17			
219	Paul	Paris			219		
037	Paul	Paris			037		

**Objet Police**

Figure XIII.6 — Un objet complexe dans une table



### 3.2 Les extensions du langage de requêtes

Au-delà du modèle, il est nécessaire d'étendre le langage de manipulation de données pour supporter les nouvelles avancées du langage de définition. SQL3 va donc comporter un langage de requêtes étendu avec notamment les appels d'opérations en résultat et en qualification, la surcharge des opérateurs de base de SQL (par exemple l'addition), le support automatique de l'héritage, le parcours de références, la constitution de tables imbriquées en résultat, etc. Nous étudierons ces différentes fonctionnalités à l'aide d'exemples.

## 4. VUE D'ENSEMBLE DE SQL3

---

SQL3 est une spécification volumineuse, qui va bien au-delà de l'objet-relationnel. Nous examinons dans cette section ses principaux composants.

### 4.1 Le processus de normalisation

Au-delà du groupe ANSI nord-américain très actif, la normalisation est conduite par un groupe international dénommé ISO/IEC JTC1/SC 21/WG3 DBL, DBL signifiant *Database Language*. Les pays actifs sont essentiellement l'Australie, le Brésil, le Canada, la France, l'Allemagne, le Japon, la Corée, la Hollande, le Royaume-Uni et bien sûr les Etats-Unis. L'essentiel du travail est bien préparé au niveau du groupe ANSI X3H2 (<http://www.ansi.org>) dont Jim Melton est le président. Ce groupe a déjà réalisé le standard SQL2 présenté dans la partie relationnelle. Les standards acceptés étaient contrôlés aux Etats-Unis par le NIST (<http://ncsl.nist.gov>) qui définissait des programmes de tests de conformité. Ceux-ci existent partiellement pour SQL2 mais pas pour SQL3. Depuis 1996, le congrès américain a supprimé les crédits du NIST si bien qu'il n'y a plus de programmes de validation espérés ...

### 4.2 Les composants et le planning

Alors que SQL2 était un unique document [SQL92], SQL3 a été divisé en composants pour en faciliter la lisibilité et la manipulation, l'ensemble faisant plus de 1.500 pages. Les composants considérés sont brièvement décrits dans le tableau représenté figure XIII.7.

Partie	Contenu	Titre	Description
1	le cadre	SQL/Framework	Une description non-technique de la façon dont le document est structuré et une définition des concepts communs à toute les parties.
2	les fondements	SQL/Foundation	Le noyau de base, incluant les types de données utilisateurs et le modèle objet-relationnel.
3	l'interface client	SQL/CLI	L'interface d'appel client permet le dialogue client-serveur pour l'accès aux données via SQL2 puis SQL3.
4	les procédures stockées	SQL/PSM	Le langage de spécification de procédures stockées.
5	l'intégration aux langages classiques	SQL/Bindings	Les liens SQL dynamique et "embedded" SQL repris de SQL-92 et étendus à SQL3.
6	la gestion de transactions	SQL/Transaction	Une spécification de l'interface XA pour moniteur transactionnel distribué.
7	la gestion du temps	SQL/Temporal	Le support du temps, des intervalles temporels et des séries temporelles.
8	abandonné		
9	l'accès aux données externes	SQL/MED	L'utilisation de SQL pour accéder à des données non-SQL.
10	l'intégration aux langages objet	SQL/OBJ	L'utilisation de SQL depuis un langage objet, tel C++, Smalltalk ou Java.

Figure XIII.7 — Les composants de SQL3

Le planning de SQL3 a été décalé a plusieurs reprises, ces décalages témoignant de la difficulté et de l'ampleur de la tâche. Les parties 1 à 7 de SQL sont à l'état de brouillons internationaux (*Draft International Standard*) depuis fin 1997. Ils devraient devenir de véritables standards fin 1998 ou en 1999. Les deux derniers sont prévus pour l'an 2000.

Au-delà de SQL3, d'autres composants sont prévus tels SQL/MM pour la spécification de types utilisateurs multimédia et SQL/RDA pour la spécification du protocole RDA de transfert de données entre client et serveur. On parle déjà de SQL4, sans doute pour après l'an 2000.

### 4.3 La base

Le composant 2 contient l'essentiel du langage SQL3, en particulier les fonctionnalités de base pour définir les autres composants (*Basic SQL/CLI capabilities*, *Basic SQL/PSM capabilities*), les déclencheurs (*Triggers*), les types utilisateurs appelés types abstraits (*Abstract Data Types*) et les fonctionnalités orientées objets que nous allons étudier en détail ci-dessous. Un mot sur les déclencheurs : bien que partie intégrante du relationnel étudiée en tant que telle au chapitre VIII,

les déclencheurs ne sont normalisés que dans SQL3. Il s'agit là de combler une lacune de la normalisation.

## 5. LE SUPPORT DES OBJETS

Dans cette section, nous allons présenter l'essentiel des commandes composant les fonctionnalités objet de SQL3 : définition de types abstraits, support d'objets complexes, héritage de type et de table. Pour chaque commande, nous donnons la syntaxe simplifiée et quelques exemples.

### 5.1 Les types abstraits

#### 5.1.1 Principes

La première nouveauté de SQL3 est l'apparition d'une commande CREATE TYPE. Au-delà des types classiques (numériques, caractères, date) de SQL, il devient possible de créer des types dépendant de l'application, multimédia par exemple (voir figure XIII.8). Une instance d'un type peut être un objet ou une valeur. Un objet possède alors un OID et peut être référencé. Une valeur peut simplement être copiée dans une ligne d'une table. Un type possède en général des colonnes, soit directement visibles, soit cachées et accessibles seulement par des fonctions encapsulant le type. Les opérateurs arithmétiques de SQL (+, \*, -, /) peuvent être redéfinis au niveau d'un type. Par exemple, on redéfinira l'addition pour les nombres complexes. Enfin, un type peut posséder des clauses de comparaison (=, <) permettant d'ordonner les valeurs et peut être représenté sous la forme d'un autre type, une chaîne de caractères par exemple.

Types simples	Numérique	245
	Caractères	"Jean"
	Date	20-Dec-98
Types multimédias	Texte	
	Spatial	
	Image	
	Vidéo	
	Spécialisé	Application

Figure XIII.8 — Exemples de types de données extensibles

#### 5.1.2 Syntaxe

La syntaxe de la commande de déclaration de type est la suivante :

```
CREATE [DISTINCT] TYPE <NOM ADT> [<OPTION OID>] [<CLAUSE SOUS-TYPE>]
[AS] (<CORPS DE L'ADT>)
```

Le mot clé DISTINCT est utilisé pour renommer un type de base existant déjà, par exemple dans la commande :

```
CREATE DISTINCT TYPE EURO AS (DECIMAL(9,2))
```

La clause :

```
<OPTION OID> ::= WITH OID VISIBLE
```

permet de préciser la visibilité de l'OID pour chaque instance (objet). Par défaut, une instance de type est une valeur sans OID. Cette clause semble plus ou moins avoir disparue de la dernière version de SQL3, le choix étant reporté sur les implémentations.

La clause :

```
<CLAUSE SOUS-TYPE> ::= UNDER <SUPERTYPE> [, <SUPERTYPE>]...
```

permet de spécifier les super-types dont le type hérite. Il y a possibilité d'héritage multiple avec résolution explicite des conflits.

La clause <CORPS DE L'ADT> se compose d'une ou plusieurs clauses membres :

```
<CLAUSES MEMBRES> ::=
    < DEFINITION DE COLONNE >
    | < CLAUSE EGALE >
    | < CLAUSE INFÉRIEURE >
    | < CLAUSE CAST >
    | < DECLARATION DE FONCTION >
    | < DECLARATION D' OPERATEUR >
```

La définition de colonne permet de spécifier les attributs du type sous la forme <NOM> <TYPE>. Un type peut bien sûr être un type de base ou un type précédemment créé par une commande CREATE TYPE. Un attribut peut être privé ou public, comme en C++. La clause égale permet de définir l'égalité de deux instances du type, alors que la clause inférieure définit l'opérateur logique <, important pour comparer et ordonner les valeurs du type. La clause CAST permet de convertir le type dans un autre type. Plusieurs clauses CAST sont possibles.

Une déclaration de fonction permet de définir les fonctions publiques qui encapsulent le type. Il existe différents types de fonctions : les **constructeurs** permettent de construire des instances du type ; ils portent en général le nom du type ; les **acteurs** agissent sur une instance en manipulant les colonnes du type ; parmi les acteurs, il est possible de distinguer les **observateurs** qui ne font que lire l'état et les **mutateurs** qui le modifient ; les **destructeurs** détruisent les instances et doivent être appelés explicitement pour récupérer la place disque (pas de ramasse-miettes). Plus précisément, la syntaxe d'une définition de fonction est la suivante :

```
<DECLARATION DE FONCTION> ::=
    [<FUNCTION TYPE>] FUNCTION <FUNCTION NAME> <PARAMETER LIST>
    RETURNS <FUNCTION RESULTS>
    { <SQL PROCEDURE> | <FILE NAME> } END FUNCTION
```

Le type est défini par :

**<FUNCTION TYPE> ::= CONSTRUCTOR | ACTOR | DESTRUCTOR**

Les fonctions SQL3 ne sont pas forcément associées à un type ; elles peuvent aussi être associées à une base ou à une table. Elles peuvent être programmées en SQL, en SQL/PSM (voir ci-dessous) ou dans un langage externe comme COBOL, C, C++ ou Java. Comme en C++, un opérateur est une fonction portant un nom particulier et déclarée comme opérateur.

### 5.1.3 Quelques exemples

Voici quelques exemples de création de types :

1. Type avec OID pouvant être utilisé comme un objet :

```
CREATE TYPE PHONE WITH OID VISIBLE (PAYS VARCHAR, ZONE VARCHAR,  
NOMBRE INT, DESCRIPTION CHAR(20))
```

2. Type sans référence :

```
CREATE TYPE PERSONNE (NSS INT, NOM VARCHAR, TEL PHONE)
```

3. Sous-type :

```
CREATE TYPE ETUDIANT UNDER PERSONNE (CYCLE VARCHAR, ANNEE INT)
```

4. Type énuméré :

```
CREATE TYPE JOUR-OUVRE (LUN, MAR, MER, JEU, VEN);
```

5. Type avec OID et fonction :

```
CREATE TYPE EMPLOYE WITH OID VISIBLE  
(NOM CHAR(10), DATENAIS DATE, REPOS JOUR-OUVRE, SALAIRE FLOAT,  
FUNCTION AGE (E EMPLOYE) RETURNS (INT)  
{ CALCUL DE L'AGE DE E }  
END FUNCTION);
```

6. Autre type sans OID avec fonction :

```
CREATE TYPE ADRESSE WITHOUT OID  
(RUE CHAR(20), CODEPOST INT, VILLE CHAR(10),  
FUNCTION DEPT(A ADRESSE) RETURNS (VARCHAR) END FUNCTION);
```

7. Création de procédure SQL de mise à jour associée au type employé :

```
CREATE procedure AUGMENTER (E EMPLOYE, MONTANT MONEY)  
{ UPDATE EMPLOYE SET SALAIRE = SALAIRE + MONTANT WHERE EMPLOYE.OID = E }  
END PROCEDURE
```

## 5.2 Les constructeurs d'objets complexes

SQL3 supporte la construction d'objets complexes par le biais de types `collection` paramétrés. Ces types génériques, appelés patrons de collections, doivent être fournis comme une bibliothèque avec le compilateur SQL3. Celui-ci doit assurer la généralité et en particulier la possibilité de déclarer des collections de types d'objets ou de valeurs quelconques. Chaque patron de collection fournit des interfaces spécifiques pour accéder aux éléments de la collection.

Tout environnement SQL3 doit offrir les patrons de base SET(T), MULTISET(T) et LIST(T). Un *multiset*, encore appelé *bag*, se comporte comme un ensemble mais permet de gérer des doubles. Par exemple, il est possible de définir un type personne avec une liste de prénoms et un ensemble de téléphones comme suit :

```
CREATE TYPE PERSONNE WITH OID VISIBLE
(NSS INT, NOM VARCHAR, PRENOMS LIST(VARCHAR), TEL SET(PHONE)).
```

Il est aussi possible de créer un ensemble de personnes :

```
CREATE TYPE PERSONNES (CONTENU SET (PERSONNE))
```

ou un ensemble d'identifiants de personnes :

```
CREATE TYPE RPERSONNES (CONTENU SET (REF (PERSONNE)))
```

Au-delà des types de collections SET, LIST et MULTISET, SQL3 propose des types additionnels multiples : piles (*stack*), files (*queue*), tableaux dynamiques (*varray*), tableaux insérables (*iarray*) pour gérer des textes par exemple. Un tableau dynamique peut grandir par la fin, alors qu'un tableau insérable peut aussi grandir par insertion à partir d'une position intermédiaire (il peut donc grandir par le milieu). Ces types de collections sont seulement optionnels : ils ne sont pas intégrés dans le langage de base mais peuvent être ajoutés.

De manière générale, il est possible de référencer des objets via des OID mémorisés comme valeurs d'attributs. De tels attributs sont déclarés références (REF) comme dans l'exemple ci-dessous :

```
CREATE TYPE VOITURE (NUMERO CHAR(9), COULEUR VARCHAR,
PROPRIETAIRE REF (PERSONNE))
```

### 5.3 Les tables

Les tables SQL restent inchangées, à ceci près que le domaine d'un attribut peut maintenant être un type créé par la commande CREATE TYPE. De plus, des fonctions peuvent être définies au niveau d'une table pour encapsuler les tuples vus comme des objets. Aussi, l'héritage de table qui permet de réutiliser une définition de table est possible.

Par exemple, la table représentée figure XIII.5 sera simplement créée par la commande :

```
CREATE TABLE ACCIDENTS (ACCIDENT INT, RAPPORT TEXT, PHOTO IMAGE)
```

en supposant définis les types TEXT (texte libre) et IMAGE, qui sont généralement fournis avec un SGBD objet-relationnel, comme nous le verrons plus loin.

La table de la figure XIII.6 pourra être définie à partir des mêmes types avec en plus :

```
CREATE TYPE CONDUCTEUR (CONDUCTEUR VARCHAR, AGE INT)
CREATE TYPE ACCIDENT (ACCIDENT INT, RAPPORT TEXT, PHOTO IMAGE)
```

par la commande :

```
CREATE TABLE POLICE (NPOLICE INT, NOM VARCHAR, ADRESSE ADRESSE, CONDUCTEURS
SET (CONDUCTEUR), ACCIDENTS LIST (ACCIDENT)).
```

SQL3 donne aussi des facilités pour passer d'un type à une table de tuples de ce type et réciproquement, d'une table au type correspondant.

Par exemple, en utilisant le type `EMPLOYE` défini ci-dessus, il est possible de définir simplement une table d'employés par la commande :

```
CREATE TABLE EMPLOYES OF EMPLOYE ;
```

En définissant une table `VINS`, il sera possible de définir le type `VIN` composé des mêmes attributs comme suit :

```
CREATE TABLE VINS (NV INT, CRU VARCHAR, DEGRE FLOAT (5.2))
OF NEW TYPE VIN ;
```

Comme mentionné, l'héritage de table qui recopie la structure est possible par la clause :

```
CREATE TABLE <TABLE> UNDER <TABLE> [WITH (LISTE D'ATTRIBUTS TYPES)] .
```

La liste des attributs typés permet d'ajouter les attributs spécifiques à la sous-table. Par exemple, une table de vins millésimé pourra être créée par :

```
CREATE TABLE VINSMILL UNDER VINS WITH (MILLESIME INT, QUALITE VARCHAR) .
```

## 5.4 L'appel de fonctions et d'opérateurs dans les requêtes

Le langage de requêtes est étendu afin de supporter les constructions nouvelles du modèle. L'extension essentielle consiste à permettre l'appel de fonctions dans les termes SQL, qui peuvent figurer dans les expressions de valeurs sélectionnées ou dans les prédicats de recherche. Une fonction s'applique à l'aide de la notation parenthésée sur des termes du type des arguments spécifiés.

Considérons par exemple la table d'employés contenant des instances du type `EMPLOYE` défini ci-dessus. La requête suivante retrouve le nom et l'âge des employés de moins de 35 ans :

```
SELECT E.NOM, AGE (E)
FROM EMPLOYES E
WHERE AGE (E) < 35;
```

La notation pointée appliquée au premier argument est aussi utilisable pour invoquer les fonctions :

```
SELECT E.NOM, E..AGE ()
FROM EMPLOYES E
WHERE E..AGE () < 35;
```

Il s'agit d'un artifice syntaxique, la dernière version utilisant d'ailleurs la double notation pointée (`..`) pour les fonctions et attributs composés, la notation pointée simple (`.`) étant réservée au SQL de base (accès à une colonne usuelle de tuple).

Au-delà des fonctions, SQL3 permet aussi l'accès aux attributs composés par la notation pointée. Supposons par exemple une table d'employés localisés définies comme suit :

```
CREATE TABLE EMPLOYESLOC UNDER EMPLOYES WITH (ADDRESS ADRESSE) .
```

La requête suivante permet de retrouver le nom et le jour de repos des employés des Bouches-du-Rhône habitant Marseille :

```
SELECT NOM, REPOS
FROM EMPLOYESLOC E
WHERE DEPT (E.ADDRESS) = "BOUCHES DU RHONE"
AND E.ADDRESS..VILLE = "MARSEILLE";
```

Notons dans la requête ci-dessus l'usage de fonctions pour extraire le département à partir du code postal et l'usage de la notation pointée pour extraire un champ composé. Nous préférons utiliser partout la notation pointée ; il faut cependant distinguer fonction et accès à attribut par la présence de parenthèses.

Afin d'illustrer plus complètement, supposons définis un type point et une fonction distance entre deux points comme suit :

```
TYPE POINT ( ABSCISSE X, ORDONNEE Y,
FUNCTION DISTANCE (P1 POINT, P2 POINT) RETURN (REAL) ) ;
```

Considérons une implémentation des employés par la table :

```
EMPLOYES (MUNEMP INT, NOM VARCHAR, ADDRESS ADRESSE, POSITION POINT) ;
```

La requête suivante recherche alors les noms et adresses de tous les employés à moins de 100 unités de distance (par exemple le mètre) de l'employé Georges :

```
SELECT E.NOM, E.ADDRESS
FROM EMPLOYES G, EMPLOYES E
WHERE G.NAME = "GEORGES" AND DISTANCE (G.POSITION, E.POSITION) < 100 .
```

Supposons de plus défini le type CERCLE comme suit :

```
TYPE CERCLE (CENTRE POINT, RAYON REAL,
CONSTRUCTOR FUNCTION CERCLE (C POINT, R REAL) RETURN (CERCLE) )
```

Ajoutons une fonction booléenne CONTIENT au type point :

```
CREATE FUNCTION CONTIENT (P POINT, C CERCLE)
{ CODE DE LA FONCTION } RETURN (BOOLEAN)
END FUNCTION .
```

La question suivante retrouve les employés dont la position est contenu dans un cercle de rayon 100 autour de l'employé Georges :



```

SELECT E.NAME, E.ADDRESS
FROM EMPLOYES G, EMPLOYES E
WHERE EMPLOYES.NOM = "GEORGES" AND
CONTIENT(E.POSITION, CERCLE(G.POSITION, 1));

```

Les deux requêtes précédentes sont de fait équivalents et génèrent les mêmes réponses.

## 5.5 Le parcours de référence

SQL3 permet aussi de traverser les associations représentées par des attributs de type références.

Ces attributs peuvent être considérés comme du type particulier référence, sur lequel il est possible d'appliquer les fonctions Ref pour obtenir la valeur de l'OID et DeRef pour obtenir l'objet pointé. Afin de simplifier l'écriture, DeRef peut être remplacée par la notation  $\rightarrow$ . Celle-ci permet donc les parcours de chemins. On peut ainsi écrire des requêtes mélangeant la notation simple point (accès à un attribut), double point (accès à un attribut ADT composé) et flèche (parcours de chemins). Espérons que la notation pointé simple pourra être utilisée partout comme une amélioration syntaxique !

Considérons le type `VOITURE` déjà défini ci-dessus comme suit :

```

CREATE TYPE VOITURE (NUMERO CHAR(9), COULEUR VARCHAR,
PROPRIETAIRE REF(PERSONNE)).

```

Créons une table de voitures :

```

CREATE TABLE VOITURES OF TYPE VOITURE.

```

La requête suivante recherche les noms des propriétaires de voitures rouges habitant Paris :

```

SELECT V.PROPRIETAIRE→NOM
FROM VOITURES V
WHERE V.COULEUR = "ROUGE" AND V.PROPRIETAIRE→ADRESSE..VILLE = "PARIS".

```

SQL3 permet de multiples notations abrégées. En particulier, il est possible d'éviter de répéter des préfixes de chemins avec la notation pointée suivie de parenthèses. Par exemple, la requête suivante recherche les numéros des voitures dont le propriétaire habite les Champs-Élysées à Paris et a pour prénom Georges :

```

SELECT V.NUMERO
FROM VOITURES V
WHERE V.PROPRIETAIRE→(ADRESSE..(VILLE = "PARIS"
AND RUE = "CHAMPS ELYSES")) AND PRENOM = "GEORGES").

```

## 5.6 La recherche en collections

Les collections de base sont donc les ensembles, listes et sacs. Ces constructeurs de collections peuvent être appliqués sur tout type déjà défini. Les collections sont rendues permanentes en qualité d'attributs de tables. La construction `TABLE` est proposée pour transformer une collection en table et l'utiliser derrière un `FROM` comme une véritable table. Toute collection peut être

utilisée à la place d'une table précédée de ce mot clé TABLE. Par exemple, supposons l'ajout d'une colonne PASSETEMPS à la table des personnes évaluée par un ensemble de chaînes de caractères :

```
ALTER TABLE PERSONNES ADD COLUMN PASSETEMPS SET (VARCHAR).
```

La requête suivante retrouve les références des personnes ayant pour passe-temps le vélo :

```
SELECT REF (P)
FROM PERSONNES P
WHERE "VELO" IN
  SELECT *
  FROM TABLE (P.PASSETEMPS).
```

Plus complexe, la requête suivante recherche les numéros des polices d'assurance dont un accident contient un rapport avec le mot clé « Pont de l'Alma » :

```
SELECT P.NPOLICE
FROM POLICE P, TABLE P.ACCIDENTS A, TABLE A.RAPPORT..KEYWORDS M
WHERE M = "PONT DE L'ALMA".
```

Cette requête suppose la disponibilité de la fonction KEYWORDS sur le type TEXT du rapport qui délivre une liste de mots clés. Nous tablons tout d'abord la liste des accidents, puis la liste des mots clés du rapport de chaque accident. Ceci donne donc une sorte d'expression de chemins multivalués. SQL3 est donc très puissant !

## 5.7 Recherche et héritage

L'héritage de tables est pris en compte au niveau des requêtes. Ainsi, lorsqu'une table possède des sous-tables, la recherche dans la table retrouve toutes les lignes qui qualifient au critère de recherche, aussi bien dans la table que dans les sous-tables

Par exemple, considérons la table BUVEURS définie comme suit :

```
CREATE TABLE BUVEURS UNDER PERSONNES WITH ETAT ENUM (NORMAL, IVRE).
```

La recherche des personnes de prénom Georges par la requête :

```
SELECT NOM, PRENOM, ADRESSE
FROM PERSONNES
WHERE PRENOM = "GEORGES"
```

retournera à la fois les personnes et les buveurs de prénom Georges.

## 6. LE LANGAGE DE PROGRAMMATION PSM

---

Le composant PSM définit un L4G adapté à SQL. Il a été adopté tout d'abord dans le cadre SQL2, puis étendu à SQL3.

## 6.1 Modules, fonctions et procédures

SQL/PSM (*Persistent Store Modules*) est un langage de programmation de modules persistants et stockés dans la base de données [Melton98]. Les modules sont créés et détruits par des instructions spéciales `CREATE MODULE` et `DROP MODULE`. Un module se compose de procédures et/ou de fonctions. Il est possible de créer directement des procédures (`CREATE PROCEDURE`) ou des fonctions (`CREATE FUNCTION`) non contenues dans un module. Un module est associé à un schéma de base de données sur lequel peuvent être définies des autorisations et des tables temporaires. Il est aussi possible de définir des curseurs partagés entre les procédures et fonctions composantes.

SQL/PSM parachève SQL pour en faire un langage complet. Cependant, il est possible d'écrire des procédures stockées en pur SQL ou dans un langage pour lequel l'intégration avec SQL est définie (ADA, C, COBOL, FORTRAN, PASCAL, PL/I et MUMPS). Pour ces langages, PSM permet de définir les paramètres d'entrée et de retour des procédures (mots clés `IN` pour les paramètres d'entrée et `OUT` pour ceux de sortie) ou des fonctions (clause `RETURN <type>`). Ces procédures sont en général invoquées depuis des programmes hôtes par des ordres `EXEC SQL` plus ou moins spécifiques du langage hôte, ou directement depuis d'autres procédures PSM par des ordres `CALL <procédure>` ou des appels directs de fonctions.

## 6.2 Les ordres essentiels

L'intérêt essentiel de SQL/PSM est de fournir un langage de programmation homogène avec SQL et manipulant les mêmes types de données. Ce langage comporte des déclarations de variables, des assignations, des conditionnels `CASE`, `IF`, des boucles `LOOP`, `REPEAT`, `WHILE`, `FOR` et des traitements d'erreurs et exceptions `SIGNAL`, `RESIGNAL`.

Une déclaration de variable est analogue à une déclaration de colonne en SQL :

```
<DECLARATION DE VARIABLE> ::= DECLARE <VARIABLE> <TYPE> [DEFAULT <VALEUR>]
```

Par exemple, une déclaration possible est :

```
DECLARE PRIX INT DEFAULT 0 ;
```

Une assignation s'effectue par une clause `SET` suivie d'une expression de valeur conforme à SQL, ou par l'assignation du résultat d'une requête dans une variable. Par exemple :

```
DECLARE MOYENNE DECIMAL (7.2)
SELECT AVG (SALAIRE) INTO MOYENNE FROM EMPLOYES
```

assigne le résultat de la requête à la variable `MOYENNE`.

L'ordre `CASE` permet de tester différents cas de valeurs d'une expression de valeur en paramètre du `CASE` (expression de valeur1) ou de plusieurs expressions de valeurs booléennes :

```

<ORDRE CASE> ::=
CASE [<EXPRESSION DE VALEUR1>]
WHEN <EXPRESSION DE VALEUR2 > THEN <ORDRE SQL> ;
[WHEN <EXPRESSION DE VALEUR> THEN <ORDRE SQL> ;]...
ELSE <ORDRE SQL> ;
END CASE

```

Par exemple :

```

CASE MOYENNE
WHEN <100 THEN CALL PROC1 ;
WHEN = 100 THEN CALL PROC2 ;
ELSE CALL PROC3 ;
END CASE

```

permet de traiter différents cas de la variable MOYENNE.

Les boucles LOOP, REPEAT et WHILE sont des boucles de calculs en mémoire classiques, qui ne font pas directement appel à la base. La boucle FOR permet au contraire de parcourir le résultat d'une requête par le biais d'un curseur. Sa syntaxe simplifiée est la suivante :

```

<ORDRE FOR> ::=
[<ETIQUETTE> :]FOR <NOM DE BOUCLE>
AS [<NOM DE CURSEUR> CURSEUR FOR] <SPECIFICATION DE CURSEUR>
DO
<ORDRE SQL>
END FOR <ETIQUETTE> ;

```

L'étiquette permet de référencer la boucle FOR. Le nom de boucle sert de qualifiant pour les colonnes de la table virtuelle correspondant au curseur, afin de les distinguer si nécessaire. Le nom de curseur est optionnel pour permettre la réutilisation du curseur. Une définition de curseur est classique en SQL : il s'agit en général d'une requête.

### 6.3 Quelques exemples

Voici la syntaxe des constructions essentielles du langage et un exemple de procédure et de fonction. Nous travaillons sur la base des vins composées des tables :

```

VINS (NV INT, CRU VARCHAR, MILL INT, DEGRE FLOAT (5.2))
COMMANDES (NCO INT, NV INT, DAT DATE, QUANTITE INT)

```

Une procédure pour calculer la variance des quantités commandées d'un cru donné en paramètre peut être définie comme suit :

```

CREATE PROCEDURE (IN C VARCHAR,
OUT VARIANCE DECIMAL(10.2))
BEGIN
DECLARE MOYENNE DECIMAL(10.2) ;
VARIANCE = 0 ;
SELECT AVG(QUANTITE) INTO MOYENNE
FROM COMMANDES C, VINS V
WHERE V.NV = C.NV AND V.CRU = C ;
BOUCLE1:
FOR M AS SELECT NCO, QUANTITE
FROM COMMANDES C, VINS V
WHERE V.NV = C.NV AND V.CRU = C
DO
SET VARIANCE = VARIANCE+(M.QUANTITE - MOYENNE)**2
END FOR BOUCLE1 ;
END

```

Tout SQL est bien sûr intégré à PSM. Il serait possible d'utiliser un curseur et une boucle WHILE avec un FETCH dans la boucle à la place de la boucle FOR.

Pour montrer la complétude du langage, nous définissons la fonction factorielle comme suit :

```

CREATE FUNCTION FACTORIELLE (N INT)
RETURNS INT
DETERMINISTIC
BEGIN
DECLARE FAC INT DEFAULT 1 ;
DECLARE I INT ;
SET I = N ;
WHILE I > 1 DO
FAC = FAC*I ;
I = I-1 ;
END WHILE ;
RETURN FAC ;
END

```

D'autres types de boucles peuvent bien sûr être utilisés pour calculer cette fonction [Melton98].

## 6.4 Place de PSM dans le standard SQL

SQL/PSM est un standard international dans sa version SQL2. Il a été intégré en 1996 au standard SQL2. Une version plus complète est en cours de spécification pour SQL3. Cette version présentera l'avantage de pouvoir utiliser les types abstraits et les collections de SQL3 dans le langage de programmation. SQL/PSM est assez proche des langages de quatrième

génération de systèmes tels qu'ORACLE, INFORMIX ou SYBASE. Malheureusement, aucun système n'implémente actuellement exactement PSM.

## 7. CONCLUSION

---

SQL3 est un standard en évolution. Comme nous l'avons vu, les composants ne sont pas tous au même niveau d'avancement. La plupart sont au stade de brouillons internationaux, et vont donc subir encore des modifications. L'ensemble devrait être terminé vers l'an 2000.

SQL3 se situe, au moins pour la partie objet et les interfaces avec les langages objet, comme un concurrent de l'ODMG. Il est supporté par tous les grands constructeurs, comme IBM, Oracle et Sybase, et est impliqué dans un processus de standardisation international. Au contraire, l'ODMG est un accord entre quelques constructeurs de SGBD objet autour d'un langage pur objet. L'ODMG traite bien les aspects collections et, dans une moindre mesure, traversée de chemins. Les deux propositions pourraient donc apparaître comme complémentaires, à condition de converger. Il existe d'ailleurs un accord entre les groupes ANSI X3 H2 responsable de SQL et ODMG pour explorer la convergence. Souhaitons que cet accord aboutisse.

D'autres, comme Stonebraker [Stonebraker96], pensent plutôt que les bases de données objet ont un créneau d'application différent de celui des bases de données objet-relationnel. Les premières seraient plutôt orientées vers les applications écrites en C++ ou Java, naviguant dans les bases mais ne formulant pas de questions complexes sur de gros volumes de données persistantes. Au contraire, les bases de données objet-relationnel profiteraient de l'orientation disque du relationnel et seraient capables de supporter des questions complexes sur des données complexes. La figure XIII.9 résume ce point de vue. Il est possible de mesurer la complexité des requêtes en nombre de jointures et agrégats, la complexité des données en nombre d'associations. Cependant, l'évolution des systèmes objet vers la compatibilité SQL, et donc vers l'objet-relationnel, poussée par le marché, ne plaide guère pour la validité de ce tableau.

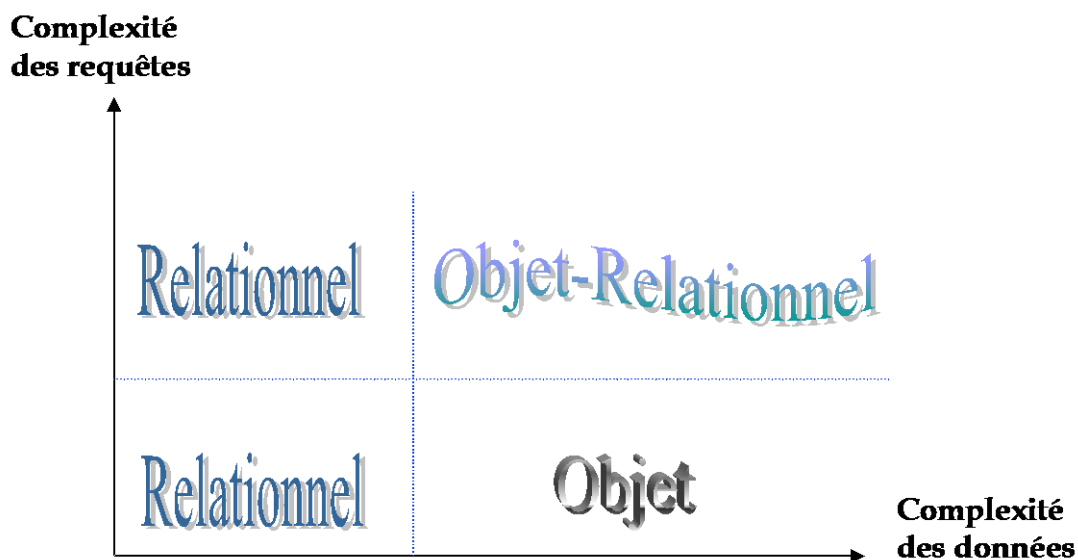


Figure XIII.9 — Relationnel, Objet ou Objet-Relationnel ?

Et après SQL3 ? On parle déjà de SQL4, notamment pour les composants orientés applications ou métiers. Il est en effet possible, voire souhaitable, de standardiser les types de données pour des créneaux d'applications, tels le texte libre, la géographie ou le multimédia. Nous étudierons plus loin ces types de données multimédia qui sont d'actualité. Au-delà, on peut aussi penser développer des types pour la CAO, la finance, l'assurance ou l'exploration pétrolière par exemple. On aboutira ainsi à des parties de schémas réutilisables, complément souhaitable d'objets métiers. Beaucoup de travail reste à faire. Reste à savoir si SQL est le cadre adéquat, et s'il n'est pas déjà trop complexe pour être étendu.

## 8. BIBLIOGRAPHIE

---

[Darwen95] H. Darwen, C-J. Date, « Introducing the Third Manifesto », *Database Programming & Design Journal*, Vol. 1, N°8, pp 25-35, Jan. 1995.

*Cet article plaide pour le modèle objet-relationnel vu comme une extension naturelle du relationnel, tel que proposé par Codd (et Date). Pour C. Date, l'apport essentiel utile du modèle objet est la possibilité de définir des types de données utilisateur. En clair, aucune modification n'est nécessaire au modèle relationnel qui avait déjà le concept de domaine : il suffit d'ouvrir les domaines et de les rendre extensibles.*

[Gardarin89] Gardarin G., Cheiney J.P., Kiernan J., Pastre D., « Managing Complex Objects in an Extensible DBMS », *15<sup>th</sup> Very Large Data Bases International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

*Une présentation détaillée du support d'objets complexes dans le SGBD extensible Sabrina. Ce système est dérivé du SGBD relationnel SABRE et fut l'un des premiers SGBD à supporter des types abstraits comme domaines d'attributs. Il a ensuite évolué vers un SGBD géographique (GéoSabrina) qui fut commercialisé par la société INFOSYS.*

[Gardarin92] Gardarin G., Valduriez P., « ESQL2: An Object-Oriented SQL with F-Logic Semantics », *8<sup>th</sup> Data Engineering International Conf.*, Phoenix, Arizona, Feb. 1992.

*Cet article décrit le langage ESQL2, précurseur de SQL3 compatible avec SQL2, permettant d'interroger à la fois des bases de données à objets et relationnelles. Le langage supporte des relations référençant des objets complexes. Une notation fonctionnelle est utilisée pour les parcours de chemins et les applications de méthodes. Une version plus élaborée de ESQL2 avec classes et relations a été spécifiée dans le projet EDS. La sémantique basée sur la F-Logic (une logique pour objets) illustre les rapports entre les modèles objets et logiques.*

[Godfrey98] M. Godfrey, T. Mayr, P. Seshadri, T. von Eicken, « Secure and Portable Database Extensibility », *Proc. of the 1998 ACM SIGMOD Intl. Conf. On Management of Data*, ACM Pub. SIGMOD Record Vol. 27, N°2, pp. 390-401, Juin 1998.

*Cet article décrit l'implémentation de types abstraits dans le SGBD PREDATOR en Java. Il montre l'intérêt de fonctions sûres et portables, mais souligne aussi les difficultés de performance par comparaison à une implémentation en C++.*

[Haas90] L. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, E. Shekita, « Starburst Mid-Flight : As the Dust Clears », *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, N°1, Mars 1990.

*Cet article décrit Starburst, un des premiers SGBD objet-relationnel. Starburst a été implémenté à IBM Almaden, et fut un des précurseurs de DB2 Universal Server. Il a conduit à de nombreux développements autour de l'objet-relationnel, notamment des techniques d'évaluation de requêtes.*

[Melton98] J. Melton, *Understanding SQL's Stored Procedures*, Morgan Kaufman Pub., 1998.

*Ce livre présente la version 96 de PSM, intégrée à SQL2. Il est très complet, aborde les différents aspects (concepts de base, création de modules et procédures, routines externes, polymorphisme, contrôles, extensions prévues) et donne un exemple d'application.*

[Scholl86] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, A. Verroust, « VERSO: A Database Machine Based On Nested Relations. Nested Relations and Complex Objects », *Intl. Workshop Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, Germany, April 1987, in *Lecture Notes in Computer Science*, Vol. 361, pp. 27-49, Springer Verlag 1989.

*La machine bases de données Verso (Recto étant le calculateur frontal) fut développée à l'INRIA au début des années 1980. Basé sur le modèle relationnel imbriqué, ce projet a développé un prototype original intégrant un filtre matériel spécialisé et la théorie du modèle relationnel NF2.*

[Seshadri97] Seshadri P., Livny M., Ramakrishnan R., « The Case for Enhanced Abstract Data Types », *Proc. of 23<sup>rd</sup> Intl. Conf. On Very Large Databases*, Athens, Greece, pp. 66-75, 1997.

*Cet article plaide pour des types abstraits améliorés exposant la sémantique de leurs méthodes sous forme de règles. Il décrit l'implémentation et les techniques d'optimisation de requêtes dans le système PREDATOR. D'autres implémentation de types abstraits sont évoquées.*

[Siberschatz91] A. Silberschatz, M. Stonebraker, J. Ullman, « Next Generation Database Systems — Achievements and Opportunities », *Comm. of the ACM*, Vol. 34, N°10, pp. 110-120, Oct. 1991.

*Cet article fait le point sur l'apport des systèmes relationnels et souligne les points essentiels que la future génération de SGBD devra résoudre.*

[Stonebraker86] M. Stonebraker, L. Rowe, M. Hirohama, « The Implementation of PostGres », *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, N°1, pp. 125-142, Mars 1990.



*Cet article décrit l'implémentation de PostGres, le système réalisé à Berkeley après Ingres. PostGres fut le premier système objet-relationnel. Il a été plus tard commercialisé sous le nom d'Illustra, qui fut racheté par Informix en 1996.*

[Stonebraker96] Stonebraker M., D. Moore, *Object-Relational DBMSs : The Next Great wave*, Morgan Kaufmann Pub., San Fransisco, CA, 1996.

*Ce livre définit ce qu'est un SGBD objet-relationnel. Les fonctionnalités du SGBD Illustra sont mises en avant pour situer les objectifs essentiels des systèmes objet-relationnels, en particulier du point de vue du modèle de données, du langage de requêtes et de l'optimisation.*

[Zaniolo83] C. Zaniolo, « The Database Language GEM », *Proc. of 1983 ACM SIGMOD Intl. Conf. on Management of Data*, San José, Ca, pp. 207-218, Mai 1983.

*Cet article propose une extension du modèle relationnel et du langage de requête QUEL pour intégrer des tables fortement typées, des généralisations, des références, des expressions de chemins et des attributs multivalués. Il décrit un langage de requêtes précurseur de SQL3.*