Christian Soutou

Avec la collaboration de Frédéric Brouard

Modélisation des bases de données

3e édition

UML et les modèles entité-association

Avec 30 exercices corrigés inspirés de cas réels

EYROLLES

Christian Soutou

Maître de conférences rattaché au département Réseaux et Télécoms de l'IUT de Blagnac, Christian Soutou intervient en licence et master professionnels. Il est aussi consultant indépendant chez Orsys et auteur de nombreux ouvrages aux éditions Eyrolles.

Frédéric Brouard

MVP SQL Server depuis plus de dix ans, Frédéric Brouard dirige la société SQL Spot qui fournit des services d'assistance, de conseil, d'audit et de formation sur SQL Server et l'architecture de données. Enseignant dans différentes écoles d'ingénieur, il a écrit plusieurs livres consacrés à SQL et rédigé de nombreux articles, notamment sur le site developpez.com.

Concevoir une base de données à l'aide d'UML ou d'un formalisme entité-association

S'adressant aux architectes logiciels, chefs de projet, analystes, développeurs, responsables méthode et étudiants en informatique, cet ouvrage explique comment créer un diagramme conceptuel pour concevoir une base de données optimisée via le langage SQL. La démarche est indépendante de tout éditeur de logiciel et aisément transposable, quel que soit l'outil de conception choisi.

Le livre décrit d'abord la construction d'un modèle conceptuel à l'aide de règles de validation et de normalisation. Tous les mécanismes de dérivation d'un modèle conceptuel dans un schéma relationnel sont clairement commentés à l'aide d'exemples concrets. Le modèle logique peut être ensuite optimisé avant l'écriture des scripts SQL. Les règles métier sont implémentées par des contraintes SQL, déclencheurs, ou dans le code des transactions. La dernière étape consiste à définir les vues pour les accès extérieurs. Le livre se clôt par une étude comparative des principaux outils de modélisation sur le marché.

En grande partie réécrite pour prendre en compte les formalismes entité-association tels que Merise ou Barker, cette troisième édition est commentée par Frédéric Brouard, expert SQL Server et auteur de nombreux ouvrages et articles sur le langage SQL. Émaillée d'une centaine de schémas et d'illustrations, elle est complétée par 30 exercices inspirés de cas réels.

À qui s'adresse ce livre ?

- Aux étudiants en IUT, master et écoles d'ingénieur, ainsi qu'à leurs professeurs
- Aux professionnels souhaitant s'initier à la modélisation de bases de données
- À tous les concepteurs de bases de données

Au sommaire

Modélisation des bases de données

3e édition

DU MÊME AUTEUR

C. Soutou. – **SQL pour Oracle (7e édition).**
N°14156, 2015, 672 pages.

C. Soutou, F. Brouard, N. Souquet et D. Barbarin. – **SQL Server 2014.**
N°13592, 2015, 890 pages.

C. Soutou. – **Programmer avec MySQL (3e édition).**
N°13719, 2013, 520 pages.

AUTRES OUVRAGES

R. Bruchez. – **Les bases de données NoSQL et le Big Data (2e édition).**
N°14155, 2015, 332 pages.

P. Roques. – **Mémento UML 2.4 (2e édition).**
N°13268, 2011, 14 pages.

P. Roques. – **Mémento UML 2.4 (2e édition).**
N°13268, 2011, 14 pages.

P. Roques. – **UML 2 par la pratique (7e édition).**
N°12565, 2009, 396 pages.

P. Roques. – **UML 2 par la pratique (6e édition).**
N°13344, 2011, 370 pages (format semi-poche).

P. Roques et F. Vallée. – **UML 2 en action (4e édition).**
N°12104, 2007, 396 pages.

H. Balzert. – **UML 2.**
N°11753, 2006, 88 pages.

F. Vallée. – **UML pour les décideurs**.
N°11621, 2005, 282 pages.

H. Bersini. – **La programmation orientée objet**. *Cours et exercices en UML 2 avec Java 6, C# 4, C++, Python, PHP 5 et LinQ*.
N°12806, 2011, 644 pages.

P. Roques. – **Mémento PHP 5 et SQL (2e édition).**
N°12457, 2009, 14 pages.

Christian Soutou

Avec la collaboration de Frédéric Brouard

Modélisation des bases de données

3e édition

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

À Louise et Émile-Barthélémy, mes grands-parents,
à Élisabeth, ma mère,
pour Paul, mon fils.

Table des matières

Avant-propos

Le but de cet ouvrage est d'expliquer tout d'abord comment construire à bon escient un modèle de données conceptuel de type entité-association (notation de Barker ou Merise) ou diagramme de classes UML pour concevoir une base de données relationnelle. La maîtrise de la traduction de ce modèle conceptuel en un script SQL vous permettra de générer des tables correctement normalisées. La démarche proposée dans ce livre est indépendante de tout éditeur de logiciel et aisément transposable, quel que soit l'outil de conception que vous adopterez.

Les deux premières éditions de cet ouvrage se focalisaient sur le formalisme d'UML. Cette troisième édition élargit le champ du conceptuel en développant les formalismes de type entité-association qui sont toujours très répandus au travers des outils de modélisation comme Data Modeler d'Oracle, MySQL Workbench ou encore Toad Data Modeler. Toujours émaillé de nombreux cas concrets présentés sous forme d'exercices, le texte est par moments commenté par Frédéric Brouard, plus connu sous le pseudo SQLPro, un expert et consultant indépendant qui n'a pas sa langue dans sa poche.

Évolution des modèles de données

Avant d'entrer dans le vif du sujet, rappelons brièvement que les bases de données relationnelles occupent toujours une part prépondérante du marché, bien que la notion de *Big Data* soit omniprésente depuis quelques mois dans les articles et discussions. Notons que la gestion de ces données souvent semi-structurées dépend davantage du monde NoSQL qui ne nécessite pas de méthodologie de conception à proprement parler (enfin, jusqu'à présent, mais l'avenir me donnera peut-être tort).

L'informatique existait déjà avant les bases de données relationnelles et les serveurs NoSQL, et Apollo 11 a bien été envoyé sur la Lune en juillet 1969 avant qu'E. Codd ne publie ses écrits sur le modèle relationnel. À l'époque, la conception des fichiers devait se faire avec bon sens, mais depuis, des méthodes plus formelles ont émergé avec de nombreux livres ou manuels volumineux. Pour toute complexité inutile, le temps fait son œuvre et que reste-t-il à appliquer, si ce n'est le bon sens ? Nous allons vous donner notre version du bon sens dans cet ouvrage : à vous de l'adopter ou sinon de vous en inspirer.

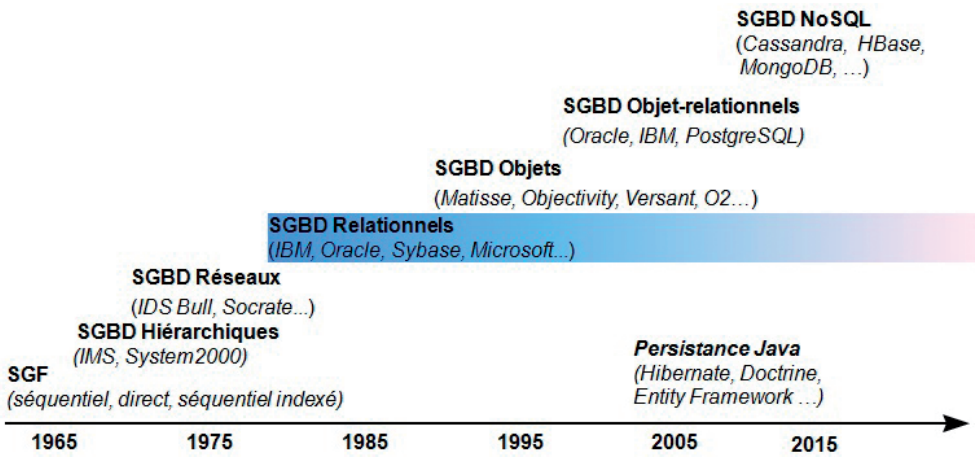

***Figure 0-1**. Historique des bases de données*

Les fichiers et COBOL

Le stockage des données a commencé dans les années 1960 avec les systèmes de gestion de fichiers et le langage COBOL (*Common Business Oriented Language*). Loin d'être dépassé, ce dernier fut le plus utilisé entre 1960 et 1980. En 2002, il permet une programmation de type objet, la gestion des informations Unicode et une intégration avec XML. En 2005, le Gartner Group estimait que COBOL manipulait près de 75 % des données de gestion stockées.

Le principal inconvénient des applications COBOL est la forte dépendance qui existe entre les données stockées et les traitements. En effet, le fait de déclarer dans chaque programme les fichiers utilisés impose une maintenance lourde si la structure d'un fichier doit être modifiée. De plus, les instructions de manipulation (ouverture, lecture, écriture et modification) sont très liées à la structure de chaque fichier. La structure des fichiers de données s'apparente à celle d'une table (suite de champs de types numériques ou alphanumériques).

Le modèle hiérarchique

Les bases de données hiérarchiques ont introduit un modèle de données du même nom. Il s'agit de déterminer une arborescence de données où l'accès à un enregistrement de niveau inférieur n'est pas possible sans passer par le niveau supérieur. Promus par IBM et toujours utilisés dans le domaine bancaire, les SGBD hiérarchiques souffrent toutefois de nombreux inconvénients.

La figure suivante illustre un modèle hiérarchique de données dans lequel des compagnies aériennes peuvent embaucher plusieurs pilotes. Un pilote peut travailler pour le compte de différentes compagnies.

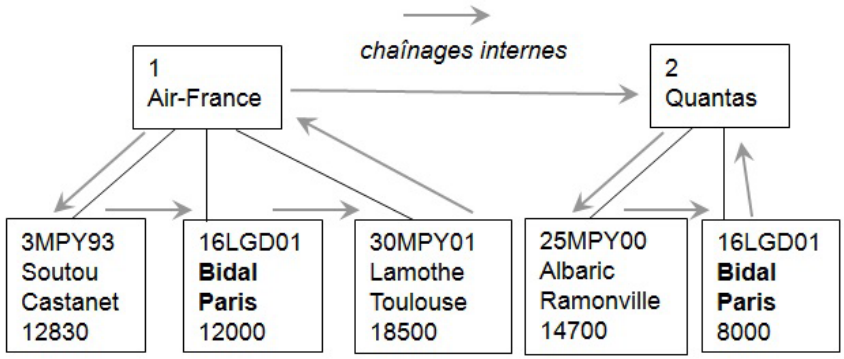

Figure 0-2. *Modèle de données hiérarchique*

Les inconvénients récurrents sont toujours la forte dépendance entre les données stockées et les méthodes d'accès. Les chaînages internes impliquent forcément une programmation complexe. Outre ces problèmes de programmation, ce modèle montre des lacunes lors de l'accès à la base.

- Extraire la liste des pilotes implique le parcours de toutes les compagnies.
- L'insertion peut se révéler problématique : l'ajout d'un pilote sans compagnie n'est pas possible, à moins d'ajouter une compagnie fictive.
- La suppression peut se révéler dangereuse : une compagnie disparaît, alors de fait ses pilotes aussi.
- La modification est souvent problématique : les incohérences proviennent d'éventuelles redondances (le nom ou l'adresse d'un pilote qui change doit se répercuter à tous les enregistrements).

Bien qu'il existe de nombreuses hiérarchies autour de nous, le monde qui nous entoure n'est pas un arbre !

Le modèle réseau

Quelques années plus tard, C. W. Bachman, pionnier dans le domaine de l'informatique, s'est essayé aux bases de données en inventant un modèle brisant cette hiérarchie plutôt arbitraire. Les bases de données réseau étaient nées avec le modèle CODASYL, première norme décidée sans IBM.

Bien que résolvant quelques limitations du modèle hiérarchique et annonçant des performances en lecture honorables, le modèle réseau n'est ni plus ni moins qu'une usine à gaz gavée de pointeurs. Pour preuve, plus personne n'utilise de tels SGBD où la dépendance entre les données stockées et les méthodes d'accès existe toujours, et l'évolution d'une base de données est très coûteuse en termes de recompilation de pointeurs.

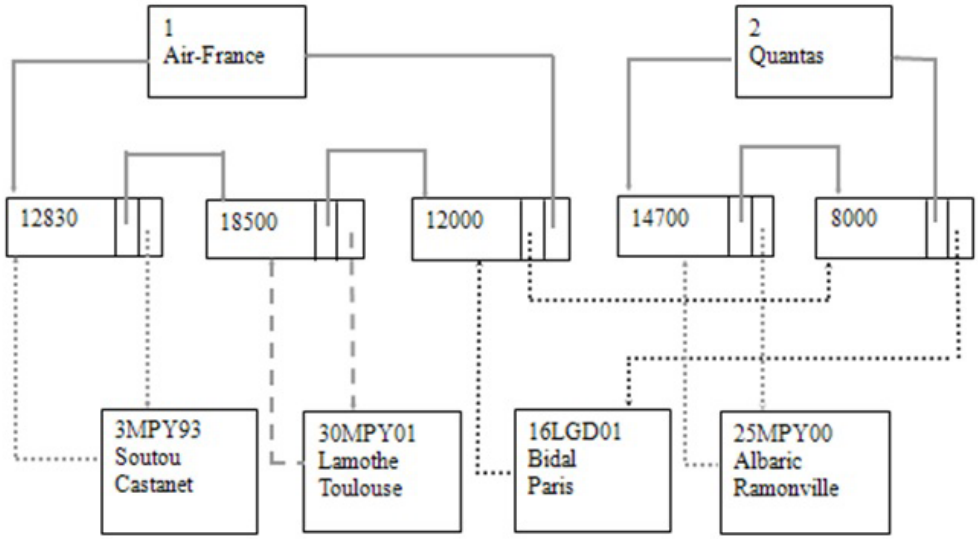

Figure 0-3. *Modèle de données réseau*

Soyons honnêtes, le monde ressemble bien à une telle usine à gaz ! Mais pas question de stocker ainsi les données, ce serait bien trop compliqué de concevoir le bon graphe. Le modèle de données se doit d'être plus simple.

Le modèle relationnel

En 1970, E. Codd publie l'article de référence posant les bases du modèle relationnel [COD 70]. D'un seul coup, toutes les limitations des précédents modèles sont résolues. Le but initial de ce modèle était d'améliorer l'indépendance entre les données et les traitements. Cet aspect des choses est réussi et avec ça d'autres fonctionnalités apparaissent :

- Normalisation (dépendances fonctionnelles) et théorie des ensembles (algèbre relationnelle).
- Cohérence des données (non-redondance et intégrité référentielle).
- Langage SQL (déclaratif et normalisé).
- Accès aux données optimisé (choix du chemin par le SGBD).
- Indexation, etc.

Les liens entre les enregistrements de la base de données sont réalisés non pas à l'aide de pointeurs physiques, mais à l'aide des valeurs des clés étrangères et des clés primaires. Pour cette raison, le modèle relationnel est dit « modèle à valeurs ».

Comment déduire de telles relations entre les tables ? C'est précisément à quoi sert le processus de modélisation que nous allons vous présenter tout au long de cet ouvrage.

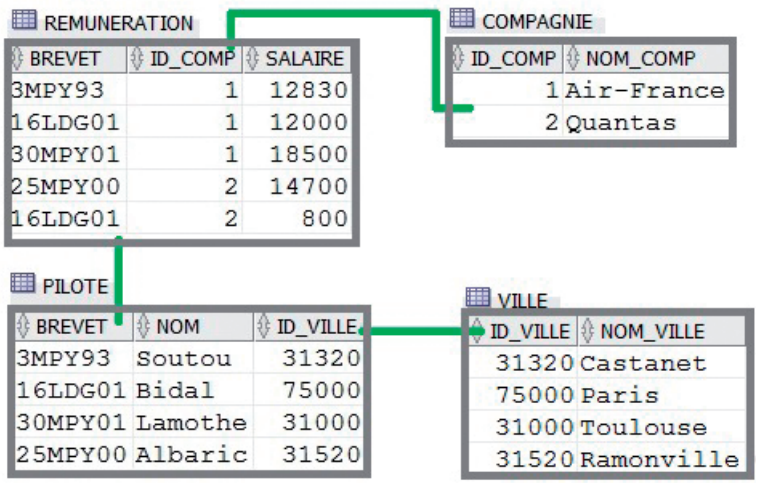

Figure 0-4. *Modèle de données relationnel*

La force de ce modèle de données réside dans le fait qu'il repose sur des principes simples et permet de modéliser des données complexes. Le modèle relationnel est à l'origine du succès que connaissent aujourd'hui les grands éditeurs de SGBD, à savoir Oracle, IBM, Microsoft et Sybase dans différents domaines :

- OLTP *(OnLine Transaction Processing)* où les mises à jour des données sont fréquentes, les accès concurrents et les transactions nécessaires.
- OLAP *(Online Analytical Processing)* où les données sont multidimensionnelles (cubes), les analyses complexes et l'informatique décisionnelle.
- Systèmes d'information géographiques (SIG) où la majorité des données sont exprimées en 2D ou 3D et suivent des variations temporelles.

Les modèles NoSQL

Depuis quelques années, le volume de données à traiter sur le Web a mis à rude épreuve les SGBD relationnels, qui ont été jugés non adaptés à de nombreuses montées en charge. Les grands acteurs du Big Data, comme Google, Amazon, LinkedIn ou Facebook, ont été amenés à créer leurs propres systèmes de stockage et de traitement de l'information (BigTable, Dynamo, Cassandra). Des implémentations d'architectures open source comme Hadoop et des

SGBD comme HBase, Redis, Riak, MongoDB ou encore CouchDB ont permis de démocratiser ce nouveau domaine de l'informatique répartie.

On distingue plusieurs modèles de données parmi les SGBD NoSQL du moment : clé-valeurs, orienté colonnes, documents et graphes. Plus le modèle est complexe, moins le système est apte à évoluer rapidement en raison de la montée en charge.

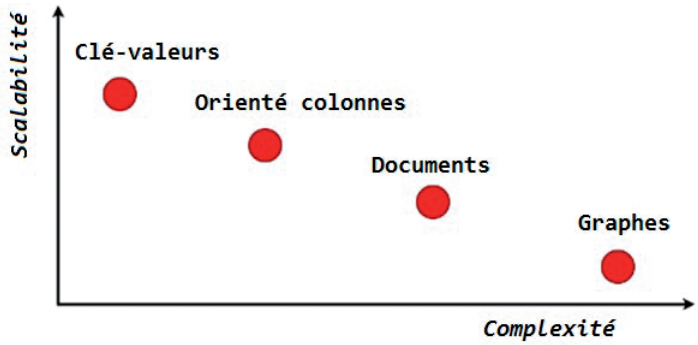

***Figure 0-5**. Modèles de données du NoSQL*

Modèle de données clé-valeurs

Le mode de stockage du modèle clé-valeurs *(key-value)* s'apparente à une table de hachage persistante qui associe une clé à une valeur (de toute nature et de type divers, la clé 1 pouvant référencer un nom, la clé 2 une date, etc.). C'est à l'application cliente de comprendre la structure de ce blob. L'intérêt de ces systèmes est de pouvoir mutualiser cette table sur un ou plusieurs serveurs. Les SGBD les plus connus sont Memcached, CouchBase, Redis et Voldemort (LinkedIn).

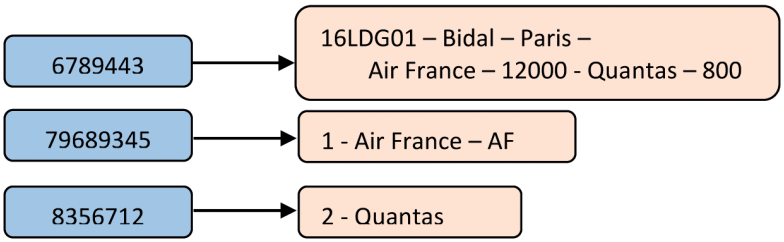

***Figure 0-6**. Modèle de données clé-valeurs*

Modèle de données orienté colonnes

Le modèle orienté colonnes ressemble à une table dénormalisée (sans la présence de NULL, toutefois) dont la structure est dynamique. Les SGBD les plus connus sont HBase (implémentation du BigTable de Google) et Cassandra (projet Apache qui reprend à la fois l'architecture de Dynamo d'Amazon et BigTable).

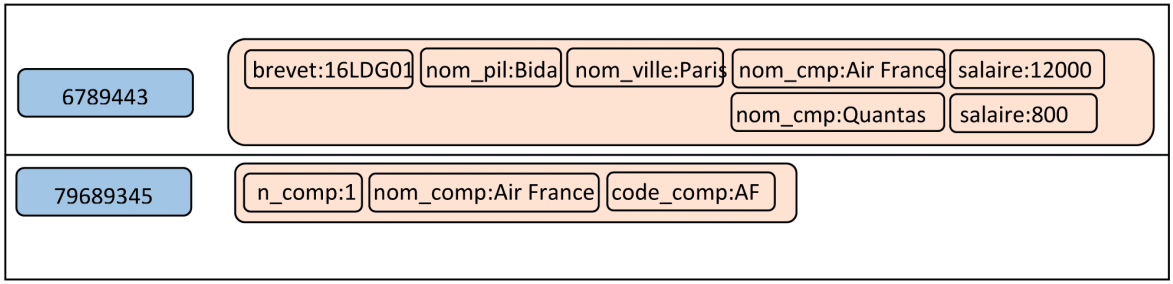

***Figure 0-7**. Modèle de données orienté colonnes*

Modèle de données orienté documents

Le modèle orienté documents est toujours basé sur une association clé-valeur dans laquelle la valeur est un document (JSON généralement, ou XML). Les implémentations les plus populaires sont CouchDB (Apache), RavenDB, Riak et MongoDB.

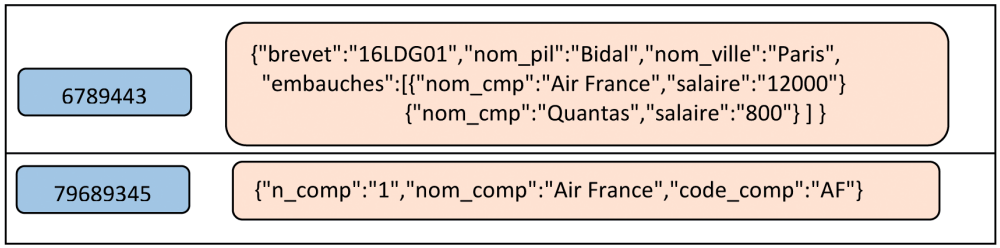

***Figure 0-8**. Modèle de données orienté documents*

Modèle de données orienté graphes

Le modèle de données orienté graphes se base sur les nœuds et les arcs orientés et éventuellement valués. Ce modèle est très bien adapté au traitement des données des réseaux sociaux où on recherche les relations entre individus de proche en proche. Les principales solutions du marché sont Neo4j (Java) et FlockDB (Twitter).

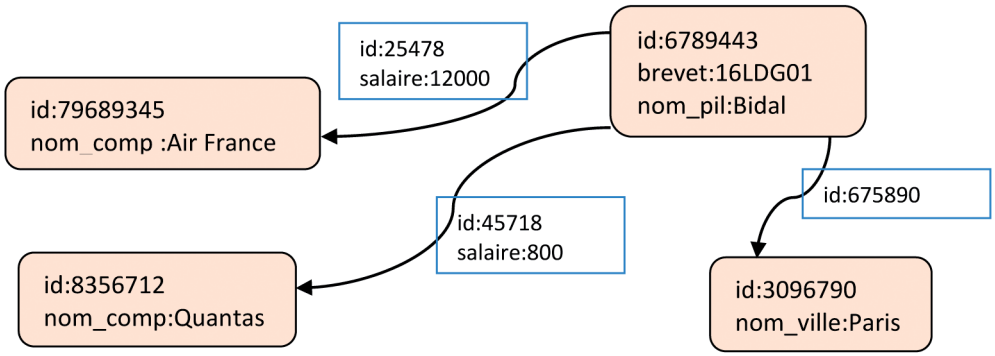

***Figure 0-9**. Modèle de données en graphes*

À qui s'adresse cet ouvrage ?

Cet ouvrage s'adresse à toutes les personnes qui s'intéressent à la modélisation et à la conception des bases de données.

- Les architectes, chefs de projet, analystes, développeurs et responsables méthode retrouveront les principes issus du modèle entité-association et y trouveront les moyens de raisonner avec le diagramme de classes UML.
- Les novices découvriront une méthode de conception, des règles de normalisation et de nombreux exercices mettant en jeu tous les niveaux du processus d'une base de données.

Quel formalisme utiliser pour les bases de données ?

Depuis plus de 30 ans, la conception des bases de données s'appuie sur un formalisme graphique appelé entité-association que la méthode Merise avait adopté en son temps. Ce formalisme a fait ses preuves et bon nombre d'outils de modélisation l'utilisent encore aujourd'hui.

La notation UML s'est imposée depuis quelques années pour la modélisation et le développement d'applications écrites dans un langage objet (C++ et Java principalement). Les entreprises du consortium initial ayant mis en place UML étaient DEC, HP, IBM, Microsoft, Oracle et Unisys pour parler des plus connues. Le marché a suivi cette tendance car, aujourd'hui, beaucoup d'outils de modélisation ont adopté cette notation.

L'adoption généralisée de la notation UML dépasse le simple effet de mode. La majorité des nouveaux projets industriels utilisent cette notation. Tous les cursus universitaires, qu'ils soient

théoriques ou plus techniques, abordent l'étude d'UML. Cela ne signifie pas qu'UML soit la panacée, mais que cette notation est devenue incontournable. La dernière version de la spécification UML, sortie en août 2011, est la 2.4.1 (http://www.omg.org/spec/UML). Ce succès s'explique aussi par l'adoption unanime des concepts objet, qui ont des avantages indéniables (réutilisabilité de composants logiciels, facilité de maintenance, prototypage et extension des applications, etc.).

Les diagrammes d'UML

Les versions 1.*x* de la notation UML définissent neuf diagrammes : cinq pour les aspects statiques (classes, objets, cas d'utilisation, composants et déploiement) et quatre pour les aspects dynamiques (séquence, collaboration, états-transitions, activités). Les spécifications d'UML 2.*x* ajoutent le diagramme d'interaction, le diagramme de structure composite et le diagramme temporel. Seul le diagramme de classes est intéressant à utiliser pour la modélisation d'une base de données.

UML concerne en premier lieu le développement logiciel et n'a pas été initialement pensé pour les bases de données. La notation UML permet toutefois d'offrir un formalisme aux concepteurs d'objets métier et aux concepteurs de bases de données. D'autre part, les concepts relatifs à la modélisation de données (entités, associations, attributs et identifiants) peuvent être parfaitement intégrés aux diagrammes de classes. De plus, d'autres concepts (notamment les classes-associations, agrégats et contraintes) permettent d'enrichir un schéma conceptuel.

Les outils

De nombreux outils informatiques basés sur le concept entité-association ou la notation UML existent depuis quelques années. Les plus sophistiqués permettent de générer des modèles logiques ou des scripts SQL. Alors que l'automatisation est quasiment assurée (sous réserve de la qualité de l'outil) entre le modèle conceptuel et la base de données, il n'en est pas de même de l'élaboration du diagramme initial qui va conditionner toute la suite. Ici l'humain est au centre de tout et il n'est pas question de penser que cette tâche puisse être automatisée (c'est heureux pour les concepteurs).

Par ailleurs, il est fort probable que les scripts SQL générés devront être modifiés manuellement par la suite, soit pour des raisons d'optimisation, soit parce que l'outil ne permet pas de générer une caractéristique particulière du SGBD (index, vues, types de données...), soit tout simplement parce que le concepteur préfère utiliser une autre possibilité d'implémentation pour traduire telle ou telle autre association.

Il est donc préférable de maîtriser les concepts, de comprendre les mécanismes de transformation de modèles et d'adopter une démarche afin d'utiliser l'outil de manière optimale. Cet ouvrage vous permettra, je l'espère, de mieux appréhender le cheminement de la conception vers le codage en donnant des règles précises à suivre dans l'élaboration des différents

modèles pour éviter de graves erreurs au niveau de la base. Le script SQL fera office de véritable révélateur.

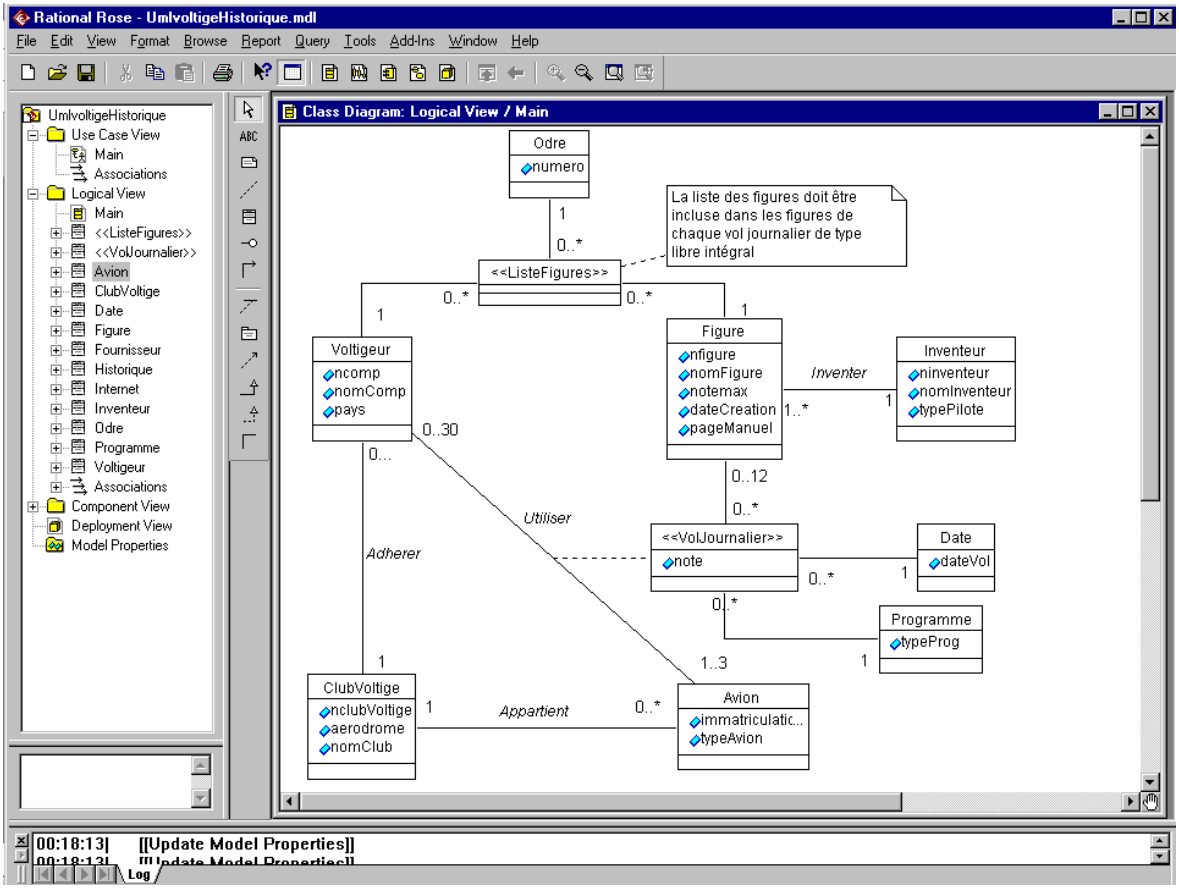

Figure 0-10. *Diagramme de classes*

Guide de lecture

Cet ouvrage s'organise en 5 chapitres qui suivent les étapes de modélisation illustrées dans la figure suivante.

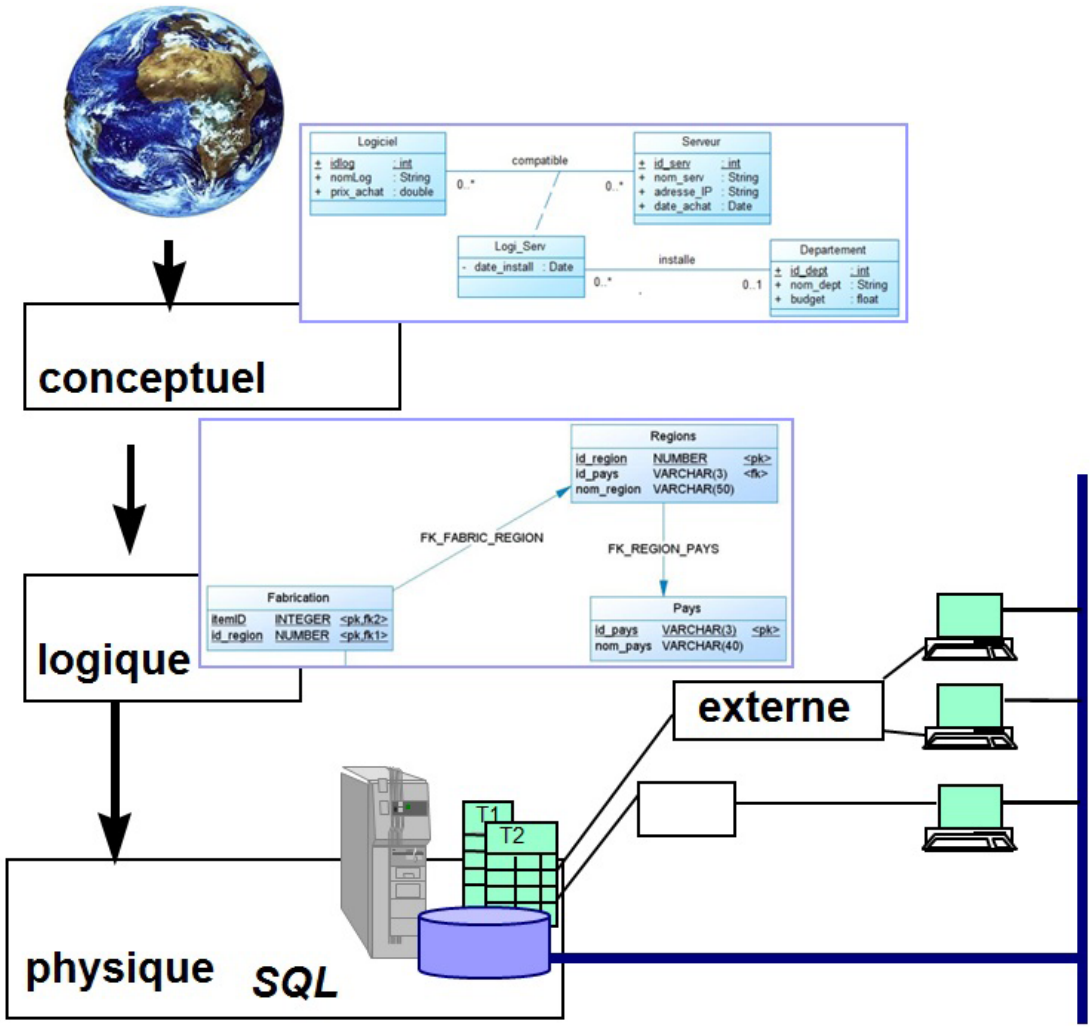

Figure 0-11. *Niveaux de conception et d'implémentation*

L'ouvrage commence par une introduction de Fédéric Brouard avant de décrire la construction d'un modèle conceptuel de type entité-association ou diagramme de classes UML en respectant des règles qui permettent de le valider et de le normaliser. Les mécanismes de dérivation d'un modèle conceptuel dans un schéma de données relationnel sont clairement expliqués à l'aide d'exemples concrets. Le modèle logique est ensuite optimisé avant l'écriture des scripts SQL. Il s'agit ensuite d'implémenter les règles métier en programmant des contraintes ou des déclencheurs SQL. La dernière étape consiste à préparer des vues qui composeront l'interface de la base aux utilisateurs extérieurs.

Niveau conceptuel

Le chapitre 1 décrit la première étape du processus de conception d'une base de données, à savoir la construction d'un schéma conceptuel. Les formalismes graphiques présentés sont les modèles entité-association et le diagramme de classes UML. Les équivalences entre ces formalismes sont clairement détaillées.

Transformation et normalisation

Le chapitre 2 décrit les concepts du modèle relationnel, puis présente les règles qui permettent de dériver un schéma logique à partir d'un modèle conceptuel. La dernière partie traite de la normalisation et du calcul de volumétrie.

Écriture des scripts SQL et programmation des contraintes

Le chapitre 3 décrit la mise en œuvre pour écrire un script SQL dérivé d'un modèle de données relationnel. Le niveau physique présenté dans ce chapitre correspond à la définition des tables, des clés étrangères ainsi que l'implémentation des éventuelles règles métier par des contraintes de clés, de validation ou par déclencheurs.

Les vues

Le niveau externe décrit au chapitre 4 correspond à la définition de vues qui agissent comme des fenêtres sur la base de données. Ce chapitre décrit les différents types de vues existantes (vues relationnelles, matérialisées ou objet).

Les outils du marché

Le chapitre 5 confronte l'offre des principaux outils UML du marché (MagicDraw, MEGA Designer, Modelio, Objecteering, PowerAMC, Rational Rose, Visual Paradigm et Win'Design). Chaque outil est évalué sur différents critères (saisie d'un diagramme de classes, génération d'un modèle relationnel, d'un script SQL et rétroconception d'une base de données).

Annexes

Les annexes contiennent les corrigés détaillés des exercices, une webographie et une bibliographie. L'index propose les termes utilisés dans la définition des concepts et de certaines instructions SQL.

Les pictogrammes

Ce pictogramme introduit une définition, un concept ou une remarque importante.

Ce pictogramme indique une astuce ou un conseil personnel.

Ce pictogramme introduit une remarque, un avis divergent, un complément ou un coup de gueule de Frédéric Brouard.

Contact avec l'auteur

Si vous avez des remarques à formuler sur le contenu de cet ouvrage, n'hésitez pas à m'écrire (christian.soutou@gmail.com). Vous trouverez les éventuels errata sur le site d'accompagnement de cet ouvrage accessible par la fiche du livre sur www.editions-eyrolles.com.

Vous pouvez aussi poster des questions sur vos modèles sur http://merise.developpez.com ou http://uml.developpez.com, de nombreux contributeurs s'y retrouvent.

Avant d'aborder la théorie

Beaucoup de développeurs sont persuadés des méfaits du respect des formes normales. N'est-il pas préférable de placer tous ses « champs » dans une même table pour acquérir de bonnes performances ? Certains internautes l'ont même écrit, mesures de performances à l'appui !

Or il n'en est rien, et le principal problème reste l'incompréhension du modèle relationnel. Bon nombre de développeurs ont succombé au NoSQL, croyant résoudre leurs problèmes alors qu'ils en créaient de pires sans le savoir.

Le modèle relationnel reste encore à ce jour le moyen le plus efficace à tout point de vue, lecture comme écriture, pour la manipulation des données de l'informatique de gestion, les transactions, et les recherches complexes. Encore faut-il en comprendre l'esprit.

C'est donc avec pragmatisme que je vais tenter de l'expliquer sans jamais passer par les « formes normales » ou les « dépendances fonctionnelles ». Seuls des éléments de base de la science mathématique seront utilisés dans cette démonstration.

Les origines

En fait, l'art de la modélisation repose sur quelques règles fondamentales facilement compréhensibles et sur la notion de relation, rarement bien comprise.

Lorsque Frank Edgar Codd invente la théorie de l'algèbre relationnelle dans les années 1970, il prétend résoudre toutes les problématiques des systèmes précédents (bases de données hiérarchiques ou en « réseau », entre autres…) aussi bien sur le plan pratique que sur le plan logique. Force est de constater que sa théorie a fort bien réussi et domine toujours la marché des bases de données de gestion. Il en va tout autrement du traitement des données documentaires dont le Big Data s'empare actuellement, dans l'indécision d'une technologie unique dont les tenants sont regroupés au sein de ce que l'on appelle désormais le NoSQL…

À partir de 1980, les premiers systèmes voient le jour (IBM System R et Oracle de Relationnal Software). Les SGBD relationnels sont donc un succès depuis plus de 35 ans… Et régulièrement, on nous annonce leur fin sans que cela n'arrive jamais. Cependant, la connaissance

du métier de la modélisation se meurt. En effet, depuis que l'objet a vu le jour, on tente de contourner le relationnel sans se rendre compte qu'il faudrait agir avec lui et non contre lui !

Il convient donc de reprendre les choses à la source pour bien les faire comprendre… C'est beaucoup plus simple qu'il n'y paraît, mais de nombreux enseignants préfèrent l'approche théorico-mathématique hyper sophistiquée à l'approche pragmatique, dégoûtant ainsi les étudiants à l'avance…

Cette introduction n'a donc pour but que de vous familiariser avec les concepts afin qu'ils vous apparaissent lumineux.

Au début était la relation

Lorsqu'on interroge les développeurs – qui pour la plupart ont déjà suivi un cours de modélisation – sur ce qu'est une relation, la réponse qui prédomine, et sur laquelle convergent les avis, est la notion de « lien ». Or il n'en est rien.

Une relation a été définie par Codd comme un objet mathématique porteur de données.

Si nous consultons un dictionnaire comme le Larousse, la relation est définie comme suit : « Action de rapporter en détail ce dont on a été le témoin ou dont on a eu connaissance ; récit qu'on en fait. »

En fait, la relation « relate » les faits… Et une base de données c'est de l'information, de la sémantique !

Pour Codd, la définition de la relation est celle-ci :

- un objet mathématique porteur d'informations ;
- contenant un ou plusieurs attribut(s) valué(s) ;
- possédant une clef ;
- doté d'un nom unique au sein de la base de données.

Il précise en outre que les attributs doivent :

- avoir un nom unique au sein de la relation ;
- contenir une information atomique ;
- posséder une valeur prise dans un domaine.

Codd part de la théorie des ensembles et ses relations doivent être considérées comme telles, chacun des éléments de l'ensemble ayant une valeur pour chaque attribut.

Exemple 1 – Une relation

La relation `Remployé` constituée des attributs `matricule`, `nom`, `prénom`, `date de naissance` ayant pour clef le seul attribut `matricule`.

On note habituellement comme ceci :

```
Remployé : matricule, nom, prénom, date de naissance
```

Le ou les attribut(s) clef(s) étant soulignés.

Complétons par la notion de tuple ou n-uplet. Il s'agit d'un ensemble de valeurs pour chacun des attributs de la relation, décrivant un élément particulier de l'ensemble.

Exemple 2 – Un tuple d'une relation

```
XD1247, DUPONT, Frédéric, 21/04/1960
```

L'élément Frédéric DUPONT, né le 21 avril 1960, dont le matricule est XD1247, est un élément de la relation `Remploye` présentée dans l'exemple 1.

Remarque : dans les ensembles, au sens mathématique du terme, il n'y pas d'ordre naturel des choses.

Voici par exemple le contenu d'une relation, représentée par un diagramme de Venn plus communément appelé « patate ».

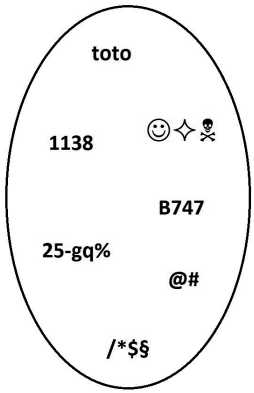

Figure I-1. *Diagramme de Venn*

Remarquez-vous un quelconque ordre des éléments ? Par exemple, une question sans réponse que l'on peut souvent lire dans les forums est la suivante : « Comment récupérer la dernière ligne que j'ai mis dans une table ? » Cette question n'a évidemment aucun sens, d'une part en raison de la concurrence d'accès, deux utilisateurs pouvant insérer une ligne au même moment et d'autre part, il n'y a pas de rangement particulier des lignes dans la table.

Puis vint l'algèbre relationnelle

À partir de la relation, Codd définit des opérateurs relationnels, c'est-à-dire des traitements mathématiques qui transforment les relations en d'autres relations.

Oui, je sais, le terme « algèbre » vous donne des boutons. Pourtant, c'est assez simple. On peut définir une algèbre comme un ensemble d'opérateurs opérant sur des objets mathématiques clairement définis.

Si je vous parle de l'algèbre en nombres entiers, cela vous parle déjà plus... Vous savez que 2 + 2 = 4, soit un nombre entier, tout comme 2 × 3 = 6, encore un entier ! Mais que dire de 7/2 ? Le tout est de savoir si vous voulez rester dans l'univers des entiers ou pas. Si oui, cette dernière opération peut avoir trois résultats différents :

- aucun ;
- 3 ;
- 3 reste 1.

À vous de vous mettre d'accord sur la règle à adopter.

Ce faisant, en restant dans l'univers des entiers, vous opérez ce que l'on appelle une « fermeture », c'est-à-dire que vous vous enfermez dans l'univers des entiers. Ceci possède un intérêt majeur... Celui de pouvoir étendre et combiner les opérations à l'infini !

Eh bien, l'algèbre relationnelle de Monsieur Codd est une fermeture. Chacune des opérations relationnelles donne à nouveau une relation en sortie qui possède une clef et des attributs atomiques tous valués.

Les opérations relationnelles définies par Codd sont les suivantes.

- La **restriction** qui ne conserve que certains tuples de la relation ayant des caractéristiques spécifiques décrites par le biais d'un prédicat (par exemple, un nom commençant par la lettre « D »).
- La **projection** qui ne retient dans la relation résultante que certains attributs et pas d'autres.
- L'**union** qui consiste à « concaténer » des relations aux caractéristiques similaires (par exemple, des clients et des prospects).
- L'**intersection** qui renvoie les éléments communs aux relations.
- La **différence** qui renvoie les éléments de l'une des relations qui n'existent pas dans l'autre.
- Le **produit cartésien** (vulgairement appelé multiplication) qui associe à tout élément d'une relation, chacun des éléments de l'autre (par exemple, une relation contenant les couleurs du jeu de cartes et une autre les figures, permettant ainsi de recomposer un jeu complet).
- La **division** qui est l'opération inverse du produit cartésien.
- Et enfin la **jointure** qui permet d'associer une relation à une autre par le biais d'un prédicat (par exemple le poids d'une lettre et la tarification d'affranchissement).

Vous noterez que certaines opérations n'utilisent qu'une seule relation (restriction, projection). On les appelle opérations unaires ou monadiques. Tandis que les autres portent sur deux relations et sont appelées opérations binaires ou dyadiques.

Et si vous vous demandez ce qu'est un prédicat, sachez que c'est une expression qui peut prendre les valeurs « vrai » ou « faux ». Quelques exemples : « les poissons n'ont pas d'os », « cette phrase compte six mots », ou « ce que vous lisez est irréel ».

Les notions de base

Détaillons maintenant tour à tour quelques-unes des notions précédemment évoquées.

Les noms

Il n'y a aucune ambiguïté sur les noms des relations comme sur les noms des attributs, dans le sens où deux relations au sein de la même base ne peuvent avoir le même nom tout comme deux attributs au sein de la même relation.

Exemple 3 – Noms des éléments

Dans l'exemple 1, le nom de la relation est « *R*employé » et les noms des attributs « nom », « prénom », « date de naissance » et « matricule ».

La notion de valeur

Pour Codd, tout attribut doit être valué. Cela sous-entend qu'un attribut est obligatoirement renseigné. Vous avez sans doute entendu parler du NULL. Cela n'existe pas dans la théorie de Codd. Tous les attributs concourant à une même relation, ou plus exactement à un même tuple (ou n-uplet) doivent posséder une valeur et pas n'importe laquelle… Une valeur vraie ! Pas des mensonges ou de valeurs qui ne veulent rien dire comme une chaîne vide, une date à zéro…

Exemple 4

Les valeurs associées au matricule XD1247 de la relation `Remployé` sont :

```
XD1247, Frédéric, DUPONT, 21/04/1960
```

Comme vous le constatez, chacun des attributs (matricule, nom, prénom et date de naissance) est valué.

La notion de clef

La clef est un moyen d'identifier un élément et un seul au sein de l'ensemble. C'est une information composée de(s) valeur(s) d'un ou plusieurs attributs qui nous permet avec certitude de retrouver un et un seul élément de l'ensemble.

Exemple 5 – Une clef

La clef de la relation `Remployé` présentée à l'exemple 1 est composée d'un seul attribut de nom `matricule`. Pour un certain Frédéric DUPONT né le 21 avril 1960, la valeur de cette clef est `XD1247`.

Si rien n'est précisé au sujet de la clef, alors, par défaut, ce sont tous les attributs de la relation qui composent cette clef.

La notion d'atomicité

Les Grecs étaient persuadés que la matière ne pouvait pas être coupée et recoupée en deux éternellement et qu'il existait une « plus petite partie insécable de la matière ». Ils ont nommé ce concept atome. En informatique, la notion d'atomicité est souvent utilisée. On dit qu'une instruction est atomique quand elle ne peut pas être interrompue. Les transactions dans les bases de données permettent de concevoir des traitements atomiques c'est-à-dire qu'ils fonctionnent alors en tout ou rien, autrement dit tout est exécuté ou rien.

Il en va de même des données.

Une donnée doit être considérée comme atomique si, en la subdivisant, ses différentes parties ne présentent aucune information sensée et nouvelle.

L'atomisation est intrinsèque à la donnée. Ce n'est pas le traitement effectué qui décide si la donnée est atomique ou non. Soit la donnée est, par nature, atomique, soit elle ne l'est pas, auquel cas il faut impérativement la subdiviser.

Exemple 6 – Une date de naissance est-elle atomique ?

Bien qu'il soit possible de subdiviser une date en différentes parties :

```
Jour
Mois
Année
Jour et mois
Jour et année
Mois et année
```

On voit tout d'abord que certaines subdivisions sont absurdes et font perdre le sens, la sémantique de la donnée initiale. Il en va ainsi des subdivisions suivantes :

```
Jour et mois
Jour et année
Jour
Mois
```

Si je vous donne rendez-vous le 13 mars, serez-vous présent ce jour-là ? Sans connaître l'année ?

Seules les subdivisions `Année` et `Mois et année` ont un sens, mais apportent-elles une information que nous ne connaissions pas déjà en ayant la date complète ? À l'évidence non, et en plus nous avons une perte de précision dans le repère temporel...

Bref une date est bien une information atomique.

Exemple 7 – Un numéro de sécurité social est-il atomique ?

Si nous ajoutons à la relation `Remployé` l'attribut `Numéro de sécurité sociale`, est-il bien atomique ?

Par exemple, en extrayant le premier caractère de ce numéro, n'obtenons-nous pas une nouvelle information parfaitement censée, celle du sexe de l'individu qui porte ce numéro ?

À l'évidence nous constatons immédiatement que ce numéro n'est pas atomique, car constitué de différentes parties ayant une sémantique propre, telles que :

```
Sexe
Mois et années de naissance
Commune de naissance (code INSEE)
Rang de naissance
```

Tant est si bien qu'un numéro de sécurité sociale est bien composé de quatre parties, et se doit d'être articulé comme suit :

```
Numéro de sécurité sociale – sexe,
Numéro de sécurité sociale - mois et années de naissance,
Numéro de sécurité sociale – code INSEE commune de naissance,
Numéro de sécurité sociale – rang de naissance.
```

La notion de domaine

Dans la théorie de Codd, les données ne sont pas « typées ». Mais l'ensemble de toutes les valeurs possibles forme un domaine qui limite l'expression des valeurs. Autrement dit, tout attribut possède un domaine, c'est-à-dire l'énumération de toutes les possibilités d'exprimer l'information.

Certains domaines sont triviaux, d'autres beaucoup plus subtils.

Exemple 8 – Domaine pour un pourcentage de réduction

À l'évidence, ce domaine doit se limiter entre 0 et 100. En dessous de 0, la réduction devient une rallonge (une réduction négative augmente !) et au-dessus de 100, non seulement le produit est gratuit, mais le vendeur vous donne de l'argent pour l'emporter ! Dans tous les cas, je ne donne pas cher de votre commerce...

Exemple 9 – Domaine pour une date de naissance d'un employé

En ce qui concerne la date de naissance d'un employé, on peut raisonnablement penser que nos employés n'ont pas plus de 130 ans, ou encore qu'ils ne sont pas nés demain !

La plage de valeurs qui forme ce domaine est alors assez claire à exprimer. Elle va de la date d'aujourd'hui à 130 ans en arrière. En SQL on l'exprimerait ainsi :

```
CHECK(date_naissance BETWEEN DATEADD(year, -130, CURRENT_DATE)
                         AND CURRENT_DATE)
```

Exemple 10 – Domaine pour un nom de personne

Il semble moins évident d'exprimer ce que peut être un domaine pour un nom de personne. Une chaîne de caractères... Oui, mais laquelle ? Un nom de personne peut-il être par exemple `#$?45)=` ? À l'évidence non !

On peut alors édicter la règle suivante : être une chaîne de caractères, composée de lettres en majuscules ou minuscules, accentuées ou non, ou bien être un espace, une apostrophe ou un tiret, mais pour ces trois derniers caractères, jamais en début ni en fin de chaîne, ce que les afficionados des expressions régulières exprimerait sous la forme :

```
[a-zA-Z][a-zA-Z\-\x20']*[a-zA-Z]
```

Les règles absolues

Les quatre règles fondamentales de l'art de la modélisation sont :

- pas de NULL ;
- des données atomiques ;
- pas de redondance ;
- la modification d'une information ne doit pas impacter plus d'une ligne.

Expliquons-les sous différents plans, notamment celui des performances…

Pas de NULL

Le fait d'avoir du NULL dans une base de données est une des plaies majeures et ce qui cause les dégâts les plus importants en termes de performances. En effet, lorsqu'une colonne d'une table contient du NULL, la place nécessaire à la donnée est quand même réservée, car cette information peut un jour prendre une valeur.

Ainsi, le fait d'intégrer des informations comme le téléphone, l'e-mail ou l'adresse dans une table de personnes n'a aucun sens, car elle peut ne pas avoir le téléphone, être SDF ou refuser le modernisme d'Internet ! Vous savez que vous aurez des cas où il n'y aura jamais cette information... Il faut donc externaliser ces informations dans d'autres tables.

Attention, la situation n'est pas la même dans une base relationnelle (OLTP), fréquemment mise à jour au cours de la journée, et dans une base décisionnelle (OLAP), alimentée une seule fois en principe. Dans la première, l'information est dynamique et peut passer subitement du NULL à une valeur à tout moment, et c'est pourquoi on réserve de la place, tandis que dans l'autre c'est un fait, une information accomplie, déjà passée et qui n'a aucune raison d'évoluer, d'où l'absence de stockage des NULL. On parle alors de compression par matrice creuse. C'est pourquoi il existe deux technologies de moteurs différents pour le stockage des données en base, la technologie relationnelle (bases OLTP, *On Line Trancation Processing*) et la technologie décisionnelle (bases OLAP, *On Line Analytical Process*).

Par exemple, dans MS SQL Server, le moteur relationnel sert au stockage des données OLTP et réserve des octets pour les NULL, tandis que SSAS *(SQL Server Analytical Service)* propose d'éviter tout stockage pour une information inexistante dans les bases OLAP. À noter que certaines bases de données évoluées permettent d'éviter de stocker le NULL, même dans les

bases relationnelles, mais il s'agit généralement d'une option que le DBA active après en avoir mesuré toutes les conséquences... Car rien n'est gratuit en informatique. Ce mécanisme existe par exemple dans MS SQL Server sous le nom de *sparse columns*.

Le principe

La règle pour décider si l'on doit mettre ou non telle ou telle information dans telle ou telle table est assez simple. Soit l'information est propre à la relation, comme le nom, le sexe, la date de naissance... et l'attribut doit apparaître dans la relation, soit l'attribut est relatif à la relation, et donc il peut ne pas être valué, et doit donc figurer autre part, comme c'est le cas des moyens de contact (téléphone, e-mail, adresse) ou du nom marital.

Une question intéressante est de savoir quel est la place du NULL dans un tri ? Comme il ne s'agit pas d'une valeur, il paraît difficile de savoir ou le placer puisqu'il ne peut être comparé à rien ! En effet, le tri suppose la comparaison, or comparer le NULL à quelque chose, même un autre NULL n'a aucune chance de produire quoi que ce soit.

Les performances

Disons-le franchement, le NULL, c'est nul en performance ! Commençons par le tri.

Nous savons maintenant que comparer un NULL avec quoi que ce soit n'est pas possible. Or, le tri impose une comparaison. Certains SGBDR placeront les NULL en tête. D'autres en queue. Quelques-uns offrent la possibilité de spécifier où (NULL FIRST, NULL LAST).

Nous voyons bien alors que ce tri est artificiel et nécessite un traitement particulier qui n'est pas celui des données valorisées.

Par conséquent, **trier des données NULLables est beaucoup moins efficace** que de ne trier que des données valorisées et rajoute ce que les américains appellent un *extra overhead*...

Recherche de données absentes

Pour accélérer l'accès aux données, il est pratique d'indexer ses tables. Un index n'est qu'une structure de données particulière dont l'organisation facilite certaines recherches. Un index dans une table est similaire à celui que vous trouvez en fin de pages dans un ouvrage technique : les mots-clefs sont listés par ordre alphabétique et chacun renvoie à une ou plusieurs pages du livre dans laquelle ce mot est présent. Mais dans un tel index, où faut-il placer les « non-mots » ? Parce qu'un NULL c'est l'absence de valeur, autrement dit un mot qui n'existe pas ! On voit tout de suite que cela n'est pas possible. Cela confirme d'ailleurs ce que nous disons précédemment : impossible de trier une non-valeur par rapport à une valeur.

Certains SGBDR n'indexent pas du tout les NULL. D'autres se forcent à les placer en début ou en fin d'index. Mais force est de constater, à nouveau qu'**un extra overhead sera nécessaire pour la recherche du NULL**, index ou pas ! **L'efficacité sera moindre ou NULL** selon le SGBDR utilisé...

Enfin, nous allons voir que même indexé par certains moteurs, il est probable que cela ne serve à rien dans bien des cas !

Les requêtes

En ce qui concerne la manipulation logique des données, la présence potentielle du NULL oblige à écrire des requêtes plus complexes, en utilisant notamment des opérateurs spécifiques comme IS NULL, COALESCE ou CASE.

Par exemple, pour écrire une requête qui renvoie le complément d'une autre requête basé sur un critère potentiellement NULLable, la complexité augmente et les performances chutent dramatiquement.

Exemple 11 – Recherche et complément

Si dans une table des employés, la colonne date de naissance a été conçue comme NULLable, la requête pour trouver les employés nés au 20e siècle est la suivante :

```
SELECT *
FROM   T_EMPLOYE
WHERE  DATE_NAISSANCE < '2001-01-01'
```

Mais quelle est la requête pour trouver le reste ?

```
SELECT *
FROM   T_EMPLOYE
WHERE  DATE_NAISSANCE < '2001-01-01' OR
       DATE_NAISSANCE IS NULL;
```

Nous voyons immédiatement que la requête est plus complexe et qu'elle utilise un opérateur logique OR réputé non « cherchable », c'est-à-dire incapable d'utiliser l'index sur la colonne DATE_NAISSANCE s'il y en avait un !

Comment interdire tout NULL ?

Est-il possible d'interdire totalement la présence du NULL dans une base de données ? La réponse est évidemment oui !

Reprenons notre exemple de relation `Remploye` et admettons que chacun des attributs en dehors de la clef peut potentiellement ne pas avoir de valeur. Imposons-nous de ne pas avoir de NULL dans la base... Comment procéder alors ? Bien entendu, il est interdit de mentir et d'introduire de fausse valeur comme la chaîne vide ou la date « zéro ».

La solution est beaucoup plus simple qu'il n'y paraît... Il suffit de décomposer la relation en trois nouvelles relations, chacune n'étant dotée, en dehors de la clef, que d'un seul attribut.

Exemple 12 – Des employés « NULLables » sans stockage de NULL

La relation `Remploye` est décomposée avec les trois relations suivantes :

```
RempNom : matricule, nom
RempPrénom : matricule, prénom
RempDateNaissance : matricule, date de naissance
```

Évidemment cela oblige à de nombreuses jointures. Mais est-ce un mal ? D'après certains développeurs, les jointures ne sont pas performantes. La légende est tenace ! Il y a bien longtemps que les SGBDR pratiquent la jointure et c'est l'une des opérations les plus courantes et les plus optimisées... Mais les développeurs rechignent à écrire des requêtes avec des jointures, alors aidons-les avec des vues qui préparent ces jointures... Par exemple comme celle-ci :

```
CREATE VIEW V_EMPLOYE AS
SELECT COALESCE(n.matricule, p.matricule, d.matricule) AS matricule,
       n.nom, p.prenom, d.date_naissance
FROM   T_EMPLOYE_NOM AS n
       FULL OUTER JOIN T_EMPLOYE_PRENOM AS p
            ON n.matricule = p.matricule
       FULL OUTER JOIN T_EMPLOYE_DATE_NAISSANCE AS d
            ON p.matricule = d.matricule;
```

Dans ce dernier exemple, cette vue ajoute des NULL dans la table réponse. Moralité, chassez le NULL, il revient au galop, notamment par le biais de la jointure externe !

Des relations mono-attribut ?

Comme nous l'avons vu à l'exemple 12, la solution parfaite de modélisation passe par des relations n'ayant au mieux qu'un seul attribut en dehors de la clef. Mais cette solution est-elle viable ? Étudions-la sous le principal aspect des performances et sous l'angle des lectures, comme des écritures...

Il semble a priori que le volume des données d'une telle conception soit plus important. Considérons que les données occupent les octets suivants :

- 6 pour le matricule ;
- 30 pour le nom ;
- 24 pour le prénom ;
- 4 pour la date de naissance.

Alors si aucune des données n'est réellement NULL, la volumétrie des données pour 1 024 occurrences est comparativement la suivante :

- 64 Ko pour la version monorelation : (6 + 30 + 24 + 4) × 1 024
- 76 Ko pour la version à 3 relations : (6 + 30 + 6 + 24 + 6 + 4) × 1 024

Il semble a priori que la table monorelation sorte vainqueur.

Avec un taux de NULL de 20 %, les choses changent :

- 64 Ko pour la version monorelation : (6 + 30 + 24 + 4) × 1 024
- 60,8 Ko pour la version à 3 relations : 0,8 × (6 + 30 + 6 + 24 + 6 + 4) × 1 024

Or, plus une base de données est grande, plus elle est lente en raison du volume !

Mais tout ceci peut être remis en cause par la création d'index... Pour être efficace dans 100 % des recherches, quelle seraient les index à créer dans les deux bases ?

Pour la version monorelation, les index à créer sont les suivants :

- X1 : nom
- X2 : prénom
- X3 : date de naissance

Mais il est courant d'effectuer des recherches combinées. Il nous faut alors aussi les index suivants :

- X4 : nom, prénom
- X5 : nom, date de naissance
- X6 : prénom, date de naissance
- X7 : nom, prénom, date de naissance

Hélas, cela ne suffit pas, car si l'index X4 permet de rechercher un « DUPONT » dont le prénom commence par « M » le tri qu'il induit s'avère inefficace dans le cas d'une recherche portant sur un « Marcel » dont le nom commence par « P ». Il convient donc de rajouter les index suivants :

- X8 : prénom, nom
- X9 : date de naissance, nom
- X10 : date de naissance, prénom
- X11 : nom, date de naissance, prénom
- X12 : prénom, nom, date de naissance
- X13 : prénom, date de naissance, nom
- X14 : date de naissance, prénom, nom
- X15 : date de naissance, nom, prénom

Et le volume global de notre base de données explose : 792 Ko !

Pour la version multirelation, trois index seulement sont nécessaires :

- X1 : nom
- X2 : prénom
- X3 : date de naissance

Soit un volume global de 152 Ko...

Battu à plate couture ! Notre optimisation est multipliée par 5... Et plus vous aurez d'attributs au sein d'une même relation et plus l'écart va augmenter de façon exponentielle, car la combinatoire des index potentiels est une factorielle !

Donc, **pour les lectures, la version multirelation est beaucoup plus intéressante**... Mais qu'en est-il pour les écritures ? Elle l'est tout autant, voire davantage !

La plupart des écritures nécessitent un index pour trouver les lignes à modifier ou supprimer. Mais aussi à insérer !

Or, chaque index ajoute du temps de traitement à chaque mise à jour. Avoir beaucoup d'index pèse donc sur les performances, car les insertions sont plus lentes dans une unique relation que dans un ensemble de relations. En outre, si votre SGDBR est haut de gamme (Oracle, SQL Server), il fera nativement du parallélisme pour répartir les insertions, ce qui bloquera la table moins longtemps.

Car là aussi l'intérêt de ventiler en plusieurs relations est qu'en cas de modification sur un attribut les autres restent lisibles, ce qui n'est pas le cas de la monorelation où la ligne est globalement bloquée à la moindre mise à jour !

Pour les écritures, la recherche des lignes à mettre à jour est au moins aussi efficace, mais **le blocage est moindre du fait que les attributs non impactés restent disponibles** pour la lecture comme pour l'écriture !

Bref, cette stratégie est gagnante sur tous les tableaux. Certains SGBDR utilisent cette technique sans vous le dire. C'est le cas par exemple de Sybase IQ. La plupart des bons SGBDR comme Oracle ou SQL Server, s'oriente vers cette façon de structurer sans vous le dire, à travers des concepts particulier tels les index « ColumnStore » disponible dans MS SQL Server depuis la version 2012...

Le compromis... NULL induit et NULL fortuit

Il convient quand même d'adopter un certain compromis. Dans notre exemple de relation éclaté en trois pour l'employé, il ne serait pas possible de faire en sorte que ces informations servent de référence à d'autres tables, tout simplement parce que l'on ne saurait pas dans quelle table chercher la clef... Une vue ne pouvant être une référence. Il faudrait donc ajouter une relation maîtresse contenant tous les matricules et lier nos trois autres à cette nouvelle. Cela devient un peu lourd !

Quelle serait alors une règle de bon compromis ? Celle de **tolérer le NULL à condition que l'on soit toujours en situation, un jour ou l'autre, d'être valué !**

Pour le savoir, il suffit de se poser la question du NULL « induit » par rapport au NULL « fortuit ».

- Un NULL sera **induit** s'il existe au moins un cas pour lequel on ne pourra jamais obtenir la valeur.
- Un NULL sera **fortuit**, si l'on peut toujours recueillir la valeur, mais que cette valeur n'est pas connue à l'instant *t*.

On en revient donc à l'essentiel et au point de départ de cette discussion (voir la section « Le principe ») : l'attribut est-il propre ou relatif à la relation ?

Exemple 13 – Attribut propre « légalement » NULLable

Rajoutons à notre relation `Remployé` l'attribut `date de décès`... Il apparaît clairement que cette dernière information ne peut pas être connue lors de la création de l'employé et heureusement ! D'où le fait d'être NULL malgré un attribut propre.

À noter que certains logiciens auraient voulu avoir d'autres marqueurs que le NULL pour tout un tas d'autres raisons. D'une part, pour connaître la raison du NULL : inapplicable, inconnu, non saisi, soit des sous-NULL ! D'autre part, pour marquer certaines valeurs particulières : infini positif, infini négatif, futur, passé... Déjà que la logique du NULL est complexe, mais alors avec autant de marqueurs, que signifierait la formule : NOT (INCONNU OR INAPPLICABLE) AND PASSÉ ?

Au niveau du modèle

Des techniques existent pour éradiquer facilement le NULL. Parmi celles-ci, ma préférée est la notion d'héritage des données. Le plus simple pour comprendre est de procéder par un exemple...

Exemple 14 – Modélisation de véhicules

Voici une relation contenant des informations sur des véhicules à moteur.

```
Rvehicule : id, immatriculation, puissance, carburant, tirant d'eau,
altitude de croisière
```

À l'évidence cette relation mélange différents types de véhicules. Des bateaux (le tirant d'eau ne s'applique qu'aux navires) et des avions (l'altitude de croisière est spécifique aux aéronefs). Lorsque nous aurons affaire à un bateau, l'altitude de croisière sera vide éternellement et le tirant d'eau renseigné. Ce sera l'inverse pour un avion. Ce modèle viole donc la règle de l'interdiction du NULL. Pour éviter cela, il faut procéder par un « héritage des données » que l'on appelle aussi spécialisation. Les attributs communs restent dans la relation générique `Rvehicule`, tandis que les attributs spécialisés rejoignent des relations spécifiques, ce qui conduit au modèle suivant :

```
Rvehicule : id, immatriculation, puissance, carburant
Rvehicule_bateau : id, tirant d'eau
Rvehicule_avion : id, altitude de croisière
```

Bien entendu, nous pouvons spécialiser encore plus. Par exemple, dans la famille des avions, nous pouvons distinguer les avions militaires et l'aviation civile, ou classer les bateaux par type, voiliers ou à moteur, etc.

L'avantage important de cette technique est qu'elle permet de « sauter » certains liens. Nous allons étudier un autre exemple d'héritage qui résout bien des problèmes. Avant cela une question... Que faites-vous si vous avez comme client des personnes physiques (moi, vous, Alain Dupont...) mais aussi des personnes morales (la mairie de Neuilly, IBM, Peugeot...) et que vous devez collecter tout un tas d'informations, notamment les moyens de contact comme les téléphones, les e-mails, les adresses ?

Exemple 15 – Héritage de personnes

Soit la relation `Rpersonne`, contenant un seul attribut `prs_id` qui en est la clef.

```
Rpersonne : prs_id
```

À partir de cette relation, nous pouvons affiner en personnes morales et physiques.

```
Rprs_morale : prs_id, raison sociale, enseigne, nature
Rprs_physique : prs_id, nom, prénom, date naissance
```

À partir de la personne physique, nous pouvons aussi spécialiser en différents éléments, par exemple les employés :

```
Rprs_employé : prs_id, matricule, service, salaire...
```

Notez que l'attribut matricule n'est plus la « clef » de cette relation, mais *une clef* « subrogée » ou « alternative ».

Puis, s'il faut encore descendre, par exemple si nous sommes dans une compagnie d'aviation :

```
Rprs_volant : prs_id, stage sécurité...
Rprs_rampant : prs_id, affectation...
```

Voire, encore plus bas :

```
Rprs_pilote : prs_id, qualification, commandant...
Rprs_pnc : prs_id, chef, responsable sécurité...
```

Et l'on voit tout de suite que trouver un commandant qualifié sur 747 est une requête relativement simple, qui évite de passer, traiter, ramener une quantité importante de colonnes inintéressantes pesant sur les performances…

```
SELECT nom, prénom
FROM   Rprs_physique AS pp
       JOIN Rprs_pilote AS pi
            ON pp.prs_id = pi.prs_id
WHERE  qualification = '747'
  AND  commandant = true
```

Des informations atomiques

Là aussi le non-respect de cette règle entraîne des catastrophes en termes de performances...

Les références multisémantiques…

Il est beaucoup plus facile de comprendre pourquoi la non-atomisation de l'information entraîne de mauvaises performances. Il suffit juste de comparer une requête effectuée sur la table dont les données ont été atomisées à la version non atomique... Il est d'ailleurs amusant de constater que parfois le développeur atomise ou n'atomise pas sans s'en rendre compte. Voici un TP de modélisation basique que je propose à mes élèves :

« Modélisez les patients d'un hôpital avec les éléments nécessaires à la prise en charge financière par les différentes couvertures sociales (sécurité sociale et mutuelles) et le paiement des forfaits hospitaliers de journées ou du ticket modérateur par prélèvement sur compte en banque. »

On constate généralement que les développeurs pensent à découper les références d'un compte bancaire en trois parties (code banque, code guichet et numéro de compte), mais pas à atomiser le numéro de sécurité sociale !

Bien évidemment, la seconde phase de ce TP consiste à écrire des requêtes dans ce modèle. Une en particulier m'amuse beaucoup :

« Pour les besoins d'une étude médicale d'un doctorant sur le cancer de la prostate concernant les gens nés à Paris dans les années 1960, retrouvez les patients cible... »

La requête SQL ressemble généralement à l'exemple suivant.

Exemple 16 – Requête sur numéro de sécurité sociale non atomisé

```
SELECT *
FROM   T_PATIENT
WHERE  NUMERO_SECU LIKE '1%'
  AND  SUBSTRING(NUMERO_SECU, 2, 2) BETWEEN '60' AND '69'
  AND  SUBSTRING(NUMERO_SECU, 6, 2) = '75'
```

Or cette requête ne sera jamais performante, car une fonction comme `SUBSTRING` appliquée à une colonne interdit l'utilisation d'un index !

Correctement atomisée en quatre parties, sexe, année/mois de naissance, numéro INSEE de commune et rang, la requête est plus simple et peut utiliser des index.

Exemple 17 – Requête sur numéro de sécurité sociale atomisé

```
SELECT *
FROM   T_PATIENT
WHERE  NUM_SECU_SEXE = '1'
  AND  NUM_SECU_ANMOIS BETWEEN '6001' AND '6912'
  AND  NUM_SECU_COMMUNE LIKE '75%'
```

Avec le bon index, cette dernière requête devient quasiment instantanée quel que soit le volume à traiter...

Le cas des e-mails

Un autre exemple plus croustillant nous est donné par les e-mails. La plupart des développeurs tombent dans le panneau, ne voyant pas que, de par sa structure, un e-mail est en réalité composé de deux parties : le nom d'utilisateur et le nom du serveur de courrier aussi appelé DNS. Il convient donc de séparer en deux, sans conserver l'arrobe (@, ou « a » commercial).

Ceci permet d'économiser de nombreux octets vu le nombre d'afficionados de Gmail, Orange, Hotmail, Yahoo!, AOL... et permet de rechercher sur des chaînes de caractères moins longues pour la partie utilisateur, ce qui au final économise de l'espace, fait gagner du temps de traitement et diminue la durée des verrous.

Le cas des adresses IP

A contrario, un exemple inverse est constitué par l'adresse IP qui n'est en fait qu'un entier (une suite de 32 bits), mais dont la présentation semble indiquer que cette information est coupée en quatre. En réalité, il s'agit de 4 octets présents sous forme d'entiers décimaux allant de 0 à 255. Cette présentation communément admise n'est là que pour faciliter le calcul mental (notamment pour les masques de sous réseau et les classes d'adresses...), mais complique les formules mathématiques de traitement... Il ne faut donc pas confondre la donnée brute et la présentation que l'on en a. L'un sera stocké dans la table, l'autre sera disponible à travers une vue.

Une adresse postale

Le cas de l'adresse postale est intéressant, parce qu'il est régi par plusieurs problématiques, dont l'atomisation naturelle bien sûr, mais aussi le formalisme induit par des éléments extérieurs…

D'innombrables sites web et beaucoup de logiciels de boutiques en ligne comme Prestashop ou Joomla ont très mal modélisé les adresses des clients, sans doute en raison de leur origine américaine, à mille lieux des préoccupations des standards européens…

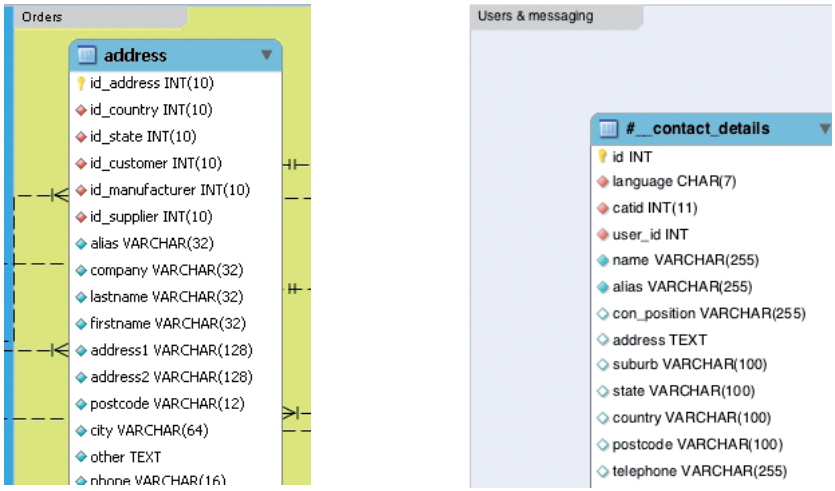

Figure I-2. *Adresses modélisées de façon inepte dans les logiciels Prestashop à gauche et Joomla à droite*

En effet, en Europe, il existe une norme de l'adresse postale, destinée à faciliter les services de transports, mais aussi en réduire les coûts, notamment pour l'envoi en masse. Ne pas s'y conformer, c'est s'interdire les rabais proposés notamment par les services postaux, mais aussi avoir moins de chances que l'envoi aboutisse.

Cette norme propose que l'adresse soit structurée en un maximum de 6 lignes de 38 caractères avec une 7e ligne optionnelle informant du pays de destination.

- La première ligne correspond au destinataire (personne morale ou physique).
- La dernière ligne est constituée du code postal et de la ville.
- L'avant-dernière doit comporter la voie et l'emplacement dans la voie (n° en général).

Entre ces éléments, différentes possibilités sont envisageables. Par exemple, si le destinataire est une personne morale, alors la 2e ligne est la personne physique ou le service auquel l'envoi est destiné. Cela peut également être des indications complémentaires de destination « géographique », par exemple le nom d'une zone industrielle (Z. I. de Fos-sur-Mer), d'un complexe de bureaux (La Défense), d'un lieu-dit…

Pour valider électroniquement une adresse, il faut transmettre les informations suivantes à un service web : code postal, ville, voie, n° dans la voie.

Celui-ci répondra en indiquant s'il connaît l'adresse complète ou partielle et dans le cas du partiel à quel degré l'information aura été trouvée :

- très probable : ville + code postal + voie + n° existant dans une plage connue ;
- probable : ville + code postal + voie ;
- peu probable : ville + code postal ou bien ville + voie ou encore code postal + voie ;
- improbable : ville ou code postal.

Exemple 18 – Modélisation normalisée d'une adresse

Du fait de la structure d'une adresse, les informations qu'elle contient doivent être modélisées comme suit (en dehors du destinataire) :

```
R_adresse_entête : adr_id, code postal, ville, voie, n° début, n° fin,
n° complément, pays
R_adresse_ligne : adr_id, position, ligne (38 caractères au maximum)
```

Il faut veiller à ce que les éléments suivants ne dépassent pas chacun 38 caractères :

- code postal + ville ;
- n° début + n° fin + n° complément voie ;
- pays.

Cette modélisation permet de stocker des adresses comme :

- chemin du Riou (pas de numéro dans la voie)
- 15-17, place d'Aligre (plage de numéro)
- 19 ter avenue du Dr. Arnold Netter (numéro avec complément)

Le problème de l'historisation

Enfin, lorsque la valeur d'un attribut évolue au cours du temps, deux techniques peuvent être utilisées : un écrasement du dernier relevé de valeur ou bien la mise en place d'un historique. Dans ce dernier cas, plusieurs solutions s'offrent alors à nous. Voyons ceci avec un exemple concret...

Exemple 19 – Attributs et historisation des données

Soit à ajouter les attributs poids et taille à la relation employé, données qui seront relevées périodiquement. Quatre solutions sont possibles :

Solution 1, ajout de 3 attributs à la relation actuelle :

```
Date mesure, poids, taille
```

Solution 2, ajout de 4 attributs à la relation actuelle :

```
Date mesure poids, date mesure taille, mesure poids, mesure taille
```

Solution 3, ajout d'un seul attribut à la relation actuelle :

```
Mesure (tableau à deux dimensions "date" et "nature" avec la mesure)
```

Solution 4, deux nouvelles relations :

```
R_histo_taille : matricule, date, taille
R_histo_poids : matricule, date, poids
```

À l'évidence, la solution 3 n'est pas atomique, car, par nature, un tableau ne l'est pas. Quant à la solution 1, elle oblige à ce que les deux mesures soient relevées en même temps. Mais dans les trois premières solutions, si on décide de ne plus relever ces informations pour les nouvelles embauches, alors nous aurons en permanence du NULL qui pèsera sur les performances.

La seule solution viable est donc la quatrième.

Malgré des attributs propres, il arrive que l'on doive externaliser certaines données, parce que l'on veut suivre leur évolution. Si par exemple le poids ou la taille est bien un attribut propre de la relation « individu », suivre son évolution au cours du temps nécessite une information supplémentaire de datation. On aura donc différentes valeurs exprimées à différents moments. Pour peu que ces deux informations ne soient pas relevées en même temps, il faudra deux relations supplémentaires... Sans le savoir, vous venez de mettre le doigt sur la 6^{e} forme normale !

Pas de redondance

Redonder les données constitue une « dénormalisation » de la base et a une conséquence dramatique évidente pour les performances : celle de nécessiter deux fois plus de travail, donc deux fois plus de ressources, de temps… Beaucoup de développeurs pensent qu'avoir une

même information présente plusieurs fois dans le Système d'information est un avantage. Hélas, cela présente trois conséquences néfastes :

- un accroissement du volume des données, toujours néfaste aux performances ;
- une indécision quant à la bonne valeur à prendre ;
- la possibilité de divergence des valeurs dupliquées.

Il arrive bien entendu que l'on soit tenté par la duplication des données afin de résoudre un problème avéré de performances. Cela peut être le cas lorsque deux informations à rapprocher fréquemment dans certaines requêtes se trouvent très éloignées du fait du modèle relationnel.

Mais dans ce cas, il faut prouver que le coût de cette dénormalisation est globalement positif en mesurant que l'économie réalisée par la redondance est très largement supérieure au coût de la double mise à jour.

En outre, il faut s'assurer que les différentes duplications ne changeront jamais de valeurs. Cela ne peut être réalisé qu'avec des techniques propres au SGBDR utilisé et surtout pas manuellement. Dans l'ordre du mieux au moins bien, ce sera :

- la création de colonnes calculées ;
- la mise en place de vue matérialisées ;
- l'utilisation de déclencheurs.

En tout état de cause, il faut donc commencer par une base de données fortement normalisée, mesurer les temps de réponse globalement, procéder au maquettage de la dénormalisation, relancer toutes les requêtes du benchmark initial et comparer…

Notons que la technique de l'héritage présentée notamment par les exemples 14 et 15 permet d'éviter de nombreuses redondances. Par exemple, si vous avez envisagé une relation contenant des patients et une autre comportant des médecins, alors il arrivera un jour où un médecin sera un patient ou vice versa et fera apparaître la redondance. La solution consiste à modéliser une relation de niveau supérieur (une généralisation) avec la notion de personne physique…

La modification d'une information ne doit pas impacter plus d'une ligne

Cette règle, en apparence des plus simples, est une des plus complexes à mettre en œuvre…

Sur le plan des performances, il est évident que mettre à jour plusieurs informations, là où je pourrais me contenter de n'en actualiser qu'une seule, coûte considérablement plus cher et donc, diminue de façon drastique les performances…

Exemple 20 – Relation contenant des personnes physiques et « civilité »

Voici une relation modélisant les personnes physiques avec un attribut `civilité` dont le domaine est {Monsieur, Madame, Mademoiselle, Maître, Monseigneur, Altesse…} :

```
Rpersonne_physique : prs_id, nom, prénom, date naissance, civilité
```

Imaginons que l'administration décide d'interdire le terme mademoiselle dans les civilités. Combien de lignes devons-nous modifier dans la base ?

Pour éviter cette modification nous aurions dû externaliser cette information en modélisant comme suit :

```
Rcivilité : cvt_id, cvt_libellé
Rpersonne_physique : prs_id, nom, prénom, date naissance, cvt_id
```

Il ne restait qu'une seule information à changer : le libellé « Mademoiselle » de la relation `Rcivilité` en « Madame ».

Bien évidemment, cette solution est un peu simpliste. Il serait préférable d'interdire l'usage de cette civilité. Quant à la décision de modifier les « Mademoiselle » en « Madame », elle n'est pas toujours opportune, car dans certaines bases de données ce peut être une information indispensable à conserver, notamment si des traitements sont basés sur cette valeurs.

En tout état de cause, une façon plus judicieuse aurait été de rajouter à la relation de référence un attribut « date d'obsolescence ».

Utilisation massive de tables de référence

L'utilisation systématique des tables de référence est un gage de bonne modélisation et apporte une très grande souplesse dans le fonctionnement d'une base de données. Voici pour ma part comment je modélise mes tables de référence :

- ref_id : la clef de la relation ;
- ref_code : une clef subrogée (ou alternative, comme vous voulez) qui est un code mnémonique utilisable dans les traitements, la valeur de l'ID pouvant changer d'une base à l'autre ;
- ref_libellé : un libellé court ;
- ref_libellé_long : un libellé long ;
- ref_description : une description ;
- ref_date_obsolete : une date d'obsolescence ;
- ref_base : un booléen qui indique si cette référence peut voir son code être modifié (impossible si des traitements sont basés dessus) ;
- ref_ordre : un numérique indiquant l'ordre de tris des valeurs de référence.

On pourrait y rajouter une date de création.

Toutes ces relations étant strictement modélisées de la même façon, il est alors facile de prévoir des objets pour les manipuler, et pourquoi pas, une vue de synthèse et des procédures génériques…

En règle générale, pour savoir si un attribut doit être externalisé dans une table de référence, vous pouvez vous poser la question de l'existence propre de l'attribut. L'attribut seul signifie-t-il quelque chose ? C'est le cas par exemple de la ville, le département, la région, le pays… Peut-il figurer comme élément de regroupement dans un diagramme statistique ? C'est le cas du sexe, de la fonction, du service, du statut…

Références internes et références externes

Nous avons vu que certaines valeurs de référence devait avoir un code immuable, notamment lorsqu'elles servent de paramètre dans des traitements. C'est indispensable pour des relations de référence que le modélisateur crée au gré de ses besoins. Mais qu'en est-il s'il doit traiter des références venant de l'extérieur ?

Pour avoir beaucoup travaillé dans le domaine de la santé où les nomenclatures externes, notamment celles venant de la sécurité sociale ou de l'OMS, sont légion, il convient de coller exactement à la réalité de l'existant, tout en pensant que la structure peut évoluer du jour au lendemain sans préavis. Autrement dit, il faut une indépendance entre les données externes de référence et les informations internes à la base, tout en assurant la continuité du lien de version en version… Exercice périlleux si l'on ne s'y est pas préparé. Par conséquent, il convient de découpler l'externe de l'interne par le biais d'informations faisant le tampon entre l'un et l'autre.

Autres cas plus complexes

Il existe de nombreux autres cas où l'on peut se trouver en situation de viol de la règle de mise à jour unitaire. Voyons cela avec un exemple concret.

Exemple 21 – Modélisation d'un dispensaire

Un dispensaire est un établissement de santé où l'on soigne gratuitement les malades, généralement indigents.

Dans notre dispensaire – celui que nous allons modéliser – il y a différentes salles de consultation et chacun des médecins qui vient y travailler bénévolement passe une journée complète à ausculter et prodiguer des soins. On leur assigne une salle pour la journée et les consultations s'enchaînent. Un même patient ne vient qu'une seule fois par jour, ou plus exactement s'il revient à cause d'un oubli de radio ou de résultats d'examens, il s'agit de la même consultation qui continue.

Pour ce faire, nous avons modélisé comme suit :

```
Rdispensaire : n° patient, date consultation, heure consultation,
médecin, salle d'auscultation
```

Par exemple, les données sont les suivantes :

```
n° patient  date cons.  heure cons.  médecin   salle d'auscult.
----------- ----------- ------------ --------- ----------------
101         20/01/16    09:00        CABROL    Pervenche
203         20/01/16    09:15        CABROL    Pervenche
309         20/01/16    09:00        BARNARD   Tulipe
077         20/01/16    09:30        CABROL    Pervenche
487         20/01/16    09:45        CABROL    Pervenche
309         20/01/16    10:00        BARNARD   Tulipe
890         20/01/16    10:00        CABROL    Pervenche
```

Que va-t-il se passer si, lorsque le professeur Cabrol arrive le matin du 20 janvier 2016, on constate que la salle est inutilisable, suite à un dégât des eaux ?

Il faut tout simplement le déplacer de salle, ce qui conduit à la modification de plusieurs lignes… Or une seule information a changé !

Il en est de même si le professeur Cabrol est empêché ce jour-là et doit être remplacé au pied levé par le docteur Charcot…

Une modélisation plus correcte est alors la suivante :

```
Rdispensaire : n° patient, date consultation, heure consultation
Rdispensaire_médecin : date consultation, salle d'auscultation,
médecin, Rdispensaire_salle : date consultation, médecin, salle
d'auscultation
```

Sans le savoir, avec cet exemple vous venez de toucher du doigt la forme normale de « Boyce Codd », mais nous avions dit que nous n'en parlerions pas. Boyce et Codd se sont associés, sans doute parce que le viol de cette forme normale suppose au moins une clef à deux attributs dans la relation de départ… Nous n'évoquerons pas non plus des formes normales 4 et 5, qui partent de dépendances multivaluées (la présence d'au moins 3 attributs dans la clef en est un signe)…

En fait, le respect des formes normales 4 et 5, ainsi que de la forme normale de Boyce Codd, permet d'éviter que la modification d'une information impacte plus d'une ligne ! Mais ça non plus je n'aurais pas dû vous le dire !

Le choix de la clef

Le dernier élément indispensable à une excellente modélisation est le choix de la clef de la relation.

Bon nombre de livres scolaires et de professeurs enseignent qu'il faut trouver parmi les attributs de la relation, celui ou ceux qui portent la clef en soi. Revenons à notre exemple de relation employé, dont voici un ensemble plus étoffé des différents attributs :

- nom
- prénom
- date de naissance
- sexe
- matricule
- numéro de sécurité sociale (atomisé en 4 parties)
- salaire
- service
- statut

- fonction
- e-mail (atomisé en deux parties)

Certains éléments de cet ensemble pourraient convenir comme clef de la relation :

- matricule : généralement fait pour être un identifiant interne à l'entreprise ;
- numéro de sécurité sociale : il est censé être unique ;
- e-mail : chaque employé a un e-mail déterminé.

Pourtant, aucun de ces attributs (ou groupe d'attributs) n'est une bonne clef.

Matricule en tant que clef

Il suffit d'imaginer les conséquences catastrophiques que pourrait avoir la fusion de deux entreprises (l'une rachetant l'autre) avec d'éventuels télescopages de valeurs, les deux entreprises ayant une même structure de matricule…

Par ailleurs, rien n'interdit de structurer le matricule pour des besoins de calculs notamment. Par exemple, un matricule contenant un premier attribut indiquant le statut (ouvrier, etam, cadre, dirigeant…), un autre l'année d'embauche, un troisième le rang…

N° de sécurité sociale en tant que clef

Un numéro de sécurité sociale est une référence externe unique. Conférer à une référence externe le rôle de clef d'une relation est la pire des choses, ne serait-ce qu'en raison de l'évolution de la structure de ce numéro (d'ailleurs l'État français songe depuis des années à le remplacer par un numéro de patient unique dans le cadre de la santé). Il suffit d'imaginer alors les conséquences pour passer d'une forme à l'autre dans toutes les tables qui le référence pour voir combien son utilisation est déconseillée comme identifiant de toute nature. De plus, la CNIL interdit un usage sauvage d'une telle information.

Et un numéro de sécurité sociale n'est pas immuable. Les transsexuels peuvent en changer.

E-mail en tant que clef

Si l'on admet ne conserver que la partie utilisateur et sans la partie DNS de l'e-mail, alors se pose le même problème que pour le matricule, si fusion d'entreprise il y a. De plus, avec le DNS, c'est une clef en deux parties…

Si l'on convient en sus que l'utilisateur peut en changer, alors l'affaire devient cauchemardesque.

À la recherche d'une clef performante…

La clef d'une relation a toutes les chances de servir fréquemment d'articulation pour l'opération de jointure dans un très grand nombre de requêtes.

À l'évidence, **plus grande ou plus complexe sera cette clef, plus longue et plus coûteuse sera l'opération de jointure**.

Si nous voulons des performances, il convient de se doter d'une clef :

- la plus courte possible (au plus la longueur du mot du processeur, soit aujourd'hui 64 bits) ;

- constituée d'un seul attribut ;
- invariante (elle ne doit pas changer de valeur au cours du temps ;
- asémantique (presque un corolaire pour l'invariance).

Différentes possibilités s'offrent alors à nous, par exemple ce peut être un littéral codé en ASCII, d'une longueur de 8 (1 caractère = 1 octet), ou encore un nombre entier de type « BIGINT ».

Mais nous devons éliminer d'emblée les valeurs littérales et cela pour deux raisons :

- la difficulté d'exprimer certains caractères (la plupart de ceux inférieurs à 31 sont non imprimables et certains de ceux supérieurs à 127 sont absents des claviers) ce qui restreint le champ des possibles ;
- la problématique de comparaison en tenant compte de la collation ou non (la sensibilité ou l'insensibilité aux majuscules/minuscules, lettres accentuées ou non…) qui entraîne un extra overhead au niveau des recherches, et donc des jointures.

Reste donc le nombre entier… Et c'est la raison pour laquelle tous les SGBDR fournissent aujourd'hui un moyen simple de calculer à la volée des clefs par le biais des auto-incréments qui sont, par nature, asémantiques donc invariants (voilà le corolaire), concis, faciles à exprimer et à calculer…

En revanche, il n'est pas interdit, bien au contraire, de disposer d'autres clefs dites clefs subrogées ou alternatives, qui sont sémantiques, mais ne serviront nullement de jointure, sauf cas tout à fait exceptionnel !

Il faut d'ailleurs remarquer que les évolutions de la norme SQL ont été développées dans ce sens avec le type « REF », mais que peu d'éditeurs de SGBDR ont hélas implémenté. Ce virage vers une clef asémantique systématique a été initié lors de la parution de la norme SQL:1999.

En revanche, l'utilisation du type GUID ou UUID comme clef d'une table est une folie dont les conséquences sont catastrophiques… La longueur de cette information nécessite davantage de ressources pour le rapprochement des valeurs, mais surtout le mécanisme de calcul unique au sein de la machine constitue un *hot spot* qui conduit à des performances lamentables, par le simple fait que toutes les tables doivent taper dans la même et unique boucle de code et cela pour des raisons algorithmiques…

En guise de conclusion…

La donnée, encore la donnée, toujours la donnée !

Ce n'est pas au regard des traitements qu'il faut décider de quelle façon on doit modéliser les données, mais simplement au regard de la sémantique des données. Les traitements doivent indiquer quelles données sont nécessaires, et une fois le « dictionnaire » des données établi, les données elles-mêmes, du fait de leur sémantique, doivent être organisées de telle ou telle façon.

Ainsi le modèle sera optimal et souple, car performant et adaptatif.

Par exemple, le fait même de se poser la question de l'atomisation d'une donnée prouve que l'on n'a pas compris ce qu'étaient la sémantique et l'organisation des données relationnelles. Soit la donnée est atomique, soit elle ne l'est pas, auquel cas il faut l'atomiser.

Par exemple, le fait de se demander dans quelle table ajouter l'information lorsqu'une base est déjà en production est un non-sens. À l'évidence, une nouvelle donnée a peu de chances d'être valuée immédiatement pour toutes les lignes déjà saisies dans la table et entraînera donc l'apparition du NULL. Il faut bien entendu intégrer une nouvelle relation en lien 0..1 (héritage) qui portera ces nouvelles informations.

On constate alors qu'une base bien modélisée contient un grand nombre de tables, dotées d'un faible nombre de colonnes par table.

Ce livre va vous enseigner comment procéder !

Bonne lecture…