100% COURS
MCours.com
Cours et mémoires gratuits
mcourscom@gmail.com

Table des matières

Introduction

Une base de données est un objet particulièrement difficile à définir puisqu'il est abordé en pratique selon différents points de vues :

- pour un utilisateur, une base de données est un espace où il peut enregistrer des informations, les retrouver et les faire traiter automatiquement par un ordinateur (on retrouve là, l'étymologie du mot informatique) ;
- pour un développeur, une base de données est un ensemble de tables, de relations et de procédures écrites en SQL (Structured Query Language) ;
- pour un administrateur informatique, une base de données est un ensemble de données à sauvegarder et sécuriser.

Nous nous contentons ici du rôle de développeur, cela signifie que nous occultons l'administration d'une base de données (puisqu'il s'agit d'un autre métier) mais que nous gardons en tête les préoccupations des utilisateurs (dont ce n'est pas le métier de développer des bases de données).

Dans une base de données personnelle (que l'on manipule dans le logiciel Access de Microsoft par exemple), on retrouve essentiellement un schéma où je suis l'unique concepteur, développeur, fournisseur et analyste des données (cf. figure 1).

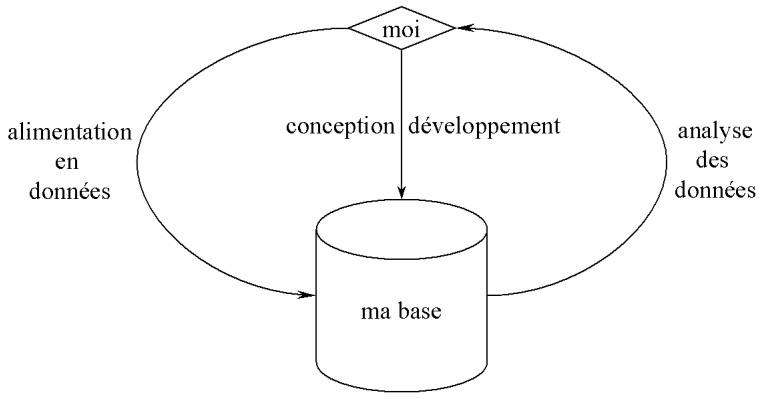

FIG. 1 – *Base de données personnelle*

Au contraire, dans un SGBD professionnel (de type SQL Server, Oracle, DB2 d'IBM et bien d'autres) le schéma est fondamentalement différent : les données sont fournies par plusieurs utilisateurs (parfois des milliers) à travers de multiples petites transactions SQL. Ces données sont stockées dans une ou plusieurs bases de production continuellement remises à jour par ces transactions. Cette partie amont du schéma constitue le système transactionnel (cf. figure 2). Les données sont en général historisées dans un entrepôt

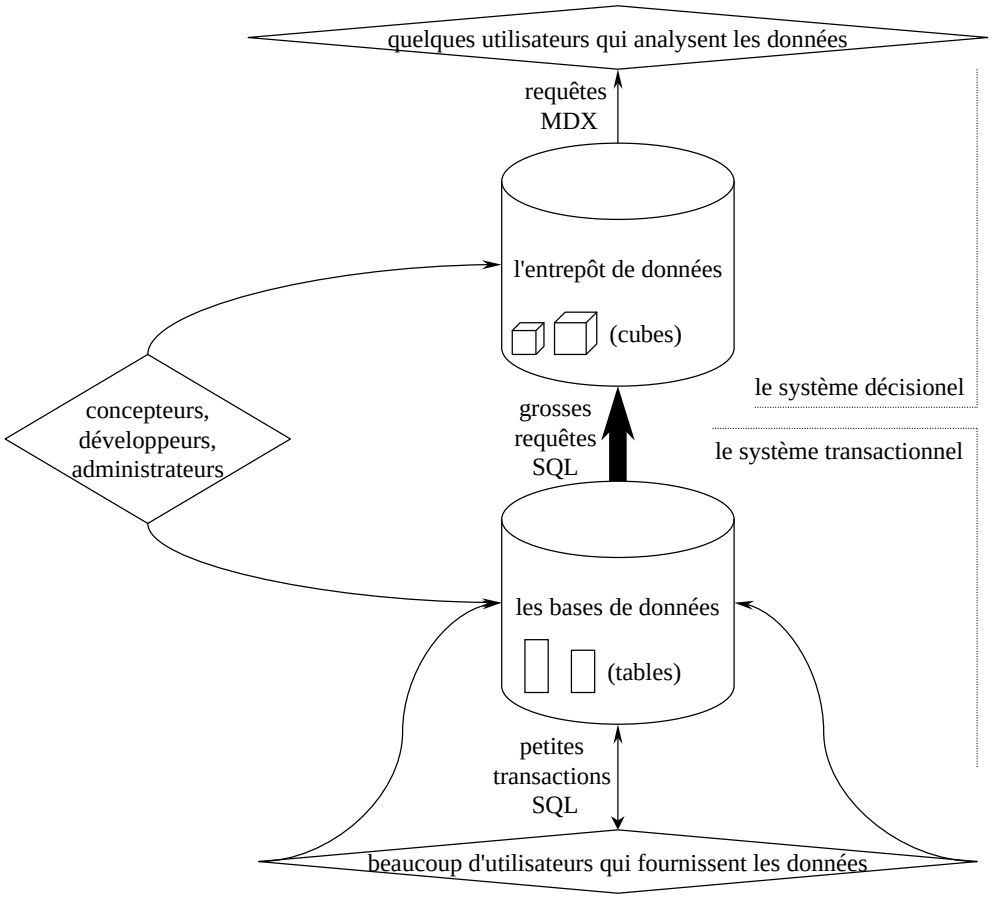

FIG. 2 – *Base de données professionnelle*

de données dont l'élément constitutif n'est plus la table mais le cube. Ceci génère de gros transferts entre les deux systèmes mais les informations utiles sont plus proches des quelques utilisateurs qui ont besoin d'analyser les données. Cette partie aval du schéma constitue le système décisionnel. L'ensemble est géré, dans l'entreprise, par les concepteurs, les développeurs et les administrateurs du service informatique.

Comme illustration nous pouvons prendre n'importe quelle entreprise qui fabrique et vend des produits (cf. figure 3). Les utilisateurs qui fournissent les données sont : les vendeurs, les interlocuteurs

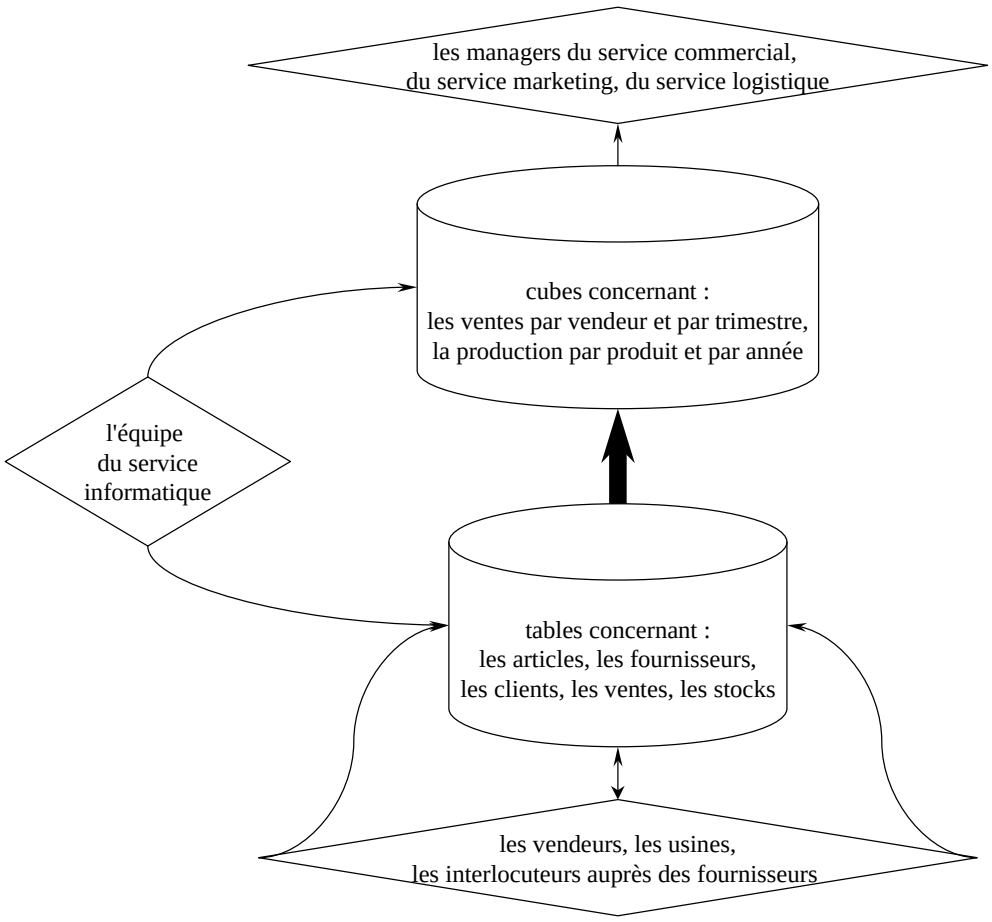

FIG. 3 – *Exemple de base de données professionnelle*

auprès des fournisseurs et des usines. On voit bien qu'ils peuvent être nombreux. Les données seront naturellement stockées dans des tables concernant : les articles, les fournisseurs, les clients, les ventes et les stocks. Toutes ces informations seront regroupées sous forme de cubes concernant notamment : les ventes par vendeur et par trimestre, la production par produit et par usine, etc. Dans cette entreprise, ces cubes sont susceptibles d'intéresser les managers du service commercial, du service marketing et du service logistique. Le rôle du service informatique étant d'échafauder ce système et de proposer des outils pour chaque métier en relation avec les données.

Physiquement, le réseau informatique concerné par le traitement des données est organisé autour d'un ordinateur (ou un cluster d'ordinateurs) équipé de SQL Server et accompagné d'une baie de disques qui stockent les données (cf. figure 4). À ce serveur sont connectés autant de stations de travail clientes que

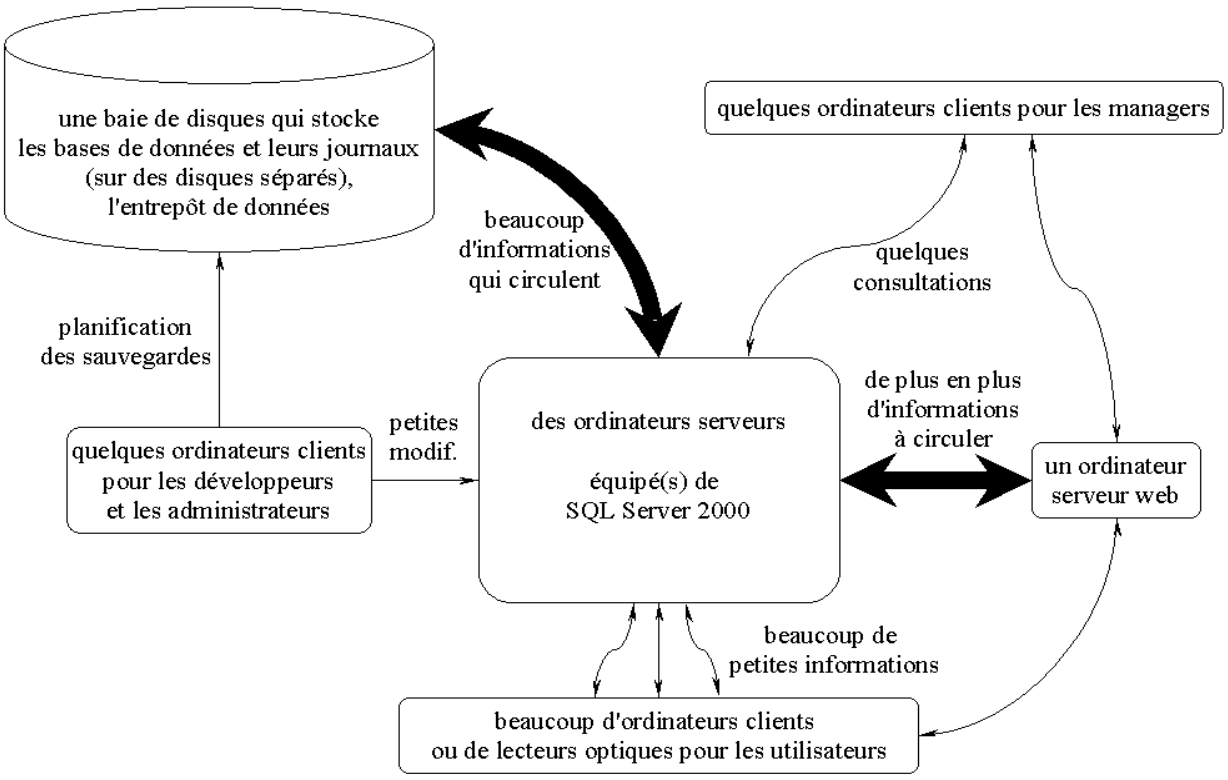

FIG. 4 – *Organisation physique du réseau base de données*

d'utilisateurs, que ce soit les opérateurs en amont, les managers en aval ou le service informatique. De plus en plus, ces utilisateurs passent par Internet, ce qui implique un nombre grandissant d'informations qui circulent entre le serveur web de l'entreprise et le serveur base de données.

D'autres remarques sont à noter concernant le logiciel présenté ici :

- comme SQL Server a toujours quelque chose à faire, il tourne en permanence sur le serveur ; c'est ce que l'on appelle un service : on ne le démarre pas comme un simple exécutable et il continue de tourner quand on se déconnecte ;
- on ne trouvera dans le logiciel SQL Server ni de formulaires ni d'états ; l'interfaçage graphique est laissé aux ordinateurs clients, comme par exemple les applications Visual Basic (dont Access), les applications textes ou encore les pages web. Par ailleurs, l'édition d'états est laissée à d'autres logiciels, comme par exemple Crystal Seagate.

Première partie

Le système transactionnel

En anglais on parle de système OLTP (On Line Transaction Processing). Il s'agit pour nous de concevoir et développer la base de données relationnelle et les transactions qui permettent de modifier les données. On propose de découvrir ici le langage Transact SQL qui est une version propre à SQL Server du langage SQL.

Le langage SQL a été initialement conçu dans les années 1970 par la firme IBM. Il a été ensuite normalisé (la norme actuelle, SQL-2, date de 1992) et est devenu le standard de tous les SGBDR. Ce langage permet de masquer aux programmeurs les algorithmes de recherche des données dans des fichiers physiques eux-même structurés de manière très complexe et différemment selon les SGBDR. Transact SQL prend certaines libertés par rapport à la norme, mais la majeure partie de ce qu'on aborde ici est réutilisable avec un autre système de gestion.

Il se décompose en quatre sous-langages qui s'occupent de :
- la définition des données : création des tables, des contraintes, etc. ;
- la manipulation des données : sélectionner, insérer, supprimer et modifier ;
- le contrôle des données : intégrité, droits d'accès, verrous et cryptage ;
- la programmation : procédures stockées, fonctions, déclencheurs.

Le lecteur ne trouvera rien ici concernant l'administration (sauvegarde, maintenance, ...), l'optimisation (index, compilation, ...) ou l'interfaçage (ADO, SQL-DMO, ...) des bases de données. Pour cela il sera libre de consulter [7] ou [9].

Le lecteur est également invité à se rappeler les méthode de conception d'un bon schéma relationnel (cf. [**?**] et les références citée à l'intérieur) et à se souvenir qu'il est essentiel de connaître le métier des utilisateurs d'une base de données avant de travailler dans celle-ci.

1 Syntaxe du langage SQL

Comme tout nouveau langage commençons par apprendre la syntaxe de base.

Tout d'abord on peut mettre autant d'espaces[1] et de sauts de ligne que l'on veut entre les mots du langage. Cependant, on respectera les règles suivantes :
- une seule instruction par ligne ;
- la même indentation[2] que dans le présent document ;
- et des lignes pas trop longues (visibles entièrement à l'écran).

1.1 Commentaires

On peut insérer des commentaires de deux façons :
- sur une ligne, à partir de deux tirets `--` ;
- dans un bloc délimité par `/*` et par `*/`.

1. dans ce cas espace est un nom féminin
2. c'est-à-dire qu'on respectera le même alignement vertical à l'aide de tabulations

Exemple :

```
1 /* cette requete selectionne
2    toutes les donnees de la
3    table Exemple */
4
5 SELECT * FROM Exemple
6 -- le * designe toutes les colonnes
```

Remarque : ne pas employer les caractères accentués (y compris dans les commentaires)

1.2 Noms

Tous les noms d'objets (table, colonne, variable, etc.) doivent respecter les règles suivantes :

- ne pas dépasser 128 caractères parmi : les lettres (non accentuées), les chiffres, `@`, `$`, `#`, `-` ;
- commencer par une lettre ;
- ne pas contenir d'espace ␣.

Si un nom ne vérifie pas les deux dernières règles, il faut le délimiter par des crochets `[` et `]` (si le nom utilise un crochet fermant, le doubler : `[Bill [William]] Clinton]`).

Par ailleurs, on est pas obligé de respecter la casse (*i.e.* il n'y a aucune différence entre les majuscules et les minuscules). Mais on prendra l'habitude de laisser en majuscule les mots-clés du langage et seulement les mots-clés du langage.

1.3 Opérateurs

- Les opérateurs arithmétiques disponibles sont : `+`, `-`, `*`, `/` et `%` le reste par division entière ;
- les opérateurs de comparaison logique sont : `<`, `<=`, `=`, `>=`, `>` et `<>` (différent) ;
- les autres opérateurs logique sont : `AND`, `OR` et `NOT` ;
- et pour la concaténation des chaînes de caractères on utilise `+`.

Les niveaux de priorité entre ces opérateurs sont usuels, il suffit donc de parenthéser quand on a un doute.

1.4 Variables

Les principaux types disponibles sont :

`INT`	entier
`DECIMAL(9,2)`	montant à 9 chiffres (décimaux) dont 2 après la virgule
`REAL`	réel flottant codé sur 24 bits
`CHAR(64)`	chaîne de caractère de longueur fixe 64
`VARCHAR(64)`	chaîne de caractère de longueur variable mais inférieure ou égale à 64
`DATETIME`	date et/ou heure avec une précision de 3.33 ms

Remarques :
- dans un soucis d'économie d'espace, on peut utiliser pour les entiers les types `SMALLINT`, `TINYINT` et même `BIT` ;
- les entiers de type `INT` peuvent aller jusqu'à un peu plus de 2 milliards, au delà il faut utiliser le type `BIGINT` qui autorise des entiers jusqu'à plus de 9000 milliards ;
- le nombre maximal de décimales est 28 ;

– on peut choisir de stocker les réels flottants sur `n` bits avec le type `FLOAT(n)` (`n` inférieur ou égale à 53) ;
– les chaînes de caractères ne peuvent pas dépasser 8000 caractères, au delà, il faut utiliser le type `TEXT` qui autorise plus de 2 milliards de caractères ;
– on peut définir son propre type, exemple[3] [4] :

```
1 sp_addtype CodePostal, CHAR(5)
```

– pour les conversions entre différents type, il faut parfois employer l'instruction `CAST`[5] (cf. l'aide en ligne).

Lorsque l'on défini une variable, on adopte la convention de faire commencer son nom par `@`.Déclaration, affectation et affichage :

```
1 DECLARE @tva DECIMAL(3,3)
2 SET @tva = 0.186
3 PRINT @tva
```

1.5 Structures

SQL offre les structures usuelles de tout langage.

1.5.1 Blocs

On peut délimiter un bloc de plusieurs instructions par `BEGIN` et par `END`. C'est la structure la plus importante du langage, elle est utilisée par toutes les autres structures, les transactions, les déclencheurs, les procédures stockées et les fonctions.

1.5.2 Branchements conditionnels

On peut parfois avoir besoin d'effectuer un branchement conditionnel, pour cela on dispose de la structure conditionnelle suivante :

`IF` expression booléenne	
...	une instruction ou un bloc
`ELSE`	faculatif
...	une instruction ou un bloc

Exemple tiré de la base de données `Northwind` : on veut supprimer le client `Frank`

```
1 IF EXISTS(SELECT OrderID FROM Orders WHERE CustomerID = 'Frank')
2 -- bref, s'il existe des commandes pour le client Frank
3   PRINT 'Impossible de supprimer le client Frank, car il fait l''objet de commandes'
4 ELSE
5   BEGIN
6     DELETE Customers WHERE CustomerID = 'Frank'
7     PRINT 'Client Frank supprime'
8   END
```

3. `sp_addtype` est une procédure stockée, cf. §9 page 44
4. on a aussi `sp_droptype` pour supprimer un type créé par l'utilisateur
5. ou `CONVERT`

Remarque : les chaînes de caractères sont délimitées par une quote ' et si la chaîne contient elle-même une apostrophe ', il suffit de doubler la quote ''.

Une erreur très fréquente consister à utiliser plusieurs instructions sans les délimiter par un bloc :

```
1 IF(@b <> 0)
2   PRINT 'On peut diviser car b est non nul'
3   @a = @a / @b
4 ELSE
5   PRINT 'On ne peut pas diviser car b est nul'
```

On dispose également de la structure plus générale suivante :

```
CASE
 WHEN expression booléenne THEN
 ...                              une instruction ou un bloc
 WHEN expression booléenne THEN
 ...                              une instruction ou un bloc
 ...                              d'autres WHEN ... THEN
 ELSE
 ...                              une instruction ou un bloc
END
```

Dans laquelle, les différents cas sont évalués successivement.

Exemple tiré de la base de données `Northwind` : on veut savoir quel produit il faut réapprovisionner

```
1  SELECT ProduitID, 'Etat du stock' =
2    CASE
3      WHEN(Discontinued = 1) THEN
4        'Ne se fait plus'
5      WHEN((UnitsInStock - UnitsOnOrder) < ReOrderLevel) THEN
6        'Seuil de reapprivionnement atteint : passer commande'
7      WHEN(UnitsInStock < UnitsOnOrder) THEN
8        'Stock potentiellement negatif : passer commande'
9      ELSE
10       'En stock'
11   END
12 FROM products
```

Exercice : une erreur s'est glissée dans l'exemple précédent.

1.5.3 Boucles conditionnelles

La seule façon d'effectuer une boucle est d'utiliser la structure suivante :

```
WHILE expression booléenne
 ...                              une instruction ou un bloc
```

On ne dispose pas de boucle `FOR` pour la simple raison que les boucles `WHILE` suffisent :

```
1 DECLARE @i
2 SET @i = 0
3
4 WHILE(@i < @n)
5   BEGIN
6     ...
7     @i = @i + 1
8   END
```

Par ailleurs, pour parcourir toutes les lignes d'une table, il suffit bien souvent d'utiliser l'instruction `SELECT`. Les boucles sont donc inutiles en général.

2 Modes d'exécution du code SQL

Une fois qu'on a écrit (sans erreur) son code, SQL étant un langage interprété, on peut décider quand et comment l'exécuter. La première étape consiste bien souvent à préciser sur quelle base de données on compte travailler. Pour cela on dispose de l'instruction `USE`. Exemple :

```
1 USE northwind
```

2.1 Exécution immédiate

Dans l'Analyseur de requête, sélectionner la partie du code à exécuter et taper sur `F5`, `CTRL+E`, `ALT+X` ou cliquer sur le bouton lecture.

2.2 Utilisation de script

On peut enregistrer le code SQL dans des fichiers textes d'extension `.sql` (il s'agit-là d'une convention que l'on adopte) pour les exécuter plus tard. Sous MS-DOS, on peut exécuter un script `truc.sql` avec l'utilitaire `osql` en tapant :

```
osql -i truc.sql
```

2.3 Exécution par lots

Dans l'utilitaire `osql` on peut également taper les lignes une par une et taper `GO` pour lancer l'exécution. Les instructions entre deux `GO` successifs forment un lot. Si une erreur existe dans un lot, aucune instruction ne sera réellement exécutée. Le lot passe donc soit en totalité, soit pas du tout.

On peut écrire les `GO` dans un script, mais on préférera utiliser les transactions.

2.4 Transactions

Une transaction est une suite d'instructions qui réussissent ou qui échouent en totalité (pas de réussite partielle). Si elle réussit, les modifications apportées à la base sont permanentes, et la transaction est inscrite au journal. Si une instruction échoue, toute la transaction est annulée et la base retrouve l'état dans lequel elle était avant la transaction.

Toutes les transactions figurent dans un fichier que l'on appelle le journal des transactions. Ce journal permet de restaurer la base de données en cas de panne sur le ou les fichiers de données. Ces fichiers de données sont évidemment sauvegardés régulièrement, mais pour pouvoir restaurer complètement la base (en cas de plantage) il faut pouvoir refaire toutes les modifications depuis la dernière sauvegarde. C'est

le rôle du journal des transactions de contenir toutes ces informations. Il est donc généralement stocké sur un autre disque.

On dit qu'une transaction est ACID :
- Atomique, au sens où on ne peut pas la diviser en une partie qui échoue et une partie qui réussit ;
- Consistante, au sens où une fois la transaction terminée, la base est de nouveau dans un état cohérent ;
- Isolée, au sens où une transaction considère que, pendant son exécution, les données qu'elle manipule ne sont pas modifiées par une autre transaction ;
- et Durable, au sens où les modifications opérées par la transaction sont enregistrées de façon permanente (et recouvrables en cas de reconstruction de la base).

La syntaxe pour délimiter une transaction est la suivante :

```
BEGIN TRAN
 ...          une suite d'instructions
COMMIT TRAN
```

C'est une notion importante : si le transfert d'une somme d'argent est encapsulé dans une transaction qui regroupe le débit du compte source et le crédit du compte destination, alors il n'y aura pas de fuite d'argent même en cas d'erreur.

2.5 Débogage

Il n'y a pas dans SQL Server de débogage à proprement parler. Tout juste dispose-t-on d'une vérification de la syntaxe des requêtes SQL. Il faut donc se débrouiller avec l'affichage des résultats à l'écran.

3 Mise en place d'une base

Toutes les opérations qui permettent de créer une base de données sont disponibles dans Enterprise Manager sous forme de boîtes de dialogue et de boutons. Mais on peut également les organiser dans un code SQL.

3.1 Une base et son journal

Une base de données SQL Server contient au minimum :
- un fichier de données principal (d'extension `.mdf`) où sont stockées les données ;
- un journal des transactions (d'extension `.ldf`) où sont répertoriées toutes les transactions.

Lorsque l'on créé une base, il faut donc préciser le nom, l'emplacement et la taille de ces deux fichiers.

Exemple : créons une base de données `papeterie`

```
CREATE DATABASE papeterie                          -- le nom de la base
  ON PRIMARY                                       -- le fichier de donnees principal
  (
    NAME = papeterie_data,                         -- nom logique
    FILENAME = 'C:\Data\papeterie.mdf',            -- emplacement et nom du fichier
    SIZE = 60MB,                                   -- taille de depart
    MAXSIZE = 70MB,                                -- taille maximale
    FILEGROWTH = 1MB                               -- increment
  )
  LOG ON                                           -- le journal
  (
    NAME = papeterie_log,
    FILENAME = 'D:\Log\papeterie.ldf',
    SIZE = 15MB,
    MAXSIZE = 20MB,
    FILEGROWTH = 1MB
  )
```

Pour modifier une base de données existante, on utilise l'instruction `ALTER DATABASE`. Par exemple :

```
ALTER DATABASE papeterie
  MODIFY NAME cartoleria
```

Remarque : d'autres modifications sont possibles.

Pour supprimer une base de données existante, il suffit de taper :

```
DROP DATABASE papeterie
```

3.2 Une table

Lors de la création d'une table dans une base de données existante, il faut préciser :
– pour chaque colonne : son nom et son type de données ;
– une clé primaire (qui permet d'identifier chaque ligne de façon unique).

On peut éventuellement préciser pour chaque colonne si vide[6] est interdit et/ou une valeur par défaut.

Exemple de création d'une table :

```
CREATE TABLE clients
(
  clt_num CHAR(8) PRIMARY KEY,    -- cle primaire
  clt_nom VARCHAR(64) NOT NULL,   -- vide interdit
  clt_ca INT DEFAULT 0            -- valeur par defaut
)
```

6. `NULL` représente une absence d'information

Pour modifier une table existante, on utilise l'instruction `ALTER TABLE`. Exemples :

```
ALTER TABLE clients
  ADD clt_adr VARCHAR(255)  -- pour ajouter la colonne adresse

ALTER TABLE clients
  DROP COLUMN clt_adr       -- pour retirer la colonne adresse

ALTER TABLE clients
  ALTER COLUMN clt_num INT  -- pour reconvertir le type de donnees
```

Pour supprimer une table existante, il suffit de taper :

```
DROP TABLE clients
```

3.3 Numérotation automatique

Pour la clé primaire d'une table, il est souvent préférable de laisser SQL Server générer des valeurs distinctes. On dispose pour cela de deux possibilités :

- une valeur entière qui s'incrémente automatiquement ;
- un identificateur unique universel (GUID), c'est-à-dire un nombre codé sur 16 octets en logique polonaise inverse.

Nous nous contentons de la première alternative :

```
CREATE TABLE clients
(
  clt_num INT PRIMARY KEY IDENTITY(4,2),
  -- les numeros des clients successifs seront 4, 6, 8, ...
  ...
)
```

Remarque : l'avantage d'avoir un incrément > 1 est de pouvoir ensuite insérer des numéros parmi les numéros automatiques (ce qui représente un intérêt limité, nous nous contentons donc bien souvent de `IDENTITY(1,1)`).

À titre d'information, la seconde alternative s'emploie ainsi :

```
ALTER TABLE clients
  ALTER COLUMN clt_num UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID()
```

3.4 Définir les relations

La commande d'un produit est forcément passée par un client. Donc la table `commandes` devra contenir une colonne pour savoir quel client est concerné. Cette colonne `cmd_clt` contiendra en fait la clé primaire du client concerné. Il y a donc une relation entre `cmd_clt` et la colonne `clt_num` de la table `clients` (cf. figure 5). Comme `cmd_clt` va chercher ses valeurs dans une autre colonne, elle constitue ce que l'on appelle une clé étrangère.

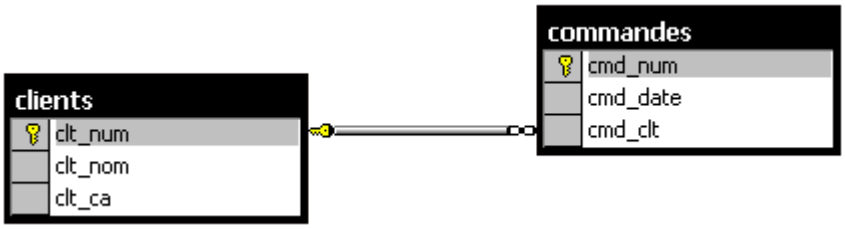

FIG. 5 – *Relation entre deux tables*

La syntaxe pour créer la table `commandes` est alors :

```
1  CREATE TABLE commandes
2  (
3    cmd_num INT PRIMARY KEY IDENTITY(1,1),
4    cmd_date DATETIME DEFAULT GETDATE(),
5    -- GETDATE() retourne la date d'aujourd'hui et l'heure courante
6    cmd_clt INT NOT NULL FOREIGN KEY REFERENCES clients(clt_num)
7  )
```

Remarques :
- `cmd_clt` et `clt_num` doivent être du même type ;
- on pourrait se contenter de `REFERENCES clients` car `clt_num` est clé primaire ;
- cette relation introduit deux contraintes :
  - lors d'une nouvelle commande, le client devra déjà exister ;
  - lors de la suppression d'un client, il ne devra plus faire l'objet de commande.

4 Sélectionner les données

On entre ici au cœur du langage SQL puisque ce sont les requêtes de sélection qui réclament le plus de conception de part du programmeur.

4.1 Sélection simple

Rappelons que la syntaxe pour effectuer une requête de sélection est :

`SELECT` colonnes
`FROM` tables
`WHERE` condition1 `AND` / `OR` condition2

Exemple : si on veut toutes les commandes du client `Razibus`

```
1  SELECT cmd_num, cmd_date
2  FROM commandes, clients
3  WHERE clt_nom = 'Razibus'
4  AND cmd_clt = clt_num      -- il faut rappeler la relation
```

Remarques :
– l'ordre n'a pas d'importance pour la relation (on aurait pu écrire `clt_num = cmd_clt` ;
– l'ordre n'a pas non plus d'importance entre les deux conditions de `WHERE` (on aurait pu mettre `clt_num = 'Razibus'` après).

Dans les conditions `WHERE` (reliées entre elles par `OR` ou `AND`) on peut utiliser
– `=`, `<>` et tous les opérateurs de comparaison :

```
WHERE clt_nom <> 'Razibus'
```

– une plage de valeurs (bornes incluses) :

```
WHERE clt_ca BETWEEN 10000 AND 100000
```

– une liste de valeurs :

```
WHERE clt_nom IN ('Razibus', 'Fricotin', 'Mironton')
```

– un filtre :

```
WHERE clt_nom  LIKE 'R%'        -- commençant par R
                                -- % remplace toute série de caractères
                                -- (y compris vide)
               LIKE 'R_zibus'   -- _ remplace un caractère
               LIKE '%[M-R]'    -- finissant par M, N, O, P, Q ou R
               LIKE '%[^FMR]%'  -- ne contenant ni F ni M ni R
```

Remarque : on dispose évidemment de

```
NOT BETWEEN ... AND ...
NOT IN ...
NOT LIKE  ...
```

Par ailleurs, on peut
– intituler les colonnes (si l'intitulé contient des espaces ou des accents, le délimiter avec des crochets `[ ]`) :

```
SELECT cmd_num AS [numéro de commande], cmd_date AS date
```

– trier les résultats :

```
SELECT ... FROM ... WHERE ...
ORDER BY cmd_date DESC, cmd_num ASC
```

c'est-à-dire par dates décroissantes, puis (pour les commandes de la même date) par numéro croissant

– n'afficher que des résultats distincts :

```
SELECT DISTINCT ...
```

– n'afficher que les premiers résultats :

```
SELECT TOP 50 ...
```

Mais malheureusement, on ne peut pas utiliser les alias définis dans la clause `SELECT` dans les autres clauses (`WHERE` et `ORDER BY` notamment).

4.2 Jointures internes

Quand on utilise deux tables ou plus, on effectue en réalité des jointures entre ces tables. Donc désormais on écrira plutôt :

```
1 SELECT cmd_num, cmd_date
2 FROM commandes
3 JOIN clients ON cmd_clt = clt_num  -- condition de jointure
4 WHERE clt_nom = 'Razibus'          -- condition de selection
```

Remarques :
- ceci permet de bien séparer les conditions de jointure des conditions de sélection ;
- une jointure est en quelque sorte un produit cartésien temporaire : la table `(cmd_num, cmd_date, clt_nom)` est provisoirement créée pour effectuer la sélection ;
- le moteur du SGBD se charge de trouver le moyen d'effectuer le moins d'opérations possible ;
- une jointure n'est pas seulement le rappel des relations entre deux tables, c'est une vraie condition (qui peut utiliser `<>` et les autres opérateur de comparaison à la place de `=` par exemple) et pas forcément entre une clé étrangère et sa référence ;
- on peut effectuer plusieurs jointures successives :

```
FROM commandes
JOIN clients ON cmd_clt = clt_num
JOIN articles ON cmd_art = art_num
```

- pour une même jointure, on peut utiliser plusieurs conditions de jointures (reliées entre elles par `AND` ou `OR`).

Pour être tout à fait rigoureux, il faut toujours préciser la table à laquelle appartiennent les colonnes utilisées (en utilisant des alias). On écrira donc désormais :

```
1 SELECT a.cmd_num, a.cmd_date
2 FROM commandes AS a
3 JOIN clients AS b ON a.cmd_clt = b.clt_num
4 WHERE b.clt_nom = 'Razibus'
```

Exercice : sélectionner les clients ayant commandé le même jour (le résultat devra se présenter sous forme de deux colonnes : un client et un autre qui a commandé le même jour).

Solution : avec les alias, l'auto-jointure devient possible

```
1 SELECT DISTINCT a.cmd_clt, b.cmd_clt
2 FROM commandes AS a
3 JOIN commandes AS b ON a.cmd_clt <> b.cmd_clt
4 -- la condition de jointure est que les deux clients ne sont pas les memes
5 WHERE a.cmd_date = b.cmd_date
6 -- parmi tous les couples de clients distincts on ne garde que ceux-la
```

4.3 Jointures externes

Imaginons maintenant que l'on dispose de la table `clients_plus` qui contient les colonnes `clt_num`, `clt_adresse`, `clt_email`, `clt_telephone` et que l'on veuille afficher la liste complète des clients ainsi que leur renseignements complémentaires s'ils existent.

La première idée consiste à effectuer la requête suivante :

```
1 SELECT a.clt_nom, b.clt_adresse, b.clt_email, b.clt_telephone
2 FROM clients AS a JOIN clients_plus AS b
3                   ON a.clt_num = b.clt_num
```

Problème : ne s'affichent que les clients ayant des informations complémentaires. La solution consiste à rendre facultative la jointure avec la table de droite.

```
1 SELECT a.clt_nom, b.clt_adresse, b.clt_email, b.clt_telephone
2 FROM clients AS a LEFT JOIN clients_plus AS b
3                   ON a.clt_num = b.clt_num
4 -- jointure facultative gauche
```

Attention : si c'est la table de droite qui est facultative, alors il s'agit d'une jointure externe gauche.

Autre cas : on veut la liste des adresses électroniques de la table `clients_plus` mais parfois il n'y a aucun client de rattaché.

```
1 SELECT b.clt_email, a.clt_nom
2 FROM clients AS a RIGHT JOIN clients_plus AS b
3                   ON a.clt_num = b.clt_num
4 -- jointure facultative droite
```

Dernier cas : on veut la liste des clients dont on a soit le nom, soit l'e-mail.

```
1 SELECT a.clt_nom, b.clt_email
2 FROM clients AS a FULL OUTER JOIN clients_plus AS b
3                   ON a.clt_num = b.clt_num
4 -- jointure facultative dans les deux sens
```

4.4 Union des sélections

On peut regrouper les résultats de plusieurs requêtes de sélection.

Exemple : à supposer que les tables `commandes2000` et `comandes2001` existent, on peut visualiser les commandes d'un client au cours de ces deux années ainsi :

```
1  SELECT a.cmd_num, a.cmd_date
2  FROM commandes2000 AS a JOIN clients AS b
3                          ON a.cmd_clt = b.clt_num
4  WHERE b.clt_nom = 'Razibus'
5
6  UNION
7
8  SELECT a.cmd_num, a.cmd_date
9  FROM commandes2001 AS a JOIN clients AS b
10                         ON a.cmd_clt = b.clt_num
11 WHERE b.clt_nom = 'Razibus'
```

Remarques :
- l'utilisation de l'opérateur `UNION` suppose que les colonnes sélectionnées des les deux requêtes sont du même nombre, du même type et dans le même ordre (mais pas forcément les mêmes) ;
- les doublons sont supprimés automatiquement (*i.e.* on ne retrouve pas deux fois la même ligne) à moins d'utiliser `UNION ALL` ;
- pour spécifier les intitulés de colonnes, préciser les alias `AS` dans la première clause `SELECT`.

Il est parfois possible de substituer un `UNION` par une jointure supplémentaire et plusieurs conditions `WHERE`, mais c'est plus facile d'utiliser `UNION`. De plus, on peut parfois obtenir de meilleures performances en décomposant une requête complexe en une série de `SELECT` combinés avec l'opérateur `UNION`.

4.5 Sous-requêtes

Les conditions de sélection de la clause `WHERE` peuvent utiliser le résultat d'une autre requête.

4.5.1 Sous-requête renvoyant une valeur

Lors que cette autre requête ne revoie qu'une valeur, on peut utiliser `=` et tous les autres opérateurs de comparaison logique[7].

Exemple : pour afficher les commandes d'un client on peut utiliser une sous-requête.

```
1 SELECT cmd_num, cmd_date
2 FROM commandes
3 WHERE cmd_clt = ( SELECT clt_num
4                   FROM clients
5                   WHERE clt_nom = 'Razibus' )
```

4.5.2 Sous-requête renvoyant une liste de valeurs

Les sous-requêtes qui renvoie une liste de valeurs peuvent être naturellement utilisé par l'opérateur `IN`.

Exemple : on veut les commandes de tous les clients dont le nom commence par `P`.

```
1 SELECT cmd_num, cmd_date
2 FROM commandes
3 WHERE cmd_clt IN ( SELECT clt_num
4                    FROM clients
5                    WHERE clt_nom LIKE 'P%' )
```

Le langage SQL offre d'autres mot-clé pour ce type de sous-requête, découvrons-les par l'exemple.Avec la table `articles` qui comporte les colonnes `art_num`, `art_nom`, `art_prix` et `art_couleur` on veut successivement :
- les articles dont le prix est supérieur à tous les articles blancs :

```
1 SELECT art_nom
2 FROM articles
3 WHERE art_prix > ALL ( SELECT art_prix
4                        FROM articles
5                        WHERE art_couleur = 'blanc' )
```

7. mais aussi `BETWEEN ... AND ...`

ceci est équivalent à

```
SELECT art_nom
FROM articles
WHERE art_prix > ( SELECT MAX(art_prix)
                   FROM articles
                   WHERE art_couleur = 'blanc' )
```

– les articles dont le prix est supérieur à l'un des articles blancs :

```
SELECT art_nom
FROM articles
WHERE art_prix > ANY ( SELECT art_prix
                       FROM articles
                       WHERE art_couleur = 'blanc' )
```

ceci est équivalent à

```
SELECT art_nom
FROM articles
WHERE art_prix > ( SELECT MIN(prix)
                   FROM articles
                   WHERE art_couleur = 'blanc' )
```

– tous les articles mais seulement s'il en existe un de blanc (pourquoi pas ?) :

```
SELECT art_nom
FROM articles
WHERE EXISTS ( SELECT art_num
               FROM articles
               WHERE art_couleur = 'blanc' )
```

– tous les articles mais seulement s'il n'en existe pas de blanc (pourquoi pas non plus ?) :

```
SELECT art_nom
FROM articles
WHERE NOT EXISTS ( SELECT art_num
                   FROM articles
                   WHERE art_couleur = 'blanc' )
```

4.5.3 Requêtes correlées

Lorsqu'une sous-requête a besoin d'information de la requête parent, on dit qu'elle est corrélée. Il suffit d'utiliser des alias `AS` pour lui passer les informations.

Exemple : quels sont les clients qui ont passé une commande d'un montant supérieur à 1 % de leur chiffre d'affaire ?

```
1 SELECT clt_nom
2 FROM clients AS a
3 WHERE (clt_ca / 100) < ANY ( SELECT cmd_montant
4                               FROM commandes AS b
5                               WHERE b.cmd_clt = a.clt_num )
```

Remarques :
– l'alias `a` est défini dans la requête appelante et est utilisable dans la sous-requête ;
– La sous-requête sera exécutée autant de fois qu'il y a de clients.

4.5.4 Requêtes imbriquées *vs.* jointures

Souvent une sous-requête peut être remplacée par une jointure :

```
1 SELECT DISTINCT a.clt_nom
2 FROM clients AS a
3 JOIN commandes AS b ON b.cmd_clt = a.clt_num
4 WHERE (a.clt_ca / 100) < b.cmd_montant
```

Lorsqu'on emploie les requêtes imbriquées, on précise à SQL Server comment effectuer la requête ; c'est une façon procédurale de voir les choses. Tandis que quand on utilise des jointures c'est une forme relationnelle et SQL Server se charge de faire pour le mieux.

Par contre, il n'y a parfois pas d'équivalence jointures à une écriture en sous-requêtes. Mais quand on a un équivalent, il vaut mieux utiliser les jointures car la requête sera optimisée par l'interprète SQL. Ceci dit, l'utilisation de sous-requête est plus lisible ...

4.5.5 Sous-requête renvoyant plusieurs colonnes

Une sous-requête renvoyant une ou plusieurs colonnes peut être utilisée comme table dans la clause `FROM`. Cela peut servir par exemple à ne sélectionner que les plus gros clients :

```
1 SELECT a.clt_nom
2 FROM ( SELECT TOP 10 *
3        FROM clients
4        ORDER BY clt_ca DESC ) AS a
5 JOIN commandes AS b ON b.cmd_clt = a.clt_num
6 WHERE (a.clt_ca / 100) < b.cmd_montant
```

Par contre, on ne peut pas utiliser ce type de sous-requête dans une clause `WHERE` (sauf avec `EXISTS`), contrairement à Oracle.

4.6 Requêtes multibases

À l'origine, SQL ne permet pas de faire de requêtes qui portent sur les tables de plusieurs bases de données et encore moins gérées par différents SGBDR. Transact-SQL offre un mécanisme de dénomination qui permet d'effectuer des jointures entre des tables issus de systèmes hétérogènes.

La syntaxe complète du nom d'une table est :

[nom du serveur] . [nom de la base] . [nom du propriétaire] . [nom de la table]

Le propriétaire est généralement `dbo` le database owner.

Cette syntaxe permet d'écrire une jointure portant sur les tables des deux bases de données différentes (sur le même serveur) :

```
1 SELECT a.cmd_num, b.art_nom
2 FROM GestionCommerciale.dbo.commandes AS a
3 JOIN GestionProductique.dbo.articles AS b ON a.cmd_art = b.art_num
```

Ou une jointure portant sur deux bases de données gérées par des serveurs différents :

```
1 SELECT a.clt_nom AS [clients communs]
2 FROM ENTREPRISE1.GestionCommerciale.dbo.clients AS a
3       JOIN ENTREPRISE2.GestionCommerciale.dbo.clients AS b ON a.clt_nom = b.clt_nom
```

Cependant, la requête étant effectuée sur un serveur SQL Server (`ENTREPRISE1` par exemple), il faut que les autres serveurs utilisés (`ENTREPRISE2` en l'occurrence) soient déclarés comme serveurs liés dans le serveur SQL Server.

Remarque : `ENTREPRISE2` n'est pas forcément un serveur SQL Server, mais peut être n'importe quel SGBD reconnu par DTS (cf. 17 page 75), ce qui permet d'écrire des requêtes sur des données hétérogènes (cf. [1]).

4.7 Quelques fonctions SQL

À tout moment dans une requête `SELECT` on peut utiliser de nombreuses fonctions, à commencer par la fonction suivante qui s'avère souvent très utile : `ISNULL` qui permet de remplacer `NULL` par une autre valeur. Par exemple, pour remplacer une absence de chiffre d'affaire par un chiffre d'affaire nul :

```
1 SELECT clt_nom, ISNULL(clt_ca, 0)
2 FROM clients
```

4.7.1 Fonctions d'agrégat

`COUNT`	dénombrement
`SUM`	
`AVG`	moyenne
`VAR`	variance
`STDEV`	écart-type
`MIN`	
`MAX`	

Exemple :

```
1 -- pour compter le nombre de client
2 SELECT COUNT(clt_num)
3 FROM clients
4
5 -- pour connaitre le chiffre d'affaire moyen des clients
6 SELECT AVG(clt_ca)
7 FROM clients
```

Remarque : toutes ces fonctions ignorent les valeurs `NULL` (surtout `COUNT`).

4.7.2 Opérateurs

C'est-à-dire : `+`, `-`, `*`, `/`, `%` et le `+` de concaténation. Exemple :

```
1 -- pour afficher le chiffre d'affaire mensuel moyen de chaque client
2 SELECT clt_nom, clt_ca / 12 AS [ca mensuel]
3 FROM clients
4
5 -- pour concatener le nom et le prenom
6 SELECT clt_nom + ' ' + clt_prenom AS [Identité]
7 FROM clients
```

4.7.3 Fonctions sur les dates

Avec `date` de type `DATETIME` :

`DATEADD(year, 4, date)`	ajoute 4 ans à `date`
`DATEADD(month, 4, date)`	
`DATEADD(week, 4, date)`	
`DATEADD(day, 4, date)`	
`DATEADD(hour, 4, date)`	
`DATEDIFF(minute, date_debut, date_fin)`	donne la différence en minutes entre
`DATEDIFF(second, date_debut, date_fin)`	`date_fin` et `date_debut`
`DATEPART(month, date)`	renvoie le numéro du mois de `date`

Remarque : `DATEDIFF` et `DATEPART` renvoient un entier.

Reprenons l'exemple de l'auto-jointure. Si on veut vraiment sélectionner les clients qui ont commandé le même jour, il faut remplacer le test d'égalité entre les dates par :

```
1 SELECT DISTINCT a.cmd_clt, b.cmd_clt
2 FROM commandes AS a
3 JOIN commandes AS b ON a.cmd_clt <> b.cmd_clt
4 WHERE DATEDIFF(day, a.cmd_date, b.cmd_date) = 0
5 -- sinon il s'agit d'une egalite a 3.33 ms pres
```

Remarque : la requête précédente n'est pas équivalente à la suivante.

```
SELECT DISTINCT a.cmd_clt, b.cmd_clt
FROM commandes AS a
JOIN commandes AS b ON a.cmd_clt <> b.cmd_clt
WHERE DATEDIFF(hour, a.cmd_date, b.cmd_date) BETWEEN -24 AND 24
/* dans ce cas les clients ont commande a moins de 24h d'intervalle
   mais pas forcement le meme jour */
```

4.7.4 Fonctions sur les chaînes de caractères

Notamment : `LEN` (longueur), `LOWER` (convertit tout en minuscule), `REPLACE`, `SUBSTRING` et `UPPER` (tout en majuscule).

4.7.5 Principales fonctions mathématiques

À savoir : `ABS` (valeur absolue), `CEILING` (partie entière +1), `COS`, `EXP`, `FLOOR` (partie entière), `LOG` (logarithme neperien), `LOG10`, `PI`, `POWER`, `SIGN`, `SIN`, `SQRT`, `SQUARE` et `TAN`.

Par exemple, on peut écrire la dernière requête ainsi :

```
SELECT DISTINCT a.cmd_clt, b.cmd_clt
FROM commandes AS a
JOIN commandes AS b ON a.cmd_clt <> b.cmd_clt
WHERE ABS(DATEDIFF(hour, a.cmd_date, b.cmd_date)) <= 24
```

4.7.6 Fonctions utilisateur

On peut aussi définir ses propres fonctions. Syntaxe :

```
CREATE FUNCTION ...   (son nom)
 (...)                (ses paramètres)
 RETURNS ...          (le type de la valeur de retour)
AS
 BEGIN
 ...
 RETURN ...           (la valeur de retour)
 END
```

La rédaction de ces fonctions est la même que celle des procédures stockées (cf. §9 page 44) :

```
CREATE FUNCTION EcartEnHeure
(
  @date1 DATETIME,
  @date2 DATETIME
)
  RETURNS INT
AS
  BEGIN
    RETURN ABS(DATEDIFF(hour, @date1, @date2))
  END
```

Puis on peut l'utiliser dans une requête :

```
SELECT DISTINCT a.cmd_clt, b.cmd_clt
FROM commandes AS a
JOIN commandes AS b ON a.cmd_clt <> b.cmd_clt
WHERE dbo.EcartEnHeure(a.cmd_date, b.cmd_date) <= 24
/* dans le cas d'une fonction utilisateur
   il ne faut pas oublier le proprietaire */
```

Remarques :
– on peut mettre jusqu'à 1024 paramètres ;
– on dispose de `ALTER FUNCTION` et de `DROP FUNCTION`.

4.8 Conclusion

On aboutit finalement à la stratégie suivante pour élaborer une requête de sélection :

1. décomposer au maximum en plusieurs sélection que l'on pourra réunir avec `UNION` ;
2. décomposer chaque sélection complexe en requête et sous-requêtes simples ;
3. et pour chaque requête et chaque sous-requête :
   (a) déterminer les tables en jeu pour remplir la clause `FROM` et les `JOIN` nécessaires ;
   (b) déterminer les colonnes à afficher pour remplir la clause `SELECT` ;
   (c) déterminer les conditions de sélection pour remplir la clause `WHERE` ;
   (d) ajouter les éventuels `ORDER BY`, `DISTINCT` et `TOP` en dernier.

5 Modifier les données

Certes, avant de sélectionner les données il faut pouvoir en ajouter dans une base *a priori* vide. Mais avant d'apprendre à modifier les données en SQL il faut savoir les sélectionner. C'est pourquoi nous n'abordons que maintenant les requêtes d'insertion, de suppression et de mise-à-jour.

Il est sous-entendu ici que l'on a les droits de modification nécessaires sur les tables concernées. Par ailleurs, il est conseillé d'inclure toutes les opérations de modifications des données dans une transaction, non seulement parce que ces opérations peuvent échouer partiellement, mais aussi afin qu'elles figurent au journal.

5.1 Insertion

En SQL on ne peut insérer des lignes que dans une table à la fois. On peut ajouter :
– des données complètes (on précise alors toutes les colonnes)

Exemple :

```
BEGIN TRAN
  INSERT clients                    -- LA table
    VALUES (16, 'Razibus', 3000000) -- toutes les colonnes et dans l'ordre
COMMIT TRAN
```

Remarque : on ne peut mettre qu'un `VALUES` par `INSERT`, mais plusieurs `INSERT` par transaction

– des données partielles (on ne précise que certaines colonnes)

Exemple :

```
1 BEGIN TRAN
2   INSERT clients(clt_nom, clt_num) -- l'ordre n'a pas d'importance
3     VALUES ('Fricotin', 18)        -- tant qu'il est le meme ici
4 COMMIT TRAN
```

Remarques : il est obligatoire d'insérer des valeurs
– compatibles avec le type de la colonne
– dans toutes les colonnes déclarées `NOT NULL` et qui n'ont pas de valeur par défaut

– des données issues d'une sélection (on introduit plusieurs lignes à la fois)

Exemple : supposons que l'on dispose d'une table `clients_importants` qui n'ai que la colonne `clt_num`

```
1 BEGIN TRAN
2   INSERT clients_importants(clt_num)
3     SELECT TOP 100 clt_num
4     FROM clients
5     ORDER BY clt_ca DESC
6 COMMIT TRAN
```

– dans une table temporaire (aucune clé primaire ni étrangère, donc aucune relation avec le schéma relationnel)

Exemple : si la table `clients_importants` n'existe pas encore

```
1 SELECT TOP 100 clt_num
2 INTO clients_importants
3 FROM clients
4 ORDER BY clt_ca DESC
```

Remarques :
– la table temporaire contient alors les mêmes colonnes que le résultat de la requête `SELECT` ;
– on ne peut pas utiliser `SELECT ... INTO` dans une transaction ;
– ne pas oublier le `DROP TABLE` une fois qu'on a plus besoin de la table temporaire.

5.2 Suppression

À nouveau, on ne peut supprimer des lignes que dans une table à la fois. La syntaxe pour effectuer une requête de suppression est :

`DELETE` table	(la table dans laquelle on supprime)
`FROM` tables	(les tables utilisées dans la clause `WHERE`)
`WHERE` conditions	(les lignes à supprimer)

Exemple : supprimer les petits clients

```
1 BEGIN TRAN
2   DELETE clients
3   FROM clients
4   WHERE clt_ca < 1000
5 COMMIT TRAN
```

Autre exemple : supprimer tous les clients (vider la table, et non pas, supprimer la table)

```
1 BEGIN TRAN
2   DELETE clients
3 COMMIT TRAN
```

Remarques :
– il est très dangereux d'oublier la clause `WHERE` dans un `DELETE` ;
– à cause de la clé étrangère dans la table `commandes`, on ne peut pas supprimer les clients qui ont des commandes.

5.3 Mise-à-jour

Encore une fois, on ne peut changer les lignes que d'une table à la fois. La syntaxe pour effectuer une requête de mise-à-jour est :

`UPDATE` table	(la table dans laquelle met à jour)
`SET` colonne1 `=` ..., colonne2 `=` ...	(les colonnes que l'on met à jour)
`FROM` tables	(les tables de la clause `WHERE`)
`WHERE` conditions	(les lignes à mettre à jour)

Exemple : pour convertir tous les prix en euros

```
1 BEGIN TRAN
2   UPDATE articles
3   SET art_prix = art_prix / 6.55957
4 COMMIT TRAN
```

Remarques :
– on ne peut pas mettre à jour une colonne `IDENTITY` ;
– comme une division est moins coûteuse qu'une multiplication, il est préférable d'inverser une bonne fois pour toute le taux de conversion et de ne plus effectuer que des multiplications :

```
1 DECLARE @taux REAL
2 SET @taux = 1.0 / 6.55957
3 BEGIN TRAN
4   UPDATE articles
5   SET art_prix = art_prix * taux
6 COMMIT TRAN
```

– il faut se méfier des mises-à-jour corrélées, puisque la requête suivante ne fait pas se que l'on pense :

```
BEGIN TRAN
  UPDATE articles
  SET art_prix = art_prix * taux, art_prixTTC = art_prix * 1.196
  /* malheureusement le art_prix utilise pour art_prixTTC
      n'est pas celui qui vient d'etre mis-a-jour */
COMMIT TRAN
```

il faut la remplacer par :

```
BEGIN TRAN
  UPDATE articles
  SET art_prix = art_prix * taux

  UPDATE articles
  SET art_prixTTC = art_prix * 1.196
COMMIT TRAN
```

6 Contraintes

Les contraintes permettent de sécuriser les données d'une table. On en connaît déjà : les clés primaire et étrangère, les valeurs par défaut. L'objet de cette section est d'apprendre à créer ces contraintes.

6.1 Syntaxe

Pour définir une contrainte sur une colonne d'une table, on dispose de deux syntaxe :
– au moment de la création de la table

Exemple : positionnons-nous dans le cas d'une mutuelle

```
CREATE TABLE assures
(
  num INT PRIMARY KEY IDENTITY(1,1),
  numSS CHAR(15),
  titre VARCHAR(5),
  age INT,
  date_entree DATETIME,
  num_rue INT,
  rue VARCHAR(255),          -- 256 est un multiple de 8
  code_postal CHAR(5),
  ville VARCHAR(63)
  CONSTRAINT cst_num_rue     -- nom de la contrainte
    CHECK(num_rue > 0)       -- corps de la contrainte
  CONSTRAINT cst_code_postal
    CHECK(code_postal LIKE ('[0-9][0-9][0-9][0-9][0-9]'))
)
```

– après la création de la table :

```
1 ALTER TABLE assures
2   ADD CONSTRAINT cst_numSS
3   CHECK (numSS LIKE ('[0-2][0-9]...'))
```

Remarques :
– pour pouvoir ajouter une contrainte, les données existantes doivent vérifier cette contrainte ;
– sur insertion ou mise-à-jour, les nouvelles données sont contrôlées (si une donnée ne vérifie pas une contrainte alors toute la transaction est annulée) ;
– on peut manipuler plusieurs contraintes dans un seul `ALTER TABLE`, il suffit de les séparer par des virgules ;
– on peut imposer plusieurs contraintes sur une même colonne.

Pour retirer une contrainte :

```
1 ALTER TABLE assures
2   DROP CONSTRAINT cst_code_postal
```

Pour modifier une contrainte, il faut d'abord la supprimer puis la créer de nouveau.

6.2 CHECK

6.2.1 Syntaxe

La syntaxe d'une contrainte de type vérification est : `CHECK(`clause `WHERE` sans le `WHERE)`.

Exemples : on peut donc
– mettre plusieurs conditions

```
1 ALTER TABLE assures
2   ADD CONSTRAINT cst_age
3   CHECK(age >= 0 AND age < 150)
```

– préciser une liste de choix desquels on ne peut pas sortir

```
1 ALTER TABLE assures
2   ADD CONSTRAINT cst_titre
3   CHECK(titre IN ('M.', 'Mme', 'Melle', 'Dr.', 'Pr.', 'SAS', 'Me'))
```

– utiliser plusieurs colonnes

```
1 ALTER TABLE articles
2   ADD CONSTRAINT cst_TTCsupHT
3   CHECK(art_prixTTC > art_prix)
```

Remarques : par contre
– la clause `CHECK` ne peut pas contenir de sous-requête ;
– la clause `CHECK` ne peut pas porter sur une colonne `UNIQUEIDENTIFIER` ou utilisant `IDENTITY` (cf. §3.3 page 17).

6.2.2 Règle

Si plusieurs colonnes (éventuellement dans des tables différentes) utilisent la même contrainte `CHECK`, alors il est intéressant de définir une règle commune à toutes ces colonnes.

Exemple :

```
1 CREATE RULE AgeRule
2 AS @age >= 0 AND @age < 150
```

Remarques :
– `@age` est une variable locale, son nom n'a pas d'importance ;
– après `AS` on peut mettre la même chose qu'après `CHECK`.

On peut ensuite attacher une règle à une colonne en utilisant la procédure stockée `sp_bindrule`. Exemple :

```
1 sp_bindrule AgeRule, [assures.age]
```

Remarques :
– une colonne peut cumuler une règle et une contrainte `CHECK` ;
– mais c'est la contrainte `CHECK` qui est vérifiée en premier ;
– on dispose de la procédure `sp_unbindrule [assures.age], AgeRule` et de `DROP RULE AgeRule`[8].

Il est également possible d'attacher une règle à un type de données, ce qui permet d'éviter de les attacher à toutes les colonnes de ce type.

Exemple :

```
1  sp_addtype CodePostalType, CHAR(5)
2
3  CREATE RULE CodePostalRule
4  AS @cp LIKE('[0-9][0-9][0-9][0-9][0-9]')
5
6  sp_bindrule CodePostalRule, CodePostalType
7
8  -- puis
9
10 CREATE TABLE assures
11 (
12   ...
13   code_postal CodePostalType,
14   ...
15 )
```

8. qui ne fonctionne que si tous les `sp_unbindrule` ont été effectués

6.3 Valeur par défaut

Pour préciser une valeur par défaut on peut le faire simplement à la création de la table (cf. §3.2 page 16), ou les ajouter *a posteriori* en tant que contrainte.

Exemple :

```
1 ALTER TABLE assures
2   ADD CONSTRAINT def_date_entree
3   DEFAULT GETDATE() FOR date_entree
```

On peut mettre après `DEFAULT` :
– une fonction niladique (*i.e.* sans argument) ;
– une constante ;
– ou `NULL`.

On peut aussi créer des valeurs par défaut partageables et l'attacher à une colonne ou à un type de données. Exemple :

```
1  CREATE DEFAULT Hier
2  AS DATEADD(day, -1, GETDATE())
3
4  sp_bindefault Hier, [assures.date_entree]
5
6  -- ou
7
8  sp_addtype DateEntree, DATETIME
9
10 sp_bindefault Hier, DateEntree
11
12 -- puis
13
14 ALTER TABLE assures
15   ALTER COLUMN date_entree DateEntree
```

Remarques :
– si la contrainte `DEFAULT` est définie, alors une éventuelle valeur par défaut partageable serait ignorée ;
– on dispose de la procédure `sp_unbindefault` et de `DROP DEFAULT`[9].

Astuce : pour utiliser les valeurs par défaut, les règles et les type de données personnels dans plusieurs bases de données, il suffit de les créer dans la base `model` car alors toute nouvelle base de données en héritera.

9. qui ne fonctionne que si tous les `sp_unbindefault` ont été effectués

6.4 Clé primaire

Les clés primaires sont aussi des contraintes et pour les ajouter *a posteriori* on peut utiliser la syntaxe :

```
CONSTRAINT nom de la contrainte
 PRIMARY KEY (colonne(s) concernée(s) par la clé primaire)
```

Cette syntaxe est la indispensable pour déclarer une clé primaire composite (c'est-à-dire portant sur plusieurs colonnes[10]).

Exemple : dans une base de donnée bibliothèque, un exemplaire d'un livre est identifié par son numéro ISBN et son numéro de copie.

```
1 ALTER TABLE ouvrages
2   ADD CONSTRAINT pk_ouvrages
3   PRIMARY KEY (isbn, no_copie)
```

6.5 UNIQUE

On peut imposer à une colonne (ou plusieurs colonnes) de prendre des valeurs uniques (c'est-à-dire sans doublons) même si ce n'est pas une clé primaire.

Exemples :

```
1 ALTER TABLE assures
2   ADD CONSTRAINT un_numSS
3   UNIQUE (numSS)
```

```
1 ALTER TABLE clients
2   ADD CONSTRAINT un_nom_prenom
3   UNIQUE (clt_nom, clt_prenom)
```

Remarque : la valeur `NULL` n'est autorisée qu'une seule fois dans une colonne `UNIQUE`.

6.6 Clé étrangère

Les clés étrangères sont aussi des contraintes, et à nouveau, si on a oublié de les préciser dès la création de la table, on peut les ajouter après. Attention : on ne peut faire de clé étrangère que vers une clé primaire ou vers une colonne `UNIQUE`.

Exemple : avec la table `feuilles_soin` qui possède la colonne `num_assure` qui doit prendre ses valeurs dans la colonne `num` de la table `assures`

```
1 ALTER TABLE feuille_soin
2   ADD CONSTRAINT fk_num_assure
3   FOREIGN KEY (num_assure) REFERENCES Assures(num)
```

Cette syntaxe est nécessaire si la clé étrangère est composite.

10. il est conseillé d'éviter les clés primaires composites à chaque fois que cela est possible

Exemple : dans une base bibliothèque un emprunt concerne un exemplaire d'un livre, les numéros ISBN et de copie doivent donc être les mêmes.

```
1 ALTER TABLE emprunts
2   ADD CONSTRAINT fk_emprunts
3   FOREIGN KEY (isbn, no_copie) REFERENCES ouvrages(isbn, no_copie)
```

6.7 Conclusion

On vient de rencontrer quelques outils qui nous permette de rendre les données plus cohérentes :
- les colonnes n'acceptent qu'un ensemble de valeurs correctes, c'est l'intégrité de domaine (on spécifie pour ça le type de données, les contraintes `CHECK`, les valeurs par défaut et aussi `NOT NULL`) ;
- les lignes doivent être identifiables de manière unique, c'est l'intégrité des entités (on utilise pour ça les clés primaires et les contraintes `UNIQUE`) ;
- on doit maintenir de bonnes relations entre les tables, c'est l'intégrité référentielle (c'est tout le travail des clés étrangères).

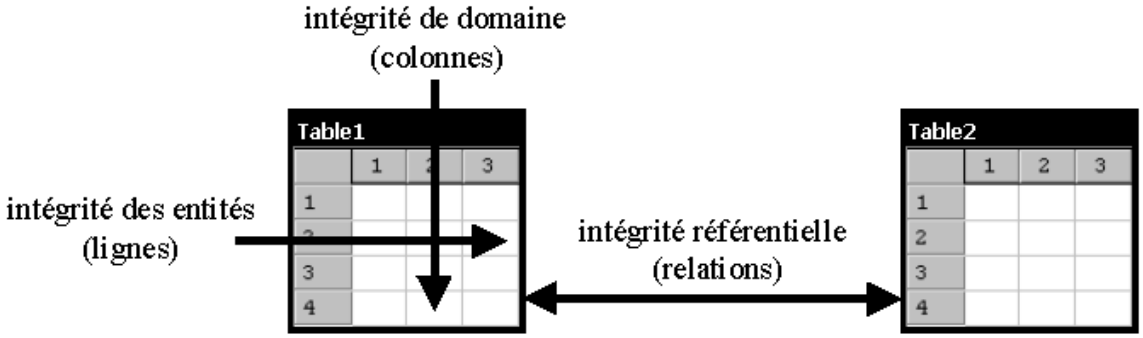

FIG. 6 – *Différents types d'intégrité*

Exemples d'intégrité référentielle :
- il est impossible de créer des factures qui ne sont reliées à aucun client ;
- et à l'inverse, il est impossible de supprimer un client à qui il reste des factures (impayées).

Il reste malgré tout un quatrième type d'intégrité qui regroupe toutes les règles (parfois complexes) propre à la politique interne de l'entreprise, c'est l'intégrité d'entreprise.

Exemples de règle spécifiques à une entreprise :
- un client ne peut pas commander lorsqu'il doit déjà trop d'argent ;
- un client qui commande régulièrement bénéficie de réductions.

Pour implémenter ce genre de règle, on a besoin d'une programmation plus élaborée que les contraintes. C'est l'objet de la section suivante.

7 Programmation événementielle

La première chose à savoir est que pour chaque table il existe en SQL trois événements (ni plus ni moins). Ils sont soulevés respectivement par les instructions `INSERT`, `DELETE` et `UPDATE` (cf. §5 page 28). L'objet de cette section est d'apprendre à les utiliser.

7.1 Mise-à-jour et suppression en cascade

Exemple : si on veut désormais que la suppression d'un client entraîne automatiquement celle de ses commandes,[11] il suffit pour cela de préciser une option lors de la définition de la contrainte clé étrangère dans la table `commandes`.

```
1 ALTER TABLE commandes
2   ADD CONSTRAINT fk_cmd_clt
3   FOREIGN KEY (cmd_clt) REFERENCES clients
4     ON DELETE CASCADE
5     ON UPDATE CASCADE
```

Remarques :
- de cette façon, la relation entre les deux tables devient non bloquante en suppression et en mise-à-jour ;
- il n'y a pas `ON INSERT CASCADE`.

Exercice : pourquoi n'y a-t-il pas d'insertion en cascade ?

7.2 Déclencheurs `AFTER`

Un déclencheur est une procédure attachée à un événement, en anglais on dit `TRIGGER`. Ces procédures se déclenchent automatiquement après que l'événement concerné a été soulevé (donc bien souvent à l'insu de l'utilisateur) et ne peuvent être appelées directement[12].

Exemple : la table `articles` contient une colonne qui précise le nombre d'articles en commande ; pour mettre à jour cette colonne lors d'insertion de nouvelles commandes on crée un déclencheur.

```
1 CREATE TRIGGER commandes_insert  -- le nom du declencheur
2   ON commandes AFTER INSERT      -- la table et l'evenement concernes
3 AS                               -- la programmation du declencheur
4   UPDATE articles SET nb_commande = nb_commande + cmd_qte
5   FROM articles AS a
6   JOIN inserted AS b ON (a.art_num = b.cmd_art)
7
8 -- (si plusieurs instructions : utiliser un bloc BEGIN ... END)
```

Quelques mots sur les tables `inserted` et `deleted` :
- il s'agit de tables temporaires créées et disponibles pendant l'événement ;
- leurs colonnes sont identiques à celles de la table sur laquelle l'événement a été levé ;
- le déclencheur `AFTER INSERT` peut utiliser la table `inserted` qui contient toutes les lignes insérées ;
- le déclencheur `AFTER DELETE` peut utiliser la table `deleted` qui contient toutes les lignes supprimées ;

11. ce n'est pas très conseillé
12. en conséquence de quoi la seule façon de les tester est de soulever l'événement par une requête appropriée

– le déclencheur `AFTER UPDATE` peut utiliser les deux tables (ce qui est logique puisqu'une mise-à-jour consiste en une insertion et une suppression).

Autre exemple avec cette fois-ci la table `deleted` :

```
1  CREATE TRIGGER commandes_delete
2    ON commandes AFTER DELETE
3  AS
4    UPDATE articles SET nb_commande = nb_commande - cmd_qte
5    FROM articles AS a
6    JOIN deleted AS b ON (a.art_num = b.cmd_art)
```

Troisième exemple, sur mise-à-jour cette fois-ci : pour être tout à fait complet, il faut également un déclencheur qui réagisse si la colonne `cmd_qte` est touchée par une mise-à-jour.

```
1  CREATE TRIGGER commandes_update
2    ON commandes AFTER UPDATE
3  AS
4    IF UPDATE(cmd_qte) -- si la colonne cmd_qte est touchee par la modification
5      BEGIN
6        UPDATE articles SET nb_commande = nb_commande - b.cmd_qte + c.cmd_qte
7        FROM articles AS a
8        JOIN deleted AS b ON (a.art_num = b.cmd_art)
9        JOIN inserted AS c ON (a.art_num = c.cmd_art)
10     END
```

Dernier exemple : on veut empêcher la modification du numéro ISBN d'un ouvrage.

```
1  CREATE TRIGGER ouvrages_update
2    ON ouvrages AFTER UPDATE
3  AS
4    IF UPDATE(isbn)
5      BEGIN
6        RAISERROR ('Le numero ISBN ne peut pas etre modifie', 0, 1)
7        -- 0 indique la gravite de l'erreur et 1 l'etat (a oublier)
8        ROLLBACK TRANSACTION
9        -- on annulle la transaction qui a declenche l'evenement
10     END
```

Remarques :
– les déclencheurs sont des transactions ;
– il faut que l'utilisateur qui tente d'insérer un emprunt, dispose des droits sur toutes les tables impliquées dans la programmation du déclencheur ;
– comme on vient de le voir, les déclencheurs sont notamment utiles pour :
  – implémenter des règles trop complexes pour les contraintes (ne serait que parce qu'une contrainte ne peut porter que sur une table) ;
  – afficher un message d'erreur personnalisé et annuler la transaction appelante.
– comme leur nom l'indique, un déclencheur `AFTER` se produisent <u>après</u> un événement ;
– du coup, les contraintes sont vérifiées avant le lancement des déclencheurs `AFTER`, ce qui a pour une conséquence fâcheuse : les mises-à-jour en cascade éventuellement soulevées par ces déclencheurs ne se font qu'après vérification des contraintes ;
– avec SQL Server il n'y a pas de déclencheurs `BEFORE` ;

– par contre les déclencheurs `INSTEAD OF` (au lieu de) existent ; c'est l'objet du paragraphe suivant.

Exercice : en quoi le cinquième point est-il fâcheux ?

7.3 Déclencheurs `INSTEAD OF`

On les utilise si on veut que leurs instructions se lancent à la place de l'insertion, de la suppression ou de la mise-à-jour qui a soulevé l'événement. Avec un déclencheur `AFTER` la modification des données a lieu puis le déclencheur est exécuté, tandis qu'avec un déclencheur `INSTEAD OF` le corps du déclencheur se substitue à la modification des données.

D'un point de vue syntaxique, il suffit de remplacer `AFTER` par `INSTEAD OF`. Exemple : on historise automatiquement les commandes insérées dans une table `historique_commmandes`.

```
1  CREATE TRIGGER commandes_insert2
2    ON commandes INSTEAD OF INSERT
3  AS
4    BEGIN
5      INSERT commandes SELECT * FROM inserted
6      -- cette ligne fais l'insertion prevue
7      INSERT historique_commmandes SELECT * FROM inserted
8    END
9
10 -- on aurait donc pu se contenter d'un declencher AFTER
11 -- avec seulement le 2e INSERT
```

Remarques :
– les tables provisoires `inserted` et `deleted` existent et sont remplies pour les déclencheurs `INSTEAD OF` (heureusement) ;
– les déclencheurs `INSTEAD OF` ne se déclenchent pas eux-mêmes (heureusement) ;
– il ne peut y avoir qu'un déclencheur `INSTEAD OF` par événement et par table (alors qu'il peut y avoir plusieurs déclencheurs `AFTER`) ;
– s'il existe une clé étrangère avec une action en cascade (`DELETE` ou `UPDATE`) dans la table, alors on ne peut pas écrire le déclencheur `INSTEAD OF` correspondant, et inversement.

Exercice : pourquoi ces trois dernières règles existent-elles ?

7.4 Compléments

Toutes les instructions SQL ne sont pas autorisées dans le code d'un déclencheur ; on se limitera généralement à : `INSERT`, `DELETE`, `UPDATE`, `RAISERROR` et `ROLLBACK TRANSACTION`.

Pour modifier un déclencheur on a :

```
1  ALTER TRIGGER commandes_insert
2   ... -- son nouveau code
```

Pour supprimer un déclencheur on a :

```
1  DROP TRIGGER commandes_insert
```

Pour suspendre provisoirement un déclencheur (sans le supprimer) on a :

```
1 ALTER TABLE commandes
2   DISABLE TRIGGER commandes_insert
3
4 ... -- d'autres instruction puis
5
6 ALTER TABLE commandes ENABLE TRIGGER commandes_insert
```

Remarque : on peut remplacer `commandes_insert` par `ALL` ou `commandes_insert, commandes_insert2`.

On peut créer un déclencheur pour deux ou trois événements à la fois. Exemple :

```
1 CREATE TRIGGER ...
2   ON ... AFTER INSERT, UPDATE
3 AS
4   ...
```

7.5 Conclusion

Faisons une synthèse sur le déroulement d'une transaction. Pour chaque instruction de la transaction on a :

vérification des autorisations de l'utilisateur (*)
puis
transfert des données nécessaires du disque dans la mémoire
puis
remplissage des tables `inserted` et/ou `deleted`
puis
modifications (prévues ou `INSTEAD OF` et/ou en cascade) des données dans la mémoire (*)
puis
vérification des contraintes (*)
puis
déclencheurs `AFTER` (*)

(*) signifie qu'à ce stade la transaction peut-être annulée.

L'écriture des données sur le disque n'intervient qu'à la fin de la transaction lorsque toutes ses instructions ont été validées.

8 Vues

Une vue est une requête `SELECT` à laquelle on donne un nom et dont on peut se servir comme s'il s'agissait d'une table. Ça n'est pas si surprenant puisque l'on peut voir une requête `SELECT` comme une fonction (au sens informatique du terme) qui retourne une table. Contrairement à ce que l'on pourrait croire, les vues ne conservent pas une copie séparée des données.

8.1 Syntaxe

Exemple de déclaration d'une vue : on désire ne garder qu'une sous table de la table `commandes` tout en affichant le nom du client et de l'article au lieu de leur numéro.

```
1 CREATE VIEW VueCommandes                  -- nom de la vue
2   ([Nom du client], [Article commandé]) -- nom des colonnes (plus parlants)
3 AS
4   SELECT b.clt_nom, c.art_nom
5   FROM commandes AS a
6   JOIN clients AS b ON a.cmd_clt = b.clt_num
7   JOIN articles AS c ON a.cmd_art = c.art_num
```

Puis on peut l'utiliser comme une table :

```
1 SELECT [Nom du client]
2 FROM VueCommandes
3 WHERE [Article commandé] = 'pinceau'
```

Remarques sur la création des vues :
- la requête `SELECT` de la vue ne doit ni contenir de clause `ORDER BY` ni faire référence à un table temporaire (cf. §5.1 page 29) ni utiliser de sous-requête ;
- il est conseillé de tester au préalable la requête `SELECT` seule ;
- on peut créer une vue à partir d'autres vues, mais pour des questions de performances il vaut mieux éviter et en revenir aux tables sous-jacentes.

Pour modifier une vue :

```
1 ALTER VIEW VueCommandes
2 ( ... )  -- les colonnes
3 AS
4   ...    -- nouvelle requete SELECT
```

Pour supprimer une vue :

```
1 DROP VIEW VueCommandes
```

8.2 Intérêts

Désormais les utilisateurs n'accéderont aux données qu'au travers des vues, seuls les développeurs manipuleront directement les tables. C'est particulièrement avantageux car :
- on peut traduire les intitulés des colonnes en différentes langues et de manière plus explicite que la nomenclature adoptée pour la base ;
- cela simplifie les requêtes que les développeurs vont écrire pour les utilisateurs (le travail de jointure est déjà fait dans la vue, les noms sont plus parlants et les colonnes utiles uniquement aux

développeurs (`clt_num` et `art_num` par exemple) sont cachées) ;
- cela simplifie la sécurisation des données (les données sensibles – responsables de l'intégrité de la base – sont masquées et il suffira de gérer les autorisations d'accès aux vues et non pas aux tables) ;
- et surtout on peut changer la structure de la base (les tables) sans avoir à modifier la programmation pour les utilisateurs (on changera éventuellement la programmation des vues mais pas celle des requêtes qui utilisent ces vues).

Illustration du dernier point : admettons que la table `commandes` soit scindée en deux tables `commandes2001` et `commandes2002`. Seules les requêtes qui utilisent la table `commandes` doivent être re-programmées.

```
1  ALTER VIEW VueCommandes
2    ([Nom du client], [Article commandé])
3  AS
4    SELECT b.clt_nom, c.art_nom
5      FROM commandes2001 AS a
6      JOIN clients AS b ON a.cmd_clt = b.clt_num
7      JOIN article AS c ON a.cmd_art = c.art_num
8    UNION
9    SELECT b.clt_nom, c.art_nom
10     FROM commandes2002 AS a
11     JOIN clients AS b ON a.cmd_clt = b.clt_num
12     JOIN article AS c ON a.cmd_art = c.art_num
```

Toutes les requêtes qui utilisent les vues restent inchangées.

```
1  SELECT [Nom du client]
2  FROM VueCommandes
3  WHERE [Articles commandé] = 'pinceau'
```

Lorsqu'une base de données est déployée à l'échelle d'une entreprise, le mécanisme des vues offre une interface entre l'implémentation (les tables) et les utilisateurs qui permet au code SQL une plus grande facilité de maintenance

8.3 Modification de données

Comme on vient de voir, la consultation des données à travers une vue ne pose pas de problème. Le problème essentiel avec les vues est la grande difficulté de modifier les données. En effet, plusieurs cas pathologiques peuvent en effet se présenter :

- il se peut qu'une colonne déclarée `NOT NULL` ne soit pas visible à travers la vue

  exemple : comment ajouter une commande avec la vue `VueCommandes` alors que :
  - la colonne `cmd_num` est clé primaire donc `NOT NULL`
  - les colonnes `cmd_clt` et `cmd_art` sont clés étrangères et `NOT NULL`

  et ne figurent pas dans la vue ?

- et comment ajouter des données dans une vue mutli-tables ?

  exemple : on voudrait par exemple ajouter automatiquement un nouveau client à sa première commande.

Malheureusement, la requête suivante n'est pas autorisée :

```
1  BEGIN TRAN
2    INSERT VueCommandes
3    VALUES('Fricotin', 'Stylo')
4  COMMIT TRAN
```

La solution consiste à employer un déclencheur `INSTEAD OF`. Exemple :

```
1  CREATE TRIGGER VueCommandes_insert
2    ON VueCommandes INSTEAD OF INSERT
3  AS
4    BEGIN
5      -- j'insere d'abord les nouveaux clients dans la table clients
6      INSERT clients(clt_nom)
7        SELECT [Nom du client]
8        FROM inserted
9        WHERE [Nom du client] NOT IN (SELECT clt_nom FROM clients)
10
11     -- j'insere ensuite les commandes elles-memes
12     -- avec tous les renseignements necessaires
13     INSERT commandes(cmd_date, cmd_clt, cmd_art)
14       SELECT GETDATE(), b.clt_num, c.art_num
15       FROM inserted AS a
16       JOIN clients AS b ON a.[Nom du client] = b.clt_nom
17       JOIN articles AS c ON a.[Article commandé] = c.art_nom
18   END
```

Avec ce déclencheur, la requête d'insertion précédente fonctionne.

Exercice : pourquoi n'a-t-on pas eu besoin de préciser ni `clt_num` dans le premier `INSERT` ni `cmd_num` dans le deuxième ?

Remarques :
- `GETDATE()` renvoie la date d'aujourd'hui ;
- on a fortement supposé dans ce déclencheur que les clients portaient un nom unique et que les articles aussi, c'est pourquoi il vaut mieux respecter les conseils suivant lors de la création d'une vue :
  - s'arranger pour ne jamais avoir de doublons dans la vue (ça peut vouloir dire par exemple ajouter une contrainte `UNIQUE` à la colonne `clt_nom` dans la table `client` ou inclure la clé primaire) ;
  - toutes les colonnes `NOT NULL` que l'on écarte doivent pouvoir recevoir une valeur calculée (c'est le cas de `cmd_date`, `cmd_clt` et `cmd_art`) ou une valeur par défaut (c'est le cas de `cmd_num`)[13] ;
- le seul déclencheur disponible pour les vues est `INSTEAD OF` (et pas `AFTER` contrairement aux tables) ;
- quand on insère dans une vue avec SQL Server, il faut malheureusement remplir toutes les colonnes et on ne peut pas faire appel à la valeur `NULL`.

13. bref, c'est difficile de cacher les clés primaires, les clés étrangères et plus généralement toutes les colonnes `NOT NULL` car une vue dénormalise les données, ce qui représente un danger

Illustration de ce dernier point : on modifie la précédente vue, en lui ajoutant deux colonnes

```
1 ALTER VIEW VueCommandes
2   ([Numéro de commande], [Nom du client], [Article commandé], Date)
3 AS
4   SELECT a.cmd_num, b.clt_nom, c.art_nom, a.cmd_date
5   FROM commandes AS a
6   JOIN clients AS b ON a.cmd_clt = b.clt_num
7   JOIN articles AS c ON a.cmd_art = c.art_num
```

on veut insérer dans cette vue (en utilisant le même déclencheur) mais en laissant SQL Server calculer le numéro de commande et la date de commande :

```
1 BEGIN TRAN
2   INSERT VueCommandes
3   VALUES('', 'Fricotin', 'Stylo', '')
4   -- on est oblige d'employer des valeurs bidons
5 COMMIT TRAN
```

9 Procédures stockées

En pratique, les programmes qui utilisent les données d'une base ne font pas directement appel aux transactions, mais plutôt à des procédures auxquelles ils peuvent passer des arguments.

9.1 Syntaxe

Le langage Transact-SQL permet de programmer ces procédures selon la syntaxe suivante :

`CREATE PROC ...`	le nom de la procédure
`(...)`	les paramètres d'entrée et de sortie séparés par des virgules
`AS`	
`DECLARE ...`	les variables locales
`BEGIN`	
`...`	les instructions, les transactions
`END`	

Remarques :
– on peut utiliser jusqu'à 1024 paramètres ;
– la syntaxe d'une procédure stockée est limitée à 128 Mo.

Exemple : une requête paramétrée

```
1 CREATE PROC InfoDuClient
2   (@numero INT)            -- ne pas oublier de preciser le type
3 AS
4   SELECT *
5   FROM clients
6   WHERE clt_num = @numero
```

Autre exemple avec un paramètre de sortie :

```
CREATE PROC NbClients
  (@resultat INT OUTPUT)
AS
  SET @resultat = (SELECT COUNT(*) FROM clients)
  -- il s'agit-la d'une sous-requete
```

Dernier exemple avec un paramètre d'entrée muni d'une valeur par défaut :

```
CREATE PROC FiltrerClients
  (@filtre VARCHAR(255) = '%')
AS
  SELECT *
  FROM clients
  WHERE clt_nom LIKE @filtre
  -- en l'absence de parametre tous les clients seront affiches
```

Pour modifier une procédure stockée :

```
ALTER PROC InfoDuClient
  (...)  -- les parametres
AS
  ...    -- nouveau corps
```

Pour supprimer une procédure stockée :

```
DROP PROCEDURE InfoDuClient
```

9.2 Utilisation

On peut ensuite utiliser ces procédures stockées dans du code SQL avec l'instruction `EXEC`.

Exemple : pour avoir les informations relatives au client `12`

```
EXEC InfoDuClient 12
-- 12 est la valeur du paramètre
```

Remarques :
- on peut aussi utiliser des variables comme valeurs de paramètre (et pas seulement des constantes comme dans l'exemple) ;
- si la procédure a besoin d'une liste de paramètres, il faut les séparer par des virgules ;
- s'il y a un paramètre de sortie, il faut en stocker la valeur de retour dans une variable.

Exemple :

```
DECLARE @NombreTotalDeClients INT
EXEC NbClients @NombreTotalDeClients OUTPUT

-- et apres, on peut utiliser le contenu de la variable @NombreTotalDeClients
```

9.3 Cryptage

Lorsque de la création ou de la modification d'un déclencheur, une vue, une fonction ou une procédure stockée (bref, tout ce qui contient le code SQL destiné aux utilisateurs), on peut préciser la clause `WITH ENCRYPTION` qui permet de crypter le code de ces objets. Cela permet de protéger la propriété intellectuelle des développeurs sous SQL Server.

Exemples :

```
1 CREATE VIEW VueCommandes(Client, Article)
2   WITH ENCRYPTION
3 AS
4   ...
```

```
1 ALTER PROC InfoDuClient
2   (@numero INT)
3   WITH ENCRYPTION
4 AS
5   ...
```

10 Verrous

Comme les transactions sont traitées en ligne sur un serveur multi-utilisateur, les accès concurrentiels aux données doivent être gérés. Pour empêcher les autres utilisateurs de modifier ou de lire des données faisant l'objet d'une transaction qui n'est pas encore terminée, il faut verrouiller ces données.

Rappelons que lors d'une transaction :
– les données nécessaires sont lues sur le disque puis chargées en mémoire centrale ;
– les opérations ont lieu dans la mémoire centrale ;

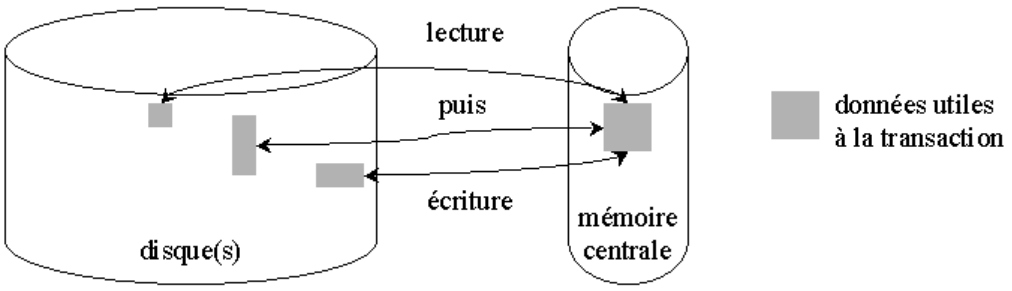

FIG. 7 – *Traitement des données d'une transaction en mémoire*

– une fois toutes les instructions validées, les nouvelles données sont écrites sur le disque.

Si les données sur le disque sont modifiées pendant la transaction, celle-ci travaille avec des données fausses. On a alors un problème de cohérence.

10.1 Isolation

Par défaut, SQL Server ne garantit pas que les données utilisées seront les mêmes pendant toute la transaction. Pour l'obliger à rendre maximal le verrouillage des données il faut lui imposer de mettre en série les transactions concurrentes. Pour cela on dipose de l'instruction :

```
1 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

Le problème majeur de la mise en série des transactions est qu'une transaction interminable bloque toutes les suivantes. Il est possible de préciser un délai d'attente maximal pour cause de verrouillage (par défaut il n'y en a pas). Pour cela on utilise l'instruction :

```
1 SET LOCK_TIMEOUT 180000
2 -- definit un delai d'expiration de 3 minutes (en millisecondes)
3 -- au dela de ce delai, la transaction en attente est annulee
```

Remarques :
– ces instructions sont attachées à la connexion qui les exécute ;
– ces instructions restent valables pour toutes les transactions qui suivent, jusqu'à la déconnexion ou jusqu'à nouvel ordre.

Le niveau d'isolation par défaut est `READ COMMITTED`, il garantit seulement que les données sont cohérentes au moment de leur lecture (et pas pendant le reste de la transaction). Pour y revenir il suffit d'écrire :

```
1 SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

10.2 Verrouillage de niveau table

Dans ce paragraphe on suppose que l'on se trouve au niveau d'isolation `READ COMMITTED`.

À chaque transaction on peut indiquer le type de verrouillage pour chaque table utilisée par les instructions `SELECT`, `INSERT`, `DELETE` et `UPDATE`. Par défaut, SQL Server verrouille les tables concernées.

On peut obliger SQL Server à laisser le verrou jusqu'à la fin de la transaction :

```
1 BEGIN TRAN
2   UPDATE clients WITH(HOLDLOCK)
3   SET ...
4 COMMIT TRAN
```

On peut se contenter de verrouiller seulement les lignes concernées par la transaction :

```
1 BEGIN TRAN
2   UPDATE clients WITH(ROWLOCK)
3   SET ...
4 COMMIT TRAN
```

Lorsqu'une première requête utilise `WITH(ROWLOCK)`, on peut indiquer à une deuxième d'ignorer les lignes verrouillées (afin de ne pas bloquer la transaction) :

```
1 SELECT AVG(clt_ca)
2 FROM clients WITH(READPAST)
```

10.3 Conclusion

On veillera à respecter les consignes suivantes pour les transactions :
– elles doivent être aussi courtes que possible afin d'éviter les files d'attente trop longues ;
– il ne faut pas les imbriquer (même si c'est possible, ça ne sert à rien) ;
– ne surtout pas interagir avec l'utilisateur pendant la transaction (mais avant).

Il est souvent bon de suivre ces quelques conseils concernant les verrous :
– mettre en série les transactions de toutes les connexions utilisateurs ;
– laisser SQL Server gérer la granularité des verrous : le laisser décider s'il faut verrouiller les lignes d'une table ou la table entière, c'est-à-dire n'utiliser ni `WITH(ROWLOCK)` ni `WITH(PAGLOCK)` ni `WITH(TABLOCK)` (dont on a pas parlé ici d'ailleurs).

11 Connexions

On a déjà plusieurs fois mentionné la nécessité d'attribuer les bons droits aux utilisateurs de notre base de données (cf. §5 page 28). L'objectif de cette section est d'apprendre à gérer ces utilisateurs, leurs droits et de protéger les développeurs.

11.1 Création

IL existe deux façon d'ajouter un nouveau compte de connexion :

– on peut le créer de toute pièce

```
1 sp_addlogin
2   'Paul',         -- le login
3   'luaP',         -- le mot de passe
4   'Northwind'     -- la base par defaut
```

– ou bien hériter d'une connexion Windows

```
1 sp_grantlogin 'STID/Henri'
2 -- STID etant le nom du domaine
3
4 sp_defaultdb 'STID/Henri', 'Northwind'
```

Il reste ensuite à lui donner accès au serveur :

```
1 sp_grantdbaccess 'Paul'
```

On dispose évidemment des procédures :

```
1 sp_revokedbaccess 'Paul'
2
3 sp_droplogin 'Paul'
```

11.2 Rôle

Il est possible (et conseillé) de regrouper les utilisateurs selon les autorisations qu'ils ont, c'est-à-dire de définir des rôles.

11.2.1 Sur le serveur

Il existe 8 rôles sur serveur dans SQL Server dont :

nom du rôle	droits de ses membres
`sysadmin`	tous les droits sur le système et toutes les base
`securityadmin`	gestion des accès à SQL Server
`dbcreator`	création de bases de données

Pour ajouter et radier un utilisateur à l'un de ces rôles :

```
1 sp_addsrvrolemember 'Paul', 'dbcreator'
2
3 sp_dropsrvrolemember 'Paul', 'dbcreator'
```

Un même utilisateur peut cumuler plusieurs rôles.

11.2.2 Dans une base de données

Dans chaque base on dispose des rôles suivants :

nom du rôle	droits de ses membres
`db_owner`	tous les droits sur les objets de la base
`db_accessadmin`	ajout d'utilisateurs et de rôles
`db_datareader`	lire le contenu des tables
`db_datawriter`	insertion, suppression et modification sur toutes les tables
`db_ddladmin`	création, modification, suppression d'objet
`db_securityadmin`	gestion des rôles et des autorisations
`db_public`	à définir

Tous les utilisateurs appartiennent au rôle `public` et peuvent appartenir à d'autres rôles.

Pour ajouter un rôle et un utilisateur à ce rôle :

```
1 sp_addrole 'ServiceInformatique'
2
3 sp_addrolemember 'ServiceInformatique', 'Henri'
```

On a aussi :

```
1 sp_droprolemember 'ServiceMarketing', 'Paul'
2
3 sp_droprole 'ServiceMarketing'
4 -- possible uniquement s'il ne reste plus aucun membre dans ce role
```

11.3 Droits

Dans ce paragraphe, on se place dans une base.

11.3.1 Sur les instructions

Exemple : pour autoriser les utilisateurs `Paul` et `Henri` à créer des tables et des déclencheurs

```
1 GRANT CREATE TABLE, CREATE TRIGGER
2   TO Paul, Henri
```

Remarque : `Paul` et `Henri` doivent déjà posséder un compte utilisateur sur SQL Server.

Autre exemple : pour empêcher `Paul` de créer des vues

```
1 DENY CREATE VIEW
2   TO Paul
```

Dernier exemple : pour lever les autorisations et les empêchements de `Paul`

```
1 REVOKE CREATE VIEW, CREATE TABLE
2   TO Paul
```

Remarques :
– `REVOKE` annule le dernier `GRANT` ou `DENY` correspondant ;
– après un `REVOKE`, SQL Server s'en remet aux autorisations par défaut du rôle dont `Paul` est membre ;
– on peut utiliser le mot-clé `ALL` pour désigner toutes les instructions.

11.3.2 Sur les objets

Dans une base de données, pour chaque table, chaque colonne et chaque instruction on peut préciser les autorisations.

Exemple : pour autoriser la sélection sur la table `clients`

```
1 GRANT SELECT ON clients
2   TO Paul
```

Autre exemple : pour empêcher les autres instructions

```
1 DENY INSERT, UPDATE, DELETE ON clients
2   TO Paul
```

Dernier exemple : pour autoriser la modification mais seulement du nom de client

```
1 GRANT UPDATE(clt_nom) ON clients
2   TO Paul
```

Remarques :
– en général on a pas besoin de descendre jusqu'à la colonne, il est préférable de créer une vue et de donner les droits sur cette vue ;

– (important) il vaut mieux utiliser `ALTER ...` car les autorisations sur l'objet concerné sont conservées, contrairement à `DROP ...` suivi de `CREATE ...`

11.3.3 Chaîne d'autorisation

Sont habilités à délivrer des autorisations `GRANT`, `DENY` et `REVOKE` :
– les membres du rôle `sysadmin` dans toutes les bases ;
– les membres du rôle `db_owner` sur les instructions et le objets de leur base ;
– les propriétaires[14] d'objet(s) sur leur(s) objet(s).

Pour habiliter un utilisateur à délivrer les autorisations dont il bénéficie, il suffit d'ajouter la clause `WITH GRANT OPTION`. Exemple :

```
1 -- avec
2 GRANT SELECT ON clients
3   TO Paul
4   WITH GRANT OPTION
5
6 -- Paul peut desormais ecrire
7 GRANT SELECT ON clients
8   TO Henri
```

Conseil : pour éviter tout problème de rupture de chaîne d'autorisation avec les vues, il faut que le `dbo`[15] soit propriétaire des vues.

12 Formulaires

Un formulaire (informatique) est un outil de saisie d'informations (en petites quantités) à l'ordinateur. Il s'agit d'une interface entre la base de données et les utilisateurs non spécialistes et/ou n'ayant pas d'accès direct à la base.

12.1 Historique

L'équivalent papier d'un formulaire est, par exemple : une fiche de renseignements ou une facture. Pendant longtemps, les formulaires informatiques étaient en mode texte (c'est-à-dire occupant l'écran d'un terminal n'affichant que des caractères). Il en reste encore beaucoup (en partie à cause de leur robustesse).

Aujourd'hui, les formulaires sont programmés en langage objet (Java ou Visual Basic par exemple), ce qui leur permet d'offrir une interface graphique (composée de boutons, fenêtres et contrôles). Les formulaires graphiques reprennent l'ergonomie des formulaires textes, mais offrent une plus grande liberté dans la disposition des champs à renseigner.

14. le propriétaire est celui qui a créé l'objet
15. le propriétaire de la base

12.2 Lien avec la base de données

Un formulaire qui permet de modifier les données d'une base est composé de :
- contrôles indépendants (essentiellement, les étiquettes et les éléments de décoration) ;
- contrôles dépendants (c'est-à-dire liés aux colonnes de la base) ;
- contrôles calculés (un montant total par exemple) qui informent l'utilisateur et qui résultent d'opérations entre plusieurs données.

Ce sont les contrôles dépendants qui nous intéressent ici. Si le système transactionnel est bien conçu, les contrôles dépendants n'accèdent aux données qu'à travers des vues et le formulaire ne peut modifier les données de la base qu'en invoquant des procédures stockées.

Le lien très fort qui uni un formulaire à sa base de données, pose de gros problèmes de maintenance (en cas de changement de SGBD, de schéma relationnel ou de type de données par exemple) et de réutilisabilité. Les vues et les modèles orientés objets traduit du même modèle conceptuel que le schéma relationnel sont des solutions. Mais, il faut souvent faire appel à une couche supérieure pour faire le lien entre ces composants (c'est le rôle des environnements de développement comme .NET et J2EE (Struts notamment)).

12.3 Validation des données

Les formulaires doivent, au moins, obéir aux mêmes règles d'intégrité que la base de données sous-jacente (afin, notamment, de gérer les erreurs de validation au niveau du formulaire). Le concepteur d'un formulaire ne doit donc pas se contenter d'une *zone de texte* (qui permet la saisie au clavier d'une chaîne de caractères, d'un entier, d'un réel ou d'une date et heure) pour chaque champ à remplir.

12.3.1 Intégrité de domaine

Le type de données d'un contrôle doit correspondre au type de données de la colonne sous-jacente.

Pour les données numériques (entier ou réel), le contrôle *potentiomètre* et *toupie* permettent leur saisie à la souris (toutefois, il est bon de les accompagner d'une zone de texte dans laquelle on peut saisir la valeur souhaitée directement au clavier). Pour les données binaires (oui ou non), le contrôle *case à cocher* est tout indiqué.

Pour les dates, les outils de création de formulaire offre souvent un contrôle *calendrier*. En tout cas, le résultat pour une date ou pour une heure est une zone de texte avec un format de contrôle strict.

Les contrôles offrent bien souvent des formats de contrôle et des règles de validation qui permettent de reproduire certaines contraintes `CHECK`. De plus, pour les contraintes `CHECK IN`, on dispose du contrôle *bouton radio* (ou *groupe d'options*) qui est utilisable lorsque la liste des choix disponibles est connue et restera inchangée (ce qui est rare ; en général, les choix sont listés dans une colonne d'une autre table).

Un contrôle peut également recevoir une valeur par défaut (cohérente avec la base) et la notion de champ obligatoire permet de traduire le caractère vide autorisé ou non.

12.3.2 Intégrité des entités

Généralement, il n'est pas nécessaire de s'en soucier dans les formulaires, car elle est assurée par la numérotation automatique (et l'intégrité référentielle lorsque la clé primaire est composée de clé(s) étrangère(s)). La seule opération à mener dans le formulaire est de rendre le champ correspondant à la clé, insaisissable afin que l'utilisateur ne puisse le modifier.

Cependant, les contraintes `UNIQUE` sont plus difficiles à traduire dans les formulaires.

12.3.3 Intégrité référentielle

Le contrôle *liste déroulante* permet de choisir une valeur parmi une liste de valeurs contenues dans une autre colonne, ce qui convient parfaitement pour remplir une clé étrangère (issue d'une association de type 1 : n, le client d'une facture par exemple). Il faut cependant spécifier au contrôle d'interdire à l'utilisateur de sortir de la liste proposée. Il est également conseillé de rendre les listes ergonomes, en faisant dérouler plusieurs colonnes si nécessaire (le numéro du client car on en a besoin, accompagné de son nom pour le retrouver facilement, par exemple) et en triant la colonne la plus pertinente (le nom du client, en l'occurrence).

Pour les associations de type n : m, on peut utiliser les *listes à choix multiples* (cela permet par exemple, pour un film, de sélectionner les acteurs qui y participent parmi une liste d'acteurs prédéfinie).

Avec la liste déroulante, il faut que la ligne référencée (le client, par exemple) par la clé étrangère (dans la commande en cours de saisie) existe déjà. Malheureusement, un formulaire est souvent chargé de saisir à la fois la ligne référencée et les lignes qui la référence (la commande et ses lignes de commandes, par exemple).

Pour cela, nous disposons du contrôle *sous-formulaire* qui permet de saisir les lignes de commandes une par une dans le même formulaire qui permet la saisie d'une commande. Le sous-formulaire doit être conçu sous la forme d'une ligne de contrôles sans intitulés dans la zone `Détail` (les intitulés sont placés en en-tête, tandis que les totaux sont placés en pied de sous-formulaire). Il apparaîtra alors sous la forme d'un tableau dans le formulaire et dans lequel il y aura autant de lignes que nécessaire.

Cette notion de sous-formulaire est très utile dans un formulaire dédié à la consultation des données. On peut par exemple, afficher synthétiquement toutes les commandes d'un client en particulier. Mais là, nous sortons du cadre des systèmes transactionnels et entrons dans le décisionnel (cf. le chapitre sur les états page 62).

12.3.4 Intégrité d'entreprise

Elle est normalement assurée par les déclencheurs dans la base de données. Cependant, il est parfois nécessaire de la reproduire dans un formulaire (afin de tenir compte de ces règles et de gérer leurs violations au niveau du formulaire).

Pour cela, nous disposons :
- des règles de validation au niveau des contrôles (qui permettent d'implémenter des règles plus générales que les contraintes `CHECK`) ;
- de la programmation événementielle (qui permet, par exemple, de désactiver le sous-formulaire concernant les enfants, si le nombre d'enfants saisi est 0) ;
- des contrôles calculés (qui permettent d'afficher le montant d'une facture en tenant compte de la remise offerte aux bons clients, par exemple).

12.4 Ergonomie

Un formulaire ne doit pas dépasser la taille de l'écran et la taille d'une feuille s'il est destiné à être imprimé. À partir de là, on peut disposer les champs à renseigner dans un ordre logique, retirer les champs insaisissables des arrêts de tabulation et s'arranger pour que les tabulations entre les champs saisissables suivent la même logique.

Les contrôles doivent être suffisamment larges pour que l'utilisateur puisse en lire le contenu.
Il est également bon de trier les listes déroulantes et les listes à choix multiples et ne pas hésiter à lister plusieurs colonnes en même temps.

Les contrôles sont des objets qui réagissent à des *événements* dont les principaux sont :
- l'entrée dans le contrôle (par le clavier ou la souris) ;
- la sortie du contrôle ;
- ou le changement de valeur du contrôle.

À ces événements, on peut associer les *actions* suivantes :
- recalculer un autre contrôle (mettre à jour une liste déroulante par exemple) ;
- activer ou désactiver un autre contrôle ;
- ou encore, afficher un message d'erreur intelligible en cas de violation d'une règle de validation des données.

Les *boutons* sont des objets qui réagissent principalement à l'événement *clic* et qui permettent, par exemple, d'enregistrer la saisie (en appelant les procédures stockées), d'imprimer la fiche ou d'ouvrir un autre formulaire (pour saisir un nouveau client, pendant qu'on établit sa première facture, par exemple).

À ce propos, il ne faut pas exagérer la décomposition de la saisie d'une fiche en plusieurs formulaires simples (certains vont jusqu'à une question par formulaire), car le passage des informations d'un formulaire à un autre est ardu. Il vaut mieux activer les contrôles les uns après les autres (au fur et à mesure de la saisie).

Finalement, l'essentiel de la programmation en langage objet dans un formulaire, porte sur la gestion des événements et la récupération des données saisies dans les contrôles afin de les envoyer au SGBD.

Conclusion sur la partie transactionnelle

Nous venons de voir comment développer des bases de données de production (c'est-à-dire mise-à-jour continuellement par les utilisateurs). Ces bases de production constituent un système client-serveur, où le client effectue ses mise-à-jour à travers des écrans fixes et où le serveur gère ces modifications sous forme de transactions programmées à l'avance.

On retrouve toujours le même schéma (cf. figure 8) dans lequel :
- les utilisateurs munis de leurs connexions et de leurs autorisations utilisent des interfaces graphiques ;
- ces formulaires font appel à des procédures stockées ;
- qui mettent en jeu des transactions (et leurs verrous) ;
- ces transactions s'appuient sur des vues (et leurs déclencheurs) ;
- qui sont les seules à accéder véritablement aux tables (et leurs contraintes).

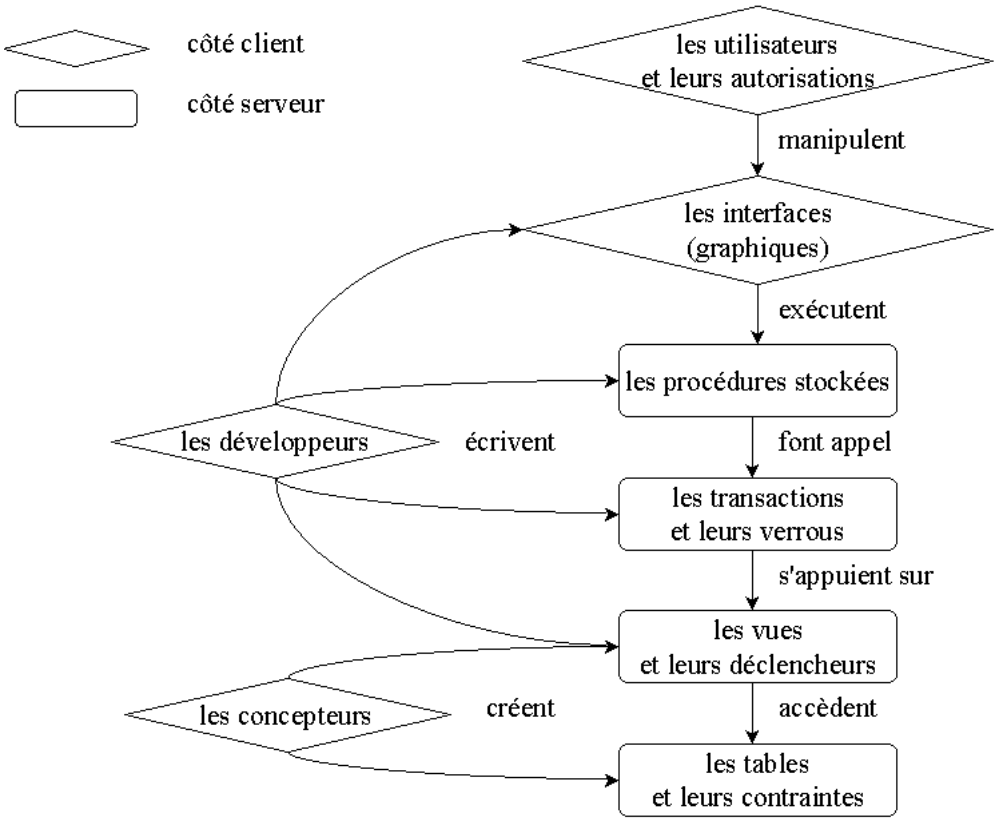

FIG. 8 – *Schéma du système transactionnel*

On peut facilement prendre comme illustration une opération bancaire à un guichet automatique (cf. figure 9) :

- le client de la banque muni de sa carte et de son code utilise l'interface graphique du distributeur de billets ;
- cette application déclenche sur le serveur de la banque la procédure de retrait d'argent ;
- cette procédure fait au moins appel à une transaction qui se charge de
  - débiter le compte (avec un verrouillage du compte pendant l'opération) ;
  - ajouter une ligne au relevé de compte ;
- on imagine que derrière cette transaction se trouve une vue regroupant les données nécessaires à l'opération et un déclencheur permettant la modification des données à travers cette vue ;
- données qui sont organisées de manière sophistiquée dans la base de données de la banque qui respecte avec la plus grande rigueur les différents type d'intégrité.

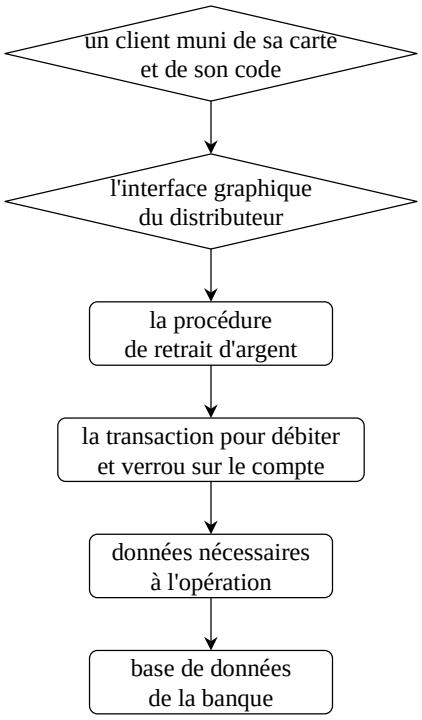

FIG. 9 – *Exemple de transaction*

Deuxième partie
Le système décisionnel

En anglais on parle de système OLAP (On Line Analytical Processing). Il s'agit ici de s'intéresser aux besoins des décideurs, à savoir, transformer les données en connaissances voire en prévisions, c'est-à-dire en information.Prenons l'exemple d'une bibliothèque : l'historique des emprunts représente les données, le nombre moyen d'emprunts d'un membre au cours d'une année est de l'ordre de la connaissance, tandis que la tendance à la baisse ou à l'augmentation des inscriptions est une prévision.

Exemple de questions décisionnelles :
- quels sont les clients les plus profitables ?
- quels sont les produits en perte de vitesse ?
- à quel chiffre d'affaire s'attendre pour l'année prochaine ?

Différences avec le système transactionnel

Contrairement au système OLTP, ici, on a besoin d'effectuer des requêtes purement consultatives et sur un grand nombre de données que l'on doit regrouper. D'une part, ces requêtes risquent de fortement perturber le système transactionnel (souvent déjà saturé), d'autre part, si elles s'appuient sur le schéma relationnel de la base de données, alors elles seront trop longues à répondre (à cause des jointures).

Par ailleurs, on ne stocke pas l'historique des données dans une base de production : généralement, la tables `clients` contient les clients qu'on a et ne s'encombre pas des clients qu'on a eu. Les informations contenues dans une base de production sont les plus à jour possible (et c'est normal).

Enfin, les données nécessaires à un processus décisionnel peuvent se trouver réparties sur plusieurs systèmes OLTP, eux-mêmes parfois gérés par différents SGBD (en raison d'un manque de centralisation des développements dans l'entreprise, ou à la suite d'une fusion de plusieurs services informatiques).

Toutes ces raisons rendent inutilisables les tableaux de bord que l'on peut mettre en place dans les bases de production et ont conduit à l'élaboration du système OLAP, plus sophistiqué.

À noter que le marché mondial de l'OLAP est très différent du marché OLTP. En effet, le marché OLAP en 2002 (comme en 2001) a été dominé par Microsoft (Analysis Services) et Hyperion (Essbase), avec Cognos (PowerPlay) et Business Objects derrière. Oracle et IBM sont loin derrière et en recul (source : [15]).

Entrepôts et cubes

Pour toutes ces raisons, la demarche OLAP consiste à copier et historiser les données dans un entrepôt de données (data warehouse)[16]. Pour des raisons qui s'éclairciront plus tard dans l'exposé, ces données sont stockées sous forme de cubes plutôt que dans des tables reliées entre elles[17].

Le premier exemple de cube que l'on peut donner est le suivant (cf. figure 10) : un cube tridimensionnel où horizontalement sont détaillées les années, verticalement les produits et, dans la profondeur, les pays. À l'intersection d'une année (2003), d'un produit A et d'un pays (France) on trouve une cellule contenant le montant des ventes et le prix unitaire du produit A en 2003 en France.

16. un magasin de données (data mart) ne s'intéresse qu'à un secteur de l'entreprise, tandis que l'entrepôt de données se situe au niveau de l'entreprise
17. c'est là toute la différence entre les entrepôt de données et les infocentres avec lesquels les données sont effectivement extraire des bases de production et consolidées sur un autre serveur, mais pas organisées en cubes

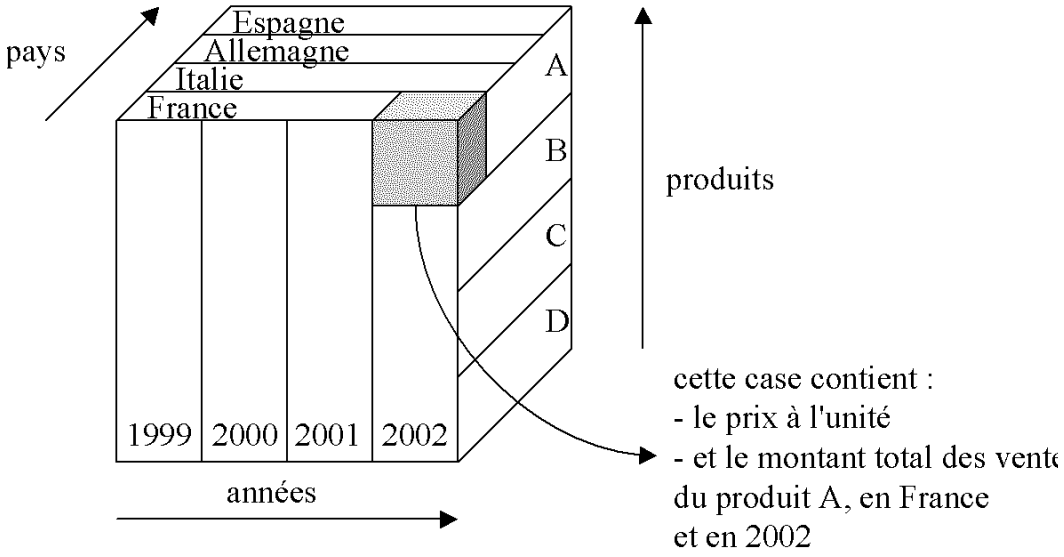

FIG. 10 – *Exemple de cube : le cube* `ventes`

L'entrepôt de données contient donc des informations datées et non volatiles. Il est alimenté par des extractions périodiques portant éventuellement sur plusieurs sources de données pendant les plages horaires peu occupées (la nuit). Ces données sont également pré-agrégées et structurées de manière multidimensionnelle, ce qui permet de simplifier et d'accélérer considérablement les requêtes. Les utilisateurs (décideurs) et la machine peuvent alors véritablement interagir.

Il faut tout de suite prendre conscience que pour bien construire un entrepôt de données dans une entreprise, il faut bien connaître les métiers utilisateurs (c'est-à-dire le metier d'analyste marketing s'il s'agit des données sur les ventes de produits par exemple).

Plan de l'exposé

Il s'agit pour nous de concevoir et de consulter les cubes avec Analysis Services et le langage MDX (MultiDimensional eXpression) qui est aux cubes ce que SQL est aux tables ...

Mais avant, nous évoquons deux éléments décisionnels complémentaires des cubes OLAP : les groupes de données et les états.

On aborde également la phase d'extraction de données, les objets virtuels ainsi que deux algorithmes de data mining :
- la clusterisation ;
- et les arbres de décisions.

De nouveau, nous laisserons de côté la construction d'interfaces graphiques de consultation et l'administration (sauvegarde, optimisation, etc.) des entrepôts. Le lecteur sera libre de consulter [8] sur ces sujets.

13 Agréger les données

Revenons un instant sur les requêtes de sélection dont nous n'avons pas totalement fait le tour au cours de la première partie.

13.1 Groupes

On peut grouper les lignes résultant d'une requête SQL et en profiter pour effectuer des mesures sur ces groupes. Comptons par exemple le nombre de commandes et calculons le montant total par client :

```
1 SELECT cmd_clt, COUNT(cmd_num)   AS NbCommandes,
2                 SUM(cmd_montant) AS [Montant total]
3 FROM commandes
4 GROUP BY cmd_clt
```

Dans le résultat de cette requête, chaque ligne est un groupe :

cmd_clt	NbCommandes	Montant total
0001	3	500.00
0007	8	1234.56
1122	1	32.60
3344	12	788.54

Rappel : `COUNT`, `SUM`, `AVG`, `VAR`, `STDEV`, `MIN` et `MAX` sont des fonctions d'agrégat. Elles retournent une information par groupe (un agrégat). Elles ignorent la valeur `NULL` (donc `COUNT(`18 lignes dont 2 `NULL)` renvoie 16) sauf `COUNT(*)`. Si on veut que la valeur `NULL` joue un rôle dans l'agrégation, alors on peut toujours utiliser la fonction `ISNULL` (cf. 4.7 §page 25).

Pour exclure certaines lignes de la table avant groupement, on utilise la clause `WHERE`. Exemple, pour ne garder que les commandes des 6 derniers mois :

```
1 SELECT cmd_clt, COUNT(cmd_num)   AS NbCommandes,
2                 SUM(cmd_montant) AS [Montant total]
3 FROM commandes
4 WHERE cmd_date >= DATEADD(MONTH, -6, GETDATE())
5 GROUP BY cmd_clt
```

Pour exclure certains groupes (donc après groupement) on ne peut pas utiliser la clause `WHERE` mais la clause `HAVING`. Exemple, pour n'afficher que les clients qui ont strictement plus de 5 commandes :

```
1 SELECT cmd_clt, COUNT(cmd_num)   AS NbCommandes,
2                 SUM(cmd_montant) AS [Montant total]
3 FROM Commandes
4 GROUP BY cmd_clt
5 HAVING COUNT(cmd_num) > 5
```

Le résultat de cette requête est :

cmd_clt	NbCommandes	Montant total
0007	8	1234.56
3344	12	788.54

Remarque : la rédaction une clause `HAVING` admet les mêmes conditions que celle d'une clause `WHERE` (cf. §4.1 page 19) et tout comme les clauses `WHERE` on ne peut malheureusement pas utiliser les alias définis dans la clause `SELECT`.

13.2 Compléments

On peut grouper selon plusieurs colonnes, et à partir de plusieurs tables. Exemple, ajoutons un sous-groupe selon le pays d'achat :

```
1 SELECT a.cmd_clt, b.btq_pays, COUNT(cmd_num)   AS NbCommandes,
2                               SUM(cmd_montant) AS [Montant total]
3 FROM commandes AS a
4 JOIN boutiques AS b ON a.cmd_btq = b.btq_num
5 GROUP BY a.cmd_clt, b.btq_pays
```

Dans le résultat de cette requête on a une ligne par sous groupe :

cmd_clt	btq_pays	NbCommandes	Montant total
0001	France	3	500.00
0007	France	4	1000.00
0007	Italie	4	234.56
1122	Italie	1	32.60
3344	Italie	6	394.27
3344	France	6	394.27

Remarques :
- l'ordre des colonnes dans la clause `GROUP BY` n'a pas d'importance ;
- toutes les colonnes de la clause `SELECT` ne faisant pas l'objet d'une fonction d'agrégat doivent figurer dans la clause `GROUP BY` et inversement ;
- la clause `WHERE` ne peut porter que sur des colonnes non agrégées (dont celle de la clause `GROUP BY`), alors que la clause `HAVING` peut porter sur toutes les colonnes de la clause `SELECT` ;
- par contre, la clause `WHERE` peut accéder aux colonnes non affichées, tandis que la clause `HAVING` ne le peut pas.

Pour ordonner les groupes et les sous-groupes (ce qui n'est pas le cas par défaut) il suffit d'utiliser la clause `ORDER BY` :

```
1 SELECT a.cmd_clt, b.btq_pays, COUNT(cmd_num)   AS NbCommandes,
2                               SUM(cmd_montant) AS [Montant total]
3 FROM commandes AS a
4 JOIN boutiques AS b ON a.cmd_btq = b.btq_num
5 GROUP BY a.cmd_clt, b.btq_pays
6 ORDER BY b.btq_pays, [Montant total]
```

Remarque : la clause `ORDER BY` s'applique après groupement et peut s'appuyer sur toutes les colonnes de la clause `SELECT`.

À noter enfin qu'à ce stade, une requête `GROUP BY` ne peut afficher que les agrégats des sous-groupes du niveau le plus bas. Si l'on désire afficher les agrégats des sous-groupes des autres niveaux (sous-total et total, par exemple), on peut ajouter à la clause `GROUP BY`, soit `WITH ROLLUP` soit `WITH CUBE` (cf. l'aide en ligne pour une description de ces deux instructions).

Exemple :

```
1 SELECT a.cmd_clt, b.btq_pays, COUNT(cmd_num)   AS NbCommandes,
2                               SUM(cmd_montant) AS [Montant total]
3 FROM commandes AS a
4 JOIN boutiques AS b ON a.cmd_btq = b.btq_num
5 GROUP BY a.cmd_clt, b.btq_pays WITH ROLLUP
```

Dans le résultat de cette requête on a, non seulement une ligne par sous groupe (niveau 2), mais aussi une ligne par groupe (niveau 1) et une ligne de niveau 0 :

cmd_clt	btq_pays	NbCommandes	Montant total
NULL	NULL	24	2555.70
0001	NULL	3	500.00
0001	France	3	500.00
0007	NULL	8	1234.56
0007	France	4	1000.00
0007	Italie	4	234.56
1122	NULL	1	32.60
1122	Italie	1	32.60
3344	NULL	12	788.54
3344	Italie	6	394.27
3344	France	6	394.27

Remarque : avec `WITH ROLLUP`, l'ordre des colonnes dans la clause `GROUP BY` a de l'importance.

13.3 Conclusion

On sait désormais rédiger une requête de sélection complète :

`SELECT` les colonnes à afficher (dans l'ordre)
`FROM` les tables et leurs conditions de jointure
`WHERE` les conditions de sélection avant groupement
`GROUP BY` les colonnes de groupement
`HAVING` les conditions de sélection sur les groupes
`ORDER BY` les colonnes à trier (dans l'ordre)

Il est donc temps de compléter la stratégie pour élaborer une requête de sélection :

1. décomposer au maximum en plusieurs sélection que l'on pourra réunir avec `UNION` ;
2. décomposer chaque sélection complexe en requête et sous-requêtes simples ;
3. et pour chaque requête et chaque sous-requête :
   (a) déterminer les tables en jeu pour remplir la clause `FROM` et les `JOIN` nécessaires ;
   (b) déterminer les colonnes de groupement pour remplir la clause `GROUP BY` ;
   (c) déterminer les colonnes à afficher pour remplir la clause `SELECT` ;
   (d) déterminer les conditions de sélection avant groupement pour remplir la clause `WHERE` ;
   (e) déterminer les conditions de sélection sur les groupes pour remplir la clause `HAVING` ;
   (f) ajouter les éventuels `ORDER BY`, `DISTINCT` et `TOP` en dernier.

14 Édition d'états

Un état est un document de synthèse, établi automatiquement à partir des données. Il peut être soit purement informatif (un annuaire, par exemple) soit à caractère décisionnel (les ventes de la veille ventilées selon les secteurs et en comparaison avec la même date l'année précédente, par exemple). En anglais, on parle de reporting.

Les états qui nous intéressent ici sont ceux qui permettent d'analyser les données (donc les états à caractère décisionnel). Ils se présentent généralement sous la forme d'un long tableau d'agrégats détaillés en groupes et sous-groupes, mais on peut considérer que les graphiques décisionnels sont aussi des états (on ne les aborde pas ici).

Certains outils d'édition d'états sont fournis avec un SGBD (Microsoft Access, par exemple) ou avec une plate-forme OLAP (c'est le cas de Crystal Reports qui est à la base de Crystal Decisions, racheté par Business Objects).

14.1 Comparaison avec les formulaires

Comme les formulaires, les états sont composés de champs et autres contrôles (essentiellement des zones de texte). Ils sont également destinés soit à être imprimés soit à être consultés à l'écran (sous la forme d'une page web, par exemple). Il faut donc veiller tout particulièrement :
- à ce que l'état ne dépasse pas la largeur de l'écran et/ou de la feuille ;
- à ce que les champs soient suffisamment larges pour afficher leur contenu ;
- et à ce que les interactions avec l'utilisateur soient ergonomiques (s'il y en a).

Un état se présente comme un formulaire muni d'un sous-formulaire, lui-même muni d'un sous-sous-formulaire, etc.

Cependant, contrairement aux formulaires, les états ne permettent pas à l'utilisateur de modifier les données, mais ont simplement une fonction de consultation des informations.

14.2 Comparaison avec les requêtes `GROUP BY`

Un état effectue essentiellement le même travail qu'une requête `GROUP BY WITH ROLLUP`, mais un état est plus lisible car mieux organisé et mieux présenté qu'un vulgaire résultat de requête.

Les problèmes des requêtes `GROUP BY WITH ROLLUP` qui sont résolus par les états sont les suivants :
- les intitulés des colonnes ne sont affichés qu'une fois, même si le résultat s'étale sur plusieurs pages ;
- on ne peut pas afficher de renseignement complémentaire (non agrégé) concernant les groupes (le nom du `client`, par exemple), à cause de la contrainte entre les clauses `SELECT` et `GROUP BY` ;
- les agrégats des niveaux supérieurs ne sont pas mis en valeur ;
- les conditions de sélection ne sont pas affichées (on ne peut donc pas diffuser le résultat).

14.3 Composition

Chaque sous-groupe de plus bas niveau occupe une ligne dans le document final. C'est cette ligne qui est décrite dans la zone `Détail`. Les intitulés des colonnes du `Détail` sont précisées dans l'en-tête de page (afin d'être répétés automatiquement à chaque changement de page).

Au dessus du `Détail` viennent les en-têtes des niveaux de groupement successifs dans lesquels on place les informations relatives à ces niveaux.

En dessous du `Détail` viennent les pieds des niveaux de groupement successifs dans lesquels on place généralement les résultats d'agrégation voulus.

Il reste :
- l'en-tête d'état pour son titre et ses conditions de sélection (notamment la ou les dates, les clients retenus, les articles concernés, etc.), il ne faut surtout pas oublier de les préciser, sans quoi le document n'a aucune valeur ;
- le pied de page pour la numérotation des pages (ce qui est important si le document est imprimé et malencontreusement mélangé) ;
- et le pied d'état pour les agrégats du niveau 0 (un total général par exemple).

Exemple :

Exemple d'état ← le titre

pour les clients ayant plus de 5 commandes ← les conditions de sélection
pour toutes les dates (dans l'en-tête d'état)

	Pays	NbCommandes	Montant	← les intitulés (en-tête de page)
client n° 0007	(Razibus)			← un en-tête de groupe
	France	4	1000.00	← une ligne par sous-groupe
	Italie	4	234.56	
	Total	8	1234.56	← un pied de groupe
client n° 3344	(Fricotin)			
	France	6	394.27	
	Italie	6	394.27	
	Total	12	788.54	
Total général		20	1923.10	← le pied d'état

page 1 ← un pied de page

15 Structurer les données en cube

On sent bien que la table est un objet bidimensionnel mal adapté à l'agrégation de données dès que le groupement porte sur deux colonnes ou plus. L'état offre une première solution, mais la feuille A4 (même orientée en paysage) est rapidement saturée quand le nombre de niveau de groupement augmente et la navigation dans un état de 200 pages est fastidieuse.

15.1 Définition d'un cube

On préférerait donc un objet :
- qui aurait autant de dimensions qu'il y a de colonnes de groupement (`clients` et `pays` dans notre exemple précédent) ;
- dont chaque dimension détaillerait les valeurs possibles (`France` et `Italie` pour la dimension `pays`) ;
- dans lequel, à l'intersection d'une valeur dans chaque dimension (`France` et `0001` par exemple) on trouverait une unique cellule ;
- une cellule qui contiendrait toutes les valeurs agrégées pour ce sous-groupe (c'est-à-dire les mesures tel que le nombre de commandes et le montant total) ;
- et aussi les agrégats des ces valeurs pour les sous-groupes supérieurs.

C'est cet objet que l'on appelle cube.

Par exemple, organisons, en cube, les données de la requête suivante :

```
1 SELECT a.cmd_clt, b.btq_pays, COUNT(cmd_num), SUM(cmd_montant)
2 FROM commandes AS a
3 JOIN boutiques AS b ON a.cmd_btq = b.btq_num
4 GROUP BY a.cmd_clt, b.btq_pays WITH CUBE
```

Résultat :

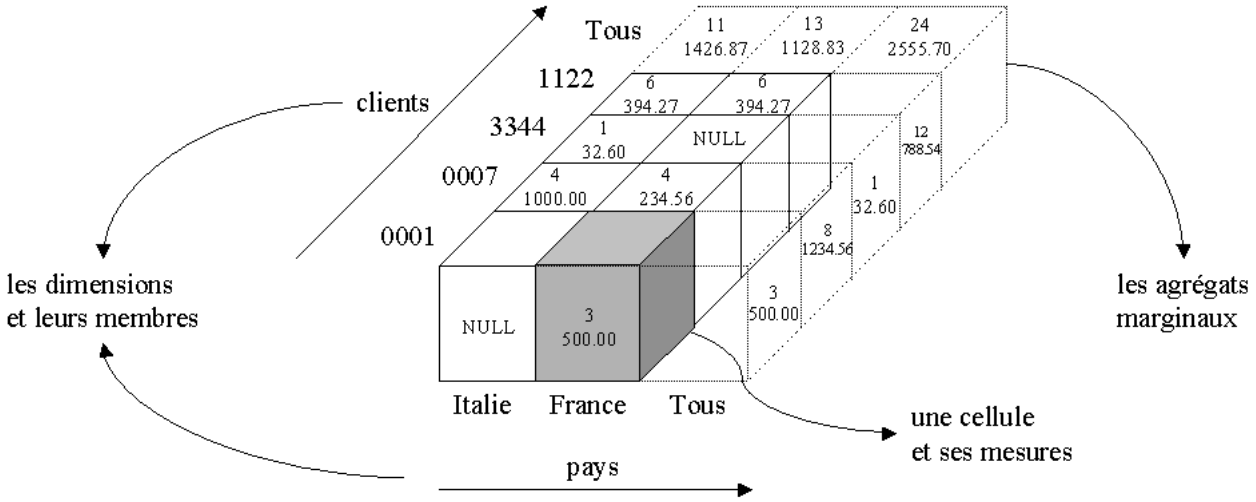

FIG. 11 – *Cube bidimensionnel*

Remarque : le terme de cube ne présupposant ni du nombre de dimensions ni de l'égalité des longueurs des dimensions (cf. figure 11).

En MDX, pour créer ce cube il suffit d'écrire :

```
1 CREATE CUBE ClientsPays
2 (
3   DIMENSION clients
4   DIMENSION pays
5   MEASURE NbCommandes FUNCTION COUNT
6   MEASURE [Montant total] FUNCTION SUM
7 )
```

Remarque : on dispose bien évidemment de `DROP CUBE` et de `ALTER CUBE`[18].

Mesures

Les mesures sont généralement additives (soit une somme, soit un dénombrement).

Il ne faut stocker dans le cube que les mesures primitives :
– le prix TTC peut être calculé à partir du prix hors taxe, ce n'est donc pas la peine de le stocker ;
– il en va de même pour les prix en dollars à partir des prix en euros ;

18. on laisse au lecteur le soin de se renseigner auprès de l'aide en ligne pour connaître les possibilités de `ALTER CUBE`

– d'un montant à partir d'un prix unitaire et d'une quantité ;
– etc.

Dans Analysis Services, les mesures non primitives peuvent être calculées avec les membres calculés (cf. §18.4.1 page 91).

Ceci dit, si les règles de calcul des mesures non primitives sont compliquées ou si elles changent souvent (c'est le cas des conversions euro/dollar, par exemple), alors il vaut mieux stocker une mesure de plus.

15.2 Hiérarchie

Complexifions progressivement la définition d'un cube.

15.2.1 Niveaux

Avec la structure de cube, on peut détailler chaque dimension en plusieurs niveaux. Par exemple, une dimension générale `geographie` peut permettre des groupements selon un `pays`, une `region` ou une `ville`. On voit alors apparaître une hiérarchie dans les dimensions que l'on dépliera et pliera à volonté, selon nos besoins d'agrégation.

On peut prendre comme illustration un cube `ventes` à trois dimensions (cf. figure 12) :
– une dimension `geographie` comprenant plusieurs pays qui se décomposent chacun en plusieurs régions qui regroupent elles-mêmes plusieurs villes dans lesquelles se situent les boutiques ;
– une dimension `produits` dans laquelle les articles sont regroupés en gammes puis en marques ;
– et une dimension temporelle détaillée en années, mois et jours.

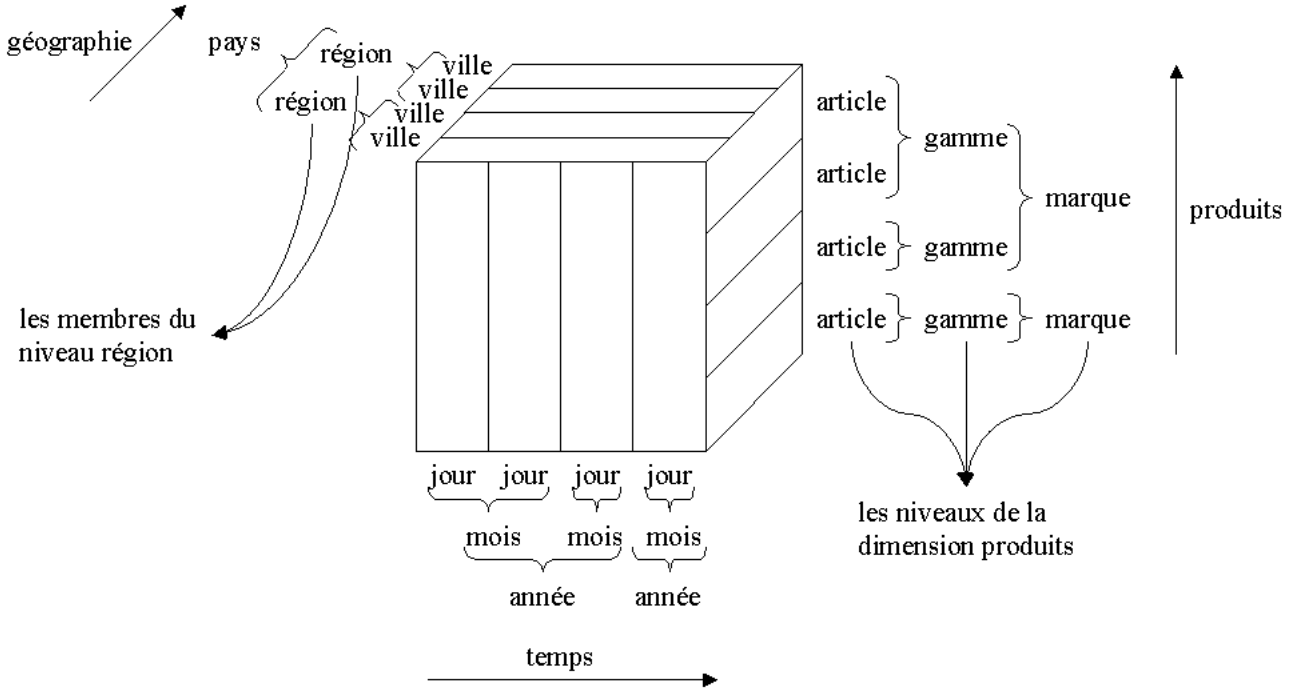

FIG. 12 – *Cube* `ventes` *hiérarchisé*

La syntaxe MDX pour déclarer ces dimensions hiérarchisées est

– pour les dimensions standards :

```
DIMENSION geographie
  LEVEL tout TYPE ALL,  -- mettre ce niveau a chaque fois
  LEVEL pays,
  LEVEL region,
  LEVEL ville

-- on separe les niveaux par des virgules mais pas les dimensions

DIMENSION produits
  LEVEL tout TYPE ALL,  -- du plus agrege
  LEVEL marque,
  LEVEL gamme,
  LEVEL article         -- au plus detaille
```

– et pour la dimension temporelle :

```
DIMENSION temps
  LEVEL tout TYPE ALL,
  LEVEL annee TYPE YEAR,
  LEVEL mois TYPE MONTH,
  LEVEL jour TYPE DAY
```

Remarques :
– le niveau `ALL` un niveau formel qui regroupe tous les autres ;
– s'il y a ambiguïté sur le nom des niveaux, il suffit de préciser la dimension concernée selon la syntaxe : `dimension.niveau` (exemple : `produits.articles`) ;
– dans une dimension hiérarchisée, les données du niveau le plus bas sont issues des bases de production ;
– mais le cube peut aussi stocker des données aux niveaux supérieurs (il s'agit le plus souvent de données agrégées, mais pas toujours).

Le niveau le plus bas d'une dimension s'appelle le grain de la dimension (le grain de la dimension `geographie` c'est la `ville`). Toutes les dimensions et leur grain doivent être choisis dès le début et doit rester inchangés pendant toute la durée de vie du cube.

15.2.2 Membres

Les différentes valeurs d'un niveau sont appelées membres. Par exemple, les membres du niveau `pays` sont `France`, `Italie`, `Allemagne` et `Espagne`. Un membre de niveau supérieur regroupe des membres du niveau immédiatement inférieur, ce sont ses enfants. Par exemple, les enfants du membre `France` sont `PACA`, `Rhones-Alpes` et `Corse`.

Remarque : un membre d'un niveau supérieur (une région) peut posséder des données (son ensoleillement, par exemple) et pas seulement des agrégats (montant total des ventes dans cette région).

15.2.3 Hiérarchies multiples

Certaines dimensions peuvent être organisées selon plusieurs hiérarchies.

C'est classiquement le cas de la dimension temporelle qui peut non seulement être organisée en :
- jours, mois, trimestres et années ;
- mais aussi en jours et semaines ;
- ou encore en jours et saisons.

Cela peut également se produire pour n'importe quelle dimension (cf. la dimension `produits` sur la figure 16).

Analysis Services ne permet malheureusement pas à une dimension de posséder plusieurs hiérarchies. La technique pour les représenter malgré tout consiste alors simplement :
- soit à introduire plusieurs dimensions (avec une hiérarchie chacune), ce qui n'est pas recommandé ;
- soit à utiliser les propriétés de membre (cf. §19.1 page 96) et les dimensions virtuelles.

15.3 Normalisation d'un cube

Le schéma relationnel d'un cube (cf. section suivante) n'a pas besoin de respecter la troisième forme normale (cf. [**?**]), ce qui permet d'utiliser des schémas en étoile. Par contre, quelques règles (tirées de [4]) doivent être respectées lors de la construction de chaque cube :

Règle 1 : dans un cube, deux membres appartenant à deux dimensions différentes doivent être indépendants .

Autrement dit, s'il n'y a qu'un vendeur par produit, il faut fusionner les dimensions `produits` et `vendeurs`.

Règle 2 : dans un cube, tous les faits doivent dépendent de toutes les dimensions.

Autrement dit, les ventes (qui dépendent du produit, du jour, du client et de la ville) et les coûts de développement (qui ne dépendent que du produit) définissent deux types de faits distincts (et conduisent donc à deux cubes distincts).

Règle 3 : dans un cube, toutes les mesures doivent respecter le grain du cube.

Si la marge n'est définie que par région et par mois, tandis que le montant des ventes le sont par ville et par jour, alors il ne faut pas chercher à les faire cohabiter par une division arithmétique mais les séparer dans deux cubes distincts.

Règle 4 : la hiérarchie d'une dimension doit être strictement arborescente.

Ce n'est pas le cas d'une dimension `organisation` dans laquelle :
- les agences sont regroupées administrativement en divisions ;
- les agences sont regroupées géographiquement en établissements, même si elles appartiennent à des divisions différentes ;
- les divisions sont regroupées en directions régionales ;
- les établissement sont également regroupés en directions régionales.

Il faut alors utiliser deux hiérarchies pour cette dimension :
- une dont les niveaux sont : agences, divisions et directions régionales ;
- et une autre dont les niveaux sont : agences, établissement et directions régionales .

Lorsqu'un cube vérifie ces quatres règles, on dit qu'il est en forme dimensionnelle normale.

16 Stockage des données

On sait que les données des systèmes transactionnels sont stockées sous une forme purement relationnelle. C'est d'ailleurs cette vision relationnelle que l'on a remplacée par une vision multidimensionnelle dans cette partie. La question qui se pose maintenant est : comment stocker des cubes ayant de nombreuses dimensions, elles-même à plusieurs niveaux ?

16.1 Schéma relationnel de l'entrepôt

La première idée est de stocker physiquement un objet multidimensionnel, c'est-à-dire une hypermatrice avec plusieurs mesures par cellule et de stocker aussi les agrégats au bout de chaque ligne, chaque colonne, etc. et pour tous les niveaux. C'est la meilleure approche du point de vue des performances de consultation (puisque tous les résultats sont présents), mais ce stockage est bien souvent trop gourmand en place mémoire.

Notre soucis est donc maintenant d'économiser de l'espace. Or on connaît déjà la façon la plus économique de stocker des données, c'est l'approche relationnelle (dans laquelle on évite toute redondance). Il se trouve justement que l'on peut voir un cube sous forme de tables et de relations.

16.1.1 Schéma en étoile

Si au sein de l'entrepôt, nous nous concentrons sur un cube, alors on peut modéliser ses dimensions, leurs niveaux, leurs membres et les cellules sous la forme d'un schéma relationnel étoilé.

Table de dimension

À chaque dimension on associe une table avec :
- une clé primaire non composite ;
- et une colonne par niveau (pour y stocker les membres).

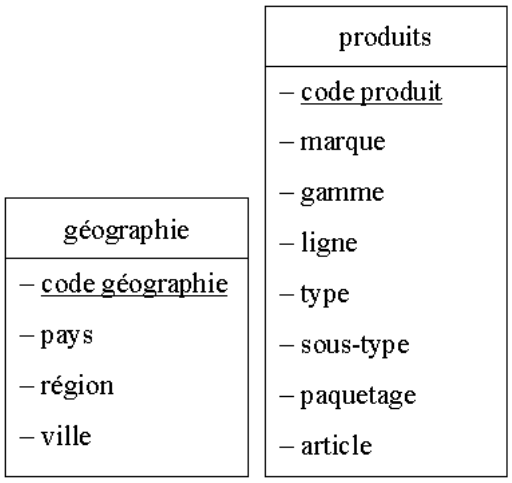

FIG. 13 – *Tables de dimension*

Par exemple, à la dimension `geographie` on associe une table (cf. figure 13(a)) ayant pour clé primaire `code geographie` et comme autres colonnes : `pays`, `region` et `ville`.

Étant donné que les données sont issues de plusieurs bases de données ayant chacune leur clés primaires, les clés primaires des tables de dimensions sont différentes car elles doivent assurer une unicité plus générale. Ces clés de substitution (surrogate keys, par opposition aux clés naturelles utilisées dans les bases de production) servent à identifier un membre pendant tout sa durée de vie au sein de l'entrepôt.

Dimension temporelle

Les dimensions temporelles ne rentrent pas dans ce cadre puisque qu'on ne s'amuse pas à détailler les années, les mois et les jours dans trois colonnes différentes, mais qu'on regroupe cette information dans une seule colonne de type `DATETIME`.

Remarque : la dimension temporelle pose toujours un problème de conception notamment parce que tous les acteurs d'une entreprise ne fonctionnent ni avec le même calendrier (civil, fiscal, scolaire, etc.) ni avec la même horloge (surtout si l'entreprise est multinationale).

Table des faits

Ensuite, chaque cellule du cube :
- est identifiée par ses coordonnées (*i.e.* son `code geographie`, son `code produits` et sa date) ;
- et contient ses mesures.

L'ensemble de ces informations (coordonnées et mesures relatives à une cellule) constitue ce que l'on appelle un fait. Si maintenant on stocke chaque fait sous la forme d'une ligne dans une table, alors on obtient un schéma relationnel étoilé par rapport à cette table (autrement dit toutes les relations partent de cette table, cf. figure 14).

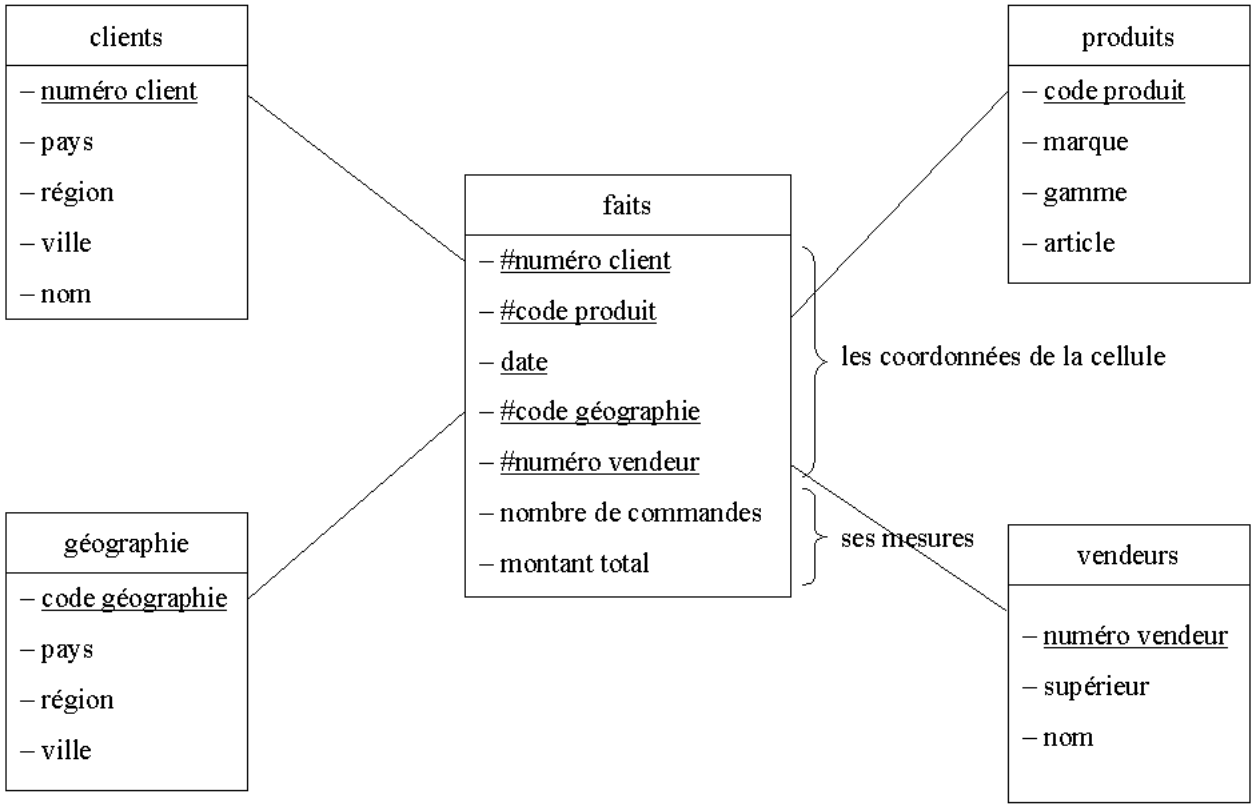

FIG. 14 – *Schéma en étoile d'un cube à 5 dimensions et 2 mesures*

Dans l'exemple du cube `ventes`, la table des faits n'aurait posséder que deux clés étrangères : `code`

`geographie` et `code produit`, une colonne pour la date ainsi que deux autres colonnes pour les mesures : `NbCommandes` et `[Montant total]`. Pour mettre en évidence le caractère étoilé du schéma relationnel, nous ajoutons deux dimensions au cube `ventes` : une relative aux clients et une relative aux vendeurs.

Remarque : la table des faits peut compter plusieurs millions de lignes et la taille des autres tables est négligeable devant celle de la table des faits.

16.1.2 Schéma en flocon

Dans le schéma en étoile (star scheme), il peut encore y avoir des redondances, pour deux types de raisons :

– n'utiliser qu'une seule table pour définir une dimension qui possède plusieurs niveaux est presque toujours en contradiction avec la troisième forme normale [**?**] (cf. figure 13(b)) ; on peut donc pousser plus loin la décomposition relationnelle (cf. figure 15)

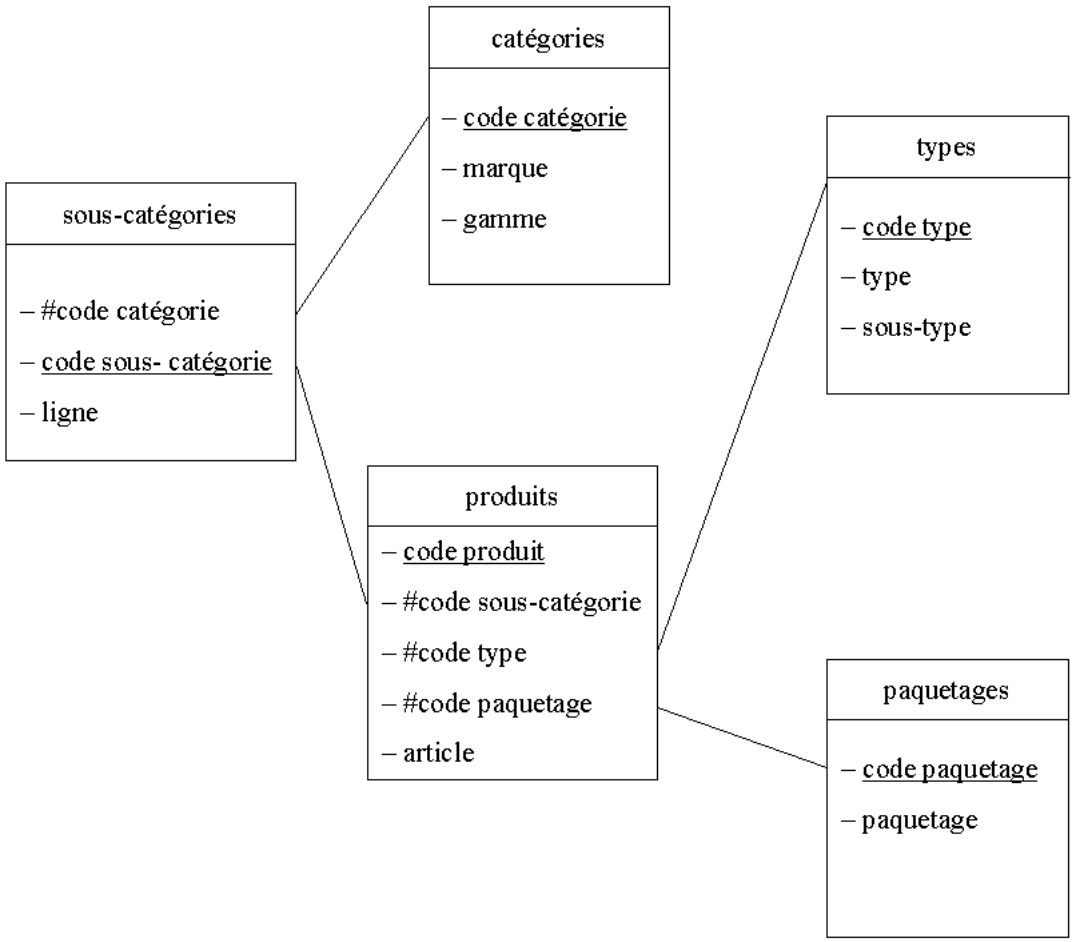

FIG. 15 – *Décomposition de la table* `produits`

– par ailleurs, on peut parfois factoriser certaines données (une seule table `geographie` pour la géographie des ventes et des clients, par exemple).

Comme la table des faits n'est plus la seule à étoiler les relations autour d'elle, dès lors qu'une dimension est basée sur deux tables ou plus (cf. figure 16), on dit que le schéma est en flocon (snowflake scheme).

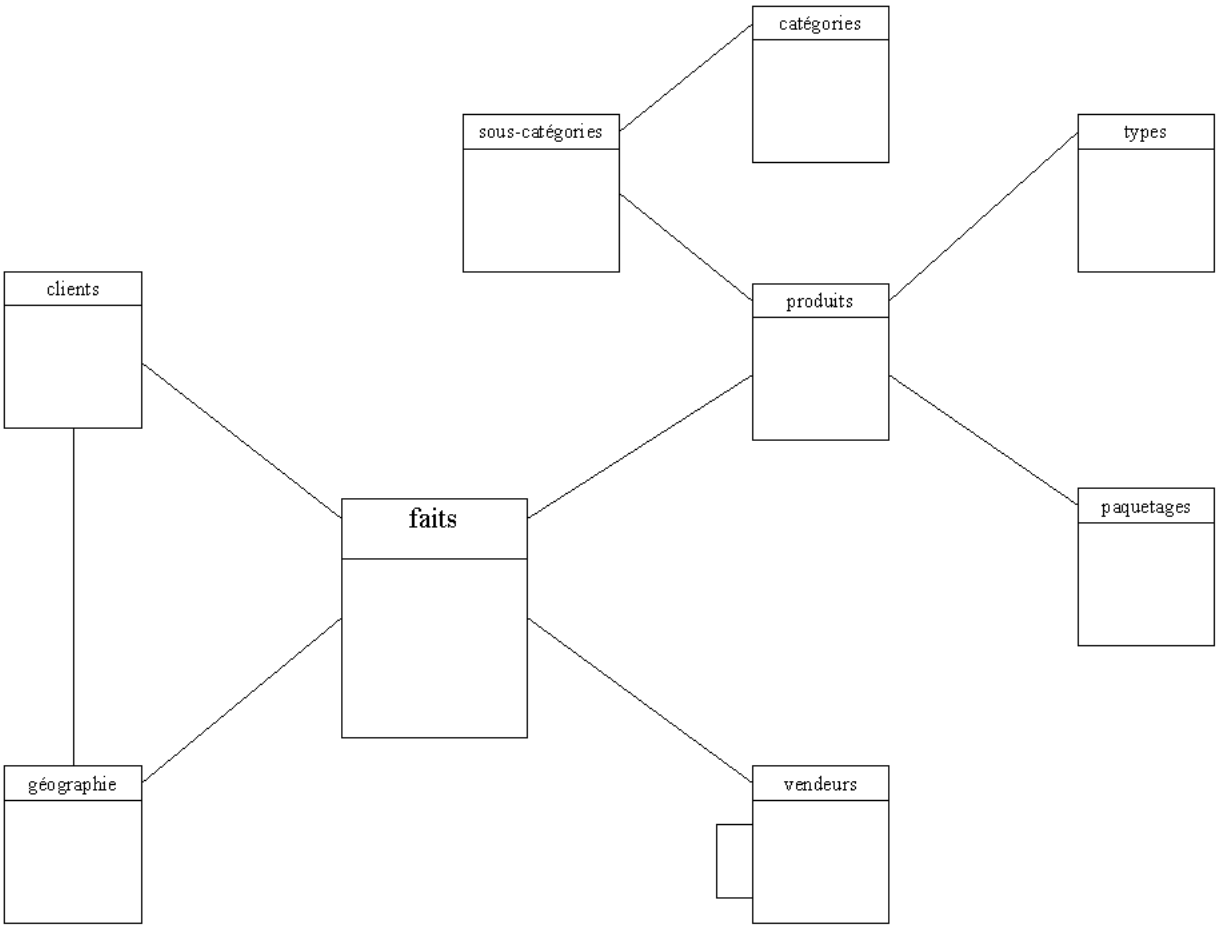

FIG. 16 – *Schéma en flocon*

Remarque : dans cet exemple, la dimension `produits` présente trois hiérarchies (une selon les types, une selon les paquetages et une selon les catégories).

Ce qu'il faut retenir de tout cela, c'est que :
- une dimension s'inscrit dans un schéma en étoile si elle n'est définie que sur une table du schéma relationnel du cube (c'est le cas de la dimension `geographie`) ;
- dès qu'elle utilise plusieurs tables du schéma relationnel, la dimension s'inscrit dans un schéma en flocon (c'est le cas des dimensions `produits` et `clients`).

Quand certaines dimensions sont en étoile et d'autre en flocon, on dit que le schéma du cube est en starflake.

Dans Analysis Services, on est donc amené à spécifier pour chaque dimension (non temporelle) :
- si elle s'inscrit dans un schéma en étoile ou en flocon ;
- la ou les tables concernées (elles doivent exister avant la création du cube).

Que choisir ?

Le schéma en étoile est, certes, plus redondant que le schéma en flocon mais :
- la redondance n'est pas un problème pour le décisionnel, puisqu'il n'y a que des requêtes de sélection ;
- l'espace occupé par les tables de dimension est négligeable devant celui de la table des faits ;
- les requêtes sont plus rapides sur un schéma en étoile ;

– l'ETL est plus simple avec un schéma en étoile.

Pour toutes ces raisons, le schéma en flocon est peu recommandé et doit se limiter à :
– la factorisation, comme celle de la table `geographie` ;
– la représentation de hiérarchie multiple, comme celle de la dimension `produits`.
Ce qui exclut la décomposition à outrance comme celle des catégories et sous-catégories.

16.1.3 Parent-enfant

D'habitude, les niveaux d'une relation sont de type ensembliste : dans la dimension `geographie`, un pays est un ensemble de régions.

Prenons maintenant une hiérarchie de dimension qui ait une signification familiale (cf. figure 17). Par exemple une dimension `personnes` avec un niveau `GrandsParents`, un niveau `parents` et un niveau

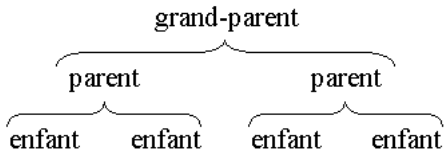

FIG. 17 – *Hiérarchie parent-enfant*

`enfants`. Le schéma relationnel du cube n'est alors plus le même, puisqu'il fait intervenir une auto-jointure sur la table `personnes` (cf. figure 18).

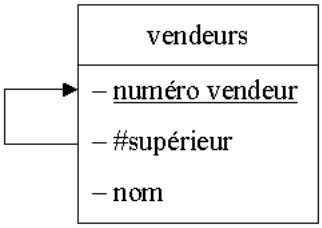

FIG. 18 – *Auto-jointure d'une table de dimension parent-enfant*

C'est ce que l'on appelle une hiérarchie parent-enfant. Elle est valable également quand il y a une relation employé-employeur.

Remarque : dans le cas d'une hiérarchie parent-enfant, les niveaux supérieurs possèdent des données propres (non agrégées). Par exemple, le nom du directeur commercial n'est pas issus de l'agrégation des noms de ses vendeurs...

16.1.4 Base décisionnelle

Un entrepôt de données peut contenir plusieurs cubes (le cube `ventes` que l'on vient de voir, et le cube `production`, par exemple). Le schéma relationnel de l'entrepôt regroupe les schémas relationnels de ses cubes. La base de données qui correspond à ce schéma relationnel global,s'appelle la base décisionnelle (par opposition aux bases de production).

Elle doit être construite avant l'entrepôt lui-même. Elle résulte d'un travail important de rétro-conception des bases de production et de normalisation à l'échelle de l'entreprise.

Certaines dimensions sont communes à plusieurs cubes (la dimension `produits` est commune aux cubes `ventes` et `production`, par exemple). Leurs tables ne sont évidemment pas répétées dans le schéma

relationnel de l'entrepôt, mais utilisées par plusieurs tables des faits. C'est pourquoi Analysis Services emploie le terme de dimensions partagées.

Il en va de même pour les mesures et toutes les autres colonnes utilisées pour définir le cube : elle sont présentes une bonne fois pour toutes dans ce schéma relationnel global, puis utilisées dans le schéma relationnel de chaque cube.

À l'instar des bases de production, le schéma relationnel de la base décisionnelle doit être conçu dans une phase modélisation et doit coller aux besoins des utilisateurs. Par contre, cette modélisation OLAP diffère sensiblement des modélisations relationnelles classiques pour les systèmes OLTP. Le schéma relationnel doit être suffisamment bien conçu dès le départ, car c'est encore plus problématique de modifier la base décisionnelle qu'une base de production.

Rappelons que cette base décisionnelle a pour but d'historiser les données et est mise-à-jour, non pas par de nombreuses transactions ACID (cf. §2.4 page 15), mais par des extractions périodiques des systèmes OLTP sous-jacents (avant qu'ils ne soient purgés).

Dans Analysis Services, c'est cette base qui est la source de données à connecter à l'entrepôt. Notons qu'Analysis Services autorise plusieurs sources de données (et donc plusieurs bases décisionnelles) pour un même entrepôt.

16.2 Modes de stockage

Classiquement, il existe trois modes de stockage pour chaque partition d'un cube. Sans entrer dans le détail, les données d'un cube peuvent être partitionnées (selon les années, généralement).

Généralement un cube présente :
- une partition pour l'année en cours ;
- une partition pour l'année précédente (ou les deux années précédentes) ;
- et une dernière partition pour les autres années.

Remarque : les partitions correspondent à un éclatement de la table des faits.

16.2.1 Multidimensional OLAP (MOLAP)

En mode MOLAP, les données de la base décisionnelle sont copiées dans une structure (hyper-)matricielle qui contient également les agrégats. Analysis Services compresse alors les données et n'alloue aucun espace aux cellules vides[19], ce qui limite la taille de stockage.

Pourtant, ce type de stockage doit se limiter aux données les plus utilisées (celle de l'année en cours, par exemple) afin que les réponses soient instantanées et que le volume de données MOLAP reste raisonnable.

16.2.2 Relational OLAP (ROLAP)

En mode ROLAP, la partition est vide. Toutes les données restent dans la base décisionnelle. Les requêtes MDX sur ce cube font donc appel à des requêtes SQL impliquant des jointures. C'est le type de stockage qui offre les temps de réponse les plus lents. Mais c'est aussi le plus économe.

Généralement, les données qui remontent à deux ans et plus (elles sont nombreuses et peu consultées) sont stockées en ROLAP.

Remarque : les requêtes sont plus rapides sur un schéma en étoile (même redondant) que sur un schéma en flocon (qui met en jeu davantage de jointures).

19. cela est heureux car, généralement, un cube OLAP est creux, c'est-à-dire essentiellement constitué de cellules vides

16.2.3 Hybrid OLAP (HOLAP)

En mode HOLAP, seuls les agrégats des niveaux supérieurs sont stockés sous forme matricielle (cf. figure 19) et les données bas niveaux restent dans la base décisionnelle. Il s'agit d'une combinaison entre l'approche MOLAP et l'approche ROLAP. Seule les requêtes MDX qui utilisent directement les données bas niveaux sont ralenties (drillthrough, par exemple).

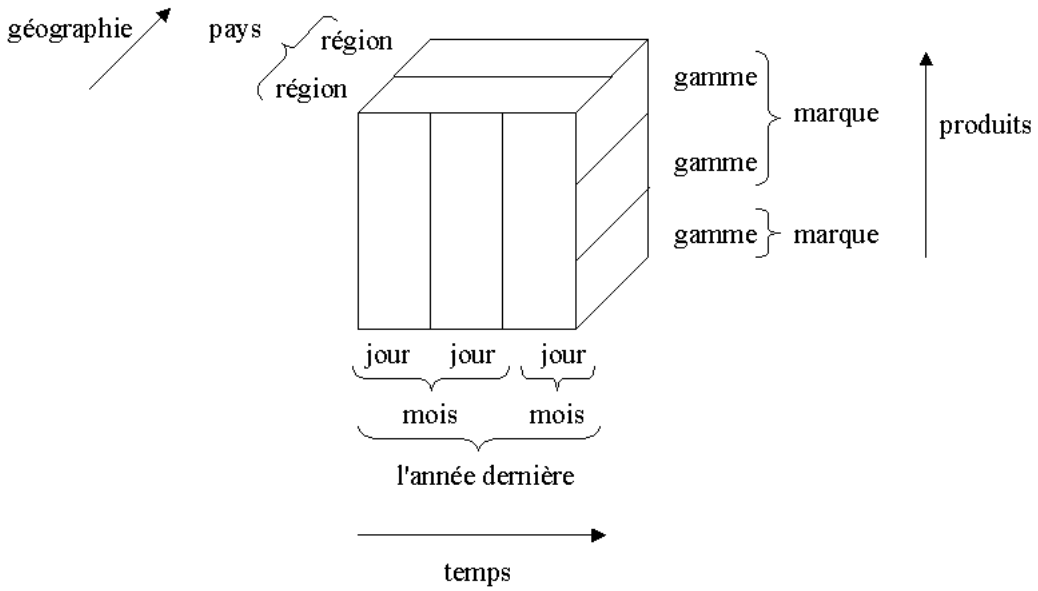

FIG. 19 – *Cube réduit pour les agrégats des niveaux supérieurs*

Ce type de stockage convient bien aux données de l'année précédente (ou des deux années précédentes).

16.3 Niveau d'agrégation

Quelque soit le mode de stockage choisi, les agrégats ne sont pas forcément tous calculés et stockés. Analysis Services peut déterminer quels agrégats stocker (en commençant par les niveaux les plus bas) en fonction de deux critères :

- la taille maximale allouée au cube (si on la connaît, autant la préciser) ;
- ou le gain de performance souhaité :

$$\text{gain en pourcentage} = 100 * \frac{T_{\max} - T}{T_{\max} - T_{\min}}$$

où $T_{\max}$ est le temps d'exécution d'une requête si aucun agrégat n'est stocké, $T_{\min}$ est le temps d'exécution de cette requête si tous les agrégats sont stockées et T est le temps d'exécution avec le gain souhaité.

17 Alimentation de l'entrepôt

Pour remplir un entrepôt de données, il faut :
- une étape d'extraction (des données pertinentes des les bases de production) ;
- une étape de transformation (nettoyage, formattage, premières agrégations et reconnaissance des membres) ;
- et une étape de chargement (des données propres dans la base décisionnelle).

En anglais on parle de phase ETL pour Extraction, Transformation and Loading[20].

La fréquence à laquelle les phases ETL sont opérées doit être cohérente avec le grain de la dimension temporelle et doit permettre d'historiser les données avant qu'elles ne soient purgées des bases de production.

Le remplissage initial des données à la création de l'entrepôt est généralement facile. C'est historiser périodiquement et automatiquement les données qui pose problème.

En effet, les sources de données sont généralement multiples et gérées par différents systèmes (géographiquement répartis dans différents sites), ce qui rend la phase ETL bien souvent très problématique. Chaque situation rencontrée est très spécifique et l'architecture ETL mise en place est souvent dédiée à l'entreprise.

D'ailleurs le marché des outils ETL (qui sont très nombreux), est particulièrement morcelé. Il est malgré tout dominé par Informatica (PowerMart/Center), Ascential (DataStage) et SAS (Warehouse Administrator). Suivent les fournisseurs de SGBD : Oracle (Warehouse Builder), IBM (DataWarehouse Manager) et Microsoft (DTS), au coude-à-coude avec Cognos et Hummingbird (parmi tant d'autres).

17.1 Data Transformation Services (DTS)

DTS est un outil fourni avec SQL Server. Il peut se connecter en lecture et en écriture :
- aux logiciels Microsoft, évidemment : SQL Server, Access, Excel et Visual FoxPro ;
- à d'autres SGBD : Corel Paradox, dBase, Oracle et même IBM DB2 ;
- à des fichiers textes ou des fichiers HTML.

DTS peut donc transférer les données non seulement d'une base SQL Server vers une base SQL Server, mais aussi d'une base DB2 vers une base Oracle, par exemple.

Cet outil permet non seulement de transférer les données, les nettoyer, les transformer, les fusionner et/ou les séparer. On entend par transformation tout calcul numérique ou toute opération sur chaînes de caractères par exemple. Ces transformations peuvent être programmées dans des scripts SQL, Visual Basic, Java ou Perl et donc être extrêmement complexes.

Une action DTS s'appelle un lot. Un lot DTS peut être exécuté régulièrement et automatiquement (en collaboration avec un autre service, SQL Server Agent, qui permet de planifier les opérations sur SQL Server).

Un lot peut comporter plusieurs tâches que l'on peut enchaîner (sur succès ou sur échec de la tâche précédente) et dont on peut détecter et gérer les erreurs.

Pour nous, il s'agit d'utiliser DTS comme outil d'extraction périodique de données depuis les bases de production vers la base décisionnelle. Parmi les tâches disponibles, nous intéressent plus particulièrement :
- les connexions aux données en lecture et en écriture ;

20. l'ETL fait partie des logiciels qui permettent aux applications hétérogènes de communiquer entre elle (middleware) au même titre que l'EAI (Enterprise Application Integration) qui permet de maintenir à jour les données entre les applications en temps réel

- l'exécution de scripts paramétrés (essentiellement en SQL) ;
- l'exécution de sous-lots (ce qui permet de décomposer le lot `ETL` très complexe en lots `extraction`, `transformation`, `chargement` et `traitement`, plus simples) ;
- le traitement des cubes.

DTS offre également la possibilité d'utiliser des variables globales visibles par toutes les tâches. Pour l'ETL, deux variables globales sont indispensables : la date de début et la date de fin qui délimite la période de temps à laquelle l'ETL s'applique. Typiquement, la date de début est la date de fin de la précédente exécution du lot et la date de fin est la date courante.

Pour un approfondissement de cet outil, le lecteur est invité à consulter [1] et [16].

17.2 Base tampon

Étant donné qu'elles ont lieu pendant les plages horaires peu occupées (la nuit), l'ETL des données OLTP pour le système OLAP entre en concurrence avec les sauvegardes des bases de production. Il faut donc que cette phase perturbe le moins longtemps possible les systèmes OLTP. Pour cela, il faut que :

- l'extraction prenne le moins de temps possible ;
- les transformations n'aient pas lieu en même temps que l'extraction et pas sur le même serveur que les systèmes OLTP ;

Bref, les données extraites doivent atterrir sur une autre base, appelée base tampon (staging area).

Une fois l'étape d'extraction terminée, les transformations nécessaires. peuvent être effectuées tranquillement dans la base tampon.

Il ne faut pas non plus que le système OLAP ne soit perturbé par la phase ETL (en particulier, par l'étape de transformation). Autrement dit, cette base tampon ne doit pas être la base décisionnelle et doit être gérée par un serveur dédié à l'ETL (cf. figure 20).

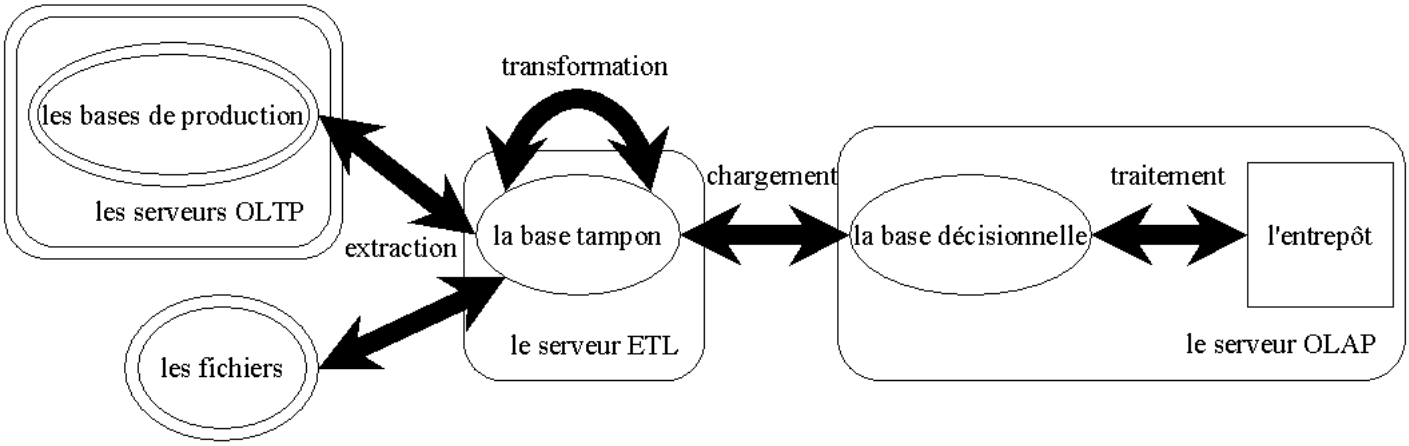

FIG. 20 – *Les étapes du processus ETL*

17.3 Étapes ETL

Détaillons maintenant les étapes de la figure 20.

17.3.1 Extraction

Pour que l'étape d'extraction dure le moins longtemps possible, il faut que :
- la requête de sélection ne comporte aucune jointure (il faut donc extraire les tables une par une) ;
- les données soient insérées dans des tables temporaires (elles n'ont aucune contrainte, aucun déclencheur et aucun index).

Par ailleurs, il est bon que dans les systèmes OLTP, chaque table concernée par l'extraction (`clients`, `produits`, etc.) soit munie d'une colonne pour la date de création et une autre pour la date de dernière modification. Sans ces colonnes, on serait obligé d'extraire toutes les lignes et il serait compliqué de déterminer (dans la base tampon) les lignes réellement modifiées depuis la dernière extraction. Avec ces colonnes, l'extraction peut être incrémentale[21].

Dans tous les cas, le volume de données à extraire est important. Il y a toujours un choix à faire entre extraire toutes les lignes d'un coup (la méthode la plus rapide, mais comme cette transaction est non atomique, la moindre erreur est fatale à tout le processus) ou les extraire une par une (ce qui prend plus de temps, mais permet de limiter l'effet d'une erreur ponctuelle).

Exemple d'extraction :

```
1  SELECT *
2  INTO clients_temporaire1
3  FROM SERVEROLTP1.BaseProduction1.dbo.clients AS a
4  WHERE a.derniere_modification BETWEEN @debut AND @fin
5
6  SELECT *
7  INTO clients_temporaire2
8  FROM SERVEROLTP2.BaseProduction2.dbo.clients AS a
9  WHERE a.last_modification_date BETWEEN @debut AND @fin
10
11 SELECT *
12 INTO commandes_temporaire1
13 FROM SERVEROLTP1.BaseProduction1.dbo.commandes AS a
14 WHERE a.date BETWEEN @debut AND @fin
```

17.3.2 Transformation

Ce n'est pas parce que les données proviennent de bases de production qui fonctionnent rigoureusement bien, que ces données sont valides pour le système décisionnel. Il faut presque toujours les transformer.

Les transformations se font en deux temps :
- d'abord, pendant le passage des données des tables temporaires aux tables tampon ;
- ensuite, des modifications sont apportées au sein des tables tampon en vue du chargement.

21. notons que l'extraction peut également être incrémentale si l'on utilise les informations contenues dans le journal des transactions

Tables tampon

À ce stade, la base tampon ne contient que des tables temporaires identiques aux tables source. L'étape de transformation consiste à consolider ces données dans les véritables tables de la base tampon (cf. figure 21).

À chaque table de la base décisionnelle correspond une table tampon qui contient :
- les colonnes de la table de dimension ou de faits correspondante ;
- les clés naturelles et les clés de substitution ;
- une colonne `exists` de type oui ou non qui dira si le membre existe déjà ou non ;

Ces tables tampon sont dépourvues de contraintes, notamment toutes ces colonnes autorisent la valeur vide.

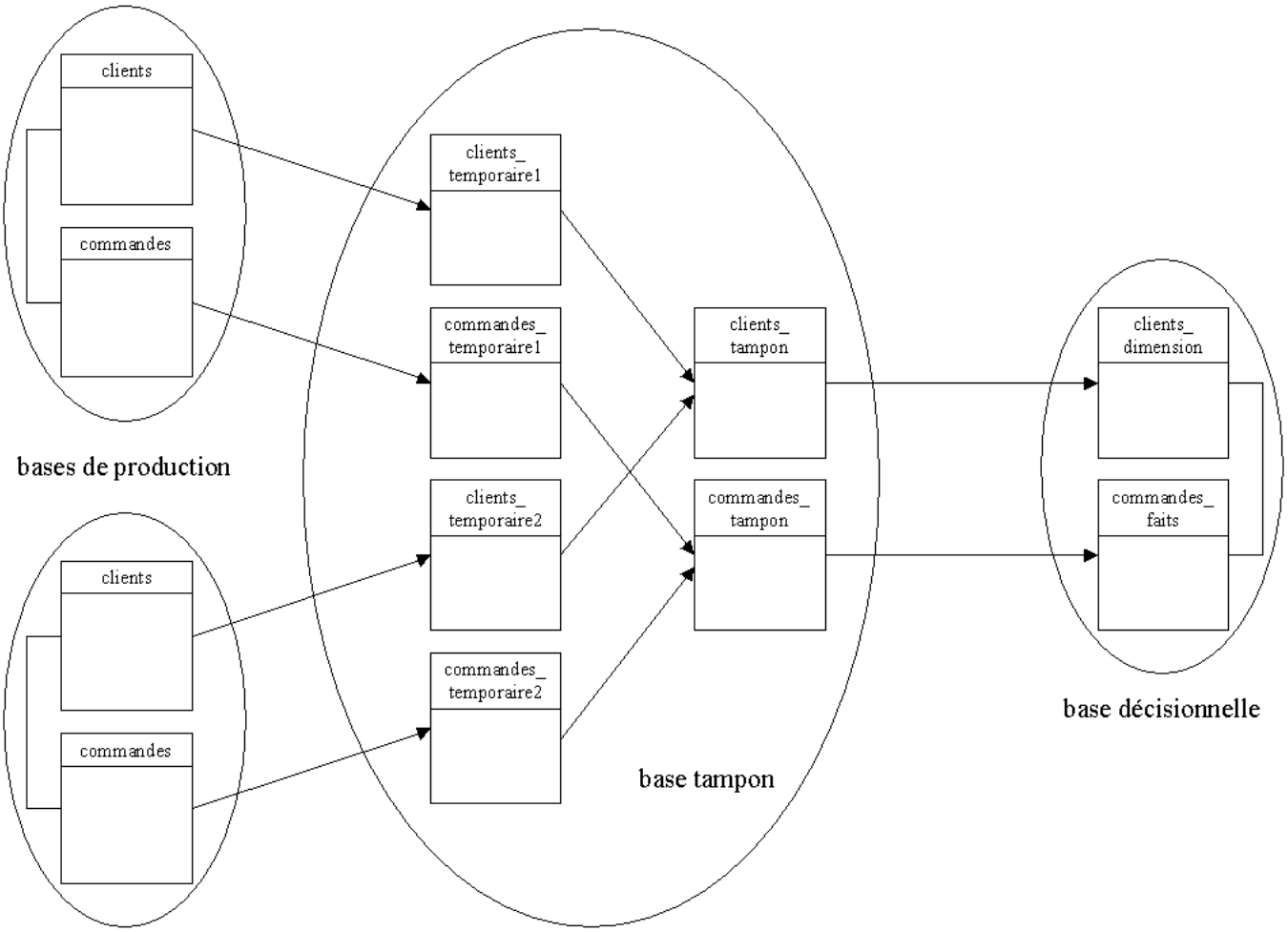

FIG. 21 – *Tables temporaires et tables tampon*

Remarques :
- pour faciliter le passage des tables temporaires aux tables tampon, il convient de supprimer, au préalable, les index sur les tables tampon ;
- comme les bases de production sont multiples, il peut y avoir plusieurs tables temporaires qui concerne les produits, par exemple ; donc l'insertion dans la table tampon qui concerne les produits se fait en plusieurs fois ;
- la base tampon est totalement dépourvue de relation car on ne peut assurer ni l'intégrité des entités ni l'intégrité référentielle, puisque les données source ne sont pas encore consolidées ;
- notons qu'il y a souvent un travail de jointure à faire sur les tables temporaires pour insérer les données dans les tables tampon ;

– notons enfin que si le grain des bases de production est plus fin que celui de la base décisionnelle, les premières agrégations nécessaires peuvent être effectuées à ce moment-là.

Réparation, complétion, synchronisation et formattage

Pendant que les données sont insérées dans les tables tampon, on peut les uniformiser, c'est-à-dire les réparer, les compléter, les synchroniser et les formatter.

Exemple de réparation des données : les codes postaux invalides peuvent être corrigés en utilisant un annuaire des codes postaux.

Exemple de complétion des données : déduire la région où est domicilié un propriétaire à partir du numéro d'immatriculation de son véhicule.

Rappelons que les bases de production n'utilisent pas forcément la même horloge. Il faut donc synchroniser toutes les dates contenues dans les tables temporaires pendant leur transfert dans les tables tampon.

Par ailleurs, quand les données arrivent dans la base tampon, elles ne sont pas toutes au même format et ne respectent pas forcément le format de la base décisionnelle (généralement, les contraintes sur les chaînes de caractères ne sont pas les mêmes dans toutes les bases et les codages de date sont hétérogènes). Il faut donc uniformiser les formats avant le chargement, c'est le formattage.

Exemple d'hétérogénéité : selon leur provenance, les noms de clients peuvent arriver sous la forme de deux colonnes (`nom` et `prenom`) ou sous la forme d'une colonne (`nom` + `prenom`, `nom` + `initiale` + `prenom`). Un autre exemple classique concerne les adresses : il y a quasiment autant de formats que de bases.

Exemple d'insertion dans une table tampon à partir d'une table temporaire :

```
1 INSERT clients_tampon
2   (cle_naturelle, source, NomPrenom, ville, region, pays, date_creation)
3 SELECT a.clt_num, 'SERVEROLTP1.BaseProduction1',  -- pour la substitution
4        a.nom + ' ' + a.prenom,                    -- formattage
5        b.ville, b.region, a.pays,                 -- completion
6        DATEADD(hour, -3, a.date_creation)         -- synchronisation
7 FROM clients_temporaire1 AS a
8 JOIN CodesPostaux AS b ON (a.CodePostal = b.CodePostal)
```

Notons qu'un client peut être inséré en double dans sa table tampon, s'il figure dans deux bases de production. Les doublons sont gérés par le paragraphe suivant. De plus, deux clients différents mais qui portent la même clé naturelle, sont distinguables par leur `source` (et leurs attributs).

Substitution des clés primaires

Une fois que les tables tampons sont remplies, on peut supprimer les tables temporaires et s'occuper de l'intégrité des données qui vont être chargées dans la base décisionnelle. Pour rendre la substitution, la validation et le chargement le plus rapide possible, il convient de re-créer les index sur les tables tampons.

Rappelons que la base décisionnelle n'utilise pas les clés naturelles des bases de production car :
– un produit peut être identifié dans deux bases de production différentes avec des clés distinctes ;
– un numéro de produit peut correspondre à deux produits distincts dans deux bases de production différentes.

Au contraire, pour identifier les membres de manière unique, la base décisionnelle utilise des clés de substitution (surrogate keys).

Au cours de l'étape de transformation, il faut donc traduire (lookup) les clés naturelles en clés de substitution et remplir la colonne `exists`. Si les bases de production ne sont pas pourvues d'une colonne pour la date de création et une colonne pour la date de dernière modification, alors cette traduction implique la recherche du membre correspondant (en fonction de leurs attributs dans la base décisionnelle) pour chaque ligne extraite. Non seulement c'est très coûteux mais en plus, cela perturbe le système OLAP.

Exemple de lookup pour les clés primaires de substitution :

```
1  -- recherche des membres deja presents
2  UPDATE clients_tampon
3  SET exists = 1, cle_substitution = b.cle
4  FROM clients_tampon AS a
5  JOIN SERVEROLAP.BaseDecisionnelle.dbo.clients_dimension AS b ON
6  (
7    ISNULL(a.NomPrenom, '') = ISNULL(b.NomPrenom, '') AND
8    ISNULL(a.pays, '')      = ISNULL(b.pays, '') AND
9    ISNULL(a.region, '')    = ISNULL(b.region, '') AND
10   ISNULL(a.ville, '')     = ISNULL(b.ville, '')
11 ) -- pays et region sont utiles pour distinguer les villes homonymes
12
13 -- nouvelle cle pour les nouveaux membres
14 UPDATE clients_tampon
15 SET cle_substitution = NEWID()
16 WHERE exists IS NULL
```

Notons que les clients insérés en double dans la table `clients_tampon`, possèdent désormais une clé de substitution unique (sauf les nouveaux membres) et que les clients distincts mais avec la même clé primaire, possèdent désormais deux clés de substitution distinctes.

Dérive dimensionnelle

Notons aussi, que si un client change de ville, il faut faire un choix entre :
- changer la ville de ce membre (et donc changer les agrégats de cette ville), ce qui n'est pas recommandé ;
- ou insérer un nouveau membre pour ce client et sa nouvelle ville (c'est le cas dans l'exemple précédent).

On peut en venir à introduire un numéro de version à chaque membre et une date de création pour chaque version. La clé primaire de la table de dimension est alors composée de deux colonnes : la clé de substitution et le numéro de version. Ça n'est pas la solution que nous avons retenue ici.

Substitution des clés étrangères

Il reste encore à traiter l'intégrité référentielle des données qui vont être chargées dans la base décisionnelle. Pour cela, il faut recalculer les clés étrangères avec les clés de substitution afin que les relations de la base décisionnelle soient vérifiées lors du chargement (cf. figure 22).

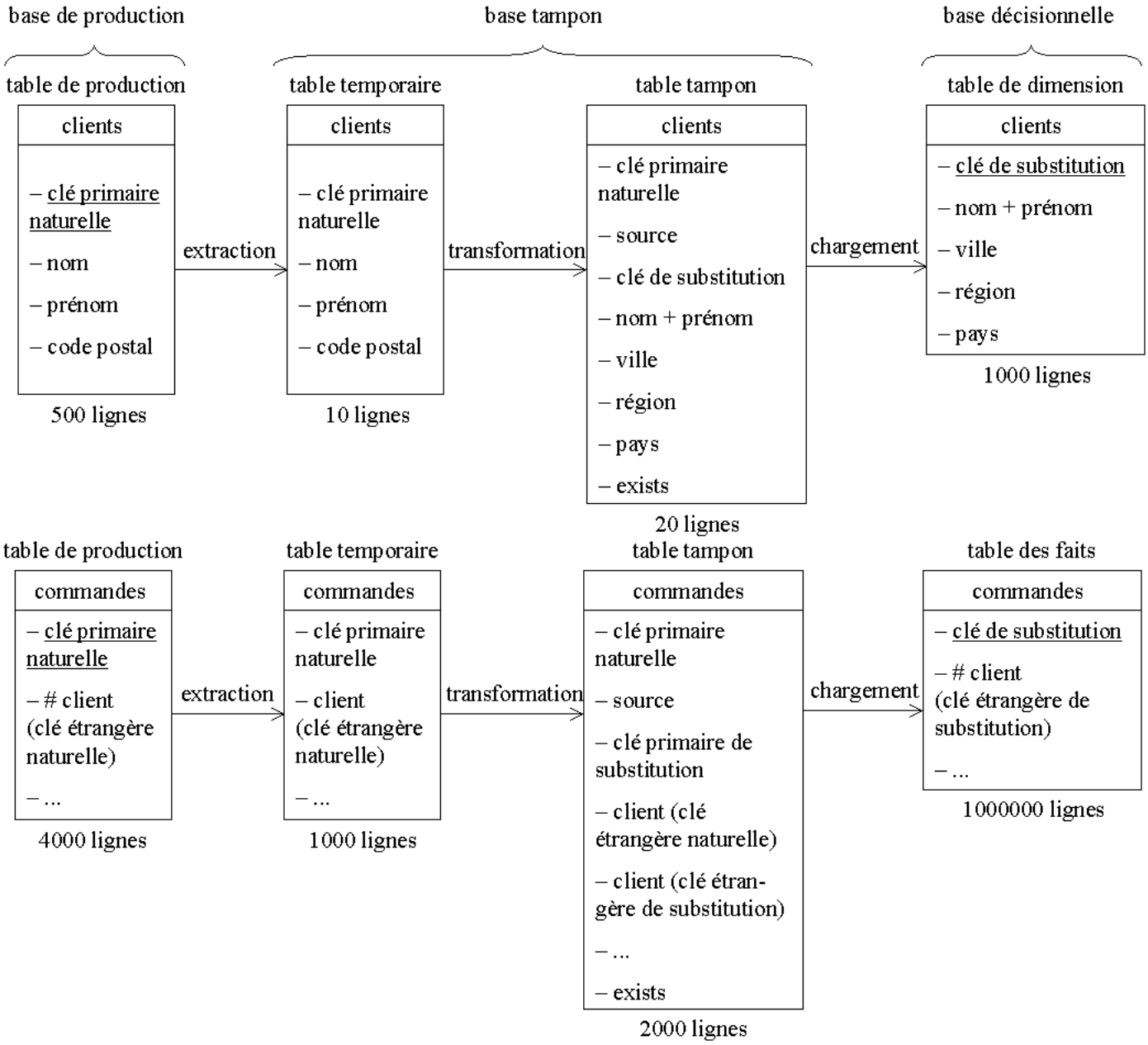

FIG. 22 – *Évolution des tables* `clients` *et* `commandes` *au cours du processus ETL*

Malheureusement, certaines clés étrangères peuvent être introuvables. Dans toutes les tables de dimension, il faut donc prévoir un membre 0 qui permet de traiter ces relations invalides[22]. Ce membre 0 est créé une bonne fois pour toutes dans chaque table de dimension de la base décisionnelle :

```
1 -- sur le server OLAP et dans la base decisionnelle
2 INSERT clients_dimension (cle, NomPrenom, pays, region, ville)
3 VALUES (0, 'autre', 'autre', 'autre', 'autre')
```

22. la numérotation des membres valides doit donc commencer à 1

Exemple de lookup pour les clés étrangères de substitution (dans la base tampon) :

```
-- traitement des relations valides
UPDATE commandes_tampon
SET client_cle_substitution = b.cle_substitution
FROM commandes_tampon AS a
JOIN clients_tampon AS b ON
(
  a.cmd_clt = b.clt_num AND
  a.source  = b.source
)

-- traitement des relations invalides
UPDATE commandes_tampon
SET client_cle_substitution = 0
WHERE client_cle_substitution IS NULL
```

Remarque : la phase de substitution est plus simple pour un schéma en étoile que pour un schéma en flocon. Il faut en tenir compte lors de la conception de la base décisionnelle.

17.3.3 Chargement

Comme les données sont chargées dans la base décisionnelle qui est muni d'un schéma relationnel, il faut charger ses tables dans cet ordre :
- d'abord les tables qui ne contiennent aucune clé étrangère ;
- ensuite les tables qui ne contiennent que des clés étrangères vers des tables déjà chargées ;
- etc.

Ensuite, pour chaque table, le chargement de décompose en deux requêtes :
- une pour les nouveaux membres ou faits ;
- et une pour les membres ou faits modifiés.

Exemple de chargement dans une table de dimension :

```
-- chargement des nouveaux membres
INSERT SERVEROLAP.BaseDecisionnelle.dbo.clients_dimension
SELECT cle_substitution, NomPrenom, ville, region, pays, age, profession
FROM clients_tampon
WHERE exists IS NULL

-- modification des anciens membres
UPDATE SERVEROLAP.BaseDecisionnelle.dbo.clients_dimension
SET age        = a.age,
    profession = a.profession
FROM clients_tampon AS a
JOIN SERVEROLAP.BaseDecisionnelle.dbo.clients_dimension AS b
  ON (a.cle_substitution = b.cle)
WHERE a.exists = 1
```

Exemple de chargement dans une table des faits :

```
-- chargement des nouveaux faits
INSERT SERVEROLAP.BaseDecisionnelle.dbo.commandes_faits
SELECT cle_substitution, client_cle_substitution, date
FROM commandes_tampon
WHERE exists IS NULL

-- modification des anciens membres
UPDATE SERVEROLAP.BaseDecisionnelle.dbo.commandes_faits
SET client_cle_substitution = a.client_cle_substitution
    date                    = a.date
FROM commandes_tampon AS a
JOIN SERVEROLAP.BaseDecisionnelle.dbo.commandes_faits AS b
  ON (a.cle_substitution = b.cle)
WHERE a.exists = 1
```

17.3.4 Traitement

Le traitement des cubes intervient juste derrière la phase ETL proprement dite. Il d'agit d'une opération entièrement réalisée par Analysis Services, mais il s'agit d'une tâche que l'on peut inclure dans le même lot DTS que l'ETL.

Hormis pour le remplissage initial de l'entrepôt, le traitement des cubes doit être incrémental. Pour cela, il est conseillé de partitionner le cube avec notamment une partition séparée pour l'année en cours, afin de ne pas traiter les autres années (qui, normalement, ne sont pas touchées par les modifications).

18 Interroger un cube

Avant de découvrir comment consulter un cube, définissons quelques opération de navigation dans un cube :

- le dépliage (drilldown) qui consiste à passer à un niveau inférieur dans une dimension ;
- le pliage (drillup ou rollup) qui consiste à passer à un niveau supérieur dans une dimension ;
- le tranchage ou découpage (slice) qui consiste à ne garder qu'une tranche du cube (une tranche étant une coupe du cube selon un membre) ;
- le cubage (dice) qui s'intéresse à plusieurs tranches à la fois ;
- et la rotation qui consiste à choisir les dimensions que l'on veut en colonnes et en lignes.

Avec ces cinq opérations, on peut naviguer dans les données (*i.e.* pratiquer le data surfing). Il faut y ajouter une sixième opération : le drillthrough (traduit maladroitement par extraction dans Analysis Services, le terme de désagrégration étant préférable) qui permet de retrouver les données qui ont permis de calculer un agrégat.

18.1 Requêtes MDX

La syntaxe pour rédiger une requête MDX est la suivante :

`WITH` certaines notations
`SELECT` les colonnes, les lignes et les autres axes
`FROM` le cube
`WHERE` les dimensions tranchées et leur tranche (au singulier)
`CELL PROPERTIES` les informations contextuelles voulues pour les cellules

Le résultat d'une telle requête est un tableau multidimensionnel dont les cellules ne contiennent qu'une valeur (contrairement aux cubes) et dans lequel on peut naviguer avec les opérations décrites précédemment.

Parmi les dimensions du cube initial, certaines servent d'axes (un axe contient les membres issus d'un dépliage et/ou d'un cubage) pour le résultat (clause `SELECT`) et d'autres de tranchage (clause `WHERE`) mais pas les deux à la fois.

Pour MDX, les mesures constituent une dimension, et peuvent donc faire l'objet d'un axe ou d'un tranchage.

Affichons par exemple, le montant des ventes, avec en colonne l'année `2002` et en ligne la région `PACA` :

```
1 SELECT {temps.annee.[2002]} ON COLUMNS,
2        {geographie.region.PACA} ON ROWS
3 FROM ventes
4 WHERE (measures.montant)
```

Le résultat de cette requête est le suivant :

	2002
PACA	56986.12

Remarques :
- n'oublier ni les accolades dans la clause `SELECT` ni les parenthèses dans la clause `WHERE` ;
- rappel : dès qu'un intitulé contient une espace, un accent ou commence par un chiffre, il faut le délimiter par des crochets ;
- si l'intitulé utilise un crochet fermant `]`, employer les doubles crochets fermants :
  `[Bill [William]] Clinton]`
- pour des raisons qui deviendront claires au §18.4.2 page 93, si l'intitulé contient une quote `'`, alors il faut la doubler :
  `[k''s Choice]`
- la syntaxe complète des noms de membres dans un cube est :
  dimension . niveau . membre
- s'il n'y a pas de confusion possible on peut omettre la dimension et/ou le niveau ;
- si dans un même niveau plusieurs membres ont le même nom, préciser autant de parents que nécessaire pour les distinguer :
  dimension . [ancêtre le plus ancien] ... [grand-père] . [père] . membre

18.1.1 Clause SELECT

La clause SELECT offre plusieurs possibilités. On peut afficher :
– plusieurs colonnes et plusieurs lignes :

```
1 SELECT {temps.annee.[1998], temps.annee.[2002]} ON COLUMNS,
2        {geographie.region.PACA, geographie.pays.France} ON ROWS
3 FROM ventes
4 WHERE (measures.montant)
```

Résultat : 2 colonnes et 2 lignes

	1998	2002
PACA	133419.96	56986.12
France	1458311.45	248996.54

– tous les enfants d'un membre :

```
1 SELECT temps.annee.[1998].CHILDREN ON COLUMNS,
2        {geographie.region.PACA} ON ROWS
3 FROM ventes
4 WHERE (measures.montant)
```

Résultat : autant de colonnes qu'il y a d'enfants

	Janvier	Février	Mars	Avril	...	Octobre	Novembre	Décembre
PACA	19856.45	11458.58	7589.47	8799.15	...	11589.45	10569.65	38360.35

– tous les membres d'un niveau :

```
1 SELECT temps.annee.MEMBERS ON COLUMNS,
2        {geographie.region.PACA} ON ROWS
3 FROM ventes
4 WHERE (measures.montant)
```

Résultat : autant de colonnes qu'il y a de membres

	1998	1999	2000	2001	2002
PACA	133419.96	121598.45	104789.56	89634.25	56986.12

– une plage de membres :

```
1 SELECT {temps.annee.[1998] : temps.annee.[2001]} ON COLUMNS,
2        {geographie.region.PACA} ON ROWS
3 FROM ventes
4 WHERE (measures.montant)
```

Résultat : autant de colonnes que d'années entre 1998 et 2001 (inclus)

	1998	1999	2000	2001
PACA	133419.96	121598.45	104789.56	89634.25

Remarques :
– il n'y a pas de confusion possible entre `MEMBERS` et `CHILDREN` puisque l'un s'applique à un niveau, l'autre à un membre :

dimension . niveau . `MEMBERS`
dimension . niveau . membre . `CHILDREN`

– on peut aussi utiliser :

dimension . `MEMBERS` (les membres de tous les niveaux de la dimension)
dimension . `CHILDREN` (les membres du niveau le plus élevé)

– avant d'utiliser l'opérateur `:`, s'assurer de l'ordre dans lequel sont stockés les membres.

18.1.2 Mesures

On peut afficher plusieurs mesures à la fois. Mais comme le résultat d'une requête n'autorise qu'une valeur par cellule, il faut aligner les mesures selon un axe :

```
1 SELECT temps.annee.MEMBERS ON COLUMNS,
2        {measures.montant, measures.NbCommandes} ON ROWS
3 FROM ventes
```

Résultat : sur tous les articles et dans tous les pays

	1998	1999	2000	2001	2002
montant	1133419.96	1121598.45	1104789.56	189634.25	156986.12
nb	2569	2107	1568	1474	978

Les mesures forment donc naturellement une dimension nommée `measures` dans chaque cube. On précise quelle mesure afficher dans la clause `WHERE` quand on en veut qu'une (c'est une tranche du cube). Et on est obligé d'en préciser au moins une, sans quoi une mesure est choisie par défaut.

18.1.3 Clause `WHERE`

On peut effectuer plusieurs découpes sur le cube. Reprenons par exemple la requête précédente :

```
1 SELECT temps.annee.MEMBERS ON COLUMNS,
2        {measures.montant, measures.NbCommandes} ON ROWS
3 FROM ventes
4 WHERE (produits.marque.Channel, geographie.pays.Italie)
```

Résultat : sur tous les articles de la marque Channel et pour l'Italie seulement

	1998	1999	2000	2001	2002
montant	419.96	598.45	789.56	634.25	986.12
nb	69	107	68	74	78

Remarques :
– une dimension ne peut apparaître qu'une fois dans la clause `WHERE` ;
– si on veut plusieurs tranches dans une dimension, il faut en faire un axe ;
– on ne peut utiliser dans la clause `WHERE` ni `MEMBERS` ni `CHILDREN`.

Le grand danger relatif aux découpes de cube, est que certaines applications MDX n'affichent pas les tranches alors que c'est une information indispensable pour comprendre les valeurs des mesures.

18.1.4 Description des axes

Une requête MDX autorise jusqu'à 127 axes, mais évidemment on ne peut pas dépasser le nombre de dimensions du cube +1 (avec la dimension `measures`). Les cinq premiers axes sont : `COLUMNS`, `ROWS`, `PAGES`, `SECTIONS` et `CHAPTERS`. Au-delà il faut utiliser : `AXIS(5)`, ..., `AXIS(126)`.

Exemple à quatre dimensions :

```
1 SELECT temps.annee.MEMBERS ON COLUMNS,
2        geographie.pays.MEMBERS ON ROWS,
3        produits.article.MEMBERS ON PAGES,
4        {measures.montant, measures.NbCommandes} ON SECTIONS
5 FROM ventes
```

On peut aussi créer un résultat ayant un seul axe :

```
1 SELECT temps.annee.MEMBERS ON COLUMNS
2 FROM ventes
3 WHERE (measures.montant)
```

Résultat : les intitulés de colonne et une seule ligne de résultats (pas d'intitulé de ligne)

1998	1999	2000	2001	2002
1133419.96	1121598.45	1104789.56	189634.25	156986.12

Ou même n'ayant aucun axe :

```
1 SELECT
2 FROM ventes
3 WHERE (measures.montant)
```

Résultat : une cellule contenant le montant total des ventes de toutes les années pour tous les produits et partout (c'est un mauvais exemple)

```
3706428.34
```

18.1.5 MDX *vs.* SQL

Comme on vient de le voir, les requêtes MDX ressemblent beaucoup aux requêtes SQL de sélection. Ceci dit, notons quand même des différences fondamentales :

- la syntaxe des clauses `SELECT` et `WHERE` n'a rien à voir ;
- la clause `FROM` n'admet qu'un seul cube, en MDX ;
- les clauses SQL `GROUP BY`, `HAVING` et `ORDER BY` n'existent pas en MDX ;
- et il existe d'autres clauses spécifiques à MDX (`WITH` et `CELL PROPERTIES`).

18.2 Filtrage des données

Jusqu'à maintenant, la clause `SELECT` n'a servi qu'à sélectionner les membres que l'on peut désigner. Si on veut faire intervenir un critère plus complexe, comme : ne garder que les membres dont une mesure est dans une certaine plage, alors il faut utiliser la fonction MDX :

`FILTER`(les membres à filter, (le critère))

Par exemple, pour n'afficher que les années pour lesquelles on a vendu pour plus de 100000 :

```
1 SELECT FILTER(temps.annee.MEMBERS, (measures.montant > 113000)) ON COLUMNS
2 FROM ventes
3 WHERE (measures.montant)
```

Résultat : il ne reste que les années concernées par le critère

1998	2001	2002
1133419.96	189634.25	156986.12

Remarques :
– la mesure dans la clause `WHERE` n'est pas obligatoirement la même que celle utilisée dans le filtre ;
– le critère peut faire appel aux fonctions MDX (cf. aide en ligne).

Exemple : pour ne garder que les pays qui ont vendu plus que la France, l'Italie et l'Allemagne

```
1 SELECT FILTER(geographie.pays.MEMBERS, (measures.montant >
2   MAX({France, Italie, Allemagne}, measures.montant)          -- ici
3   )) ON COLUMNS
4 FROM ventes
5 WHERE (measures.montant)
```

Remarque : MDX offre toutes les fonctions d'agrégat (mais avec deux arguments) ainsi que `MEDIAN`.

Si le critère ne porte pas directement sur les agrégats des membres à filtrer mais sur des valeurs plus fines, alors il faut faire appel à la notion de tuple pour décrire ces valeurs. Le tuple est la généralisation de la notion de couple et de triplet, il correspond aux coordonnées des valeurs voulues dans le cube.

Exemple de tuple : le montant des ventes pour le mois de janvier 2002 se note par le couple :

`([2002].janvier, measures.montant)`

Autre exemple : le nombre de commandes du client `Razibus` en France et en 1999 se note par le quadruplet :

`(clients.Razibus, geographie.France, temps.[1999], measures.NbCommandes)`

Remarques :
– l'ordre a parfois de l'importance dans un tuple ;
– les composantes d'un tuple doivent être issues de dimensions différentes.

Avec cette notation, on peut par exemple filtrer les articles pour lesquels les ventes ont augmenté entre janvier 2001 et janvier 2002 :

```
SELECT FILTER(produits.article.MEMBERS,
  (([2001].janvier, measures.montant) < ([2002].janvier, measures.montant))
  ) ON COLUMNS
FROM ventes
WHERE (measures.NbCommandes) -- on n'est pas oblige d'afficher le montant
```

Remarque : les membres à filtrer peuvent être définis en plusieurs fois. Par exemple, pour filtrer les mois de 2001 et de 2002 pendant lesquels il a été vendu pour un montant supérieur à 10 000 :

```
SELECT FILTER({[2001].CHILDREN, [2002].CHILDREN},
  (measures.montant > 10000)) ON COLUMNS
...
```

Remarque : on peut afficher les deux montants utilisés dans le filtre

```
SELECT FILTER(produits.article.MEMBERS,
  (([2001].janvier, measures.montant) < ([2002].janvier, measures.montant))
  ) ON COLUMNS,
  {([2001].janvier, measures.montant), ([2002].janvier, measures.montant)}
    ON ROWS
FROM ventes
```

Compléments

Sans utiliser la fonction `FILTER`, on peut éliminer simplement les membres qui ne contiennent pas de données :

`SELECT NON EMPTY {...} ON COLUMNS`

Par ailleurs, on peut ne garder que les membres extremaux avec la fonction :

`TOPCOUNT(`les membres à filtrer`,` le nombre voulu`,` `(`la mesure pour établir le classement`))`

Filtrons par exemple les 10 articles les plus vendus :

```
SELECT TOPCOUNT(article.MEMBERS, 10, (measures.NbCommandes))
  ON COLUMNS
...
```

Remarques :
- là aussi, on peut décrire les membres à filtrer en plusieurs fois (en utilisant des accolades) ;
- là aussi, on peut employer un tuple en troisième argument ; exemple : les 10 articles les plus vendus en 1998 :

```
SELECT TOPCOUNT(article.MEMBERS, 10, (measures.NbCommandes, [1998]))
  ON COLUMNS
...
```

- on a évidemment la fonction `BOTTOMCOUNT` (même syntaxe) ;
- il existe aussi `TOPSUM` et `TOPPERCENT` (cf. l'aide en ligne).

18.3 Disposition des résultats

On peut encore rendre les requêtes MDX plus complexe lorsque l'on chercher à réorganiser les résultats.

18.3.1 Ordonner les axes

Pour ordonner les membres dans un axe et selon une mesure (et non pas par ordre alphabétique), on utilise dans la clause `SELECT` la fonction :

`ORDER`(les membres à trier, (la mesure selon laquelle trier), `ASC` ou `DESC`)

Exemple, pour ordonner les années de la plus lucrative à la moins lucrative :

```
1 SELECT ORDER(annee.MEMBERS, (measures.montant), DESC) ON COLUMNS
2 ...
```

Remarques :
- on peut faire appel aux tuples pour désigner le critère de tri ; exemple, les années de la plus lucrative à la moins lucrative en France :

```
1 SELECT ORDER(annee.MEMBERS, (measures.montant, pays.France), DESC) ON COLUMNS
2 ...
```

- à nouveau, on peut décrire les membres à trier en plusieurs fois (en utilisant des accolades). Par exemple, trions les mois de 2000 et de 2001 selon le montant des ventes de pinceaux :

```
1 SELECT ORDER({[2000].CHILDREN, [2001].CHILDREN},
2   (produits.pinceau, measures.montant), ASC) ON COLUMNS
3 ...
```

Mais le problème avec `ASC` et `DESC` est que la hiérarchie est respectée (dans notre exemple, les mois de 2000 seront triés séparemment des mois de 2001). Pour ne pas tenir compte de la hiérarchie, il suffit d'utiliser `BASC` et `BDESC`.

18.3.2 Axes pluridimensionnels

On a parfois besoin que le résultat ne comporte que deux axes (ne serait-ce que pour l'imprimer), sans pour autant perdre la possibilité d'utiliser 3 dimensions ou plus dans la clause `SELECT`. La solution consiste à représenter plusieurs dimensions par axe. Pour cela, on utilise les tuples.

Exemple, présentons à la fois les années et les pays en colonne :

```
1 SELECT {(France,[2000]), (France,[2001]),
2         (Italie,[2000]), (Italie,[2001])} ON COLUMNS
3 ...
```

Résultat : certains intitulés sont multicolonnes

	France		Italie	
	2000	2001	2000	2001

Remarques :
– dans ce cas, l'ordre à l'intérieur de chaque tuple a de l'importance ;
– les tuples doivent être homogènes (c'est-à-dire présenter les mêmes dimensions et dans le même ordre).

Si on veut ensuite détaillé chaque année selon les produits `A`, `B` et `C`, on voit tout de suite que la clause `SELECT` serait longue à écrire (12 triplets). Heureusement, on peut générer les tuples par produit cartésien :

```
1 SELECT CROSSJOIN({France, Italie}, {[2000], [2001]}) ON COLUMNS
2 ...
3 -- donne la meme chose que precedemment
4
5 SELECT CROSSJOIN(CROSSJOIN({France, Italie}, {[2000], [2001]}),
6                  {A, B, C}) ON COLUMNS
7 ...
```

Résultat : trois dimensions sur l'axe des colonnes

	France						Italie					
	2000			2001			2000			2001		
	A	B	C	A	B	C	A	B	C	A	B	C

18.4 Clause `WITH`

Comme on l'a déjà vu, les requêtes MDX présentent une dernière clause, la clause `WITH` qui offre la possibilité de définir certains objets avant le début du `SELECT`.

Les objets définis dans une clause `WITH` ne sont visibles que dans la clause `SELECT` qui suit. C'est pourquoi Analysis Services propose également de les définir dans l'éditeur de cube afin qu'ils soient utilisables par toutes les requêtes.

Comme la syntaxe est la même, nous nous contentons ici d'utiliser la clause `WITH`.

18.4.1 Membres calculés

Un membre calculé est un membre supplémentaire dont la définition repose sur les membres déjà présents dans le cube. Il s'agit d'un calcul entre membres dont le résultat peut être utilisé comme un membre à part entière.

Exemple : à partir des membres `juillet 2001` et `juillet 2002`, on peut définir un membre calculé qui représente la progression entre juillet 2001 et juillet 2002 ainsi :

```
1 WITH MEMBER temps.[de juillet 2001 a juillet 2002] -- le nom complet
2   AS 'temps.[2002].juillet - temps.[2001].juillet' -- l'expression du calcul
3 SELECT {temps.[2001].juillet, temps.[2002].juillet
4   temps.[de juillet 2001 a juillet 2002]} ON COLUMNS
5 FROM ventes
6 WHERE (measures.montant)
```

Résultat : la troisième colonne présente la progression du chiffre d'affaire entre juillet 2001 et juillet 2002

juillet 2001	juillet 2002	de juillet 2001 a juillet 2002
10186.12	9486.78	-699.35

Jusqu'ici on s'est contenté d'utiliser les mesures brutes (tel qu'elles sont stockées dans le cube). Si on veut afficher des mesures plus complexes, il suffit de définir un membre calculé en fonction d'autres mesures.

Si, par exemple, on dispose des mesures `montant` et `quantite` alors on peut définir le membre calculé `prix_unitaire` :

```
1 WITH MEMBER measures.prix_unitaire    -- le nom complet
2   AS 'measures.montant / measures.quantite' -- l'expression du calcul
3 SELECT article.MEMBERS ON COLUMNS
4 FROM ventes
5 WHERE (measures.prix_unitaire)
```

Les membres calculés ne sont pas retournés par défaut par la fonction `MEMBERS`, il faut utiliser la fonction `ADDCALCULATEDMEMBERS`. Si on veut voir apparaître le nouvelle mesure `prix_unitaire` par exemple :

```
1 WITH MEMBER measures.prix_unitaire
2   AS 'measures.montant / measures.nb'
3 SELECT ADDCALCULATEDMEMBERS(measures.MEMBERS) ON COLUMNS
4 FROM ventes
5 WHERE (temps.[2001])
```

Remarques :
- les valeurs des membres calculés ne sont pas stockées dans le cube, mais calculées à la volée (ce qui ralentit raisonnablement les requêtes) ;
- le membres calculés peuvent être utilisés dans la clause `WHERE` (c'est d'ailleurs la seule façon d'effectuer une tranche qui concerne plusieurs membres de base du cube).

Ordre de résolution

Il est parfois nécessaire de préciser dans quel ordre les membres calculés doivent être calculés. C'est le cas notamment lorsque l'on combine pourcentage et différence.

Considérons l'exemple suivant :

```
1 WITH MEMBER temps.[de juillet 2001 a juillet 2002]
2   AS 'temps.[2002].juillet - temps.[2001].juillet'
3   MEMBER measures.[profit (en pourcentage)]
4   AS '100 * (measures.montant - measures.cout) / measures.cout'
5 SELECT {measures.montant, measures.cout, measures[profit (\%)]} ON COLUMNS,
6   {temps.[2001].juillet, temps.[2002].juillet
7   temps.[de juillet 2001 a juillet 2002]} ON ROWS
8 FROM ventes
```

Cette requête produit le résultat suivant : la dernière cellule ne contient pas forcément ce que l'on veut

	montant	cout	profit (%)
juillet 2001	10186.12	8451.00	20
juillet 2002	9486.78	7569.50	25
de juillet 2001 a juillet 2002	-699.35	-881.5	-20

Pour obtenir la progression du profit en pourcentage il suffit de préciser que le membre calculé `[de juillet 2001 a juillet 2002]` doit être calculé après le membre calculé `[profit (%)]` :

```
1  WITH MEMBER temps.[de juillet 2001 a juillet 2002]
2    AS 'temps.[2002].juillet - temps.[2001].juillet',
3    SOLVE_ORDER = 2
4    MEMBER measures.[profit (en pourcentage)]
5    AS '100 * (measures.montant - measures.cout) / measures.cout',
6    SOLVE_ORDER = 1
7  SELECT {measures.montant, measures.cout, measures[profit (\%)]} ON COLUMNS,
8    {temps.[2001].juillet, temps.[2002].juillet
9    temps.[de juillet 2001 a juillet 2002]} ON ROWS
10 FROM ventes
```

Auquel cas nous obtenons bien le pourcentage recherché :

	montant	cout	profit (%)
juillet 2001	10186.12	8451.00	20
juillet 2002	9486.78	7569.50	25
de juillet 2001 a juillet 2002	-699.35	-881.5	5

Mise en forme des membres calculés

D'autres options sont disponibles dans la clause `MEMBER` (cf. l'aide en ligne). Comme par exemple, une description du format numérique à employer :

```
1  WITH MEMBER measures.prix_unitaire
2    AS 'measures.montant / measures.quantite',
3    FORMAT_STRING = '#.## euros'
4  SELECT ...
```

18.4.2 Jeux nommés

Si un ensemble de membres est souvent utilisés dans des requêtes MDX, il est intéressant de le définir une bonne fois pour toutes et de lui donner un nom. C'est ce que l'on appelle un jeu (sous-entendu de membres) nommé.

Si, par exemple, les 10 articles les plus vendus reviennent souvent alors on peut définir un jeu nommé ainsi :

```
1  WITH SET MeilleursArticles                                  -- le nom
2    AS 'TOPCOUNT(article.MEMBERS, 10, measures.quantite)' -- l'expression
3  SELECT MeilleursArticles ON COLUMNS
4  FROM ventes
5  WHERE (measures.nb)
```

Dans la définition d'un jeu nommé, on peut utiliser les fonctions ensemblistes `UNION`, `EXCEPT` et `INTERSECT`. Exemple :

```
WITH SET MeilleursArticles
  AS 'TOPCOUNT(article.MEMBERS, 10, (measures.quantite))'
  SET ArticlesLesPlusCher
  AS 'TOPCOUNT(article.MEMBERS, 10, (measures.prix_unitaire))'
  SET MeilleursArticlesEtArticlesLesPlusCher
  AS 'UNION(MeilleursArticles, ArticlesLesPlusCher)'
  SET MeilleursArticlesSaufLesPlusCher
  AS 'EXCEPT(MeilleursArticles, ArticlesLesPlusCher)'
  SET MeilleursArticlesQuiSoientParmisLesPlusCher
  AS 'INTERSECT(MeilleursArticles, ArticlesLesPlusCher)'
SELECT ...
```

18.4.3 Cellules calculées

Il existe un troisième objet que l'on peut définir dans la clause `WITH`, il s'agit des cellules calculées (`CELL CALCULATION`). Mais cet objet est trop complexe pour entrer dans le cadre de ce document. Le lecteur est donc dirigé vers l'aide en ligne et [12] pour découvrir cette notion.

18.4.4 Précisions

Si un intitulé comporte une quote ', alors elle doit être doublée afin de ne par interférer avec la délimitation de l'expression :

```
WITH MEMBER article.[Tous les albums de k''s Choice]
  AS 'SUM({[k''s Choice - Cocoon Crash], [k''s Choice - Paradise in me]})'
```

Si on désire introduire plusieurs notations dans la clause `WITH`, il suffit de les juxtaposer (les virgules sont réservées aux propriétés des membres calculés) :

```
WITH MEMBER ... AS '...'
     MEMBER ... AS '...'
     SET ... AS '...'
SELECT ...
```

18.5 Clause `CELL PROPERTIES`

Par ailleurs, on peut contrôler les propriétés de cellule que l'on veut afficher dans la fenêtre contextuelle (qui apparaît au clic droit sur une cellule) à l'aide la dernière clause des requêtes MDX. Par exemple, pour n'afficher que l'ordinal et la valeur formattée :

```
SELECT ...
FROM ...
WHERE ...
CELL PROPERTIES CELL_ORDINAL, FORMATTED_VALUE
```

18.6 Fonctions MDX

Dans ces requêtes on peut à tout moment utiliser une multitude d'autres fonctions offertes par MDX (cf. l'aide en ligne et [12]). À commencer par la fonction :

`IIF(`condition`,` si vrai`,` si faux `)`

Exemple : affichons oui ou non en deuxième colonne selon que les articles se sont vendus moins de 200 fois ou non :

```
1 WITH MEMBER measures.MauvaisArticle
2   AS 'IIF(measures.quantite < 200, "oui", "non")'
3 SELECT {measures.quantite, measures.MauvaisArticle} ON COLUMNS,
4   article.MEMBERS ON ROWS
5 FROM ventes
6 WHERE (temps.[2001])
```

MDX offre aussi la fonction `ISEMPTY` et qui permet de remplacer la valeur `NULL` par 0 (par exemple) :

```
1 WITH MEMBER measures.[quantite corrigee]
2   AS 'IIF(ISEMPTY(measures.quantite), 0, measures.quantite)
3 SELECT temps.[2003].CHILDREN ON COLUMNS, article.MEMBERS ON ROWS
4 FROM ventes
5 WHERE (measure.[quantite corrigee])
```

Remarque : dans la condition de la fonction `IIF`, on peut utiliser le mot-clé `NOT`.

Autre exemple, pour retrouver un ancêtre : sélectionner la région dans laquelle se trouve la ville de Nice

```
1 SELECT {ANCESTOR(Nice,region)} ON COLUMNS
2 ...
```

Remarques :
– le premier argument doit être un membre unique ;
– le deuxième argument est soit le niveau auquel on monte, soit le nombre de niveaux à monter.

Exemple pour retrouver les descendants : sélectionner les articles de la marque Channel

```
1 SELECT DESCENDANTS(Channel, article) ON COLUMNS
2 ...
```

Remarques :
– le premier argument doit être un membre unique ;
– le deuxième argument est soit le niveau auquel on descend, soit le nombre de niveaux à descendre.

Dernier exemple : pour définir une colonne d'agrégat qui comprend la France et l'Italie par exemple

```
1 WITH MEMBER geographie.[France et Italie]
2   AS  'AGGREGATE({France, Italie})'
3 SELECT {France, Italie, [France et Italie]} ON COLUMNS,
4        Measures.MEMBERS ON ROWS
5 FROM ventes
```

La fonction `AGGREGATE` utilise alors la fonction d'agrégation appropriée à chaque mesure.

18.7 Conclusion

On aboutit à la stratégie suivante pour l'élaboration d'une requête MDX :

1. remplir la clause `FROM` avec le cube sur lequel on travaille ;
2. définir dans la clause `WITH` les membres calculés, les jeux nommés et les cellules calculées locaux ;
3. déterminer les tranches voulues pour remplir le tuple de la clause `WHERE` ;
4. pour chaque axe de la clause `SELECT` (et dans l'ordre) :
   (a) déterminer les dimensions et les membres concernés ;
   (b) filtrer éventuellement ces membres avec `FILTER`, `NON EMPTY` et/ou `TOPCOUNT` ;
   (c) ordonner éventuellement les membres restants avec `ORDER` :
   (d) lister après le mot-clé `PROPERTIES` les propriétés de membres (cf. §19.1 page 96) que l'on veut ajouter aux propriétés de cellule.
5. lister dans la clause `CELL PROPERTIES` les propriétés de cellule (cf. §18.5 page 94) que l'on souhaite avoir à disposition.

19 Objets virtuels

Sont regroupées ici quelques notions importantes qui permettent une plus grande souplesse dans la gestion des informations.

19.1 Propriété de membre

Une propriété de membre est une information relative à un niveau, stockée dans la table de dimension concernée, mais qui ne participe pas à la hiérarchie.

Dans la dimension `temps` par exemple, l'information `lundi`, `mardi`, ..., `dimanche` relative au niveau `jour` est une propriété des membres `1`, `2`, ..., `31`. Dans la table de dimension `temps`, cette données que l'on peut appeler `JourDeLaSemaine` est stockée dans une colonne supplémentaire contenant les valeurs `1`, `2`, ..., `7` (se méfier car 1 ne correspond pas forcément au lundi).

Autres exemples de propriétés de membres :
– le coloris d'un article ;
– la population d'une ville.

Il va sans dire que pour être utilisables, ces propriétés de membres doivent être présentes dans la base décisionnelle (et donc prévues à l'avance afin que l'ETL puisse les alimenter).

Afin d'afficher une propriété de membre dans un requête MDX, il suffit de la préciser dans la zone `PROPERTIES` de la clause `SELECT`.

Exemple : on désire afficher la couleur et la taille des articles ainsi que le genre des clients :

```
1 SELECT produits.article.MEMBERS
2        PROPERTIES produits.article.couleur,
3                   produits.article.taille ON COLUMNS,
4        clients.MEMBERS
5        PROPERTIES clients.genre ON ROWS
6 FROM ventes
7 WHERE (measures.montant)
```

Dans certaines applications MDX, ces informations ne sont pas disponibles directement sur le résultat de la requête. Il faut alors cliquer droit sur la cellule désirée afin d'obtenir les propriétés de la cellule.

Mais par ailleurs, les propriétés de membres permettent de répartir les données en catégories au même titre que les niveaux. Mais pour qu'elles soient utilisées comme les niveaux il faut introduire la notion de dimension virtuelle.

19.2 Dimension virtuelle

Jusqu'à maintenant nous n'avons abordé que les dimensions physiques d'un cube. Une dimension virtuelle est une dimension purement logique, fondée sur une dimension physique, et dont les niveaux sont choisis parmi toutes les colonnes ce cette dimension, y compris les colonnes propriétés de membre.

L'introduction d'une colonne virtuelle ne modifie pas le stockage du cube. C'est simplement une nouvelle façon d'organiser les données.

Par exemple, avec une dimension virtuelle `semaine` dont le seul niveau est `JourDeLaSemaine`, on peut comparer les ventes des lundis aux ventes des mardis (ce que l'on ne pouvait pas faire avec la dimension physique `temps` sous-jacente).

Remarques :
- on peut utiliser les dimensions virtuelles dans les requêtes MDX (en tant qu'axe ou découpage) ;
- les agrégats relatifs aux dimensions virtuelles ne sont pas stockés dans le cube mais calculés à la volée ce qui induit un ralentissement des requêtes y faisant appel.

Exemple :

```
1 SELECT semaine.JourDeLaSemaine.MEMBERS ON COLUMNS
2 FROM ventes
3 WHERE (measures.montant)
```

19.3 Cube virtuel

Le cube virtuel est aux cubes, ce que la vue est aux tables, c'est-à-dire :
- soit un sous-cube ;
- soit une combinaison de plusieurs cubes (qui ont des dimensions communes) en un cube logique.

Exemples de cubes virtuels :
- le cube `MiniVentes` qui ne garde que les dimensions `temps` et `geographie` et uniquement la mesure `montant` du cube physique `ventes` ;
- si on a les cubes physiques `VentesEntreprise1` et `VentesEntreprise2`, on peut utiliser un cube virtuel pour les regrouper en un seul (mauvais exemple, car on on aurait plutôt tendance à fusionner les bases décisionnelles et les cubes physiques dans ce cas) ;
- les cubes sur les ventes, sur la production et sur l'approvisionnement peuvent être physiquement séparés et rassemblés dans un cube virtuel grâce à leurs dimensions communes (le temps et les produits, par exemple).

Remarques :
- les cubes virtuels ne contiennent que des définitions, pas de données ;
- il faut pourtant les traiter, ce traitement consiste uniquement à établir les liens internes vers les dimensions et les mesures concernées et éventuellement à déclencher le traitement des cubes sous-jacents ;
- de même que la vue, le cube virtuel est un élément :
  - de sécurité car il peut masquer aux utilisateurs les données qui ne le concernent pas ;

– et de simplicité car il permet de masquer les données inutiles et de regrouper les données utiles selon les utilisateurs ;
– une requête MDX peut porter sur un cube virtuel (clause `FROM`), c'est d'ailleurs la seule façon d'utiliser plusieurs cubes.

Exemple : le cube utilisé dans la requête donnée en exemple pour introduire l'ordre de résolution (cf. page) est vraisemblablement virtuel car les deux mesures `montant` et `cout` n'ont pas le même grain (le coût d'un produit ne dépend pas du client qui l'achète). Elles appartiennent donc à deux cubes physiques distincts (si l'entrepôt est bien conçu) qui ont pour dimensions communes `temps` et `produits`.

20 Exploration de données

En anglais on parle de data mining. C'est l'ensemble des techniques qui permettent de construire des modèles d'un entrepôt de données historisées (*i.e.* avec une dimension `temps`) afin de décrire et/ou de prédire des tendances et les règles qui les régissent.

Le marché mondial du data mining est fortement dominé par Enterprise Miner (SAS). Les autres produits disponibles sont : Clementine de SPSS, Knowledge Seeker de Angoss et Intelligent Miner de IBM. Oracle propose aussi des fonctionnalités de data mining depuis le rachat de Thinking Machines.

Il s'agit simplement ici de découvrir les quelques fonctionnalités offertes par Analysis Services.

20.1 Modèles

Il existent de nombreux algorithmes de data mining (cf. [11]), Analysis Services en offre deux :
– l'organisation en clusters (groupage) ;
– et l'arborescence de décision.

20.1.1 Clustering

Avant toute chose, on appelle classification toute technique visant à composer des classes d'objets homogènes (autrement dit pour nous : former des ensembles de données ayant des caractéristiques communes).

Par exemple si on note sur un graphe les boissons favorites de différentes personnes classées selon leur âge en abscisse et selon la boisson proposée en ordonnée, alors on pourra vraisemblablement regrouper les jeunes autour des sodas, les plus âgés autour des vins et les autres autour des bières (cf. figure 23).

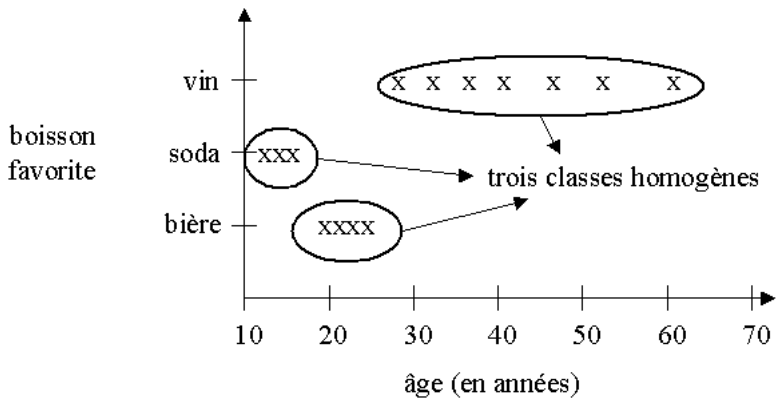

FIG. 23 – *Exemple de classification*

Le clustering est une technique de classification utilisant une fonction distance pour séparer les groupes.

Exemple de distance : la distance euclidienne dans $\mathbb{R}^M$, avec M le nombre de caractéristiques (ou variables explicatives) m (pour nous, ces caractéristiques sont des mesures ou propriétés de membre) utilisées pour distinguer les groupes. La distance séparant deux objets i et j étant alors :

$$d(i,j) = \left(\sum_m \big(m(j) - m(i)\big)^2 \right)^{1/2}$$

Exemple de groupage s'appuyant sur une distance : la méthode des k-moyennes (ou méthode des centres mobiles). On veut classer N objets en K groupes (avec $K \ll N$), pour cela :
- choisir K objets initiaux appelés centres des K-groupes (au hasard) ;
- placer chacun des N objets dans le groupe dont le centre est le plus proche (utilisation de la distance d) ;
- recalculer le centre de chaque groupe (barycentrage) ;
- et itérer les deux étapes précédentes jusqu'à ce que plus aucun objet ne change de groupe.

Une fois que les groupes ont été établis sur un échantillon représentatif, on est en mesure de mieux connaître tout nouvel objet, grâce à son appartenance à une classe homogène (dont on connaît le comportement), simplement en examinant ses caractéristiques.

Exemple d'utilisation :
- (commercial) grouper les clients afin de cibler l'envoi d'offres promotionnelles ;
- (assurances) grouper les sociétaires afin de déterminer si un nouveau client est fiable.

Remarques :
- un cluster solide est constitué d'une population significative (*i.e.* dont la tendance centrale est foncièrement différente des autres, et d'une dispersion faible) ;
- si la population d'un cluster est trop faible, il est préférable de le grouper avec un autre ;
- si le cluster est trop dispersé, il est préférable de le scinder et de relancer le processus sur les sous-groupes ;
- certains cluster peuvent être difficiles à expliquer.

20.1.2 Arbre de décision

Pour aller plus loin dans l'exploration des données, on peut essayer de déterminer des règles de comportement.

Exemple de règle : un client qui a acheté un monospace est généralement marié avec au moins deux enfants.

Le principe de fonctionnement d'un arbre de décision est le suivant :
- pour expliquer une variable, le système recherche le critère le plus pertinent et découpe la population en sous-populations possédant la même valeur pour ce critère (phase d'expansion) ;
- un nœud dans l'arbre est terminal (c'est-à-dire une feuille) si sa population ne contient plus assez d'individus pour être subdivisée ;
- les branches non pertinentes sont éliminées (phase d'élagage) ;
- le processus reprend avec les autres nœuds jusqu'à ce qu'il ne reste que des feuilles ou jusqu'à épuisement des critères.

Exemple d'arbre : la variable à expliquer est le fait d'acheter un monospace. Le premier critère trouvé par le système est le nombre d'enfants. Les branches à 2 enfants et 3 enfant ou plus peuvent être détaillées selon un deuxième critère, à savoir le fait d'être marié ou non. Les branches les plus pesantes sont alors

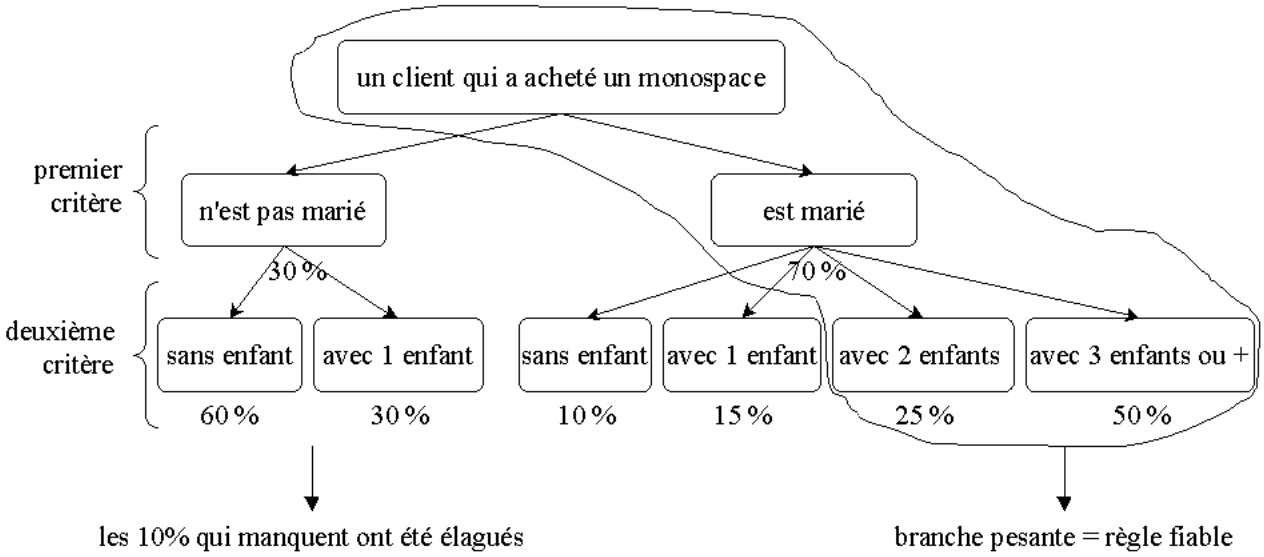

FIG. 24 – *Exemple d'arbre de décision*

pour les personnes mariées. On peut alors conclure à la règle ci-dessus.

Remarques :
- la construction d'un arbre fait appel à plusieurs notions statistiques :
  - au test du χ^2 :
  - à l'indice de pureté de Gini ;
  - à un fonction d'entropie ;
- les arbres sont généralement bien appréciés car les règles trouvées sont très explicites et la visualisation est intuitive ;
- mais l'algorithme est très coûteux.

20.2 Implémentation

Intéressons-nous maintenant à l'implémentation des ces algorithmes. D'une manière générale, il est très difficile de savoir comment sont implémentées les choses dans SQL Server. Cette section se fonde sur deux articles publiés par le centre de recherche de Microsoft (cf. [2] et [5]).

20.2.1 Vocabulaire

Les données nécessaires au data mining se présentent sous la forme d'une table ayant N lignes, M colonnes explicatives $A_1, \ldots, A_M$ (critères, caractéristiques) et une colonne à expliquer B. B est forcément qualitative[23], sinon il s'agit d'un problème de régression.

Analysis Services emploie le vocabulaire suivant :
- un cas est une des N lignes ;
- l'entité prévue est la variable à expliquer B ;
- les attributs sont les variables explicatives $A_1, \ldots, A_M$;
- et la table des cas est :

A_1	A_2	...	B
a_1	a_2		b_1
a_1	a'_2		b_2
$\vdots$	$\vdots$		$\vdots$

23. si B est continue, elle est discrétisée en plusieurs intervalles

En pratique pour nous, les colonnes $A_1, \ldots, A_M$ sont soit des mesures, soit des propriétés de membres contenues dans un cube (éventuellement virtuel). La table des cas est donc construite à partir de la table des faits en jointure avec les tables de dimension concernées par les propriétés de membres.

20.2.2 Préparation des données

Les données de la table des cas sont trop volumineuses pour entrer en mémoire centrale. En réalité, un ensemble réduit de statistiques sur ces cas suffit pour mener les algorithmes précédents. Il s'agit de la table de comptage des co-occurrences (notée CC) constituée de quatre colonnes :

colonne	valeur	classe	nombre
A_1	a_1	b_1	20
A_1	a_1	b_2	38
A_2	a_2	b_1	44
A_2	a'_2	b_2	12
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Remarques :
- la table CC est beaucoup moins volumineuse ;
- les calculs s'effectuent ensuite uniquement à partir de la table CC ;
- il est possible de construire la table CC en ne lisant les données qu'une seule fois.

Pour calculer l'arbre de décision, lorsque l'attribut le plus pertinent est A_2, le calcul du poids de la branche a'_2 utilisera les lignes de CC où $A_2 = a'_2$. Ensuite, si l'attribut suivant est A_1 le poids de la branche a_1 dans la branche a'_2 utilisera les lignes de CC où $A_2 = a'_2$ et $A_1 = a_1$.

Table des cas non pivotée

Pour remplir la table CC, il est préférable de ne pas partir directement de la table des cas, mais de mettre la table des cas sous la forme suivante (appelée `UnpivotedCases`) :

ligne	classe	colonne	valeur
1	b_1	A_1	a_1
1	b_1	A_2	a_2
2	b_2	A_1	a_1
2	b_2	A_2	a'_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$

En effet, la table CC est alors le résultat de la requête simple suivante :

```
1 SELECT colonne, valeur, classe, COUNT(*)
2 FROM UnpivotedCases
3 GROUP BY colonne, valeur, classe
```

Remarques :
– `UnpivotedCases` est calculée à partir de la table des cas grâce à un nouvel opérateur :

```
1 UnpivotedCases = Cases.UNPIVOT(valeur FOR colonne IN(A1, A2, ...))
```

– la force de cet opérateur est que la création de CC fait appel à une seule lecture des données.

Bref (cf. figure 25) :
– les calculs se font sur la table CC ;
– la table CC est calculée à partir de la table des cas non pivotée par une requête de dénombrement simple ;
– la table des cas non pivotée est issue de la table des cas grâce à l'opérateur `UNPIVOT` ;
– la table des cas est obtenue par jointure entre la table des faits et les tables de dimension.

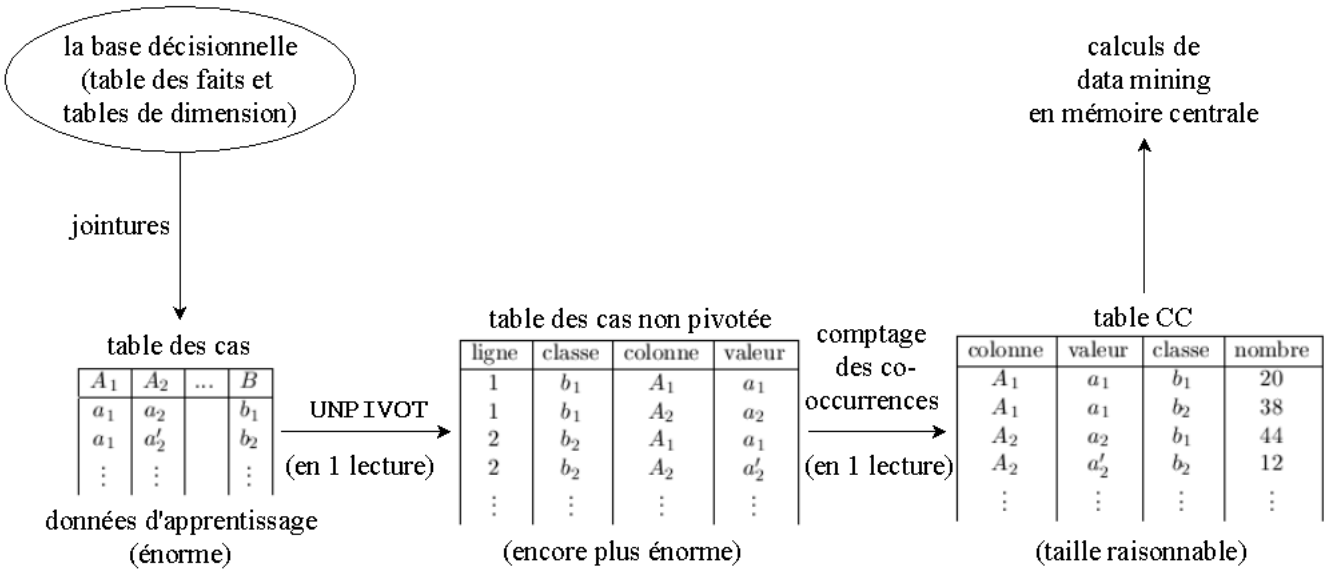

FIG. 25 – *Préparation des données pour le data mining*

20.2.3 Objets supplémentaires

Le résultat d'un modèle de mining dans un cube, constitue une nouvelle dimension de ce cube. Cette dimension est créée automatiquement et :
– pour le clustering, chaque cluster constitue un membre (un seul niveau) ;
– pour les decision trees, l'arbre correspond à la hiérarchie de cette dimension.

Cette dimension est virtuelle. Les données relatives à un modèle de mining sont alors consultables à travers un cube virtuel (créé automatiquement) regroupant le cube de départ et cette nouvelle dimension.

Exemple :

```
1 SELECT produits.monospace.MEMBERS ON COLUMNS,
2   mining_clients.[marié].CHILDREN ON ROWS
3 FROM ventes
4 WHERE (measures.NbCommandes)
```

20.3 Prédiction

Les modèles de mining précédents permettent d'extraire des tendances et des règles de nos données. Il est intéressant maintenant de se servir de ces connaissances pour prévoir les futures données.

20.3.1 Réseau de dépendances

C'est un réseau dont les nœuds sont les variables (explicatives ou à expliquer) et dont les liens sont dirigés de la variable qui prédit vers celle qui est prédite. Les liens sont d'autant plus forts que la prédiction est fiable.

Ce réseau permet de voir précisément quels facteurs sont utiles à la prédiction de tel facteur (en se basant sur les données du modèle).

20.3.2 Données prévisionnelles

Après avoir créé un modèle de mining, il est possible avec l'utilitaire DTS (cf. §17 page 75) de remplir une nouvelle colonne B avec de nouvelles valeurs pour les colonnes $A_1, \ldots, A_M$ (cf. figure 26). Il s'agit

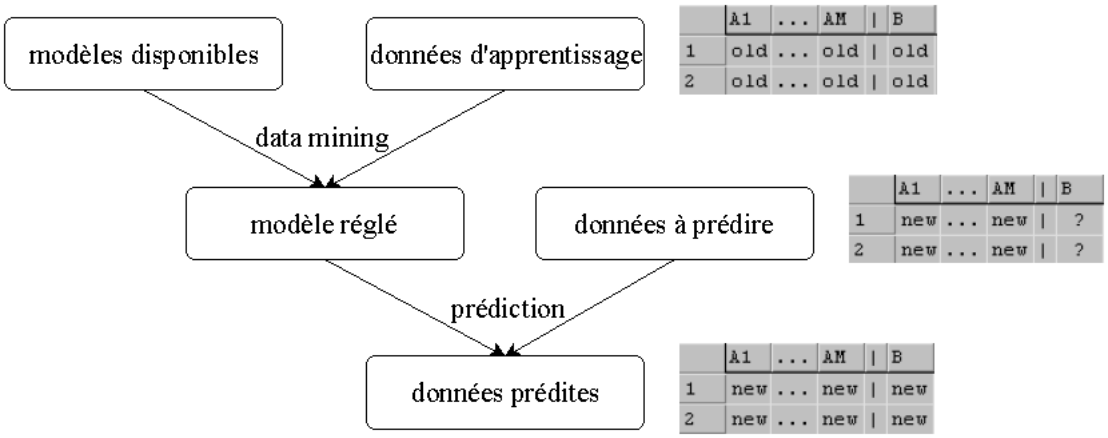

FIG. 26 – *Mécanisme de prédiction*

simplement d'une tâche que l'on peut insérer dans un lot.

20.4 Conclusion

Le data mining fait appel à de nombreux calculs de statistiques et permet d'enrichir la valeur des données contenues dans l'entrepôt. D'autres techniques (comme les réseaux neuronaux ou l'analyse de panier) seront sans doutes fournies dans les prochaines versions du logiciel.

Conclusion sur la partie décisionnelle

On peut résumer le système décisionnel ainsi (cf. figure 27) :
- les bases de production gérées en OLTP fournissent les données ;
- l'utilitaire DTS traite ces données pour alimenter les cubes OLAP (phase d'extraction) ;
- ces cubes sont là pour organiser et agréger les données pour les requêtes MDX et les modèles de mining (phase de stockage) ;
- des interfaces graphiques (que ce soit des requêteurs comme Business Objects, des tableurs comme Excel ou des SIAD pour Système Interactif d'Aide à la Décision, ou en anglais, EIS pour Executive Information System) permettent grâce à cela de sélectionner et d'explorer les données (phase de consultation) ;
- les informations et les connaissances sont alors exploitables (phase de présentation).

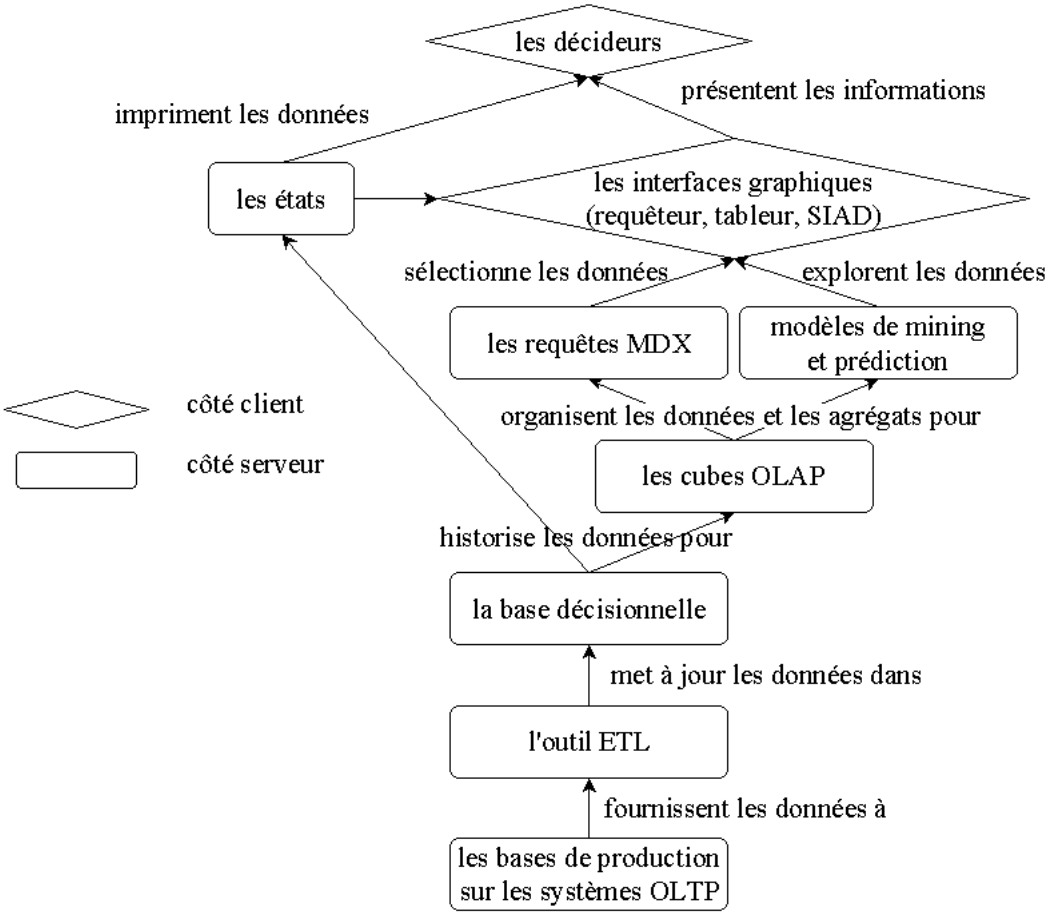

FIG. 27 – *Schéma du système décisionnel*

Dans l'entreprise les rôles concernant le système décisionnel se répartissent ainsi :
- les concepteurs de l'entrepôt de données se charge de mettre en place le schéma relationnel de la base décisionnelle, la structure des cubes et les modèles de mining ;
- les développeurs ont pour tâche, la mise en place de la phase ETL (environ 50% du temps), la programmation des requêtes MDX et des interfaces graphiques ;
- enfin, il reste aux décideurs de s'appuyer sur les résultats pour prendre les bonnes décisions.

Schématiquement, dans des entreprises comme Air France ou la SNCF (cf. figure 28) :
- le système (gigantesque) de billetterie constitue le système transactionnel de production ;
- ces données alimentent le cube des réservations ;
- des consultations MDX permettent de connaître la fréquentation des lignes ;
- tandis que des modèles de mining permettent d'établir la tendance de cette fréquentation ;
- finalement le responsable commercial pourra décider de la tarification optimale sur telle ligne à tel moment et le responsable logistique pourra augmenter ou réduire le nombre de train ou d'avion, etc.

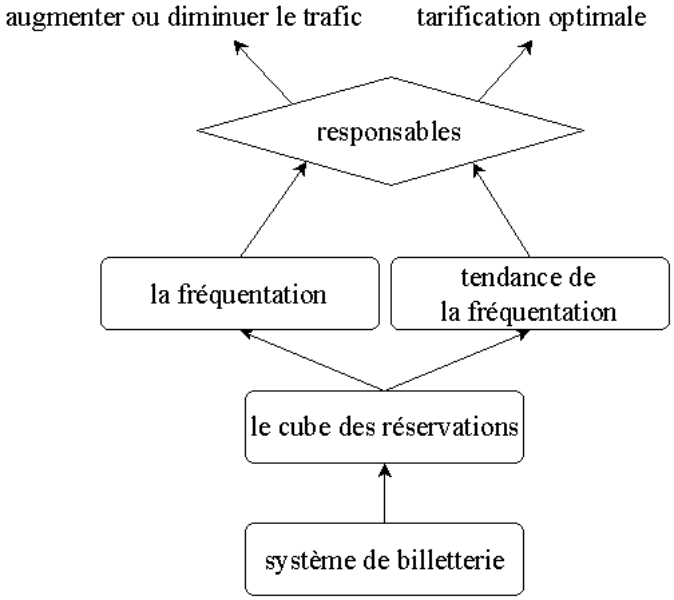

FIG. 28 – *Exemple de problématique décisionnelle*

Notons que d'autres utilisations du système décisionnel sont possibles :
- la simulation (qui permet de répondre aux questions du type « que se passerait-il si ... ? ») ;
- l'emission d'alertes automatiques (quand certains secteurs sont en perte de vitesse, par exemple) ;
- le contrôle des bases de production (le système décisionnel peut détecter certaines anomalies).

Table des figures

Références

Bibliographie

[1] Chaffin, Mark, Knight, Brian et Robinson, Todd. *Professional SQL Server 2000 DTS*. Wrox, 2000.
Cet ouvrage volumineux offre de nombreuses solutions pour gérer les transformations de données avec SQL Server.

[2] Chaudhuri, Surajit, Fayyad, Usama et Bernhardt, Jeff. *Scalable Classification over SQL Databases*. Microsoft Research, Redmond, USA, 1998
Cet article démontre la scalabilité de l'algorithme de classification de SQL Server.

[3] Gardarin, Georges. *Internet/Intranet et Bases de Données*. Eyrolles, 1999
Cet ouvrage théorique définit proprement les notions relatives au système décisionnel et les différences avec le système transactionnel.

[4] Gouarné, Jean-Marie. *Le projet décisionnel*. Eyrolles, 1998
Cet ouvrage détaille la démarche de conception d'un système décisionnel.

[5] Graefe, Goetz, Fayyad, Usama et Chaudhuri, Surajit. *On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases*. Microsoft Corporation, Redmond, USA, 1998
Cet article présente les statistiques cachées derrière le Data Mining de SQL Server ainsi que l'opérateur `UNPIVOT`.

[6] Grin, Richard. *Langage SQL*. Université de Nice Sophia-Antipolis, 1998
Ce support de cours présente la programmation SQL pour Oracle.

[7] Gunderloy, Mike et Jorden, Joseph L. *Mastering SQL Server 2000*. Sybex, 2000
Cet ouvrage volumineux constitue une référence sur l'apprentissage de SQL Server 2000.

[8] Gunderloy, Mike et Sneath, Tim. *SQL Server Developer's Guide to OLAP With Analysis Services*. Sybex, 2001
Cet ouvrage constitue une référence sur la programmation OLAP avec SQL Server 2000.

[9] Israel, Marc. *SQL Server 2000*. Eyrolles, 2001
Cet ouvrage volumineux et en français permet d'apprendre facilement à utiliser SQL Server 2000.

[10] Marée, Christian et Ledant, Guy. *SQL 2 : initiation, programmation*. Dunod, 1999
Cet ouvrage permet de découvrir la norme 92 du langage SQL.

[11] Nakache, Didier, Caulier Donneger, Anne, Rivelois Dugresson, Pascale, Dassonville, Philippe et Delebecq, Jean-Louis. *Data warehouse et data mining*. C.N.A.M. de Lille, 1998
Ce support de cours détaille le monde du décisionnel, y compris du data mining.

[12] Spofford George. *MDX Solutions*. Wiley, 2001
Ce livre en anglais constitue une référence du langage MDX et offre de nombreuses solutions pratiques.

Pages web

[13] `www.gartner.com/reprints/informatica/106602.html`
Sur ce site en anglais, les outils ETL sont classés selon l'étendue de leurs services et leur facilité d'utilisation.

[14] `www.informatica.com/news/awards/giga_etl.pdf`
Dans ce document en anglais, on trouvera notamment la repartition du marché ETL et une brève étude des différents outils ETL.

[15] `www.olapreport.com`
Les principales informations sur les produits qui se partagent le marché de l'OLAP actuellement sont regroupées sur ce site en anglais.

[16] `sqldts.com`
Ce site en anglais propose des tutoriels, une FAQ et des liens pertinents concernant les DTS.

[17] `www.swynk.com/faq/sql/sqlserverfaq.asp`
On trouvera sur cette page en anglais une FAQ correcte concernant SQL Server.

Newsgroups

[18] `comp.databases.ms-sqlserver`
Ce newsgroup anglophone répond efficacement aux problèmes posés par les utilisateurs de SQL Server.

[19] `microsoft.public[.fr].sqlserver` et `microsoft.public.sqlserver.olap`
À noter enfin l'existance de ces newsgroups maintenus par Microsoft.

Index

Symboles

A

B

C

D

E

F

G

H

I

Q

R

S

T

U

V

Z