III. Exploiter les données

III.1. Bases de l'algèbre relationnelle

III.1.1. Définitions préliminaires

Nous donnons ci-dessous une définition formelle de termes déjà rencontrés (cf. exemple figure 8) :

- *Domaine* : Ensemble des valeurs admissibles pour un composant d'une relation.
  Plusieurs attributs peuvent avoir le même domaine (ex. H_Dep et H_Arr: Heure = 00:00..23:59)
- *Relation* : Sous-ensemble du produit cartésien d'une liste de *n* domaines D_i : $R \subseteq D_1 \times D_2 \times \ldots \times D_n$, où *n* est appelé *arité* ou *degré* de la relation. En d'autres termes, une relation est une table dans laquelle chaque colonne correspond à un domaine et porte un nom, ce qui rend leur ordre sans importance. L'arité d'une relation est son nombre d'attributs, sa cardinalité est son nombre de tuples.
- *Attribut* : colonne d'une relation caractérisée par un nom unique et un domaine de définition.
- *Schéma de relation* : nom de la relation, suivi de la liste des attributs avec leurs domaines.
  Ex. VOL1(N_Vol: **NumeroVol**, Vil_Dep: **Ville**, Vil_Arr: **Ville**, H_Dep: **Heure**, H_Arr: **Heure**)
- *Base de données relationnelle* : Base de données dont le schéma est un ensemble de schémas de relations, où les données sont des tuples de ces relations, et dont la cohérence est définie par un ensemble de contraintes d'intégrité.
- *Système de gestion de bases de données relationnelles* : logiciel supportant le modèle relationnel et qui peut manipuler les données avec des opérateurs de l'algèbre relationnelle.
  Ces derniers se divisent en *opérateurs ensemblistes* et *opérateurs relationnels*.

III.1.2. Opérateurs ensemblistes

- Deux relations R et S sont dites *définies sur le même schéma* (ou encore *compatibles avec l'union*) si elles ont même nombre d'attributs (même arité) et que ceux-ci sont définis sur les mêmes domaines (le $i^{ème}$ attribut de R et le $i^{ème}$ attribut de S ont des domaines identiques, ou sont de même type).
- L'***union*** de 2 relations R et S définies sur le même schéma est une relation ($T=R \cup S$ ou T=UNION(R,S)) de même schéma contenant l'ensemble des tuples de R et S distincts (ceux qui figurent à la fois dans R et S ne sont conservés qu'une fois).
- L'***intersection*** de 2 relations R et S définies sur le même schéma est une relation ($T=R \cap S$ ou T=INTERSECT(R,S)) de même schéma et contenant les tuples communs à R et S.
- La ***différence*** de 2 relations R et S définies sur le même schéma est une relation (T=R-S ou T=R\S ou T=DIFF(R,S)) contenant l'ensemble des tuples de R qui n'appartiennent pas à S.
- Le ***produit*** *(cartésien)* de 2 relations R et S (par forcément définies sur le même schéma) est une relation ($T=R \times S$ ou T=PROD(R,S)) contenant l'ensemble de toutes les combinaisons possibles des tuples de R avec ceux de S.

VOL1	N_Vol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_Av
	IT100	Lille	Paris	7:00	7:20	100
	IT101	Paris	Lille	11:00	11:20	100
	IT102	Lyon	Paris	14:00	14:35	101

VOL2	CodeVol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_Av
	IT105	Lille	Toulouse	9:00	9:50	101
	IT101	Paris	Lille	11:00	11:20	100

VOL3=VOL1∪VOL2	N_Vol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_Av
	IT100	Lille	Paris	7:00	7:20	100
	IT101	Paris	Lille	11:00	11:20	100
	IT102	Lyon	Paris	14:00	14:35	101
	IT105	Lille	Toulouse	9:00	9:50	101

VOL3=VOL1∩VOL2	N_Vol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_Av
	IT101	Paris	Lille	11:00	11:20	100

VOL4=VOL1-VOL2	N_Vol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_A
	IT100	Lille	Paris	7:00	7:20	100
	IT102	Lyon	Paris	14:00	14:35	101

AVION	N_A	Typ_Av
	100	A300
	101	B747

VOL5=VOL2×AVION	CodeVol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_Av	N_A	Typ_Av
	IT105	Lille	Toulouse	9:00	9:50	101	100	A300
	IT105	Lille	Toulouse	9:00	9:50	101	101	B747
	IT101	Paris	Lille	11:00	11:20	100	100	A300
	IT101	Paris	Lille	11:00	11:20	100	101	B747

Figure 8: Exemples de résultats avec les opérateurs ensemblistes

III.1.3. Opérateurs relationnels

- La ***projection*** (notée $\pi_M(R)$ ou PROJECT(R,M)) est une opération unaire consistant à supprimer des attributs d'une relation et à éliminer les tuples en doubles dans la nouvelle relation. Formellement, la projection de la relation $R(A_1,A_2,\ldots,A_n)$ sur un sous-ensemble de ses attributs $M=\{A_{i1},A_{i2},\ldots A_{ip}\}$ (avec $i_j \neq i_k$ et $p<n$) produit la relation $R'(A_{i1},A_{i2},\ldots,A_{ip})$ dont les tuples sont obtenus par élimination des valeurs des attributs de R n'appartenant pas à R' et par suppression des tuples en double.

- La ***sélection*** (ou ***restriction***, notée $\sigma_F(R)$ ou RESTRICT(R,F)) est une opération unaire qui fournit comme résultat les seuls tuples de la relation R qui satisfont la condition de sélection F. Cette dernière est une formule comportant comme opérandes des attributs de R ou des constantes, liés par des opérateurs de comparaison ("<", ">", "<=", ">=", "<>" ou "=") ou logique (AND, OR et NOT).

- La ***jointure*** de deux relations R et S d'après le prédicat P (notée $R\bowtie_P S$ ou JOIN(R,S,P)) est une combinaison de tous les tuples de R avec ceux de S, qui satisfont le prédicat de jointure P. Ce dernier contient un attribut de R et un attribut de S – qui doivent être définis sur le même domaine –, liés par l'un des opérateurs de comparaison. Si cet opérateur est l'égalité, on parle d'*équi-jointure*.

  Remarque : L'opérateur de jointure $\bowtie_P$ équivaut à une restriction selon P du produit cartésien ×, c'est-à-dire que $R\bowtie_P S=\sigma_P(R\times S)$; si P est vide, on retrouve ce produit cartésien : $R\bowtie_{P=\{\}} S=R\times S$.

- La ***division*** (ou le quotient, notés R÷S) d'une relation R par une relation S n'est possible que si S est une sous-table de R. Si R a pour schéma $R(T_1,T_2,\ldots,T_j,\ldots T_m)$ et $S(T_{p+1},T_{p+2},\ldots,T_m)$, le résultat est une sous-table de R, $Q=R\div S$ de schéma $Q(T_1,T_2,\ldots,T_j,\ldots,T_p)$, formée de tous les tuples qui, concaténés à chacun des tuples de S, donnent toujours un tuple de R. En d'autre termes, le produit cartésien Q×S doit être contenu dans la table R.

VOL6=π_{Vil_Arr}(VOL3)	Vil_Arr
	Paris
	Lille
	Toulouse

VOL7=π_{Vil_Arr,Vil_Dep} (VOL5)	Vil_Arr	Vil_Dep
	Lille	Toulouse
	Paris	Lille

VOL8=$\sigma_{Vil\_Arr="Paris"}$ (VOL1)	N_Vol	Vil_Dep	Vil_Arr	H_Dep	H_Arr
	IT100	Lille	Paris	7:00	7:20
	IT102	Lyon	Paris	14:00	14:35

VOL9=$\sigma_{H\_Dep>=11:00 \text{ AND } H\_Arr<15:00}$ (VOL3)	N_Vol	Vil_Dep	Vil_Arr	H_Dep	H_Arr
	IT101	Paris	Lille	11:00	11:20
	IT102	Lyon	Paris	14:00	14:35

VOL0=VOL2$\bowtie_{N_Av=N_A}$AVION	CodeVol	Vil_Dep	Vil_Arr	H_Dep	H_Arr	N_Av	N_A	Typ_Av
	IT105	Lille	Toulouse	9:00	9:50	101	101	B747
	IT101	Paris	Lille	11:00	11:20	100	100	A300

Figure 9 : Exemples de résultats avec les opérateurs relationnels

III.2. Le langage SQL (Structured Query Language)

Nous ne présentons ici que les fonctionnalités principales de SQL. Pour des informations complémentaires, le lecteur peut se reporter au fichier fourni en annexe ainsi qu'aux ouvrages (principalement les sites internet) cités en bibliographie.

III.2.1. Introduction

Les langages utilisés dans les bases de données relationnelles se fondent sur l'algèbre relationnelle et/ou le calcul relationnel. Ce dernier est basé sur la logique des prédicats (expressions d'une condition de sélection définie par l'utilisateur). On peut distinguer 3 grandes classes de langages :

- les *langages algébriques*, basés sur l'algèbre relationnelle de CODD, dans lesquels les requêtes sont exprimées comme l'application des opérateurs relationnels sur des relations. Le langage SQL, standard pour l'interrogation des bases de données, fait partie de cette catégorie.
- les *langages basés sur le calcul relationnel de tuples*, construits à partir à partir de la logique des prédicats, et dans lesquels les variables manipulées sont des tuples (ex. : QUEL, QUEry Language).
- les *langages basés sur le calcul relationnel de domaines*, également construit à partir de la logique des prédicats, mais en faisant varier les variables sur les domaines des relations.

Les langages de manipulation de données sont dits *déclaratifs*, ou *assertionnels*, c'est-à-dire que l'on spécifie les ***propriétés*** des données que l'on manipule et pas – comme dans un langage *impératif* – comment y accéder.

Un langage d'*interrogation* de bases de données est dit *relationnel complet* s'il permet au moins l'emploi des trois opérateurs ensemblistes (union, différence et produit cartésien) et des deux opérateurs relationnels de projection et sélection. Plus généralement, outre la recherche ou sélection de données, les langages de *manipulation* de données permettent souvent :

- la création de table, l'insertion, la suppression et la modification de tuples vérifiant certains critères ;
- de réaliser des calculs sur des agrégats (ex. somme, minimum ou moyenne de valeurs d'un attribut) ;

- de présenter les données sous certains formats (ex. séquences de tris) ;
- de gérer les autorisations d'accès pour assurer la protection de la base ;
- de garantir la sécurité des données en prenant en compte l'environnement multi-utilisateurs.

SQL est un langage normalisé (dernière version : SQL:2011) permettant :

- l'interrogation de la base (clause `SELECT`)
- la manipulation de données (`UPDATE`, `INSERT`, `DELETE`)
- la définition des données (`ALTER`, `CREATE`, `DROP`)
- le contrôle des données (`GRANT`, `REVOKE`, `COMMIT`, `ROLLBACK`)

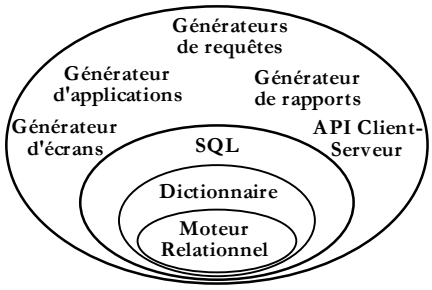

Figure 10 : SQL au sein d'un SGBD

Tout SGBD contient un *dictionnaire* (figure 10) qui contient à tout moment le descriptif complet des données, sous la forme d'un ensemble de tables comprenant les bases de données présentes, les tables et leurs attributs, les clés candidates, primaires et étrangères, les différentes contraintes et les vues créées sur les tables. SQL utilise ce dictionnaire lors de l'exécution de requêtes pour vérifier les contraintes, optimiser les requêtes et assurer le contrôle des accès.

III.2.2. Interroger une base de données

III.2.2.1. Introduction

La commande **`SELECT`** constitue, à elle seule, le langage d'interrogation d'une base. Elle permet :

- de sélectionner certaines colonnes d'une table (opération de *projection*) ;
- de sélectionner certaines lignes d'une table en fonction de leur contenu (*restriction*) ;
- de combiner des informations venant de plusieurs tables (*jointure*, *union*, *intersection*, *différence*) ;
- de combiner ces différentes opérations

Une *interrogation*, ou *requête de sélection*, est une combinaison d'opérations portant sur des tables et dont le résultat est lui-même une table, mais éphémère ou *dynamique* (car elle n'existe que le temps de la requête).

Dans ce qui suit, nous utiliserons les conventions typographiques suivantes :

- en MAJUCULES les commandes ou opérateurs qu'il faut recopier tels quels (ex. `SELECT`)
- en *italique* les paramètres devant être remplacés par une valeur appropriée (ex. *`table`*, *`alias`*)
- en souligné la valeur par défaut (ex. {ASC|DESC})
- des crochets […] encadrent une valeur optionnelle (ex. `[DISTINCT]`, `[AS alias]`)
- des accolades {…} encadrent des valeurs (séparées par |) dont l'une doit être saisie (ex. `{ON|OFF}`)
- des points de suspension … indiquent que les valeurs précédentes peuvent être répétées plusieurs fois (ex. *`table`*`, ...` peut prendre comme valeur `table1, table2, table3`)
- les signes de ponctuation (parenthèses, virgules et points-virgules) doivent être saisis comme présentés. En particulier, ne pas oublier le **point-virgule obligatoire** à la fin de chaque requête.

Par ailleurs, on désigne par :

- ***alias*** un synonyme d'un nom de table, de colonne, ou d'expression calculée
- ***condition*** une expression prenant la valeur vrai ou faux
- ***sous-interrogation*** (ou *sous-requête*) une expression de requête (SELECT) figurant dans une clause WHERE d'une requête principale
- *expr* une colonne ou un attribut calculé par une expression
- *num* un numéro de colonne

Enfin, dans ce qui suit, nous illustrerons les requêtes sur une base formée des relations suivantes :

Employes(num_emp, nom, fonction, _superieur, date_embauche, salaire, comm, _num_dep)

EMPLOYES	num_emp	nom	fonction	_superieur	date_embauche	salaire	comm	_num_dep
	7369	Coquillet	Technicien	7902	17/12/80	800		20
	7499	Parent	Commercial	7698	20/02/81	1600	300	30
	7521	Henry	Commercial	7698	22/02/81	1250	500	30
	7566	Dumoulin	Directeur	7839	02/04/81	2975		20
	7654	Ferron	Commercial	7698	28/09/81	1250	1400	30
	7698	Capon	Directeur	7839	01/05/81	2850		30
	7782	Martin	Directeur	7839	09/06/81	2450		10
	7788	Colin	Analyste	7566	19/04/87	3000		20
	7839	Patton	Président		17/11/81	5000		10
	7844	Benami	Commercial	7698	08/09/81	1500	0	30
	7876	Calamon	Technicien	7788	23/05/87	1100		20
	7900	James	Technicien	7698	03/12/81	950		30
	7902	Henry	Analyste	7566	03/12/81	3000		20
	7934	Tozzi	Technicien	7782	23/01/82	1300		10

Departements(num_dep, nom, ville)

DEPARTEMENTS	num_dep	nom	ville
	10	Comptabilité	Paris
	20	Recherche	Lille
	30	Ventes	Marseille
	40	Opérations	Lyon

III.2.2.2. Syntaxe de base

La syntaxe de base de la commande `SELECT` est la suivante :

SELECT ***attributs***
FROM ***tables*** } Correspond à l'opérateur de projection $\pi_{attributs}(tables)$

WHERE ***prédicats;*** Correspond aux opérateurs de jointure $\bowtie_{prédicat}$ et/ou de sélection $\sigma_{prédicat}$

Les *attributs* sont les colonnes que l'on souhaite voir apparaître dans la réponse. Si l'on souhaite toutes les colonnes, il faut utiliser le symbole *.

Ex.

`SELECT * FROM employes;` Affiche la table *employes* avec toutes ses colonnes

`SELECT nom, fonction FROM employes;` Affiche le nom et la fonction de tous les employés

Il est possible de faire précéder les noms des attributs de la table où ils figurent. Cela est obligatoire si l'on extraie les données de plusieurs tables et que deux attributs sélectionnés portent le même nom :

```
SELECT employes.nom, departements.nom FROM employes, departements;
```

Il est aussi possible de renommer une colonne ou une table par un ***alias***, ce qui est utile lorsqu'on veut donner un nom plus « parlant » ou plus court pour faciliter sa désignation ultérieurement dans la requête. Pour renommer une colonne, on utilise la syntaxe `Col1 AS 'Nouveau nom'` dans le SELECT (les guillemets ne sont obligatoires que si l'alias comporte des espaces ou des caractères spéciaux). Pour renommer une table, il suffit de faire suivre son nom par son alias dans la clause FROM :

```
SELECT emp.nom, emp.salaire AS 'salaire mensuel' FROM employes emp;
```

Ceci est particulièrement utile lorsque l'on définit un ***attribut calculé*** à partir des attributs de tables. Il est en effet possible de faire figurer comme résultat d'un SELECT une expression composée d'opérateurs, de fonctions prédéfinies (*cf*. liste en annexe), de variables et de constantes :

```
SELECT nom, salaire+comm AS 'salaire total' FROM employes;
```

La clause WHERE permet de spécifier quelles sont les lignes à sélectionner. Le prédicat de sélection, formé d'une expression comportant des opérandes liés par des opérateurs, sera évalué pour chaque ligne de la table, et seules les lignes pour lesquelles il est vrai seront retenues

Les opérandes figurant dans le prédicat peuvent être des noms de colonnes, ou des constantes de type :

- nombres : peuvent inclure un signe, un point décimal et une puissance de dix (ex. -1.2E-5)
- chaînes : chaînes de caractères entre apostrophes (Attention : 'Martin' ≠ 'MARTIN')
- dates : chaînes de caractères entre apostrophes dans un format spécial. Pour s'affranchir des problèmes inhérents aux incompatibilités dans les différentes langues, il est conseillé d'utiliser le format suivant : 'aaaa-mm-jj' (ex. {d '2001-10-31'})
- la valeur NULL (valeur non définie), que l'on doit tester avec l'opérateur IS, et pas = (cf. ci-dessous).

Les opérateurs figurant dans le prédicat peuvent être :

- les opérateurs de comparaison traditionnels (= (égal), <> (différent de), <, <=, >, >=)
- les opérateurs spéciaux
  - BETWEEN … AND (valeur comprise entre 2 bornes) : `expr1 BETWEEN expr2 AND expr3`
  - IN (valeur comprise dans une liste) : `expr1 IN (expr2, expr3, …)`
  - LIKE (chaîne semblable à une chaîne générique) : `expr1 LIKE chaine`

    La chaîne générique peut comporter les caractères jokers '_' (remplace 1 seul caractère quelconque) et '%' (remplace une chaîne de caractères quelconque, y compris de longueur nulle)
- les opérateurs logiques AND, OR et NOT (inversion logique) pour combiner plusieurs prédicats. Des parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation du prédicat (par défaut, l'opérateur AND est prioritaire par rapport à OR).

Exemples.

- Employés dont la commission est supérieure au salaire
```
SELECT nom, salaire, comm FROM employes WHERE comm > salaire;
```
- Employés gagnant entre 1500 et 2850€
```
SELECT nom, salaire FROM employes WHERE salaire BETWEEN 1500 AND 2850;
```
- Employés commerciaux ou analystes
```
SELECT num_emp, nom, fonction FROM employes
WHERE fonction IN ('Commercial', 'Analyste');
```
- Employés dont le nom commence par M
```
SELECT nom FROM employes WHERE nom LIKE 'M%';
```
- Employés du département 30 ayant un salaire supérieure à 2500€
```
SELECT nom FROM employes WHERE _num_dep=30 AND salaire>2500;
```
- Employés directeurs ou commerciaux travaillant dans le département 10
```
SELECT nom, fonction FROM employes
WHERE (fonction='Directeur' OR fonction ='Commercial') AND _num_dep=10;
```
- Employés percevant une commission
```
SELECT nom, salaire, comm FROM employes WHERE comm IS NOT NULL;
```

III.2.2.3. Les jointures

La jointure est une opération permettant de combiner des informations issues de plusieurs tables. Elle se formule simplement en spécifiant, dans la clause FROM , le nom des tables concernées et, dans la clause WHERE, les conditions qui vont permettre de réaliser la jointure (en effet, si l'on ne précise pas de condition de sélection, on obtient le produit cartésien des tables présentes derrière le FROM, ce qui n'est en général pas souhaité).

Il existe plusieurs types de jointures (*cf.* figure 11) :

- ***equi-jointure*** (ou *jointure naturelle*, ou *jointure interne*) : permet de relier deux colonnes appartenant à deux tables différentes mais ayant le même "sens" et venant vraisemblablement d'une relation 1-N lors de la conception. Les tables sont reliées par une relation d'égalité entre leur attribut commun ou, à partir de SQL2, avec la clause INNER JOIN.

  Ex. Donner le nom de chaque employé et la ville où il/elle travaille
  ```
  SELECT employes.nom, Ville FROM employes, departements
  WHERE employes._num_dep=departements.num_dep;
  ```
  ou
  ```
  SELECT employes.nom, departements.ville FROM employes
  INNER JOIN departements ON employes._num_dep = departements.num_dep;
  ```

- ***auto-jointure*** : cette jointure d'une table à elle-même permet de relier des informations venant d'une ligne d'une table avec des informations venant d'une autre ligne de la même table. Dans ce cas, il faut renommer au moins l'une des deux occurrences de la table pour pouvoir préfixer sans ambiguïté chaque nom de colonne.

  Ex. Donner pour chaque employé le nom de son supérieur hiérarchique
  ```
  SELECT employes.nom, superieurs.nom FROM employes, employes AS superieurs
  WHERE employes._superieur=superieurs.num_emp;
  ```
  ou
  ```
  SELECT employes.nom, superieurs.nom FROM employes
  INNER JOIN employes AS superieurs
  ON employes._superieur=superieurs.num_emp;
  ```

- ***θ-jointure*** : si le critère d'égalité correspond à la jointure la plus naturelle, les autres opérateurs de comparaison sont également utilisables dans le prédicat de jointure. Néanmoins, cette possibilité doit être utilisée avec précaution car elle peut entraîner une explosion combinatoire (on rappelle qu'une jointure ne constitue qu'une restriction du produit cartésien de deux relations ou plus).

  Ex. Quels sont les employés gagnant plus que Martin ?
  ```
  SELECT e1.nom, e1.salaire, e1.fonction FROM employes AS e1, employes AS e2
  WHERE e1.salaire>e2.salaire AND e2.nom='Martin';
  ```

- ***jointure externe*** : lorsqu'une ligne d'une table figurant dans une jointure n'a pas de correspondant dans les autres tables, elle ne satisfait pas au critère d'équi-jointure et ne figure donc pas dans le résultat de la requête. Une jointure externe est une jointure qui favorise l'une des tables en affichant toutes ses lignes, qu'il y ait ou non correspondance avec l'autre table de jointure. Les colonnes pour lesquelles il n'y a pas de correspondance sont remplies avec la valeur NULL. Pour définir une telle jointure, on utilise les clauses LEFT OUTER JOIN (*jointure externe gauche*) et RIGHT OUTER JOIN (*jointure externe droite*).

  Ex. Le département de Lyon n'apparaissait pas dans l'exemple de l'équi-jointure, mais figurera ici :
  ```
  SELECT employes.nom, departements.ville FROM employes
  RIGHT JOIN departements ON employes._num_dep = departements.num_dep;
  ```
  Ex. Le Président Patton, sans supérieur hiérarchique, n'apparaissait pas dans l'exemple de l'auto-jointure, mais figurera ici (avec NULL comme nom de supérieur) :
  ```
  SELECT employes.nom, superieurs.nom FROM employes
  LEFT JOIN employes AS superieurs ON employes._superieur=superieurs.num_emp;
  ```

Ex. Retrouver les départements n'ayant aucun employé :

```
SELECT departements.num_dep, employes.nom FROM employes
RIGHT JOIN departements ON employes._num_dep = departements.num_dep
WHERE employes.nom IS NULL;
```

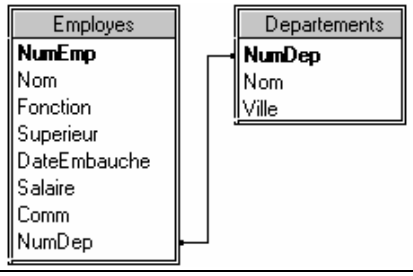

Figure 11-1 : Equi-jointure

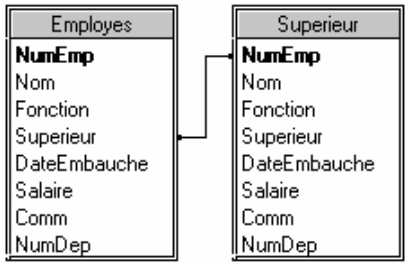

Figure 11-2 : Auto-jointure

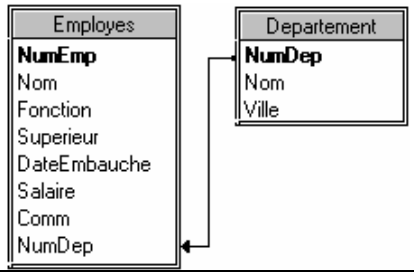

Figure 11-3 : Jointure externe

Figure 11 : Les différents types de jointures

III.2.2.4. Fonctions de groupes

Dans les exemples précédents, chaque ligne résultat d'un SELECT était calculée sur les valeurs d'une seule ligne de la table consultée. Il existe un autre type de SELECT qui permet d'effectuer des calculs sur l'ensemble des valeurs d'une colonne, au moyen de l'une des ***fonctions de groupe*** suivantes (toutes portent sur une expression, qui peut-être un attribut d'une table ou un attribut calculé) :

- SUM(expr)	Somme des valeurs
- AVG(expr)	Moyenne des valeurs
- COUNT([DISTINCT] expr)	Nombre de valeurs (différentes si DISTINCT est présent)
- COUNT(*)	Compte toutes les lignes de la table
- MIN(expr) *ou* MAX(expr)	Minimum ou maximum des valeurs
- VARIANCE(expr) *ou* STDDEV(expr)	Variance ou écart-type des valeurs

Ex. : Donner le total des salaires des employés travaillant dans le département 10 :

```
SELECT SUM(salaire) AS som_salaires FROM employes WHERE _num_dep=10;
```

Calcul de résultats sur plusieurs groupes. Il est possible de subdiviser la table en ***groupes***, chaque groupe étant l'ensemble des lignes ayant une valeur commune pour les expressions spécifiées. C'est la clause GROUP BY qui permet de découper la table en plusieurs groupes :

```
GROUP BY expr1, expr2, …
```

Si l'on regroupe suivant une seule expression, ceci définit les groupes comme les ensembles de lignes pour lesquelles cette expression prend la même valeur. Si plusieurs expressions sont présentes, le groupement va s'effectuer d'abord sur la première, puis sur la seconde, et ainsi de suite : parmi toutes les lignes pour lesquelles *expr1* prend la même valeur, on regroupe celles ayant *expr2* identique, etc. Un SELECT de groupe avec une clause GROUP BY donnera une ligne résultat pour chaque groupe.

Ex. : Donner le total des salaires des employés pour chaque département :

```
SELECT _num_dep, SUM(salaire) AS som_salaires FROM employes GROUP BY _num_dep;
```

Sélection des groupes. De même que la clause WHERE permet de sélectionner certaines lignes, il est possible de ne retenir, dans un SELECT comportant une fonction de groupe, que les lignes répondant à un critère donné par la clause HAVING. Cette clause se place après la clause GROUP BY et son prédicat suit les mêmes règles de syntaxe que celui de la clause WHERE, à la différence près qu'il ne

peut porter que sur des caractéristiques du groupe (fonction de groupe ou expression figurant dans la clause GROUP BY).

Ex. : Donner la liste des salaires moyens par fonction pour les groupes ayant plus de 2 employés :

```
SELECT fonction, COUNT(*) AS nb_employes, AVG(salaire) AS moy_salaires
FROM employes GROUP BY fonction HAVING COUNT(*)>2;
```

Remarques importantes :

- Dans la liste des colonnes résultat d'un SELECT comportant une fonction de groupe, ne peuvent figurer que des caractéristiques de groupe, c'est-à-dire soit des fonctions de groupe, soit des expressions figurant dans la clause GROUP BY. En effet, de manière générale, lorsqu'un attribut est sélectionné dans une clause SELECT, le résultat peut comporter de zéro à n valeurs ; cela pourrait provoquer des conflits si l'on utilisait conjointement des fonctions statistiques qui, elles, ne retournent qu'une seule valeur.
- Un SELECT de groupe peut contenir à la fois les clauses WHERE et HAVING. La première sera d'abord appliquée pour sélectionner les lignes, puis les fonctions de groupe seront évaluées, et les groupes ainsi constitués seront sélectionnés suivant le prédicat de la clause HAVING.

  Ex. : Pour les départements comportant au moins 2 employés techniciens ou commerciaux, donner le nombre de ces employés par département :

  ```
  SELECT _num_dep, COUNT(*) AS nb_techn_ou_comm FROM employes
  WHERE fonction in('Technicien','Commercial')
  GROUP BY _num_dep HAVING COUNT(*)>=2;
  ```

- Un SELECT de groupe peut être utilisé dans une sous-interrogation ; inversement, une clause HAVING peut comporter une sous-interrogation.

III.2.2.5. Sous-interrogations

Une caractéristique puissante de SQL est la possibilité qu'un critère de recherche employé dans une clause WHERE (expression à droite d'un opérateur de comparaison) soit lui-même le résultat d'un SELECT ; c'est ce que l'on appelle une ***sous-interrogation*** ou un *SELECT imbriqué*. Cela permet de réaliser des interrogations complexes qui nécessiteraient sinon plusieurs interrogations avec stockage des résultats intermédiaires.

Un SELECT peut retourner de zéro à n lignes comme résultat. On a donc les cas suivants :

- Une sous-interrogation ne retournant aucune ligne se termine avec un code d'erreur, à moins d'utiliser l'opérateur EXISTS qui retourne *faux* dans ce cas, *vrai* si la sous-interrogation retourne au moins une ligne. Cet opérateur permet d'empêcher l'exécution de l'interrogation principale si la sous-interrogation ne retourne aucun résultat, mais s'emploie surtout avec les sous-interrogations synchronisées (*cf.* plus loin).

  Ex. : Donner la liste des noms et salaires des employés si un des salaires est inférieur à 1000 :

  ```
  SELECT nom, salaire FROM employes
  WHERE EXISTS(SELECT * FROM employes WHERE salaire<1000);
  ```

- Si le résultat d'une sous-interrogation ne comporte qu'une seule ligne, il peut être utilisé directement avec les opérateurs de comparaison classiques.

  Ex. : Donner le nom des employés exerçant la même fonction que Martin :

  ```
  SELECT nom FROM employes
  WHERE fonction =(SELECT fonction FROM employes WHERE nom='Martin');
  ```

- Une sous-interrogation peut retourner plusieurs lignes à condition que l'opérateur de comparaison admette à sa droite un ensemble de valeurs. Les opérateurs permettant de comparer une valeur à un ensemble de valeurs sont :

IN	Retourne vrai si la valeur est égale à l'une des valeurs retournées par la sous-interrogation.
ANY	Retourne vrai si la comparaison est vraie pour l'une au moins des valeurs retournées par la sous-interrogation. Peut suivre l'un des opérateurs =,<>,>,>=,<, ou <=.
ALL	Retourne vrai si la comparaison est vraie pour toutes les valeurs retournées par la sous-interrogation. Peut suivre l'un des opérateurs =,<>,>,>=,<, ou <=.

Ex. : Donner la liste des employés gagnant plus que tous ceux du département 30 :
```
SELECT nom, salaire FROM employes
WHERE salaire > ALL(SELECT salaire FROM employes WHERE _num_dep=30);
```

Sous-interrogation synchronisée avec l'interrogation principale. Dans les exemples précédents, la sous-interrogation était évaluée d'abord, puis son résultat était utilisé pour exécuter l'interrogation principale. SQL sait également traiter une sous-interrogation faisant référence à une colonne de la table de l'interrogation principale. Dans ce cas, le traitement est plus complexe car il faut évaluer la sous-interrogation pour chaque ligne de l'interrogation principale.

Ex. : Donner la liste des employés ne travaillant pas dans le même département que leur supérieur :
```
SELECT nom FROM employes e
WHERE _num_dep<>(SELECT _num_dep FROM employes WHERE
e._superieur=num_emp)
AND _superieur IS NOT NULL;
```

Il a fallu ici renommer la table Employes de l'interrogation principale pour pouvoir la référencer sans ambiguïté dans la sous-interrogation. La dernière ligne est utile car le président ne possède pas de supérieur (valeur NULL) et la sous-interrogation ne retourne alors aucune valeur.

Ex. : Donner les employés des départements qui n'ont pas embauché depuis le début de l'année 1982 :
```
SELECT * FROM employes e
WHERE NOT EXISTS(SELECT * FROM employes WHERE
date_embauche>=to_date('01/01/1982','DD-MM-YYYY') AND _num_dep=e._num_dep);
```

III.2.2.6. Récapitulatif : syntaxe étendue

La syntaxe étendue de l'instruction SELECT est la suivante :

```
SELECT [DISTINCT] {* | expr [AS alias], ... }
  FROM table [alias], ...
  [WHERE { conditions | sous conditions} ]
  [GROUP BY expr, ...] [HAVING conditions]
  [ORDER BY {expr | num}{ASC | DESC}, ...];
```

Clause	Précise …	Notes
SELECT	Colonnes qui vont apparaître dans la réponse (DISTINCT permet d'éliminer les tuples en double)	* = toutes les colonnes AS pour renommer une expression
FROM	Table(s) intervenant dans l'interrogation	Alias surtout utile dans les jointures
WHERE	Conditions à appliquer sur les lignes. Peut inclure : - comparateurs : =, >, <, >=, <=,<> - opérateurs logiques : AND, OR, NOT - prédicats : IN, LIKE, ALL, SOME, ANY, EXISTS, ...	Jointures aussi définies par : - INNER JOIN (équi- et auto-jointure) - LEFT OUTER JOIN et RIGHT OUTER JOIN (jointures externes)
GROUP BY	Colonne(s) de regroupement	
HAVING	Condition(s) associée(s) à un regroupement	
ORDER BY	Ordre dans lequel vont apparaître les lignes de la réponse	DESC = ordre descendant

III.2.3. Modifier une base de données

III.2.3.1. Manipulation de données (LMD)

Le langage de manipulation de données permet de modifier les informations contenues dans la base de données. Les modifications ne peuvent se faire que sur une seule table à la fois, et l'unité manipulée est la ligne. Il existe une commande SQL correspondant à chaque type de modification de données : ajout, modification et suppression.

Remarques importantes : Les opérations de mise à jour sont susceptibles d'être rejetées par le moteur SQL, notamment si elles violent des contraintes d'intégrité des données. Par ailleurs, ces commandes doivent être utilisées avec précaution car elles modifient les données de manière définitive (il est impossible de revenir sur un ordre de mise à jour si l'on s'est trompé).

La commande INSERT permet d'insérer une ligne dans une table existante en spécifiant les valeurs à insérer. On peut spécifier explicitement les colonnes pour lesquelles on va fournir des valeurs ; si elle est omise, la liste des colonnes sera par défaut celle de la table dans l'ordre donné lors de sa création.

```
INSERT INTO nom_table (nom_col1, nom_col2,…) VALUES (val_col1, val_col2,…)
```

Ex. : Insérer un tuple dans la table Employes :

```
INSERT INTO employes(nom,fonction,salaire,comm)
VALUES ('Copin','Commercial',1500,150)
```

Il est possible d'insérer dans une table les lignes provenant d'une requête d'interrogation SELECT qui peut contenir n'importe quelle clause sauf un ORDER BY (ce classement des lignes serait contraire à l'esprit du relationnel) :

```
INSERT INTO nom_table (nom_col1, nom_col2, …) SELECT…
```

La commande UPDATE permet de modifier les valeurs d'une ou plusieurs colonnes, dans une ou plusieurs lignes existantes d'une table :

```
UPDATE nom_table
SET nom_col1 = expr1, nom_col2 = expr2, …
WHERE prédicat;
```

Les valeurs des colonnes *nom_col1*, *nom_col2*, … sont modifiées dans toutes les lignes satisfaisant au prédicat ; celui-ci peut contenir une sous-requête afin de filtrer de manière plus complète. En l'absence de clause WHERE, toutes les lignes sont mises à jour.

Les expression de mise à jour *expr1*, *expr2*, … peuvent contenir des constantes (numériques, dates, chaînes), des opérateurs et fonctions, et peuvent même faire référence aux anciennes valeurs de la ligne.

Ex. : Augmenter de 10% les analystes :

```
UPDATE employes SET salaire=salaire*1.1 WHERE fonction='Analyste'
```

La commande DELETE permet de supprimer des lignes d'une table :

```
DELETE FROM nom_table WHERE prédicat;
```

Toutes les lignes pour lesquelles le prédicat est vrai sont supprimées. En l'absence de clause WHERE, la table est vidée de toutes ses lignes.

III.2.3.2. Description des données (LDD)

Note : Les notions évoquées ici ne sont qu'un survol très rapide du langage de description des données. Pour un exposé complet des commandes SQL de création de schéma, le lecteur peut se reporter au site de Frédéric Brouard (*cf.* bibliographie).

La commande CREATE TABLE permet de définir une table, avec ses colonnes et ses contraintes :

```
CREATE TABLE nom_table  {colonne|contrainte}, {colonne|contrainte_tbl}, …
```

où *colonne* définit à la fois le nom de la colonne, mais aussi son type, sa valeur par défaut et ses éventuelles contraintes de colonne (la colonne peut en effet accepter ou non les valeurs nulles, certaines valeurs de validation, les doublons, ou bien être une clé primaire ou étrangère) :

```
colonne:: nom_colonne {type|domaine} [DEFAULT val_default] [contrainte_col]
contrainte_col:: [NOT] NULL | CHECK (prédicat) | UNIQUE | PRIMARY KEY |
          FOREIGN KEY [colonne] REFERENCES table(colonne) spécif_référence
```

et où les contraintes de table permettent de définir une clé multi-colonnes ainsi que des contraintes portant sur plusieurs colonnes (unicité globale, validation multi-attributs ou intégrité référentielle) :

```
contrainte_tbl:: CONSTRAINT nom_contrainte
{PRIMARY KEY (liste_cols) | UNIQUE | CHECK (prédicat_tbl) | FOREIGN KEY
(liste_cols) REFERENCES table(liste_cols) spécif_référence}
```

Ex. : Si l'on souhaite imposer les contraintes suivantes à la table Employés :
- les valeurs de département doivent appartenir à l'intervalle 10..100
- les fonctions sont forcément 'Directeur', 'Analyste', 'Technicien', 'Commercial' ou 'President'
- tout employé embauché après 1986 doit gagner plus de 1000€

sa définition serait :

```
CREATE TABLE employes (
     num_emp       SMALLINT   NOT NULL PRIMARY KEY,
     nom           VARCHAR(9),
     fonction      CHAR(10)   CHECK (fonction IN ('Directeur', 'Analyste',
                                  'Technicien', 'Commercial', 'President')),
     _superieur    SMALLINT   FOREIGN KEY REFERENCES employes (num_emp),
     Date_embauche DATE,
     salaire       DECIMAL(7,2),
     comm          DECIMAL(7,2),
     _num_dep      SMALLINT CHECK (_num_dep BETWEEN 10 AND 100),
     CONSTRAINT ChkSal CHECK (YEAR(date_embauche)<=1986 OR salaire>1000) );
```

La commande ALTER TABLE permet de modifier la définition d'une table en y ajoutant ou en supprimant une colonne ou une contrainte (mais elle ne permet pas de changer le nom ou le type d'une colonne, ni d'ajouter une contrainte de ligne [NOT] NULL) :

```
ALTER TABLE nom_table {
     ADD definition_colonne |
     ADD CONSTRAINT definition_contrainte |
     ALTER nom_colonne {SET DEFAULT valeur_def|DROP DEFAULT} |
     DROP nom_colonne [CASCADE|RESTRICT] |
     DROP CONSTRAINT nom_contrainte [CASCADE|RESTRICT] };
```

L'option CASCADE / RESTRICT permet de gérer l'intégrité référentielle de la colonne ou la contrainte.

Ex. : Modifier la définition de la table Employés pour que le salaire total dépasse forcément 1000€ :

```
ALTER TABLE employes ADD CONSTRAINT ChkSalTot CHECK (salaire+comm > 1000)
```

La commande DROP TABLE permet de supprimer la définition d'une table et, dans le même temps, de tout son contenu et des droits d'accès des utilisateurs à la table (à utiliser avec précaution, donc !) :

```
DROP TABLE nom_table
```

III.2.3.3. Contrôle des données (LCD)

La *protection des données* consiste à identifier les utilisateurs et gérer les autorisations d'accès. Pour cela, une mesure fondamentale est de ne fournir aux utilisateurs qu'un accès aux tables ou aux parties de tables nécessaires à leur travail, ce que l'on appelle des ***vues***. Grâce à elles, chaque utilisateur pourra avoir sa vision propre des données. Une vue est définie en SQL grâce à la commande CREATE VIEW comme résultat d'une requête d'interrogation SELECT :

```
CREATE VIEW nom_vue AS SELECT …
```

Ex. : Créer une vue constituant une restriction de la table Employés aux nom et salaire des employés du département 10 :

```
CREATE VIEW emp10 AS SELECT nom, salaire FROM employes WHERE _num_dep = 10;
```

Il n'y a pas de duplication des informations, mais stockage de la définition de la vue. Une fois créée, la vue peut être interrogée exactement comme une table. Les utilisateurs pourront consulter la base à travers elle, c'est-à-dire manipuler la table résultat du SELECT comme s'il s'agissait d'une table réelle. En revanche, il existe deux restrictions à la manipulation d'une vue par un INSERT ou un UPDATE :

- le SELECT définissant la vue ne doit pas comporter de jointure ;
- les colonnes résultats du SELECT doivent être des colonnes réelles et non pas des expressions.

Ex. : Augmentation de 10% des salaires du département 10 à travers la vue Emp10 :

```
UPDATE emp10 SET salaire=salaire*1.1;
```

Une vue peut être supprimée ou renommée par les commandes :

```
DROP VIEW nom_vue  ou  RENAME ancien_nom TO nouveau_nom
```

Une protection efficace des données nécessite en outre d'autoriser ou d'interdire certaines opérations, sur les tables ou les vues, aux différentes catégories d'utilisateurs. Les commandes correspondantes sont :

```
GRANT type_privilege1, type_privilege2,… ON {table|vue|*} TO utilisateur;
REVOKE type_privilege1, type_privilege2,… ON {table|vue|*} FROM utilisateur;
```

Les types de privilèges correspondent aux commandes SQL (SELECT pour la possibilité d'interroger, UPDATE pour celle de mise à jour, etc.) et les utilisateurs concernés sont désignés par leur identifiant de connexion au serveur, ou par le mot-clé PUBLIC pour désigner tous les utilisateurs.

Ex. : Autoriser la suppression de la vue Emp10 au seul représentant du personnel identifié par ID7788 :

```
GRANT DELETE ON emp10 TO ID7788;
```

Gestion des transactions. Une transaction est un ensemble de modifications de la base qui forment un tout indivisible, qu'il faut effectuer entièrement ou pas du tout, sous peine de laisser la base dans un état incohérent. SQL permet aux utilisateurs de gérer leurs transactions :

- la commande COMMIT permet de valider une transaction. Les modifications deviennent définitives et visibles à tous les utilisateurs ;
- la commande ROLLBACK permet à tout moment d'annuler la transaction en cours. Toutes les modifications effectuées depuis le début de la transaction sont alors défaites.

Pour assurer ainsi l'intégrité de la base en cas d'interruption anormale d'une tâche utilisateur, les SGBD dits *transactionnels* utilisent un mécanisme de verrouillage qui empêche deux utilisateurs d'effectuer des transactions incompatibles.

- Bibliographie -

- A. Meier (2002). *Introduction pratique aux bases de données relationnelles*. Springer-Verlag France, Collection Iris (B.U. 005.756 MEI).
- S. Maouche (2000). *Bases de données et Systèmes de gestion de bases de données*. Cours universitaire (non publié).
- A. Abdellatif, M. Limame, A. Zeroual (1998). *Oracle 7 : langage - architecture - administration*. Eyrolles, Paris.
- J. Akoka, I. Comyn-Wattiau (2001). *Conception des bases de données relationnelles, en pratique*. Vuibert, Paris, Collection Informatique (B.U. 005.756 AKO).
- Sites web
  - Introduction aux modèles EA et relationnel : www.iutc3.unicaen.fr/~moranb/cours/acsi/menu.htm
  - Bases de données, SGBD: www-lsr.imag.fr/Les.Personnes/Herve.Martin/HTML/FenetrePrincipale.htm
  - SGBD et SQL : wwwdi.supelec.fr/~yb/poly_bd/node1.html
  - Algèbre relationnelle, SQL (exo en ligne): tlouahlia.free.fr/cours/tmp/SQL/ar&sql.htm
  - SQL (apprentissage interactif) : www.marc-grange.net/SQL.htm
  - Site de F. Brouard sur SQL (très complet) : sqlpro.developpez.com/indexSQL.html