Définition des données

Le langage SQL permet de déclarer tous les éléments d'une base de données, en particulier les tables, qui sont les conteneurs d'informations.

Création de tables

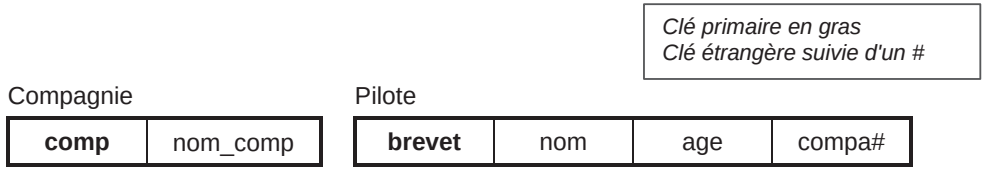
Dans notre premier exemple, nous allons travailler sur deux tables : une qui contiendra les compagnies aériennes, et l'autre qui décrira les pilotes rattachés à leur compagnie. Les commandes suivantes créent les tables `Compagnie` et `Pilote` avec les syntaxes SQL d'Oracle et Microsoft. La clause `PRIMARY KEY` permet de déclarer une clé primaire, la clause `FOREIGN KEY`, une clé étrangère.

Avec la clause `CONSTRAINT`, vous pouvez programmer tout autre type de contrainte en la nommant (valable également pour les clés primaires ou étrangères). Nous avons limité à l'aide d'une contrainte le domaine de valeurs de l'âge des pilotes (ici la contrainte se nomme `ck_age_pilote` et assurera que l'âge de chaque pilote sera toujours compris entre 20 et 60 ans).

```
CREATE TABLE compagnie
(comp VARCHAR(4), nom_comp VARCHAR(30),
 CONSTRAINT pk_compagnie PRIMARY KEY(comp))
CREATE TABLE pilote
(brevet VARCHAR(8), nom   VARCHAR(30),
 age    INTEGER,    compa VARCHAR(4),
 CONSTRAINT pk_pilote PRIMARY KEY(brevet),
 CONSTRAINT fk_pilote_compa_compagnie FOREIGN KEY(compa)
            REFERENCES compagnie(comp),
 CONSTRAINT ck_age_pilote CHECK (age BETWEEN 20 AND 60))
```

Afin d'exécuter ces scripts, sous Oracle, il faudra ajouter un point-virgule après chaque instruction et sous Microsoft la directive `GO`. La structure des tables est illustrée figure 3-1.

Figure 3-1 *Structure des tables*



Remarque

Il faut d'abord supprimer avec la directive `DROP TABLE` les tables *fils* puis *père*, ensuite créer les tables *père* puis les tables *fils*. Cela permet de pouvoir relancer le script à la demande. Le script précédent devrait donc contenir en en-tête les instructions :

Tableau 3.1 Destruction du schéma

Oracle	Microsoft
DROP TABLE pilote; DROP TABLE compagnie;	DROP TABLE pilote GO DROP TABLE compagnie GO

Noms des contraintes

Nommez `pk_nomtable` la contrainte clé primaire de la table.

Nommez `fk_T1_ce_T2` la contrainte clé étrangère `ce` de la table `T1` vers la table `T2` .

Manipulation des données

Les commandes `INSERT`, `UPDATE`, et `DELETE` permettent respectivement d'insérer, de modifier et de supprimer des enregistrements d'une table. Concernant les deux dernières fonctionnalités, on peut filtrer ces instructions selon des critères définis avec la directive `WHERE`.

Le script SQL suivant insère des compagnies et des pilotes rattachés chacun à une compagnie.

```
INSERT INTO compagnie VALUES('AF','Air France')
INSERT INTO compagnie VALUES('CAST','Castanet Air Lines')
INSERT INTO pilote VALUES ('3MPY93', 'Soutou', 36, 'CAST')
INSERT INTO pilote VALUES ('16AGN65', 'Bidal', 36, 'CAST')
INSERT INTO pilote VALUES ('9PAR64', 'Rival', 37, 'AF')
INSERT INTO pilote VALUES ('30MPY67', 'Lamothe', 34, 'AF')
INSERT INTO pilote VALUES ('25MPY67', 'Albaric', 34, 'CAST')
```

Le contenu des tables est illustré figure 3-2 :

Figure 3-2 *Contenu des tables*

Compagnie

comp	nom_comp
AF	Air France
CAST	Castanet Air Lines

Pilote

brevet	nom	age	compa#
3MPY93	Soutou	36	CAST
16AGN65	Bidal	36	CAST
9PAR64	Rival	37	AF
30MPY67	Lamothe	34	AF
25MPY67	Albaric	34	AF

Le script SQL suivant affecte le pilote de code '`Lamothe`' à la compagnie de code '`CAST`' :

```
UPDATE pilote SET compa = 'CAST' WHERE nom = 'Lamothe'
```

Le script SQL suivant supprime les pilotes qui ont plus de 36 ans et qui appartiennent à la compagnie de code '`AF`' :

```
DELETE FROM pilote WHERE (age > 35 AND compa = 'AF')
```

Le contenu des tables est maintenant illustré par la figure 3-3 :

Figure 3-3 *Contenu des tables*

Compagnie

comp	nom_comp
AF	Air France
CAST	Castanet Air Lines

Pilote

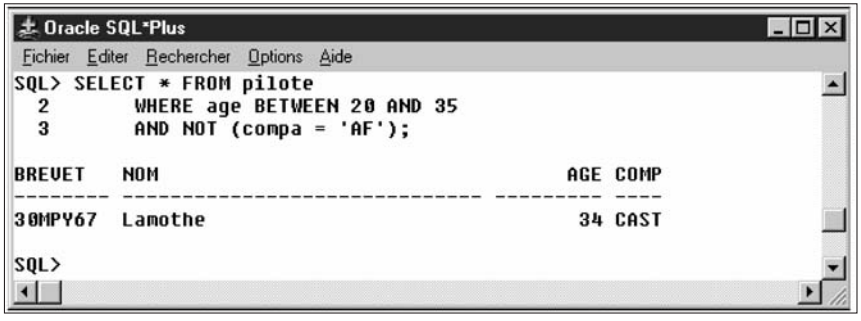
brevet	nom	age	compa#
3MPY93	Soutou	36	CAST
16AGN65	Bidal	36	CAST
30MPY67	Lamothe	34	CAST
25MPY67	Albaric	34	AF

Interrogation des données

Le langage SQL permet d'extraire des informations de la base de données en fonction de critères. Pour ce faire, il faut recourir à une instruction de type `SELECT` appelée « requête ». À titre d'exemple, nous cherchons les pilotes âgés de 20 à 35 ans, qui n'appartiennent pas à la compagnie de code 'AF'. La requête qu'il convient d'utiliser est la suivante (le signe « * » sélectionne toutes les colonnes de la table).

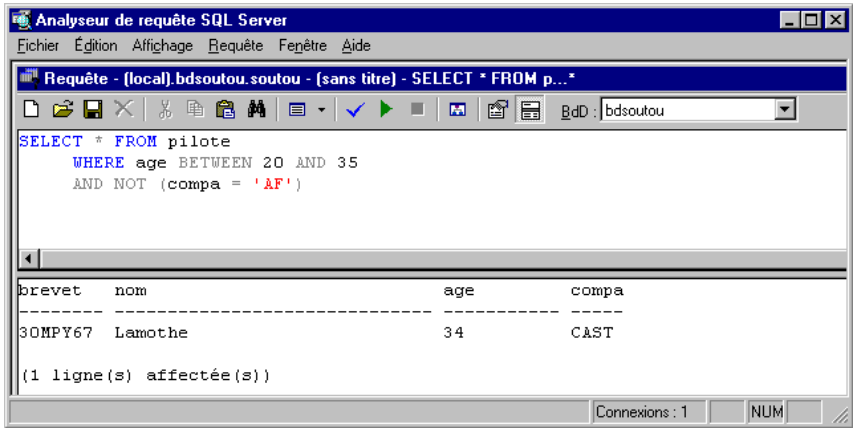
*Interface SQL*Plus d'Oracle*

Figure 3-4 *Résultat sous Oracle*



Interface ISQL/W de Microsoft

Figure 3-5 *Résultat sous Microsoft*



Requête avec jointures

Il est possible de rédiger des questions plus complexes mettant en jeu plusieurs tables, et faisant la plupart du temps intervenir des jointures. Supposons que le contenu des tables soit à présent celui de la figure 3-6, et que vous souhaitiez connaître le nom des pilotes âgés de 35 à 37 ans, qui appartiennent à une compagnie embauchant plus de deux pilotes, et dont le budget dépasse la moyenne des budgets des compagnies.

Figure 3-6 *Contenu des tables*

Compagnie

comp	nom_comp	budget
AF	Air France	300 000
CAST	Castanet Air Lines	400 001
ASO	Air Sud-Ouest	500 000

Pilote

brevet	nom	age	compa#
3MPY93	Soutou	36	CAST
2MTB98	Laroche	39	CAST
30MPY67	Lamothe	34	AF
25MPY67	Albaric	34	AF
16AGN65	Bidal	36	ASO
21PAU99	Labat	33	ASO
6MPY97	Tauzin	34	ASO

La requête qu'il conviendra d'utiliser sera alors la suivante :

```
SELECT nom, age, compa FROM pilote
  WHERE compa IN (SELECT comp FROM compagnie WHERE budget >=
                          (SELECT AVG(budget) FROM compagnie))
```

```
  AND compa IN   (SELECT compa FROM pilote GROUP BY compa
                         HAVING COUNT(*) >= 2)
  AND age BETWEEN 35 AND 38
```

Le résultat retourne :

```
NOM                                AGE COMP
------------------------------ --------- ----
Bidal                               36 ASO
Soutou                              36 CAST
```

On se rend compte que le concepteur de cette base n'a pas forcément pris en compte la nécessité d'effectuer cette requête lors de la conception. Cela illustre une des fonctionnalités des bases de données, à savoir le fait de ne pas connaître exhaustivement les requêtes à soumettre, et qu'on peut déduire des informations en rapprochant des faits élémentaires entre eux.

Contrôle des données

Dans un contexte multi-utilisateur, SQL a dû s'adapter pour :

- contrôler l'accès aux données (privilèges en lecture, modification ou suppression) contenues dans les tables ;
- assurer la confidentialité et l'intégrité des informations. La confidentialité est souvent réalisée par l'utilisation de vues *(views)*.

Attribution de privilèges

Les instructions SQL qui définissent les privilèges sur les données sont fondées sur les directives `GRANT` (pour autoriser) et `REVOKE` (pour interdire).

Exemple

Tableau 3.2 Privilèges

Utilisateur *Soutou*

Pilote

brevet	nom	age	compa
3MPY93	Soutou	36	CAST
16AGN65	Bidal	36	CAST
30MPY67	Lamothe	34	CAST
25MPY67	Albaric	34	AF

Soutou désire autoriser l'utilisateur *Tremont* à lire la table `Pilote` et à modifier la colonne `age`.

Utilisateur *Tremont*

Compagnie

comp	nom_comp
AF	Air France
CAST	Castanet Air Lines

Tremont autorise l'utilisateur *Soutou* à lire et à insérer des enregistrements dans la table `Compagnie`.
Tremont désire interdire désormais à l'utilisateur *Soutou* d'insérer des enregistrements dans la table `Compagnie`.

Mise en œuvre sous Oracle

Tableau 3.3 Privilèges sous Oracle

Utilisateur *Soutou*	Utilisateur *Tremont*

Autorisation à l'utilisateur *Tremont* de lire la table `Pilote`.
`GRANT SELECT ON Pilote TO Tremont;`
`Autorisation de privilèges (GRANT) acceptée.`
Autorisation à l'utilisateur *Tremont* de modifier la colonne `age` de la table `Pilote`.
`GRANT UPDATE(age) ON Pilote TO Tremont;`
`Autorisation de privilèges (GRANT) acceptée.`	

	Lecture de la table `Pilote` de *Soutou*.
	`SELECT nom,age,compa FROM Soutou.Pilote;`
	`NOM AGE COMPA`
	`-------------------- --- ----`
	`Soutou 36 CAST`
	`Bidal 36 CAST`
	`Lamothe 34 AF`
	`Albaric 34 AF`

	Modification de l'âge d'un des pilotes.
	`UPDATE Soutou.Pilote SET age = age+1 WHERE nom = 'Bidal';`
	`1 ligne mise à jour.`
	Tentative de suppression de tous les pilotes.
	`DELETE FROM Soutou.Pilote; *`
	`ERREUR à la ligne 1 :`
	`ORA-01031: privilèges insuffisants`
	Tremont autorise *Soutou* à lire et à insérer des enregistrements dans la table `Compagnie`.
	`GRANT SELECT,INSERT ON Compagnie TO Soutou;`
	`Autorisation de privilèges (GRANT) acceptée.`

Tableau 3.3 Privilèges sous Oracle (suite)

Utilisateur *Soutou*	Utilisateur *Tremont*

Lecture de la table `Compagnie` de l'utilisateur *Tremont*.
`SELECT * FROM Tremont.Compagnie;`
`COMP NOM_COMP`
`---- -------------------------`
`AF Air France`
`CAST Castanet Air Lines`
Insertion d'une nouvelle compagnie dans la table de l'utilisateur *Tremont*.
`INSERT INTO Tremont.Compagnie`
`VALUES('IFID','Aéro-IFIDEC');`
`1 ligne créée.`
`SELECT * FROM Tremont.Compagnie;`
`COMP NOM_COMP`
`---- -------------------------`
`AF Air France`
`CAST Castanet Air Lines`
`IFID Aéro-IFIDEC`
Tentative de supprimer toutes les compagnies.
`DELETE FROM Tremont.Compagnie;`
`*`
`ERREUR à la ligne 1 :`
`ORA-01031: privilèges insuffisants`	

	Tremont désire interdire désormais à l'utilisateur *Soutou* d'insérer des enregistrements dans la table `Compagnie`.
	`REVOKE INSERT ON Compagnie`
	`FROM Soutou;`
	`Suppression de privilèges (REVOKE) acceptée.`

Insertion d'une nouvelle compagnie dans la table `Compagnie` de l'utilisateur *Tremont*.
`INSERT INTO Tremont.Compagnie`
`VALUES('Essai','Aéro Test');`
`*`
`ERREUR à la ligne 1 :`
`ORA-01031: privilèges insuffisants`	

Intégrité des données

Les SGBD prennent en compte l'intégrité des données définies *via* la déclaration de contraintes ou la programmation de fonctions ou de procédures cataloguées, de paquetages *(packages)* ou de déclencheurs *(triggers)*. Le principe est simple : assurer la cohérence de la base après chaque modification (par `INSERT`, `UPDATE` ou `DELETE`).

Exemples de contraintes

Dans l'exemple, trois contraintes ont été déclarées sur la table `Pilote`, elles permettent de programmer les domaines d'attributs du modèle relationnel. La première (`pk_pilote`) indique que le numéro du brevet est unique, la deuxième (`fk_pilote_compa_compagnie`) indique que la compagnie du pilote doit être référencée dans la table compagnie, la troisième (`ck_age_pilote`) définit un intervalle d'âge possible pour tout pilote. Chaque contrainte est nommée. Ce principe facilite la désactivation temporaire et la réactivation des contraintes avec la commande `ALTER TABLE`.

```
CREATE TABLE pilote
(brevet VARCHAR(8), nom VARCHAR(30), age NUMBER, compa VARCHAR(4),
 CONSTRAINT pk_pilote PRIMARY KEY(brevet),
 CONSTRAINT fk_pilote_compa_compagnie FOREIGN KEY(compa)
            REFERENCES compagnie(comp),
 CONSTRAINT ck_age_pilote CHECK (age BETWEEN 20 AND 60))
```

Supposons que l'état des tables soit le suivant :

```
SQL> SELECT * FROM compagnie;
COMP NOM_COMP
---- ------------------------------
AF   Air France
CAST Castanet Air Lines

SQL> SELECT * FROM pilote;
BREVET   NOM                                AGE COMP
-------- ------------------------------ --------- ----
3MPY93   Soutou                              36 CAST
16AGN65  Bidal                               36 CAST
9PAR64   Rival                               37 AF
30MPY67  Lamothe                             34 AF
25MPY67  Albaric                             34 AF
```

Essayons à présent d'insérer des enregistrements de cette table en ne respectant pas les règles de gestion déclarées, c'est-à-dire en donnant un numéro de brevet, un code compagnie et un âge incorrects.

```
SQL> INSERT INTO pilote VALUES ('3MPY93', 'Tuffery', 32, 'CAST');
ERREUR à la ligne 1 :
ORA-00001: violation de contrainte unique (SOUTOU.PK_PILOTE)

SQL> INSERT INTO pilote VALUES ('1MPY93', 'Tuffery', 32, 'RIEN');
ERREUR à la ligne 1 :
ORA-02291: violation de contrainte (SOUTOU.FK_PILOTE_COMPA_COMPA-
GNIE) d'intégrité - touche parent introuvable

SQL> INSERT INTO pilote VALUES ('1MPY93', 'Tuffery', 62, 'CAST');
ERREUR à la ligne 1 :
ORA-02290: violation de contraintes (SOUTOU.CK_AGE_PILOTE) de
vérification
```

Notons que le SGBDR renvoie chaque fois une erreur, qui évite au programmeur de prendre en compte toute une série de tests. Avec Oracle, le nom de la contrainte est préfixé par le nom du propriétaire de la table sur laquelle se porte cette contrainte (ici l'utilisateur Soutou).

Intégrité référentielle

L'intégrité référentielle, mise en œuvre à l'aide des clés étrangères, est l'une des caractéristiques majeures des SGBD. Elle permet d'assurer la cohérence d'une base de données.

Dans notre exemple, il était incorrect d'insérer un pilote d'une compagnie non référencée dans la base. Il est tout aussi incorrect de vouloir supprimer une compagnie à laquelle des pilotes sont rattachés. La suppression d'une compagnie est envisageable en effectuant en contrepartie des actions compensatoires, comme modifier tous les pilotes qui dépendent de cette compagnie (en affectant la valeur nulle à la colonne `compa`) ou en supprimer tous les pilotes concernés ! Ces actions peuvent être définies avec la directive `CASCADE` et sont déclarées avec la table concernée comme une contrainte.

Dans notre exemple, nous n'avons pas pris en compte de telles contraintes. La suppression d'une compagnie ayant des pilotes entraîne donc l'incohérence de la base. Le script suivant montre qu'Oracle gère intrinsèquement l'intégrité référentielle.

```
-- Modif de la compagnie du premier pilote
SQL> UPDATE pilote SET compa = 'AOM' WHERE brevet = '3MPY93';
ERREUR à la ligne 1 :
ORA-02291: violation de contrainte (SOUTOU.FK_PILOTE_COMPA_COMPA-
GNIE) d'intégrité - touche parent introuvable

--Suppression d'une compagnie ayant des pilotes
SQL> DELETE FROM compagnie WHERE comp = 'AF';
ERREUR à la ligne 1 :
ORA-02292: violation de contrainte (SOUTOU.FK_PILOTE_COMPA_
COMPAGNIE) d'intégrité – enregistrement fils existant
```

Dans le premier message d'erreur, Oracle signale qu'un *fils* n'a pas de *père*. Dans le second message, il signale qu'un *père* possède un *fils*.

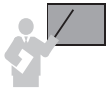
Passage du logique à SQL2

Cette section décrit la traduction SQL2 d'un schéma logique relationnel. Nous expliquons comment traduire les associations et les contraintes découvertes au niveau conceptuel.

Traduction des relations

Les instructions SQL que nous allons détailler seront à placer dans un script qui permettra la création de la base de données relationnelle.

Principe



Une relation devient une table, ses attributs sont les colonnes de la table. La clé primaire est définie sur les colonnes traduites de l'identifiant de la relation avec la contrainte `PRIMARY KEY`.

Exemple

L'exemple décrit un pilote caractérisé par un numéro de brevet, un nom et un âge. La figure 3-7 illustre cet exemple. La règle R1 de transit du niveau conceptuel au niveau logique est décrite au chapitre 2 (section *Du conceptuel au logique*).

Figure 3-7 *Premier exemple*

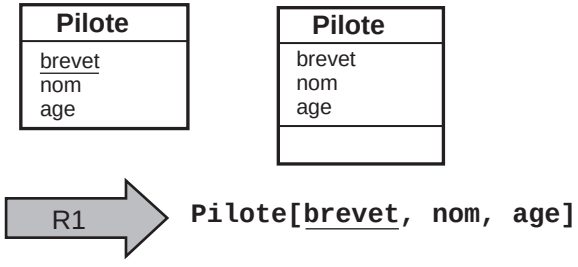


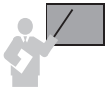
Tableau 3.4 Règle R1

Schéma logique	Script SQL2

Pilote[brevet, nom, age]	`CREATE TABLE pilote`
	`(brevet VARCHAR(8),`
	`nom VARCHAR(30),`
	`age NUMBER,`
	`CONSTRAINT pk_pilote PRIMARY KEY(brevet))`

Traduction des associations binaires

Nous étudions la traduction des associations *un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs* et réflexives. Nous illustrons nos exemples avec les formalismes Merise et UML de manière à ce que le lecteur puisse mieux appréhender les niveaux conceptuel, logique et physique.



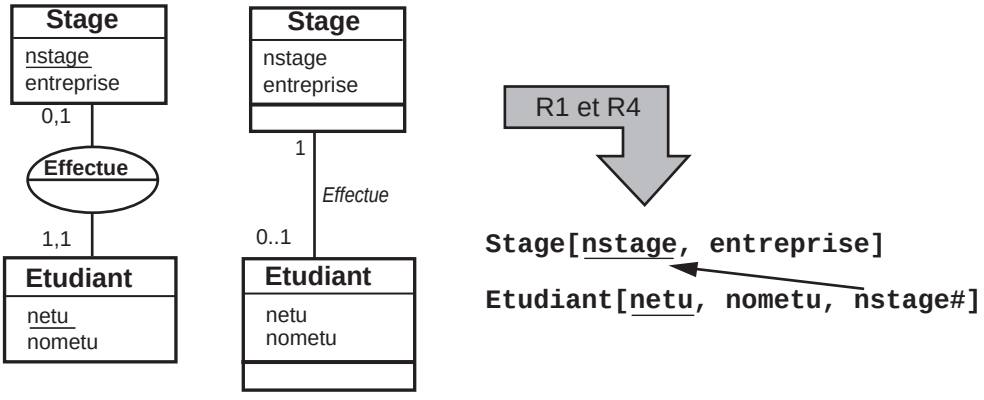
Les clés étrangères sont traduites avec des contraintes `FOREIGN KEY… REFERENCES…`

Associations un-à-un

L'exemple 3-8 décrit l'inscription d'un étudiant à un stage. Les règles R1 et R4 doivent être appliquées. Il est préférable de placer la clé étrangère dans la table `Etudiant` pour éviter les valeurs nulles en base.

La particularité du script SQL réside dans la définition des deux dernières contraintes qui

Figure 3-8 *Exemple d'association un-à-un*



traduisent les deux cardinalités minimales (à 1) de l'association `Effectue`. Ces contraintes expriment le fait qu'un étudiant doit être affecté à un seul stage (contrainte `NOT NULL`), et qu'un stage n'est attribué qu'à un seul étudiant (contrainte `UNIQUE`). La contrainte `NOT NULL` interdira les valeurs nulles dans la colonne `nstage`. La contrainte `UNIQUE` interdira qu'une valeur dans la colonne `nstage` puisse apparaître pour différents étudiants.

Tableau 3.5 Association un-à-un

Schéma logique	Script SQL2
Stage[nstage, entreprise] Etudiant[netu, nometu, nstage#]	CREATE TABLE stage (nstage VARCHAR(4), entreprise VARCHAR(30), CONSTRAINT pk_stage PRIMARY KEY(nstage)) CREATE TABLE etudiant (netu VARCHAR(2),nometu VARCHAR(30), nstage VARCHAR(4), CONSTRAINT pk_etudiant PRIMARY KEY(netu), CONSTRAINT fk_etudiant_nstage_stage FOREIGN KEY(nstage) REFERENCES stage(nstage), **CONSTRAINT nn_etudiant_nstage** **CHECK (nstage IS NOT NULL),** **CONSTRAINT unique_etudiant_nstage UNIQUE** **(nstage))**

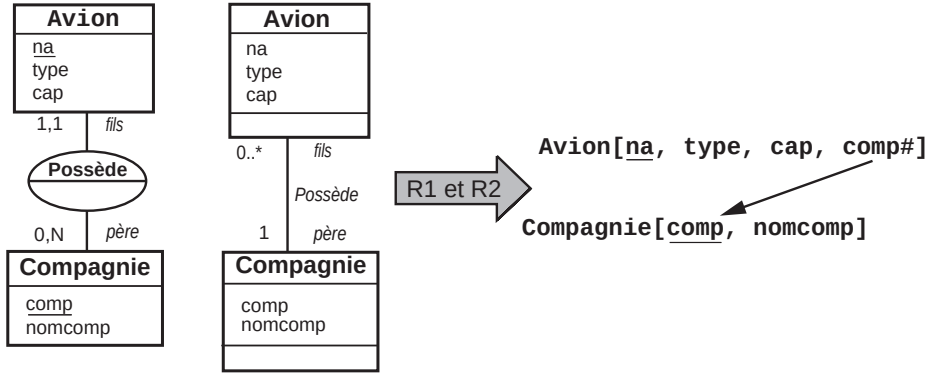
Nommez `nn_T1_col` une contrainte de type `NOT NULL` sur la colonne `col` de la table `T1`.

Nommez `unique_T1_col` une contrainte de type `UNIQUE` sur la colonne `col` de la table `T1`.

Associations un-à-plusieurs

L'exemple 3-9 décrit une association *un-à-plusieurs* entre une compagnie et ses avions. Les règles à adopter sont R1 et R2.

Figure 3-9 *Exemple d'association un-à-plusieurs*



La cardinalité minimale 1 de l'association `Possede` se traduit à l'aide d'une contrainte de type `NOT NULL` sur la clé étrangère.

La majorité des outils génèrent les clés étrangères après les tables par l'instruction ALTER TABLE.

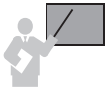
Tableau 3.6 Association un-à-plusieurs

Schéma logique	Script SQL2

	`CREATE TABLE compagnie`
	`(comp VARCHAR(4),nomcomp VARCHAR(30),`
	`CONSTRAINT pk_compagnie PRIMARY KEY(comp))`

Compagnie[comp, nomcomp]
Avion[na, type, cap, comp#]
	`CREATE TABLE avion`
	`(na VARCHAR(2), type VARCHAR(4),`
	`cap NUMBER(3), comp VARCHAR(4),`
	`CONSTRAINT pk_avion PRIMARY KEY(na),`
	`CONSTRAINT fk_avion_comp_compagnie`
	`FOREIGN KEY(comp) REFERENCES compagnie(comp),`
	`CONSTRAINT nn_avion_comp`
	`CHECK (comp IS NOT NULL))`

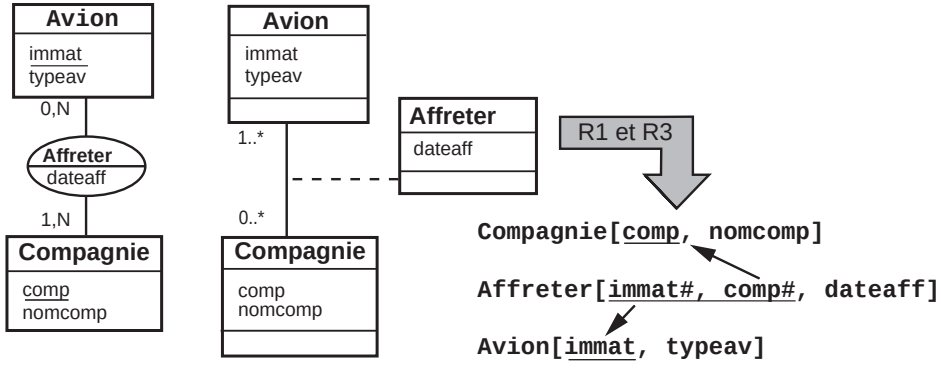
Associations plusieurs-à-plusieurs



Une relation dont l'identifiant est composé de plusieurs attributs devient une table pour laquelle la clé primaire se compose de plusieurs colonnes à l'aide de la contrainte de type `PRIMARY KEY(col1,col2,…)`.

Les règles R1 et R3 ont été appliquées à l'exemple 3-10 de manière à dériver trois relations, dont l'une (`Affreter`) possède une clé primaire composée.

Figure 3-10 *Exemple d'association plusieurs-à-plusieurs*



La cardinalité (multiplicité) minimale 1 d'une association *plusieurs-à-plusieurs* ne se traduit pas au niveau physique.

Dans l'exemple, cette cardinalité indique que tous les codes des compagnies référencées dans la table `Compagnie` doivent se trouver dans la table `Affreter`. Il n'est pas possible que cette contrainte soit respectée au début du cycle de vie de la base (lors du premier affrètement par exemple). En revanche, il sera possible de programmer, ultérieurement, que toute compagnie est référencée par un affrètement.

Tableau 3.7 Association plusieurs-à-plusieurs

Schéma logique	Script SQL2

	`CREATE TABLE compagnie`
	`(comp VARCHAR(4), nomcomp VARCHAR(30),`
	`CONSTRAINT pk_compagnie PRIMARY KEY(comp))`

Compagnie[comp, nomcomp]
Affreter[immat#, comp#, dateaff]
Avion[immat, typav]
	`CREATE TABLE avion`
	`(immat VARCHAR(6), typav VARCHAR(10),`
	`CONSTRAINT pk_avion PRIMARY KEY(immat))`
	
	`CREATE TABLE affreter`
	`(immat VARCHAR(6),comp VARCHAR(4),dateaff DATE,`
	`CONSTRAINT pk_affreter PRIMARY KEY (immat,comp),`
	`CONSTRAINT fk_affreter_immat_avion`
	`FOREIGN KEY(immat)REFERENCES avion(immat),`
	`CONSTRAINT fk_affreter_comp_compagnie`
	`FOREIGN KEY(comp) REFERENCES compagnie(comp))`

Associations n-aires

Une association *n*-aire généralise une association *plusieurs-à-plusieurs*. La clé primaire de la table réalisant l'association sera donc composée de *n* colonnes. Comme pour les associations *plusieurs-à-plusieurs*, toute cardinalité minimale 1 de l'association ne se traduit pas au niveau physique.

Les règles R1 et R3 sont appliquées à l'association *3*-aire de l'exemple 3-11. On ne dérive pas une relation de l'entité temporelle `Jour` (explication au chapitre 2). En conséquence, seuls deux attributs de la table `Affreter` sont des clés étrangères (`immat` et `comp`). La différence avec le schéma précédent réside dans le fait qu'un avion donné pour une compagnie donnée peut être affrété à différentes dates.

Figure 3-11 *Exemple d'association* n*-aire*

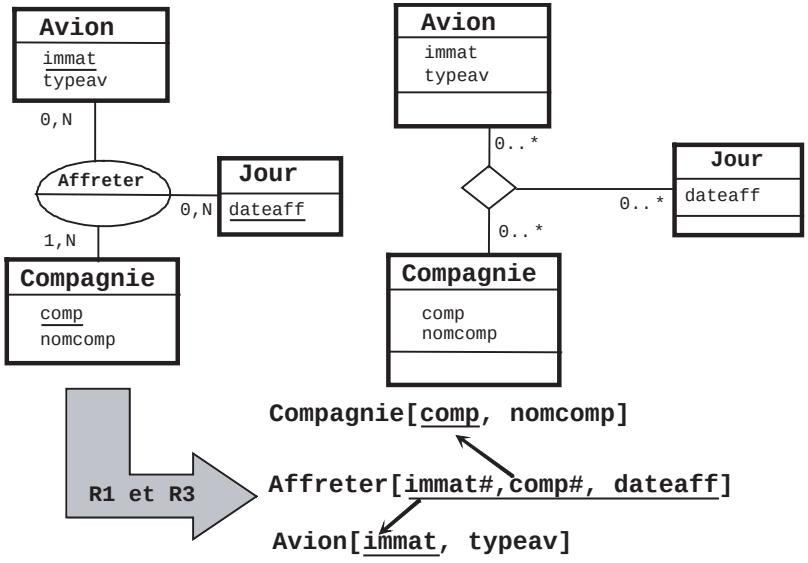


Tableau 3.8 Association n-aire

Schéma logique	Script SQL2

	`CREATE TABLE compagnie`
	`(comp VARCHAR(4), nomcomp VARCHAR(30),`
	`CONSTRAINT pk_compagnie PRIMARY KEY(comp))`

Compagnie[comp, nomcomp]	`CREATE TABLE avion`
	`(immat VARCHAR(6), typav VARCHAR(10),`
	`CONSTRAINT pk_avion PRIMARY KEY(immat))`

Affreter[immat#, comp#, dateaff]
Avion[immat, typav]
	`CREATE TABLE affreter`
	`(immat VARCHAR(6), comp VARCHAR(4),dateaff DATE,`
	`CONSTRAINT pk_affreter`
	`PRIMARY KEY (immat,comp,dateaff),`
	`CONSTRAINT fk_affreter_immat_avion`
	`FOREIGN KEY(immat) REFERENCES avion(immat),`
	`CONSTRAINT fk_affreter_comp_compagnie`
	`FOREIGN KEY(comp) REFERENCES compagnie(comp))`

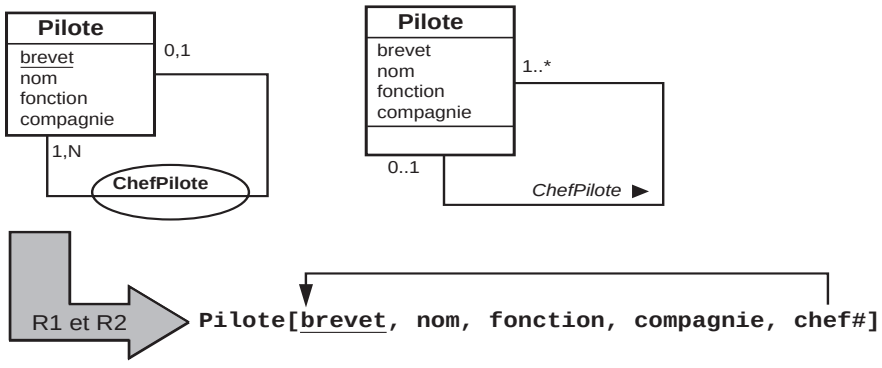
Associations réflexives

Les associations réflexives sont des associations binaires (*un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs*) ou *n*-aires. Les transformations sont analogues aux associations non réflexives.

Un-à-plusieurs

Les règles R1 et R2 sont appliquées à l'association réflexive *un-à-plusieurs* de l'exemple 3-12. La clé étrangère contiendra le code du chef pilote pour chaque pilote. Pour tout chef, cette clé étrangère ne contiendra pas de valeur (`NULL`).

Figure 3-12 *Exemple d'association réflexive un-à-plusieurs*



La partie SQL indiquée en gras souligne la particularité de l'association réflexive.

Tableau 3.9 Association réflexive un-à-plusieurs

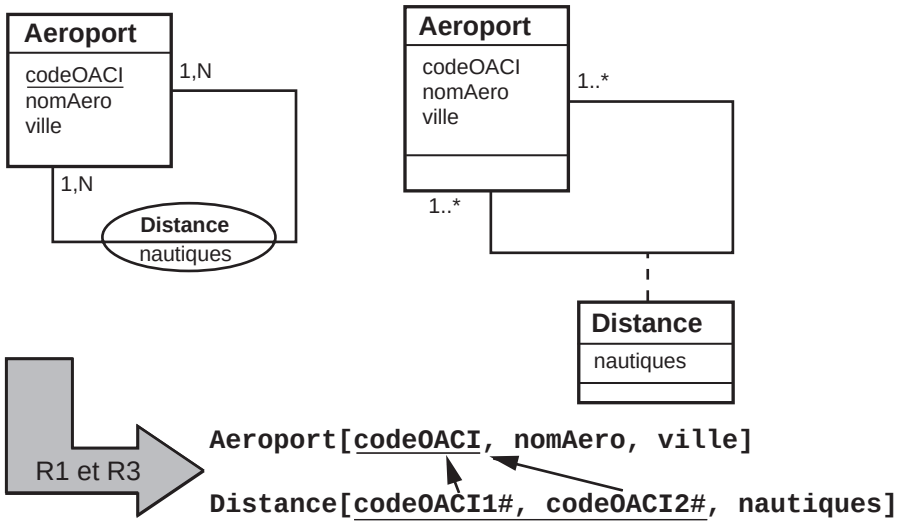
Schéma logique	Script SQL2

Pilote[brevet, nom, fonction, compagnie, chef#]	`CREATE TABLE pilote`
	`(brevet VARCHAR(8), nom VARCHAR(30),`
	`fonction VARCHAR(4), compagnie VARCHAR(4),`
	`chef VARCHAR(8),`
	`CONSTRAINT pk_pilote PRIMARY KEY(brevet),`
	`CONSTRAINT fk_pilote_chef_pilote`
	`FOREIGN KEY(chef) REFERENCES pilote(brevet))`

Plusieurs-à-plusieurs

Les règles R1 et R3 sont appliquées à l'association réflexive *plusieurs-à-plusieurs* de l'exemple 3-13 (modélisation de la distance entre deux aéroports).

Figure 3-13 *Exemple d'association réflexive plusieurs-à-plusieurs*



Le script SQL est le suivant.

Tableau 3.10 Association réflexive plusieurs-à-plusieurs

Schéma logique	Script SQL2

Aeroport[codeOACI, nomAero, ville]
Distance[OACI1#, OACI2#, nautiques]
	`CREATE TABLE Aeroport`
	`(codeOACI VARCHAR(8), nomAero VARCHAR(30),`
	`ville VARCHAR(20), CONSTRAINT pk_Aeroport`
	`PRIMARY KEY(codeOACI))`
	
	`CREATE TABLE Distance`
	`(OACI1 VARCHAR(8),OACI2 VARCHAR(8),`
	`nautiques NUMBER(4), CONSTRAINT pk_Distance`
	`PRIMARY KEY(OACI1,OACI2),`
	`CONSTRAINT fk_Distance_Aeroport1`
	`FOREIGN KEY(OACI1)`
	`REFERENCES Aeroport(codeOACI),`
	`CONSTRAINT fk_Distance_Aeroport2`
	`FOREIGN KEY(OACI2)`
	`REFERENCES Aeroport(codeOACI))`

Solution universelle

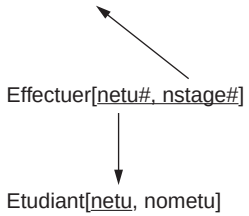
Il est possible de modéliser toute association par une table supplémentaire, qui contiendra autant de clés étrangères qu'il y a de tables à relier, et sur lesquelles existera ou non une contrainte de type `UNIQUE`. Ce principe s'apparente à la règle R3.

Cette solution présente l'avantage de pouvoir faire évoluer le schéma plus facilement si les cardinalités viennent à changer dans le temps. En effet, il n'y a besoin de modifier la structure d'aucune table, seules des contraintes `UNIQUE` devront être désactivées.

Considérons à nouveau l'exemple des stages des étudiants. La table supplémentaire (`Effectuer`) contient deux colonnes réalisant l'association. Sur chaque colonne, il sera impératif de définir une contrainte `UNIQUE` qui garantira les cardinalités maximales de l'association (ici *un-à-un*).

Tableau 3.11 Solution universelle pour une association un-à-un

Schéma logique	Script SQL2

	`CREATE TABLE Stage`
	`(nstage VARCHAR(4), entreprise VARCHAR(30),`
	`CONSTRAINT pk_stage PRIMARY KEY(nstage))`
	
	`CREATE TABLE Etudiant`
	`(netu VARCHAR(2), nometu VARCHAR(30),`
	`CONSTRAINT pk_etudiant PRIMARY KEY(netu))`
	
	`CREATE TABLE Effectuer`
	`(netu VARCHAR(2), nstage VARCHAR(4),`
	`CONSTRAINT pk_Effectuer PRIMARY KEY(netu,nstage),`
	`CONSTRAINT fk_Effectuer_nstage_Stage`
	`FOREIGN KEY(nstage) REFERENCES Stage(nstage),`
	`CONSTRAINT fk_Effectuer_netu_Etudiant`
	`FOREIGN KEY(netu) REFERENCES Etudiant(netu),`
	`CONSTRAINT unique_Effectuer_netu UNIQUE (netu),`
	`CONSTRAINT unique_Effectuer_nstage UNIQUE (nstage))`

Ce schéma est évolutif. En effet, si un stage peut être effectué par plusieurs étudiants, il faudra simplement désactiver la contrainte (`ALTER TABLE Effectuer DISABLE CONSTRAINT unique_Effectuer_nstage`). On retrouvera une association *un-à-plusieurs*. Si un étudiant peut effectuer plusieurs stages, il faudra désactiver l'autre contrainte. On se retrouvera alors avec une association *plusieurs-à-plusieurs*.

Traduction des associations d'héritage

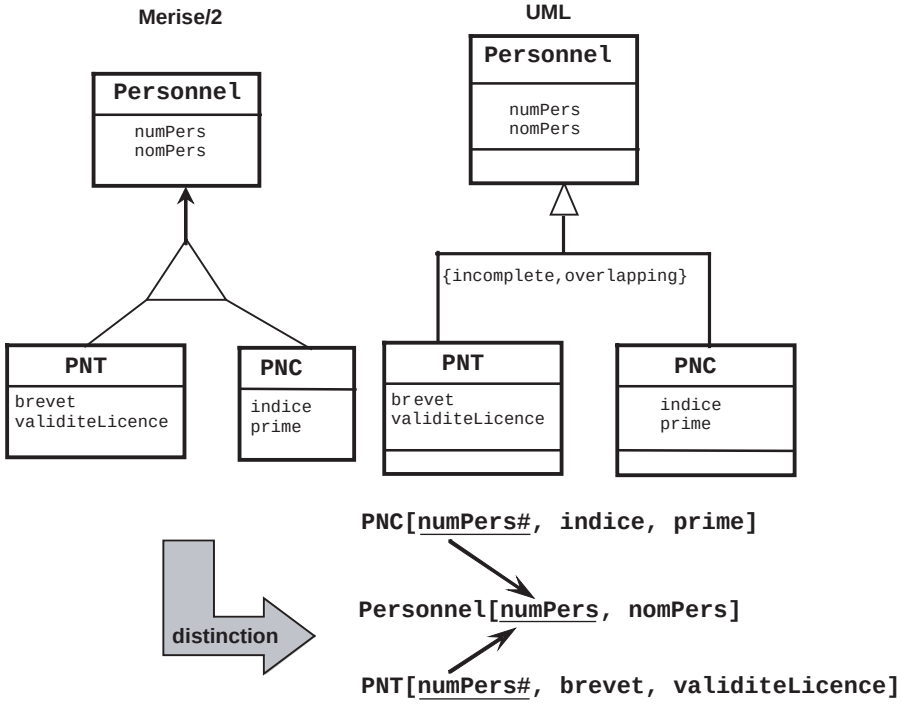
Nous expliquerons dans un premier temps la traduction SQL2 des associations d'héritage en fonction de la décomposition choisie (chapitre 2, section *Héritage*). Nous détaillerons ensuite la traduction d'éventuelles contraintes (partition, totalité et exclusivité).

Nous avons recensé au chapitre précédent trois familles de décomposition au niveau logique pour traduire une association d'héritage : décomposition par distinction, descendante *(push-down)* et ascendante *(push-up)*.

Décomposition par distinction

L'héritage 3-14 (sans contrainte avec Merise/2 se modélisant par une contrainte UML) est traduit suivant le principe de décomposition par distinction. La clé primaire issue de la sur-entité (sur-classe) est dupliquée dans les deux tables déduites des sous-entités (sous-classes).

Figure 3-14 *Décomposition par distinction d'une association d'héritage*



Le schéma physique est composé de trois tables dotées de la même clé primaire. La table `Personnel` stockera notamment les personnels n'étant ni PNT ni PNC.

Tableau 3.12 Héritage par distinction

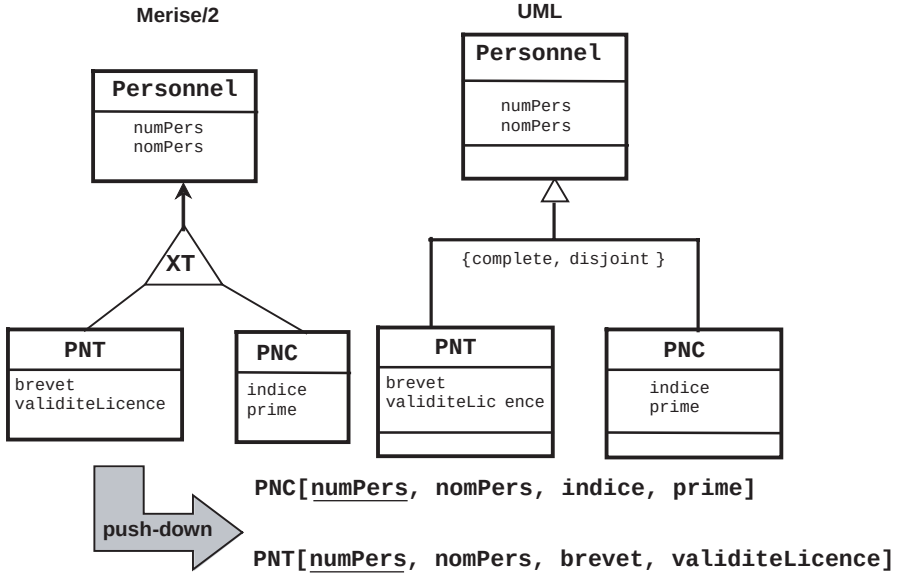
Schéma logique	Script SQL2

PNC[numPers#, indice, prime]
Personnel[numPers, nomPers]
PNT[numPers#, brevet, validiteLicence]
	`CREATE TABLE Personnel`
	`(numPers NUMBER, nomPers VARCHAR(20),`
	`CONSTRAINT pk_Personnel PRIMARY KEY(numPers))`
	
	`CREATE TABLE PNC`
	`(numPers NUMBER,indice NUMBER,prime NUMBER(8,2),`
	`CONSTRAINT pk_PNC PRIMARY KEY(numPers),`
	`CONSTRAINT fk_PNC_Personnel FOREIGN KEY(numPers)`
	`➥REFERENCES Personnel(numPers))`
	
	`CREATE TABLE PNT`
	`(numPers NUMBER, brevet VARCHAR(10),`
	`validiteLicence DATE,`
	`CONSTRAINT pk_PNT PRIMARY KEY(numPers),`
	`CONSTRAINT fk_PNT_Personnel FOREIGN KEY(numPers)`
	`➥REFERENCES Personnel(numPers))`

Décomposition descendante (push-down)

L'héritage de partition 3-15 exprime le fait qu'aucun personnel ne peut être à la fois PNT et PNC, et qu'il n'existe pas non plus de personnel n'étant ni PNT ni PNC. Nous verrons plus loin comment traduire cette contrainte. Il est question ici uniquement de traduire les relations déduites au niveau logique en tables.

Figure 3-15 *Exemple de décomposition descendante* (push-down) *d'une association d'héritage*



Le schéma physique est composé de deux tables dans lesquelles le contenu de la relation issue de la sur-entité (ici `Personnel`) a été intégralement dupliqué. La table `Personnel` n'est pas nécessaire car aucun personnel ne peut être ni PNT ni PNC.

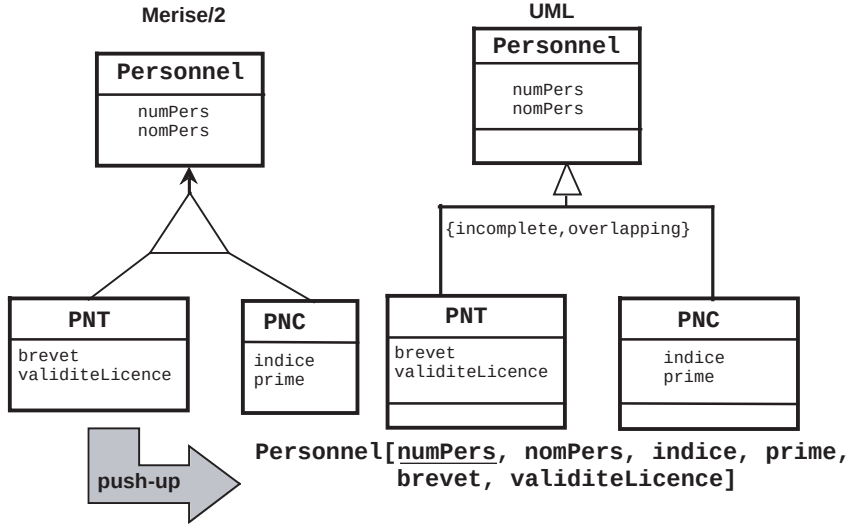
Tableau 3.13 Héritage push-down

Schéma logique	Script SQL2
PNC[numPers, nomPers, indice, prime]	`CREATE TABLE PNC (numPers NUMBER, nomPers VARCHAR(20), indice NUMBER, prime NUMBER(8,2), CONSTRAINT pk_PNC PRIMARY KEY(numPers))`
PNT[numPers, nomPers, brevet, validiteLicence]	`CREATE TABLE PNT (numPers NUMBER, nomPers VARCHAR(20), brevet VARCHAR(10), validiteLicence DATE, CONSTRAINT pk_PNT PRIMARY KEY(numPers))`

Décomposition ascendante (push-up)

L'exemple 3-16 illustre la décomposition ascendante du graphe d'héritage.

Figure 3-16 *Décomposition ascendante* (push-up) *d'une association d'héritage*



Le schéma physique est constitué d'une seule table dans laquelle se trouvent tous les attributs des sous-entités (sous-classes). La table `Personnel` contiendra des valeurs nulles dans certains cas (par exemple pour un personnel seulement PNT, les colonnes `indice` et `prime` vaudront `NULL`).

Tableau 3.14 Héritage push-up

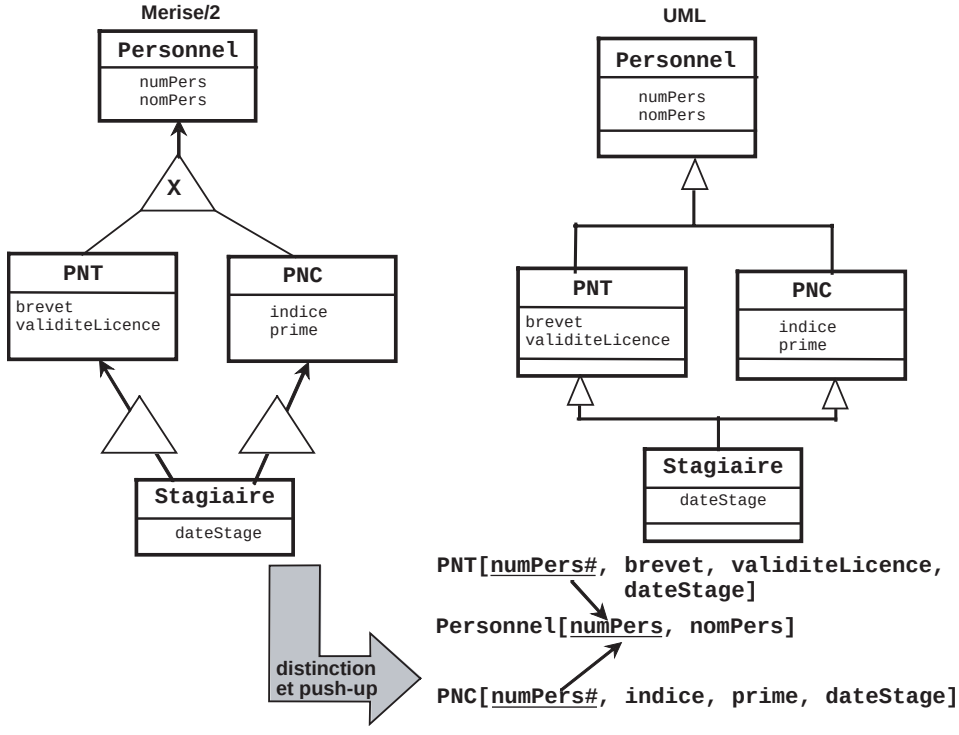
Schéma logique	Script SQL2
Personnel[numPers, nomPers, indice, prime, brevet, validiteLicence]	`CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), indice NUMBER, prime NUMBER(8,2), brevet VARCHAR(10), validiteLicence DATE, CONSTRAINT pk_Personnel PRIMARY KEY(numPers))`

Traduction des associations d'héritage multiple

Les trois familles de décomposition précédemment étudiées peuvent s'appliquer aux associations d'héritage multiple. Chaque association du graphe d'héritage pourra ainsi être traduite soit par distinction, soit de manière descendante, soit de manière ascendante, dans la mesure du possible en fonction des contraintes existantes par ailleurs (contraintes d'héritage mais aussi d'autres contraintes comme une sous-entité connectée à une entité non rattachée au graphe d'héritage, etc.).

Dans le graphe d'héritage suivant, nous choisissons d'utiliser la décomposition par distinction pour le premier niveau d'héritage, et la décomposition ascendante *(push-up)* pour le second niveau d'héritage.

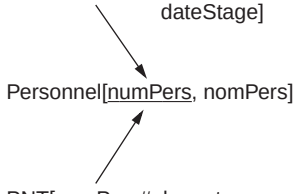
Figure 3-17 *Décomposition d'une association d'héritage multiple*



Le schéma physique sera constitué de trois tables. Les enregistrements des tables `PNT` et `PNC`, dont la colonne `dateStage` sera non nulle, seront des stagiaires.

Tableau 3.15 Héritage multiple

Schéma logique	Script SQL2

	`CREATE TABLE Personnel`
	`(numPers NUMBER, nomPers VARCHAR(20),`
	`CONSTRAINT pk_Personnel PRIMARY KEY(numPers))`
	
	`CREATE TABLE PNC`
	`(numPers NUMBER, indice NUMBER,`
	`prime NUMBER(8,2),` **`dateStage DATE`**`,`
	`CONSTRAINT pk_PNC PRIMARY KEY(numPers),`
	`CONSTRAINT fk_PNC_Personnel FOREIGN KEY(numPers)`
	`REFERENCES Personnel(numPers))`
	
	`CREATE TABLE PNT`
	`(numPers NUMBER, brevet VARCHAR(10),`
	`validiteLicence DATE,` **`dateStage DATE`**`,`
	`CONSTRAINT pk_PNT PRIMARY KEY(numPers),`
	`CONSTRAINT fk_PNT_Personnel FOREIGN KEY(numPers)`
	`REFERENCES Personnel(numPers))`

Traduction des contraintes d'héritage

Nous étudions à présent les moyens qu'offre le langage SQL pour traduire les contraintes possibles d'un graphe d'héritage. Cela faisant, nous pourrons discuter des avantages et inconvénients des différentes décompositions possibles précédemment citées.

Le tableau 3.16 résume les contraintes à mettre en place pour les décompositions d'héritage possibles, à partir de l'exemple en cours. Nous expliquerons comment programmer avec SQL les différents prédicats (numérotés de *A* à *D*) composant chaque contrainte, en fonction du schéma relationnel choisi. Nous considérons que les tables relationnelles sont déjà créées.

Décomposition par distinction

Contrainte de partition

La contrainte de partition exprime le fait qu'il n'existe aucun personnel pouvant être à la fois PNT et PNC (prédicat *A*), et qu'il n'existe pas non plus de personnel n'étant ni PNT ni PNC (prédicat *B*).

Le prédicat *A* se programme à l'aide de déclencheurs *(triggers)* sur les tables `PNT` et `PNC`. Concernant les ajouts dans la table `PNT` et la modification du numéro (`numPers`), le déclencheur devra s'assurer qu'il n'existe pas d'enregistrement `PNC` ayant le même numéro.

Tableau 3.16 Différents cas d'héritage

Héritage \ Décomposition	distinction	descendante	ascendante
Partition Merise/2 :  UML : {complete, disjoint} ∄ un personnel à la fois PNT et PNC *(A)* ∄ un personnel ni PNT ni PNC *(B)*	Personnel[numPers, nomPers] PNC[numPers#, indice, prime] PNT[numPers#, brevet, validiteLicence]	PNC[numPers, nomPers, indice, prime] PNT[numPers, nomPers, brevet, validiteLicence]	Personnel[numPers, nomPers, indice, prime, brevet, validiteLicence]
Totalité Merise/2  UML : {complete, overlapping} ∄ un personnel ni PNT ni PNC *(B)* ∃ un personnel à la fois PNT et PNC *(C)*			
Exclusivité Merise/2 :  UML : {incomplete, disjoint} ou rien ∄ un personnel à la fois PNT et PNC *(A)* ∃ un personnel ni PNT ni PNC *(D)*		Personnel[numPers, nomPers] PNC[numPers#, nomPers, indice, prime] PNT[numPers#, nomPers, brevet, validiteLicence]	
Sans contrainte Merise/2 :  UML : {incomplete, overlapping} ∃ un personnel à la fois PNT et PNC *(C)* ∃ un personnel ni PNT ni PNC *(D)*			

Il en va de même pour le déclencheur de la table PNC qui devra s'assurer qu'il n'existe pas d'enregistrement de PNT ayant même numéro.

Bien que les deux déclencheurs soient écrits avec une syntaxe Oracle, il est aisé de les programmer par analogie avec un autre SGBD.

La clause RAISE_APPLICATION_ERROR génère un ROLLBACK et envoie un message applicatif qui n'est pas dans la plage réservée des erreurs Oracle. Nous verrons dans le cas de la décomposition descendante un exemple d'exécution de ce déclencheur.

La clause NO_DATA_FOUND permet de lever l'exception lors d'une sélection ne ramenant aucun enregistrement.

```
-- déclencheur sur PNT
CREATE TRIGGER Tri_B_IU_PNT
     BEFORE INSERT OR UPDATE OF numPers ON PNT FOR EACH ROW
DECLARE
  num NUMBER;
BEGIN
  SELECT numPers INTO num FROM PNC WHERE numPers = :NEW.numPers;
  RAISE_APPLICATION_ERROR(-20001,'Le personnel'||TO_CHAR(num)||'
est déjà PNC...');
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     NULL;
END;

-- déclencheur sur PNC
CREATE TRIGGER Tri_B_IU_PNC
    BEFORE INSERT OR UPDATE OF numPers ON PNC FOR EACH ROW
DECLARE
  num NUMBER;
BEGIN
  SELECT numPers INTO num FROM PNT WHERE numPers = :NEW.numPers;
  RAISE_APPLICATION_ERROR(-20001,'Le personnel'||TO_CHAR(num)||'
est déjà PNT...');
EXCEPTION
  WHEN NO_DATA_FOUND THEN NULL;
END;
```

Étiquette prédéfinie d'Oracle

Artifice pour éviter que le déclencheur ne renvoie pas d'erreur si le SELECT ne renvoie aucune ligne

Le prédicat *B* se programme à l'aide de deux procédures cataloguées et d'un déclencheur. L'ajout d'un personnel sera répercuté dans la table Personnel et dans la table PNT ou dans la table PNC. Il en va de même pour la modification du numéro d'un personnel ou pour la suppression d'un personnel qui devra répercuter la suppression dans la table PNT ou dans la table PNC.

Les deux procédures cataloguées réalisent respectivement l'ajout d'un personnel PNT et d'un personnel de type PNC. L'appel EXECUTE ajout_PNT(6,'Laurent','4MPY01',SYSDATE) déclenchera deux insertions : une dans la table Personnel avec les valeurs (6,'Laurent'), et l'autre dans la table PNT avec les valeurs (6,'4MPY01',SYSDATE). SYSDATE désignant la date du jour Oracle.

```
-- Ajout d'un PNT
CREATE PROCEDURE ajout_PNT(num NUMBER, nom VARCHAR,
       brevet VARCHAR, licence DATE) IS
BEGIN
  INSERT INTO Personnel VALUES (num,nom);
  INSERT INTO PNT VALUES (num,brevet,licence);
END;
-- Ajout d'un PNC
CREATE PROCEDURE ajout_PNC(num NUMBER, nom VARCHAR,ind NUMBER,
prim NUMBER) IS
BEGIN
  INSERT INTO Personnel VALUES (num,nom);
  INSERT INTO PNC VALUES (num,ind,prim);
END;
```

Les procédures suivantes réalisent respectivement la suppression d'un personnel PNT et d'un personnel de type PNC. On aurait pu également gérer la suppression à l'aide de déclencheurs sur les trois tables.

```
-- Suppression d'un PNT
CREATE PROCEDURE enleve_PNT(num NUMBER) IS
BEGIN
  DELETE FROM PNT       WHERE numPers=num;
  DELETE FROM Personnel WHERE numPers=num;
END;
-- Suppression d'un PNC
CREATE PROCEDURE enleve_PNC(num NUMBER) IS
BEGIN
  DELETE FROM PNC       WHERE numPers=num;
  DELETE FROM Personnel WHERE numPers=num;
END;
```

La modification d'un numéro de personnel au niveau de la table `Personnel` doit être répercutée dans la table `PNT` ou `PNC`. Pour réaliser ces répercussions, nous programmons le déclencheur suivant.

```
-- Répercussion de Personnel vers PNT ou PNC
CREATE TRIGGER Tri_B_U_Personnel
       BEFORE UPDATE OF numPers ON Personnel FOR EACH ROW
BEGIN
     UPDATE PNC SET numPers = :NEW.numPers
              WHERE numPers = :OLD.numPers;
     UPDATE PNT SET numPers = :NEW.numPers
              WHERE numPers = :OLD.numPers;
END;
```

Contrainte de totalité

La contrainte de totalité exprime le fait qu'il n'existe aucun personnel n'étant ni PNT ni PNC (prédicat *B*), et qu'il peut exister un personnel à la fois PNT et PNC (prédicat *C*). Le prédicat *B* a été programmé plus avant. Le prédicat *C* revient à ne pas programmer le prédicat *A* précédent (ne pas mettre en œuvre les déclencheurs des tables `PNT` et `PNC`).

Contrainte d'exclusivité

La contrainte d'exclusivité exprime le fait qu'il n'existe aucun personnel pouvant être à la fois PNT et PNC (prédicat *A*), et qu'il peut exister un personnel ni PNT ni PNC (prédicat *D*). Le prédicat *A* a été programmé plus avant. Le prédicat *D* revient à ne pas programmer le prédicat *B* précédent (ne pas mettre en œuvre les procédures d'ajout et de suppression et le déclencheur de la table `Personnel`).

Sans contrainte

Aucune contrainte n'est à programmer.

Décomposition descendante

Contrainte de partition

Le prédicat *A* se programme à l'aide des mêmes déclencheurs sur les tables `PNT` et `PNC` que pour la décomposition ascendante. Le code suivant illustre une tentative d'ajout d'un PNT, qui est déjà PNC lorsque les déclencheurs sont actifs.

```
-- État des tables
SQL> SELECT * FROM PNT;
  NUMPERS NOMPERS              BREVET     VALIDITE
--------- -------------------- ---------- --------
        1 Tanguy               4MPY68     10/12/02
        3 Laverdure            3MPY69     11/12/03
SQL> SELECT * FROM PNC;
  NUMPERS NOMPERS                  INDICE     PRIME
--------- -------------------- --------- ---------
        2 Natacha                     567   15000,2
-- tentative d'ajout d'un PNT déjà PNC
SQL> INSERT INTO PNT values (2,'Natacha','3MPY99','11-12-2003');
ERREUR à la ligne 1 : ORA-20001: Le personnel 2 est déjà PNC...
ORA-06512: à "SOUTOU.TRI_B_IU_PNT", ligne 6
```

Le prédicat *B* n'a pas à être programmé si la table `Personnel` est inexistante (voir chapitre 2, section *Décomposition ascendante* : quand il existe une contrainte de totalité ou de partition sur l'association d'héritage, il est possible de ne pas traduire la table issue de la sur-entité). En effet, avec les tables `PNT` et `PNC` seules, il n'est pas possible d'avoir un enregistrement ni PNT ni PNC. En revanche, si la table `Personnel` est présente, il faut procéder de la même façon que pour la décomposition ascendante (procédures et déclencheur sur la table `Personnel`).

Contrainte de totalité

La contrainte de partition doit rendre valide le prédicat *B* et il peut exister un personnel à la fois PNT et PNC (prédicat *C*). Nous avons parlé plus haut du prédicat *B*. Le prédicat *C* revient à ne pas programmer le prédicat *A* précédent (ne pas mettre en œuvre les déclencheurs des tables `PNT` et `PNC`).

Contrainte d'exclusivité

Dans ce cas et dans le suivant, la table `Personnel` doit exister pour contenir les employés n'étant ni PNT ni PNC (prédicat *D*). Pour l'exclusivité, le prédicat *A* se programme comme pour la décomposition ascendante. Le prédicat *D* revient à ne pas programmer le prédicat *B* précédente (ne pas mettre en œuvre les procédures d'ajout et de suppression et le déclencheur de la table `Personnel`).

Sans contrainte

Aucun prédicat n'est à programmer et la table `Personnel` doit être présente dans la base.

Décomposition ascendante (push-up)

Dans cette décomposition, on ne retrouve qu'une seule relation au niveau logique, qui devient une table (`Personnel`) au niveau physique. Cette table contient toutes les colonnes issues des sous-entités (sous-classes).

Contrainte de partition

Les prédicats *A* et *B* se programment au niveau de la table `Personnel` à l'aide de contraintes de validation `CHECK` :

- Pour le prédicat *A*, il faut que les colonnes `indice`, `prime`, `brevet` et `validite` ne soient pas toutes initialisées.
- Pour le prédicat *B*, il faut que les colonnes `indice`, `prime`, `brevet` et `validite` ne soient pas toutes nulles.

L'ordre `ALTER TABLE` modifie la structure d'une table (colonne ou contrainte SQL), ici on ajoute deux contraintes.

```
--- prédicat A
ALTER TABLE Personnel ADD CONSTRAINT ck_predicat_A
     CHECK ((indice IS NULL AND prime IS NULL)
        OR  (brevet IS NULL AND validiteLicence IS NULL));
--- prédicat B
ALTER TABLE Personnel ADD CONSTRAINT ck_predicat_B
     CHECK ((indice IS NOT NULL OR prime IS NOT NULL)
        OR  (brevet IS NOT NULL OR validiteLicence IS NOT NULL));
```

Les tentatives d'ajout d'un personnel à la fois PNT et PNC et d'un personnel n'étant ni l'un ni l'autre se traduiront par un échec.

```
-- État de la base
SQL> SELECT * FROM Personnel;
  NUMPERS NOMPERS            INDICE     PRIME BREVET     VALIDITE
--------- --------------- --------- --------- ---------- --------
        1 Tanguy                                4MPY68     10/12/02
        2 Natacha               567   15000,2
        3 Laverdure                             3MPY69     11/12/03
-- Ajout d'un personnel PNT et PNC
SQL> INSERT INTO Personnel VALUES (4,'Soutou', 780, 100.00,
'3MPY93', '11-12-2003');
ERREUR à la ligne 1 : ORA-02290: violation de contraintes (SOU-
TOU.CK_PREDICAT_A) de vérification
-- Ajout d'un personnel ni PNT ni PNC
SQL> INSERT INTO Personnel VALUES (5,'Bidal',NULL,NULL,NULL,NULL);
ERREUR à la ligne 1 : ORA-02290: violation de contraintes (SOU-
TOU.CK_PREDICAT_B) de vérification
```

Contrainte de totalité

Les prédicats *B* et *C* sont à programmer. Nous avons parlé plus haut de la contrainte *B*. La contrainte *C* revient à supprimer la contrainte *A* précédente avec l'option `DROP CONSTRAINT`.

```
--- Désactivation du prédicat A
ALTER TABLE Personnel DROP CONSTRAINT ck_predicat_A;
```

On peut désormais insérer un personnel à la fois PNT et PNC comme le montre le script suivant :

```
SQL> INSERT INTO Personnel VALUES (4,'Soutou', 780, 100.00,
'3MPY93', '11-12-2003');
SQL> SELECT * FROM Personnel;
  NUMPERS NOMPERS            INDICE     PRIME BREVET     VALIDITE
--------- --------------- --------- --------- ---------- --------
        1 Tanguy                                4MPY68     10/12/02
        2 Natacha               567   15000,2
        3 Laverdure                             3MPY69     11/12/03
        4 Soutou                780       100 3MPY93     11/12/03
```

Contrainte d'exclusivité

Il faut redéfinir le prédicat *A* en supprimant au préalable les enregistrements ne répondant pas à ce critère, et programmer le prédicat *D* en supprimant la contrainte vérifiant le prédicat *B*.

```
--- Activation du prédicat A
ALTER TABLE Personnel ADD CONSTRAINT ck_predicat_A
     CHECK ((indice IS NULL AND prime IS NULL)
        OR  (brevet IS NULL AND validiteLicence IS NULL));
--- Activation du predicat D
ALTER TABLE Personnel DROP CONSTRAINT ck_predicat_B;
```

Sans contrainte

Aucune contrainte de validation (`CHECK`) n'est à programmer.

Bilan

Dressons un bilan des avantages et des inconvénients de chacune de ces approches. Aucune des solutions ne constitue la panacée. Il semblerait que la décomposition descendante soit la plus pénalisante. La décomposition ascendante semble être la plus souple au point de vue de la programmation des contraintes, mais n'est pas optimale d'un point de vue stockage des informations. La décomposition par distinction peut paraître comme un compromis entre ces deux solutions.

Tableau 3.17 Bilan des traductions de l'héritage

Décomposition	distinction	descendante	ascendante
Avantages	Pas de redondances de données. Prédicats *C* et *D* implicites.	Diminution du nombre de jointures en exploitation. Prédicats *C* et *D* implicites.	Simplicité des prédicats *A* et *B* à programmer. Prédicats *C* et *D* implicites.
Inconvénients	Prédicats *A* et *B* à programmer.	Redondances possibles de données. Prédicats *A* et *B* à programmer.	Possibilité de valeurs nulles dans les données.

Transformation des agrégations

Pour UML, l'agrégation renforce le couplage d'une association. Pour le modèle entité-association, il s'agit d'affiner une association *n*-aire (qui se traduit en classes-associations dans la notation UML).

Agrégations UML

UML distingue une agrégation partagée (ou simple) d'une composition. L'agrégation partagée se traduit sous SQL comme une simple association. La composition nécessite d'inclure dans la clé primaire de la table composant l'identifiant de la table composite.

L'exemple 3-18 décrit une agrégation simple *plusieurs-à-plusieurs* et une composition *un-à-plusieurs* (une composition est toujours *un-à-plusieurs* ou *un-à-un*).

La première association est agrégée dans le sens où la suppression d'un immeuble doit se répercuter sur la suppression des achats relatifs à cet immeuble. Il ne s'agit pas de détruire les copropriétaires de l'immeuble supprimé, car ils peuvent être copropriétaires d'autres immeubles. En appliquant les règles R1 et R3, on obtient trois relations `CoProprietaire`, `Immeuble` et `Achat`.

La programmation SQL de l'agrégation UML (partagée ou composition) se réalise à l'aide des directives `UPDATE CASCADE` et `DELETE CASCADE` qui permettent de propager la suppression (ou la modification) d'un enregistrement composite au niveau du composant.

Figure 3-18 *Agrégations UML*

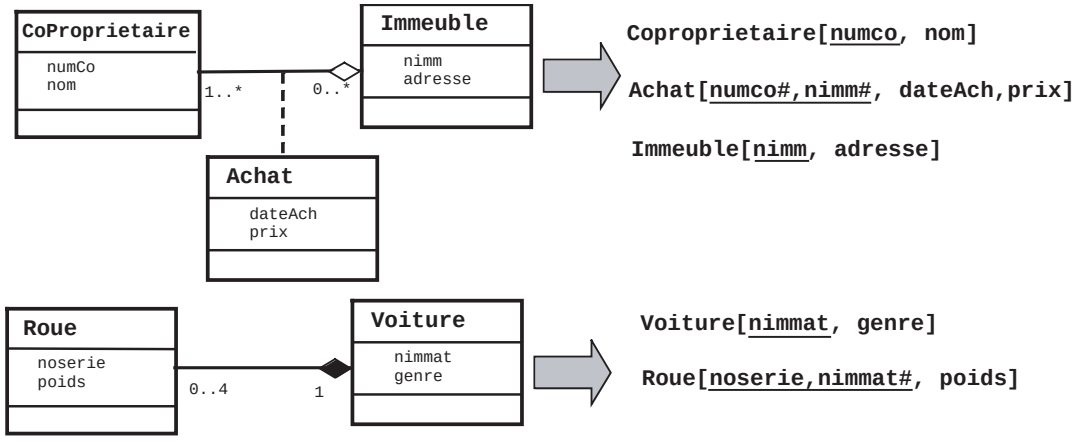


Tableau 3.18 Agrégations UML

Schéma logique	Script SQL2 (programmation MySQL)

	`CREATE TABLE Coproprietaire`
	`(numCo INTEGER, nom CHAR(30),`
	`CONSTRAINT pk_Coproprietaire PRIMARY KEY(numCo));`

CoProprietaire[numCo, nom]	`CREATE TABLE Immeuble`
	`(nimm INTEGER, adresse CHAR(40),`
	`CONSTRAINT pk_Immeuble PRIMARY KEY(nimm));`

Achat[numCo#, nimm#, dateAch, prix]	`CREATE TABLE Achat`
	`(numCo INTEGER, nimm INTEGER, dateAch DATETIME,`
	`prix INTEGER,`
	`CONSTRAINT pk_Achat PRIMARY KEY (numCo,nimm),`
	`CONSTRAINT fk_Achat_numCo_Coproprietaire`
	`FOREIGN KEY(numCo)REFERENCES Coproprietaire(numCo)`
	`**ON DELETE CASCADE ON UPDATE CASCADE**,`
	`CONSTRAINT fk_Achat_nimm_Immeuble`
	`FOREIGN KEY(nimm) REFERENCES Immeuble(nimm)`
	`**ON DELETE CASCADE ON UPDATE CASCADE**);`

Immeuble[nimm, adresse]	

Voiture[nimmat, genre]	`CREATE TABLE Voiture`
	`(nimmat CHAR(10), genre CHAR(30),`
	`CONSTRAINT pk_Voiture PRIMARY KEY(nimmat));`

Roue[noserie#, nimmat#, poids]	`CREATE TABLE Roue`
	`(noserie INTEGER, nimmat CHAR(10), poids INTEGER,`
	`CONSTRAINT pk_Roue PRIMARY KEY(noserie,nimmat),`
	`CONSTRAINT fk_Roue_nimmat_Voiture`
	`FOREIGN KEY (nimmat) REFERENCES Voiture(nimmat)`
	`**ON DELETE CASCADE ON UPDATE CASCADE**);`

Modèle entité-association

En considérant l'agrégat comme une entité dans le passage au niveau logique, il n'y a pas de difficulté particulière au niveau de SQL pour traduire une association reliant l'agrégat à une entité. Puisque les associations d'agrégation du modèle entité-association se traduisent sous UML à l'aide de classes-associations, il est intéressant d'étudier leur transformation avec SQL2.

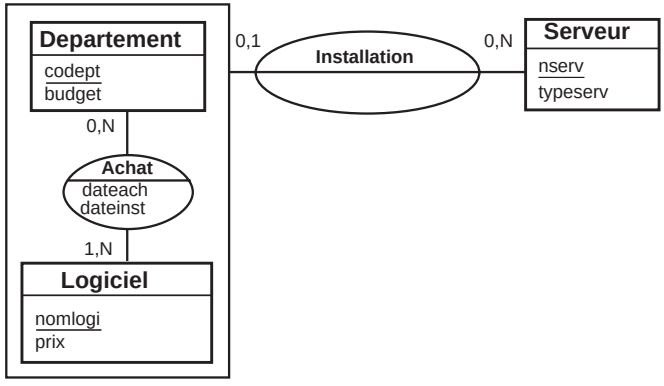
L'exemple 3-19 décrit les installations des logiciels d'un département sur des serveurs. L'association *3*-aire s'affine à l'aide d'une association d'agrégation dans le sens où la base de données ne devra pas stocker des installations de logiciels non achetés. Nous poserons par la suite des conditions supplémentaires de manière à examiner les trois autres cardinalités possibles.

Associations d'agrégation un-à-plusieurs

Un logiciel acheté n'est installé que sur un seul serveur qui peut héberger plusieurs logiciels achetés.

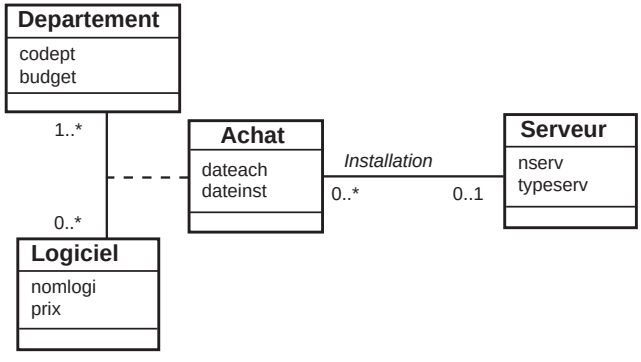
Modèle entité-association

Figure 3-19 *Association d'agrégation un-à-plusieurs*



Notation UML

Figure 3-20 *Classe-association un-à-plusieurs*

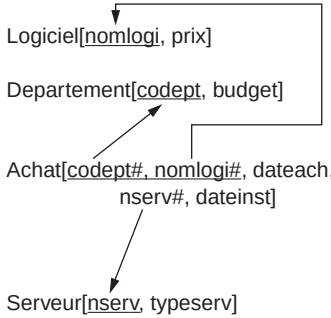


Script SQL2

En appliquant la règle R1, on obtient trois relations `Departement`, `Logiciel` et `Serveur`. En appliquant la règle R3, on obtient la relation `Achat`. La règle R2 traduit l'association `Installation` en faisant migrer dans la relation *fils* (ici l'agrégat `Achat`) l'identifiant de la relation *père* (ici `Serveur`).

Tableau 3.19 Association d'agrégation un-à-plusieurs

Schéma logique	Script SQL2 (Oracle)

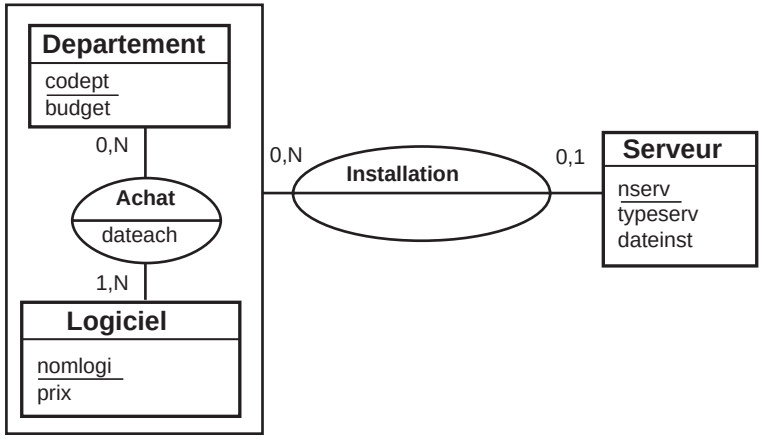
	`CREATE TABLE logiciel`
	`(nomlogi VARCHAR(20), prix NUMBER,`
	` CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi))`
	
	`CREATE TABLE departement`
	`(codept VARCHAR(6), budget NUMBER,`
	` CONSTRAINT pk_departement PRIMARY KEY(codept))`
	
	`CREATE TABLE serveur`
	`(nserv NUMBER, typeserv VARCHAR(20),`
	` CONSTRAINT pk_serveur PRIMARY KEY(nserv))`
	
	`CREATE TABLE achat`
	`(codept VARCHAR(6), nomlogi VARCHAR(20),`
	` dateach DATE, nserv NUMBER, dateinst DATE,`
	` CONSTRAINT pk_achat PRIMARY KEY (codept,nomlogi),`
	` CONSTRAINT fk_achat_codept_departement`
	`FOREIGN KEY(codept)`
	`REFERENCES departement(codept),`
	` CONSTRAINT fk_achat_nomlogi_logiciel`
	`FOREIGN KEY(nomlogi)`
	`REFERENCES logiciel(nomlogi),`
	` CONSTRAINT fk_achat_nserv_serveur`
	`FOREIGN KEY(nserv) REFERENCES serveur(nserv))`

Associations d'agrégation plusieurs-à-un

Un serveur n'héberge qu'un seul logiciel acheté qui peut être installé sur plusieurs serveurs.

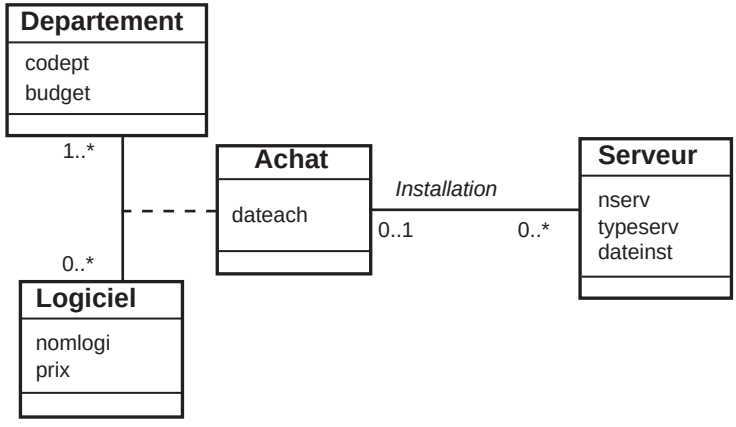
Modèle entité-association

Figure 3-21 *Association d'agrégation plusieurs-à-un*



Notation UML

Figure 3-22 *Classe-association plusieurs-à-un*

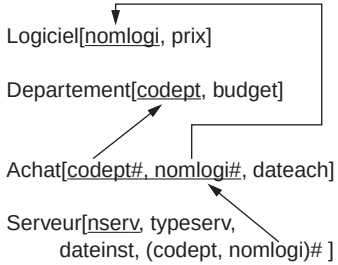


Script SQL2

En appliquant la règle R1, on obtient les relations `Departement`, `Logiciel` et `Serveur`. En appliquant la règle R3, on obtient la relation `Achat`. La règle R2 traduit l'association `Installation` en faisant migrer dans la relation *fils* (ici `Serveur`) l'identifiant de la relation *père* (le couple `codept`,`nomlogi` de l'agrégat `Achat`). Nous indiquons en gras cette migration dans le script SQL2 du tableau 3.20.

Tableau 3.20 Association d'agrégation plusieurs-à-un

Schéma logique	Script SQL2 (Oracle)

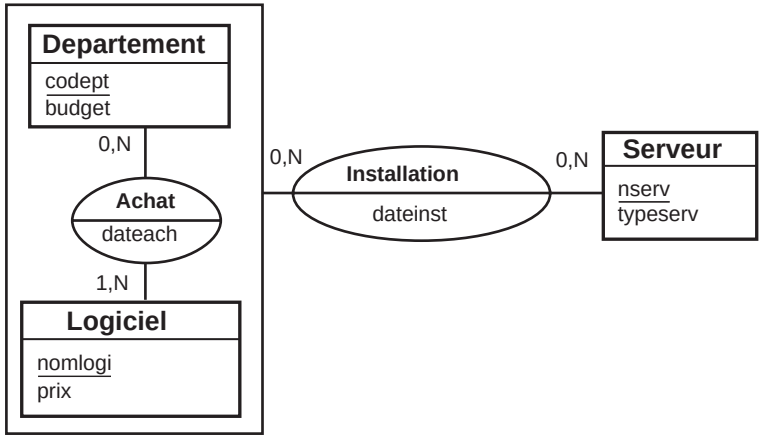
	`CREATE TABLE logiciel`
	`(nomlogi VARCHAR(20), prix NUMBER,`
	` CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi))`
	
	`CREATE TABLE departement`
	`(codept VARCHAR(6), budget NUMBER,`
	` CONSTRAINT pk_departement PRIMARY KEY(codept))`
	
	`CREATE TABLE achat`
	`(codept VARCHAR(6), nomlogi VARCHAR(20),`
	` dateach DATE,`
	` CONSTRAINT pk_achat PRIMARY KEY`
	`(codept,nomlogi),`
	` CONSTRAINT fk_achat_codept_departement`
	`FOREIGN KEY(codept)`
	`REFERENCES departement(codept),`
	` CONSTRAINT fk_achat_nomlogi_logiciel`
	`FOREIGN KEY(nomlogi)`
	`REFERENCES logiciel(nomlogi));`
	
	`CREATE TABLE serveur`
	`(nserv NUMBER, typeserv VARCHAR(20),`
	` dateinst DATE,` **`codept VARCHAR(6),`**
	` **nomlogi VARCHAR(20),**`
	` CONSTRAINT pk_serveur PRIMARY KEY(nserv),`
	` **CONSTRAINT fk_serveur_achat**`
	`FOREIGN KEY(codept,nomlogi)`
	`REFERENCES achat(codept,nomlogi))`

Associations d'agrégation plusieurs-à-plusieurs

Un serveur héberge plusieurs logiciels achetés. Un logiciel pouvant être installé sur différents serveurs.

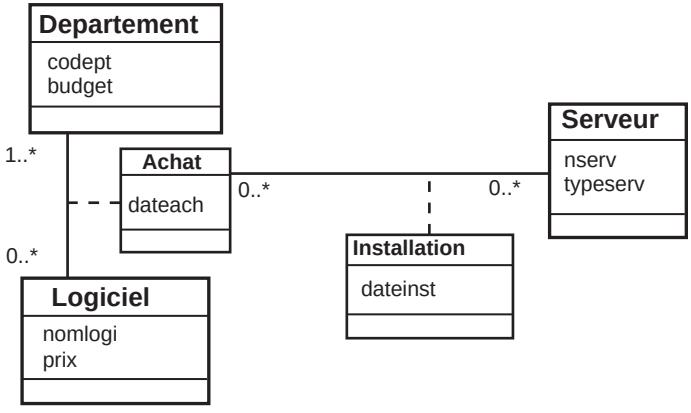
Modèle entité-association

Figure 3-23 *Association d'agrégation plusieurs-à-plusieurs*



Notation UML

Figure 3-24 *Classe-association plusieurs-à-plusieurs*



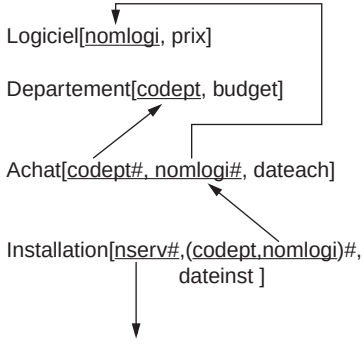
Script SQL2

En appliquant la règle R1, on obtient les relations `Departement`, `Logiciel` et `Serveur`. La règle R3 s'applique pour les deux associations *plusieurs-à-plusieurs* (`Achat` et `Installation`). La clé primaire de la relation `Installation` contient les identifiants des entités/classes connectées (`nserv` avec le couple `codept`,`nomlogi` de l'agrégat représenté par la classe-association `Achat`). Comme ce couple est par ailleurs clé primaire de la relation `Achat`, il est devient clé étrangère dans la relation `Installation`.

C'est cette contrainte de clé étrangère qui renforce l'intégrité de la base de données, car il ne sera pas possible d'ajouter l'installation d'un logiciel si ce précédent n'a pas été acheté au préalable par le département. Ce schéma est plus intègre qu'un schéma traduisant l'association *3*-aire sans contrainte entre `Departement`, `Logiciel` et `Serveur`. La clé étrangère qui traduit l'association d'agrégation est indiquée en gras.

Tableau 3.21 Association d'agrégation plusieurs-à-plusieurs

Schéma logique	Script SQL2 (Oracle)

	`CREATE TABLE logiciel`
	`(nomlogi VARCHAR(20), prix NUMBER,`
	`CONSTRAINT pk_logiciel`
	`PRIMARY KEY(nomlogi))`
	`CREATE TABLE departement`
	`(codept VARCHAR(6), budget NUMBER,`
	`CONSTRAINT pk_departement`
	`PRIMARY KEY(codept))`
	`CREATE TABLE serveur`
	`(nserv NUMBER, typeserv VARCHAR(20),`
	`CONSTRAINT pk_serveur`
	`PRIMARY KEY(nserv))`
	`CREATE TABLE achat`
	`(codept VARCHAR(6), nomlogi VARCHAR(20),`
	`dateach DATE,`
	`CONSTRAINT pk_achat`
	`PRIMARY KEY(codept,nomlogi),`
	`CONSTRAINT fk_achat_codept_departement`
	`FOREIGN KEY(codept)`
	`REFERENCES departement(codept),`
	`CONSTRAINT fk_achat_nomlogi_logiciel`
	`FOREIGN KEY(nomlogi)`
	`REFERENCES logiciel(nomlogi))`
	`CREATE TABLE installation`
	`(nserv NUMBER,` **`codept VARCHAR(6),`**
	`nomlogi VARCHAR(20)``, dateinst DATE,`
	`CONSTRAINT pk_installation`
	`PRIMARY KEY(nserv,codept,nomlogi),`
	`CONSTRAINT fk_installation_serveur`
	`FOREIGN KEY(nserv)`
	`REFERENCES serveur(nserv),`
	`CONSTRAINT fk_installation_achat`
	`FOREIGN KEY(codept,nomlogi)`
	`REFERENCES achat(codept,nomlogi))`

Traduction des contraintes

Cette section est consacrée à la programmation SQL2 des différentes contraintes du niveau conceptuel (partition, exclusivité, totalité, inclusion et simultanéité).

Contrainte de partition

Considérons l'exemple des pilotes (caractérisés par un numéro, un nom et un grade) qui partent soit en mission sanitaire, soit en mission d'entraînement. La contrainte de partition détermine le fait qu'aucun pilote n'est au repos ou ne mène de front des missions des deux types. Les modèles conceptuels sont illustrés figures 1-37 et 1-40.

La partition se programme par une contrainte SQL de validation (CHECK).

Tableau 3.22 Contrainte de partition

Schéma logique	Script SQL2 (Oracle)

	`CREATE TABLE sanitaire`
	`(codesan VARCHAR(10),`
	`organisme VARCHAR(20),`
	`CONSTRAINT pk_sanitaire PRIMARY KEY(codesan))`

Sanitaire[codesan, organisme]	`CREATE TABLE entrainement`
	`(codent VARCHAR(10), datent DATE,`
	`region VARCHAR(20),`
	`CONSTRAINT pk_entraine PRIMARY KEY(codent))`

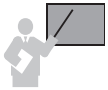
Pilote[numpil, nom, grade, sani#, entraine#]	`CREATE TABLE pilote`
	`(numpil NUMBER, nom VARCHAR(10),`
	`grade VARCHAR(10),`
	`sani VARCHAR(10), entraine VARCHAR(10),`
	`CONSTRAINT fk_pilote_sanitaire`
	`FOREIGN KEY(sani)`
	`REFERENCES sanitaire(codesan),`
	`CONSTRAINT fk_pilote_entrainement`
	`FOREIGN KEY(entraine)`
	`REFERENCES entrainement(codent),`

Entrainement[codent, datent, region]	`CONSTRAINT ck_partition CHECK`
	`((sani IS NOT NULL OR entraine IS NOT NULL)`
	`AND NOT`
	`(sani IS NOT NULL AND entraine IS NOT NULL)),`

	`CONSTRAINT pk_pilote PRIMARY KEY(numpil))`

Contrainte d'exclusivité

Dans notre exemple, un pilote peut être au repos (affecté à aucune mission). Si un pilote est affecté à un exercice d'entraînement, alors il ne peut pas être affecté à une mission sanitaire et réciproquement.

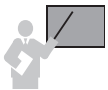


L'exclusivité se programme par une contrainte SQL de validation (CHECK).

```
CREATE TABLE pilote
(numpil NUMBER, nom VARCHAR(10),
 grade  VARCHAR(10), sani  VARCHAR(10), entraine VARCHAR(10),
 CONSTRAINT fk_pilote_sanitaire
        FOREIGN KEY(sani)REFERENCES sanitaire(codesan),
 CONSTRAINT fk_pilote_entrainement
        FOREIGN KEY(entraine) REFERENCES entrainement(codent),
CONSTRAINT ck_exclusivite CHECK
   (sani ISNULL OR entraine IS NULL),
CONSTRAINT pk_pilote PRIMARY KEY(numpil))
```

Contrainte de totalité

Dans notre exemple, un pilote peut être affecté à la fois à une mission sanitaire et à un exercice d'entraînement, et tous les pilotes participent à au moins une mission.

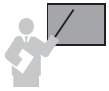


La totalité se programme par une contrainte SQL de validation (CHECK).

```
CREATE TABLE pilote
(numpil NUMBER, nom VARCHAR(10),
 grade  VARCHAR(10), sani VARCHAR(10), entraine VARCHAR(10),
 CONSTRAINT fk_pilote_sanitaire
        FOREIGN KEY(sani) REFERENCES sanitaire(codesan),
 CONSTRAINT fk_pilote_entrainement
        FOREIGN KEY(entraine) REFERENCES entrainement(codent),
CONSTRAINT ck_totalité CHECK
➥(NOT (sani IS NULL AND entraine IS NULL)),
CONSTRAINT pk_pilote PRIMARY KEY(numpil))
```

Contrainte de simultanéité

Dans notre exemple, un pilote peut être affecté à la fois à une mission sanitaire et à un exercice d'entraînement. En outre, il peut n'être affecté à aucune mission.



La simultanéité se programme par une contrainte SQL de validation (CHECK).

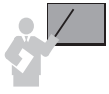
```
CREATE TABLE pilote
 (numpil NUMBER, nom VARCHAR(10), grade VARCHAR(10),
  sani VARCHAR(10), entraine VARCHAR(10),
  CONSTRAINT fk_pilote_sanitaire FOREIGN KEY(sani)
   REFERENCES sanitaire(codesan),
  CONSTRAINT fk_pilote_entrainement FOREIGN KEY(entraine)
   REFERENCES entrainement(codent),
  CONSTRAINT ck_simultaneite CHECK
  ➥((sani IS NULL AND entraine IS NULL) OR
  ➥(sani IS NOT NULL AND entraine IS NOT NULL)),
CONSTRAINT pk_pilote PRIMARY KEY(numpil))
```

Contrainte d'inclusion

Selon la contrainte d'inclusion, toutes les occurrences d'une association doivent être incluses dans les occurrences d'une autre association.

Entre deux associations binaires

Considérons l'exemple 1-43 (1-47 pour la notation UML) dans lequel chaque étudiant formule des vœux concernant des stages. Pour modéliser le fait qu'un étudiant accède à un stage à condition qu'il le demande explicitement, il faut utiliser une contrainte d'inclusion. Les règles R1 et R3 s'appliquent pour traduire l'association `Voeux`. Les règles R1 et R4 s'appliquent pour traduire l'association `Effectuer`.



L'inclusion entre deux associations se programme par une contrainte SQL de clé étrangère (`FOREIGN KEY`).

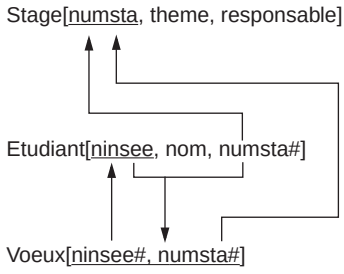
Puisqu'il n'est pas possible de déclarer la contrainte lors de la création de la table `Etudiant` (car la table `Voeux` référence la table `Etudiant`), il faut utiliser l'instruction `ALTER TABLE`.

Entre trois associations binaires

Considérons l'exemple 1-45 dans lequel des départements achètent des logiciels. La contrainte d'inclusion exprime qu'un logiciel *l* acheté par le département *d* est installé sur un serveur *s*, destiné, entre autres, à ce département.

Tableau 3.23 Contraintes d'inclusion

Schéma logique	Script SQL2 (Oracle)

	`CREATE TABLE stage`
	`(numsta VARCHAR(10), theme VARCHAR(20),`
	`responsable VARCHAR(20),`
	`CONSTRAINT pk_stage PRIMARY KEY(numsta))`
	
	`CREATE TABLE etudiant`
	`(ninsee VARCHAR(13), nom VARCHAR(30),`
	`numsta VARCHAR(10),`
	`CONSTRAINT pk_etudiant PRIMARY KEY(ninsee),`
	`CONSTRAINT fk_etudiant_stage`
	`FOREIGN KEY(numsta) REFERENCES stage(numsta))`
	
	`CREATE TABLE voeux`
	`(ninsee VARCHAR(13), numsta VARCHAR(10),`
	`CONSTRAINT fk_voeux_stage`
	`FOREIGN KEY(numsta) REFERENCES stage(numsta),`
	`CONSTRAINT fk_voeux_etudiant FOREIGN KEY(ninsee)`
	`REFERENCES etudiant(ninsee),`
	`CONSTRAINT pk_voeux PRIMARY KEY(ninsee,numsta))`
	
	`ALTER TABLE etudiant ADD CONSTRAINT fk_inclusion`
	`FOREIGN KEY(ninsee,numsta)`
	`REFERENCES voeux(ninsee,numsta))`

On peut formuler cette contrainte de la manière suivante : les occurrences de l'association `Installe` doivent être incluses dans les occurrences issues de la jointure entre les associations `Achat` et `Utilisation` (voir la figure 3-25).

L'inclusion entre trois associations (et plus) se programme à l'aide d'un un déclencheur (situé dans la table ciblée par la contrainte).

```
CREATE TABLE Logiciel
(nomlogi VARCHAR(20), editeur VARCHAR(30),
 CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi));
CREATE TABLE Departement
(codedept VARCHAR(8), nomdept VARCHAR(30), budget NUMBER,
 CONSTRAINT pk_departement PRIMARY KEY(codedept));
CREATE TABLE Serveur
(nomserv VARCHAR(8), typeserv VARCHAR(30),
 CONSTRAINT pk_serveur PRIMARY KEY(nomserv));

CREATE TABLE Achat
```

```
(codedept VARCHAR(8), nomlogi VARCHAR(20), dateachat DATE,
 CONSTRAINT pk_Achat PRIMARY KEY(codedept,nomlogi),
 CONSTRAINT fk_Achat_dept FOREIGN KEY(codedept)
            REFERENCES Departement(codedept),
 CONSTRAINT fk_Achat_logi FOREIGN KEY(nomlogi)
            REFERENCES Logiciel(nomlogi));
CREATE TABLE Utilisation
(codedept VARCHAR(8), nomserv VARCHAR(8),
 CONSTRAINT pk_Utilisation PRIMARY KEY(codedept,nomserv),
 CONSTRAINT fk_Utilisation_serv FOREIGN KEY(nomserv)
            REFERENCES Serveur(nomserv),
 CONSTRAINT fk_Utilisation_dept FOREIGN KEY(codedept)
            REFERENCES Departement(codedept));
CREATE TABLE Installe
(nomserv VARCHAR(8), nomlogi VARCHAR(20),
 CONSTRAINT pk_Installe PRIMARY KEY(nomserv,nomlogi),
 CONSTRAINT fk_Installe_serv FOREIGN KEY(nomserv)
            REFERENCES Serveur(nomserv),
 CONSTRAINT fk_Installe_logi FOREIGN KEY(nomlogi)
            REFERENCES Logiciel(nomlogi));
```

```
CREATE TRIGGER tri_contrainte_inclusion
BEFORE INSERT ON Installe FOR EACH ROW
DECLARE
   V1 VARCHAR(20);
   V2 VARCHAR(8);
BEGIN
SELECT a.nomlogi,u.nomserv INTO v1,v2
  FROM Achat a, Utilisation u
  WHERE a.codedept = u.codedept
  AND a.nomlogi = :NEW.nomlogi
  AND u.nomserv = :NEW.nomserv;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RAISE_APPLICATION_ERROR(-20100, 'Le logiciel doit être
          placé sur un serveur du département acheteur');
END;
```

Le code suivant décrit deux insertions dans la table `Installe` (la première est correcte du fait des données de l'exemple 3-25, la seconde, incohérente, est rejetée automatiquement par le déclencheur).

```
SQL> INSERT INTO Installe VALUES ('S2','L1');
1 ligne créée.
SQL> INSERT INTO Installe VALUES ('S3','L2');
ERREUR à la ligne 1 : ORA-20100: Le logiciel doit être placé sur un
serveur du département acheteur
ORA-06512: à "SOUTOU.TRI_CONTRAINTE_INCLUSION", ligne 12
```

Figure 3-25 *Représentation tabulaire des occurrences*

Achat

Dept.	Logiciel
D1	L1
D1	L2
D1	L4
D2	L1

Utilisation

Dept.	Serveur
D1	S1
D1	S2
D2	S3

Du modèle objet à SQL3

Alors que la majorité des outils de conception sont capables de générer des classes C++, Java et C# à partir de spécifications UML, aucun n'est adapté à la génération de script SQL3 pour la conception de bases de données objet-relationnelles.

Cette section présente les techniques permettant de dériver un diagramme UML en langage SQL3. Nous utilisons la syntaxe d'Oracle mais l'analogie avec un autre SGBD incluant les références, collections et l'héritage est réalisable à moindre frais.

Traduction des classes UML

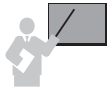
Une classe UML se traduit en un type de données avec la syntaxe CREATE TYPE… et une table objet-relationnelle issue de ce type.

Création d'un type

La commande CREATE TYPE déclare une structure de données simple ou complexe pouvant contenir des déclarations de méthodes. Cette structure est utilisée dans un programme pour définir des objets non persistants ou pour former des tables objet-relationnelles. Le code suivant définit trois types SQL3. Les deux premiers types sont des structures de données simples. Le troisième type inclut deux références (attributs REF) vers des types prédéfinis.

```
CREATE TYPE sanitaire_type AS OBJECT
       (codesan CHAR(10), organisme CHAR(20))
CREATE TYPE entraine_type AS OBJECT
       (codent CHAR(10), datent DATE, region CHAR(20))
CREATE TYPE pilote_type AS OBJECT
      (numpil NUMBER, nom CHAR(10), grade CHAR(10),
       refSani REF sanitaire_type, refEntraine REF entraine_type)
```

Création d'une table objet-relationnelle



La commande CREATE TABLE *nomtable* OF *nomdetype* déclare une table objet-relationnelle issue d'un type. Chaque enregistrement de la table est doté d'un unique OID (Object IDentifier) qui pourra être utilisé par d'autres objets comme une référence.

La table objet-relationnelle `Pilote`, qui pourra stocker des objets de type `pilote_type`, est déclarée sous SQL3 de la manière suivante.

```
CREATE TABLE Pilote OF pilote_type
```

Le script permettant de supprimer le schéma entier doit détruire d'abord les tables puis les types dans l'ordre inverse de leur création.

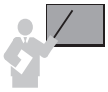
```
DROP TABLE Pilote
DROP TYPE pilote_type
DROP TYPE entraine_type
DROP TYPE sanitaire_type
```

La commande DESC affiche la structure d'un type ou d'une table. On retrouve la déclaration des deux références au niveau de la table.

```
SQL> DESC pilote
 Nom                                Null ?   Type
 ---------------------------------- -------- ---------------
 NUMPIL                                      NUMBER
 NOM                                         CHAR(10)
 GRADE                                       CHAR(10)
 SANI                                        REF OF SANITAIRE_
 ENTRAINE                                    REF OF ENTRAINE_T
```

Associations un-à-un

Examinons les différentes possibilités qu'offre SQL3 pour implanter une association *un-à-un* entre deux classes `C1` et `C2`. Nous recensons deux familles de possibilités.



Solutions 1 et 2 Définition d'un attribut `REF` ou `FOREIGN KEY` dans le type qui décrit `C1`. Cet attribut pointe vers le type qui décrit `C2`. La deuxième solution est symétrique à la première.

Solution 3 Définition d'un type qui contient deux attributs `REF` ou `FOREIGN KEY` vers les types décrivant `C1` et `C2`.

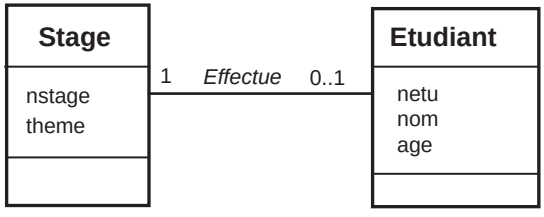
En utilisant deux attributs `REF` dans la dernière solution, on obtient la solution universelle, qui convient à tous les types d'associations.

Afin de maintenir la cohérence des cardinalités maximales `1`, il conviendrait de définir une contrainte `UNIQUE` sur chaque référence. Pour les attributs de type `FOREIGN KEY`, l'opération est possible. En revanche, pour les attributs de type `REF`, ce n'est pas toujours permis par le SGBD.

L'exemple suivant concerne l'association *un-à-un* entre `Stage` et `Etudiant`.

Notation UML

Figure 3-26 *Association un-à-un UML*



Script SQL3

Adoptons la première solution d'implantation. La multiplicité minimale 1 côté `Stage` se traduit à l'aide d'une contrainte de type `NOT NULL` sur la référence. Oracle permet en outre de limiter la portée par la directive `SCOPE IS` ou de poser une contrainte de clé étrangère `REFERENCES` (voir tableau 3-25).

Tableau 3.24 Association un-à-un SQL3

Schéma logique	Script SQL3 (Oracle)

Stage[nstage, theme]
Etudiant[netu, nom, age, refSta]
	`CREATE TYPE stage_type AS OBJECT`
	`(nstage CHAR(4), theme CHAR(30))`
	
	`CREATE TYPE etudiant_type AS OBJECT`
	`(netu CHAR(4), nom CHAR(30), age NUMBER,`
	`refSta REF stage_type)`
	
	`CREATE TABLE Stage OF stage_type`
	`(CONSTRAINT pk_stage PRIMARY KEY(nstage));`
	
	`CREATE TABLE Etudiant OF etudiant_type`
	`(CONSTRAINT pk_etudiant PRIMARY KEY(netu),`
	`CONSTRAINT nn_refSta CHECK (refSta IS NOT NULL),`
	`CONSTRAINT scope_refSta refSta SCOPE IS Stage)`

À propos des références

Bien que les références permettent d'implanter une association entre deux tables, comme le faisaient les clés étrangères, elles n'offrent pas encore toutes les fonctionnalités de ces dernières. Il est possible que l'intégrité référentielle soit à programmer partiellement ou totalement.

Avec Oracle, il est possible de savoir si l'objet *père* d'un objet *fils* donné a été supprimé par la directive `DANGLING` [SOU 04]. En revanche, il n'est pas encore possible de définir une clé primaire, une contrainte `UNIQUE` ou un index sur une référence.

Que reste-t-il aux références ? La limitation du nombre de jointures entre des tables et la possibilité de définir des vues objet-relationnelles de tables SQL2.

Associations un-à-plusieurs

Nous recensons trois bases de travail sous SQL3 pour décrire une association *un-à-plusieurs* entre deux classes `C1` *(père)* et `C2` *(fils)*.

Solution 1 Définition d'une collection (`NESTED TABLE` ou `VARRAY` pour Oracle) dans le type dérivé de `C1`. Cette collection contient une référence vers le type dérivé de `C2`.

Solution 2 Définition d'un attribut `REF` ou `FOREIGN KEY` dans le type dérivé de `C2` qui référence le type dérivé de `C1`.

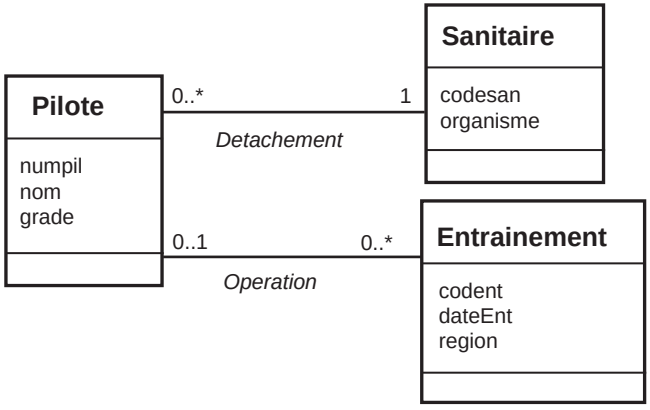
Solution 3 Définition d'un troisième type contenant deux attributs `REF` ou `FOREIGN KEY` qui référencent les types dérivés de `C1` et `C2`.

La dernière solution correspond à la solution universelle.

Dans l'exemple 3-27, choissisons de dériver l'association `Detachement` par la première solution, et l'autre avec la deuxième solution.

Notation UML

Figure 3-27 *Associations un-à-plusieurs UML*



Script SQL3

Pour traduire l'association `Detachement`, la référence `refSani` est placée dans le type *fils* (`pilote_type`) et une contrainte de clé étrangère est définie au niveau de la table `Pilote`. La multiplicité 1 du côté `Sanitaire` est programmée à l'aide d'une contrainte `NOT NULL`. Une collection de références est utilisée pour l'association `Operation` (voir tableau 3-25).

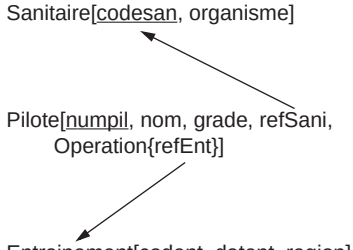
À propos des collections

L'utilisation d'une collection peut privilégier l'accès aux données au niveau de la table qui la contient. Il n'est en général pas possible de définir des contraintes de clé étrangère au niveau d'une référence dans une collection.

Dans notre exemple, l'extraction des missions d'entraînement pour un pilote donné sera plus aisée que l'obtention de la liste des pilotes concernés par une mission d'entraînement.

Si on utilise la solution universelle, il convient de définir une contrainte `UNIQUE` sur la référence qui répertorie la classe affectée de la multiplicité `*`. Oracle ne permet pas de définir une telle contrainte sur un attribut `REF`.

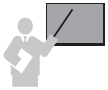
Tableau 3.25 Association un-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)
	CREATE TYPE sanitaire_type AS OBJECT (codesan CHAR(10), organisme CHAR(20)) CREATE TYPE entrainement_type AS OBJECT (codent CHAR(10),dateEnt DATE,region CHAR(20)) CREATE TYPE elt_collection_ent AS OBJECT (refEnt REF entrainement_type) CREATE TYPE collection_ent **AS TABLE OF** elt_collection_ent CREATE TYPE pilote_type AS OBJECT (numpil NUMBER, nom CHAR(10), grade CHAR(10), **refSani REF sanitaire_type,** Operation collection_ent) CREATE TABLE Sanitaire OF sanitaire_type (CONSTRAINT pk_sanitaire PRIMARY KEY(codesan)) CREATE TABLE Entrainement OF entrainement_type (CONSTRAINT pk_ent PRIMARY KEY(codent)) CREATE TABLE Pilote OF pilote_type (CONSTRAINT pk_pilote PRIMARY KEY(numpil), CONSTRAINT nn_refSani **CHECK** **(refSani IS NOT NULL),** CONSTRAINT fk_refSani **refSani REFERENCES** Sanitaire) NESTED TABLE Operation STORE AS temp

Pour profiter des avantages des collections de références (accès rapide aux données par une table sans jointures) sans en subir les inconvénients (requête qui interroge la table non privilégiée et qui nécessite une jointure avec la table contenant la collection), il est possible de définir des vues objet-relationnelles [SOU 04], qui privilégieront tantôt l'accès *via* une table, tantôt l'accès par une autre. Les données pourront être stockées sous la forme relationnelle ou suivant la solution universelle.

Associations plusieurs-à-plusieurs

Nous recensons deux bases de travail sous SQL3 pour décrire une association *plusieurs-à-plusieurs* entre deux classes `C1` et `C2`. La première famille de solutions privilégie un accès aux données par une table.



Solutions 1 et 2 Définition d'une collection dans le type dérivé de `C1`. Cette collection contient une référence vers le type dérivé de `C2`. La deuxième solution est l'inverse de la première, à savoir que la collection se trouve dans `C2` et référence `C1`.

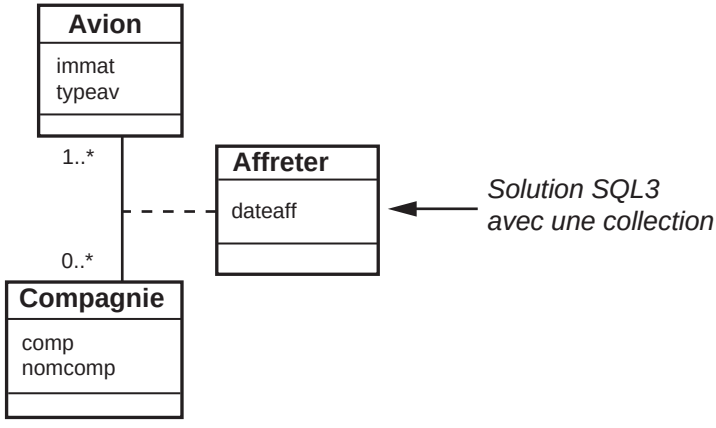
Solution 3 Définition d'un type contenant deux attributs `REF` ou `FOREIGN KEY`, qui référencent les types dérivés de `C1` et `C2`.

La dernière solution avec deux références correspond à la solution universelle.

Dérivons l'association `Affreter` de l'exemple 3-28 à l'aide de la première solution (en privilégiant les accès aux données par les compagnies).

Notation UML

Figure 3-28 *Association plusieurs-à-plusieurs UML*



Script SQL3

La collection `collAvi` contient la référence `refAvi` et l'attribut de l'association.

Tableau 3.26 Association plusieurs-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)

Compagnie[comp, nomcomp, collAvi{refAvi, dateaff}]
↓
Avion[immat, typeav]
	`CREATE TYPE avion_type AS OBJECT`
	`(immat VARCHAR(6), typav VARCHAR(10))`
	
	`CREATE TYPE elt_coll_avi AS OBJECT`
	`(refAvi REF avion_type, dateaff DATE)`
	
	`CREATE TYPE coll_avi_type AS TABLE OF elt_coll_avi`
	
	`CREATE TYPE compagnie_type AS OBJECT`
	`(comp VARCHAR(4), nomcomp VARCHAR(30),`
	`collAvi coll_avi_type)`
	
	`CREATE TABLE Avion OF avion_type`
	`(CONSTRAINT pk_avion PRIMARY KEY(immat))`
	
	`CREATE TABLE Compagnie OF compagnie_type`
	`(CONSTRAINT pk_compagnie PRIMARY KEY(comp))`
	`NESTED TABLE collAvi STORE AS tempAvi`

Associations *n*-aires

Nous recensons *n*+1 bases de travail sous SQL3 pour décrire une association *n*-aire entre *n* classes `C1`, `C2`... `Cn`. La première solution ne privilégie aucun accès aux données, il s'agit de la solution universelle. Les autres solutions privilégient l'accès aux données par une des *n* tables.

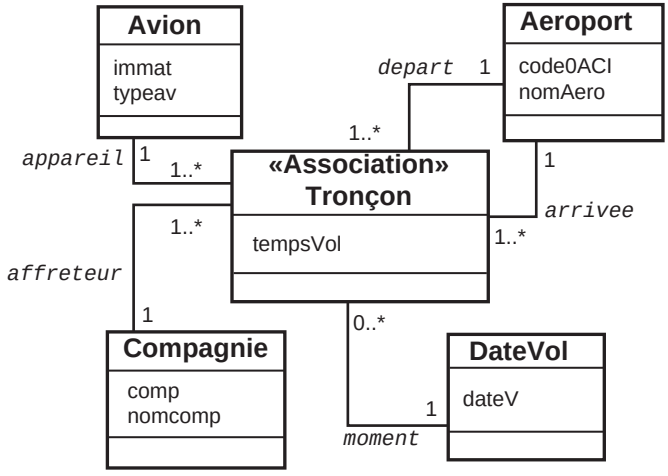
Solutions 1 Définition d'un type contenant *n* attributs `REF` ou `FOREIGN KEY`, qui référencent les *n* types dérivés des classes `C1` à `Cn`.

Solutions 2 à *n*+1 Définition d'une collection dans un type dérivé de la classe `Cj`. Cette collection contient *n*-1 références vers les types dérivés des *n*-1 autres classes.

L'exemple 3-29 modélise les temps de vol des avions sur chaque tronçon. Un tronçon est caractérisé par un aéroport de départ et un aéroport d'arrivée. Supposons qu'on désire aussi connaître la compagnie qui a affrété le vol, la date du vol et l'appareil.

Notation UML

Figure 3-29 *Association* n-aire UML

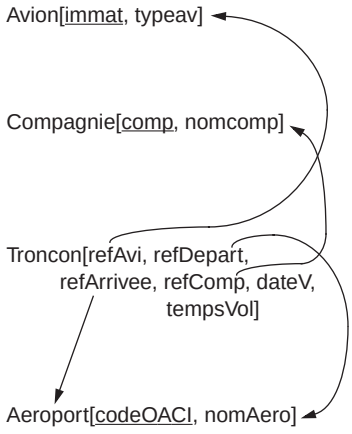


Dérivons l'association `Tronçon` dans un premier temps avec la première solution, puis dans un second temps en privilégiant les accès aux données par la compagnie.

Solution universelle

Le type `troncon_type` contient autant de références qu'il faut relier de tables concernées par l'association et les éventuels attributs de l'association (ici `tempsVol`). La table associée à ce type doit mettre en œuvre autant de clés étrangères que nécessaire (tableau 3-27).

Tableau 3.27 Association *n*-aire par la solution universelle SQL3

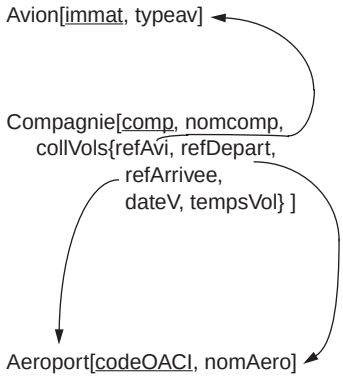
Schéma logique	Script SQL3 (Oracle)
	CREATE TYPE avion_type AS OBJECT (immat VARCHAR(6), typeav VARCHAR(10)) CREATE TYPE aeroport_type AS OBJECT (codeOACI VARCHAR(6), nomAero VARCHAR(30)) CREATE TYPE compagnie_type AS OBJECT (comp VARCHAR(4), nomcomp VARCHAR(30)) CREATE TYPE troncon_type AS OBJECT (refAvi REF avion_type, refDepart REF aeroport_type, refArrivee REF aeroport_type, refComp REF compagnie_type, dateV DATE, tempsVol NUMBER) CREATE TABLE Avion OF avion_type (CONSTRAINT pk_avion PRIMARY KEY(immat)) CREATE TABLE Aeroport OF aeroport_type (CONSTRAINT pk_aeroport PRIMARY KEY(codeOACI)) CREATE TABLE Compagnie OF compagnie_type (CONSTRAINT pk_compagnie PRIMARY KEY(comp)) CREATE TABLE Troncon OF troncon_type (CONSTRAINT nn_refAvi CHECK (refAvi IS NOT NULL), CONSTRAINT fk_refAvi **refAvi REFERENCES** Avion, CONSTRAINT nn_refDepart CHECK(refDepart IS NOT NULL), CONSTRAINT fk_refDep refDepart **REFERENCES** Aeroport, CONSTRAINT nn_refArrivee CHECK (refArrivee IS NOT NULL), CONSTRAINT fk_refArr refArrivee **REFERENCES** Aeroport, CONSTRAINT nn_refComp CHECK (refComp IS NOT NULL), CONSTRAINT fk_refCom refComp **REFERENCES** Compagnie, CONSTRAINT nn_dateV CHECK (dateV IS NOT NULL))

Avec une collection

La collection `collVols` contient les références vers les classes concernées par l'association ainsi que les attributs `tempsVol` et `dateV`. Aucune contrainte de clé étrangère ne peut être programmée pour l'heure avec Oracle.

Tableau 3.28 Association *n*-aire par une collection SQL3

Schéma logique	Script SQL3 (Oracle)

	`CREATE TYPE avion_type AS OBJECT`
	`(immat VARCHAR(6), typav VARCHAR(10))`
	
	`CREATE TYPE aeroport_type AS OBJECT`
	`(codeOACI VARCHAR(6), nomAero VARCHAR(30))`
	
	`CREATE TYPE elt_collVols AS OBJECT`
	`(refAvi REF avion_type,`
	`refDepart REF aeroport_type,`
	`refArrivee REF aeroport_type,`
	`dateV DATE, tempsVol NUMBER)`
	
	`CREATE TYPE collVols_type AS TABLE OF`
	`elt_collVols`
	
	`CREATE TYPE compagnie_type AS OBJECT`
	`(comp VARCHAR(4), nomcomp VARCHAR(30),`
	`collVols collVols_type)`
	
	`CREATE TABLE Avion OF avion_type`
	`(CONSTRAINT pk_avion PRIMARY KEY(immat))`
	
	`CREATE TABLE Aeroport OF aeroport_type`
	`(CONSTRAINT pk_aeroport PRIMARY KEY(codeOACI))`
	
	`CREATE TABLE Compagnie OF compagnie_type`
	`(CONSTRAINT pk_compagnie PRIMARY KEY(comp))`
	`NESTED TABLE collVols STORE AS tempVols`

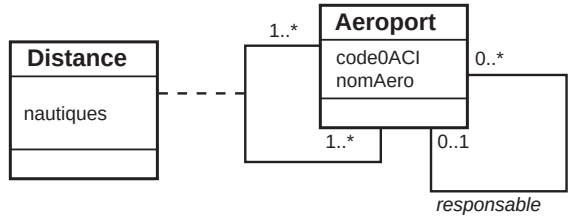
Associations réflexives

Pour traduire une association réflexive, il suffit d'adopter une des solutions précédemment recensées selon la nature de l'association (*un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs* ou *n*-aire). La solution universelle peut convenir dans tous les cas.

Dans l'exemple 3-30, l'association réflexive *plusieurs-à-plusieurs* modélise la distance séparant deux aéroports. Par ailleurs, l'association réflexive *un-à-plusieurs* modélise le fait qu'un aéroport peut avoir sous sa responsabilité plusieurs autres aéroports.

Notation UML

Figure 3-30 *Associations réflexives UML*



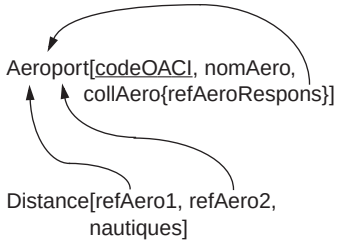
Choisissons de traduire l'association réflexive *plusieurs-à-plusieurs* par la solution universelle et l'association réflexive *un-à-plusieurs* par une collection de références (première solution d'implantation d'une association *un-à-plusieurs*).

Script SQL3

L'association `Distance` met en œuvre des références et une table précisant les clés étrangères. L'association `responsable` nécessite une collection (`collAero`) de références (`refAero-Respons`).

Tableau 3.29 Associations réflexives SQL3

Schéma logique	Script SQL3 (Oracle)

	`CREATE TYPE aeroport_type`
	
	`CREATE TYPE elt_collAero AS OBJECT`
	`(refAeroRespons REF aeroport_type)`
	
	`CREATE TYPE collAero_type AS TABLE OF elt_collAero`
	
	`CREATE TYPE aeroport_type AS OBJECT`
	`(codeOACI VARCHAR(6), nomAero VARCHAR(30),`
	`collAero collAero_type)`

	`CREATE TYPE distance_type AS OBJECT`
	`(refAero1 REF aeroport_type,`
	`refAero2 REF aeroport_type, nautique NUMBER)`
	
	`CREATE TABLE Aeroport OF aeroport_type`
	`(CONSTRAINT pk_aeroport PRIMARY KEY(codeOACI))`
	`NESTED TABLE collAero STORE AS tempAeros`

Tableau 3.29 Associations réflexives SQL3 (suite)

Schéma logique	Script SQL3 (Oracle)

	`CREATE TABLE Distance OF distance_type`
	`(CONSTRAINT nn_refAero1 CHECK`
	`(refAero1 IS NOT NULL),`
	`CONSTRAINT fk_Aero1`
	`refAero1 REFERENCES Aeroport,`
	`CONSTRAINT nn_refAero2 CHECK`
	`(refAero2 IS NOT NULL),`
	`CONSTRAINT fk_Aero2`
	`refAero2 REFERENCES Aeroport)`

Notez qu'il est nécessaire d'utiliser un artifice pour définir une collection de références vers le type contenant la collection. Avec Oracle, la solution consiste à déclarer un type incomplet (première instruction du script). Le type est ensuite redéfini plus loin. Pour régénérer ce schéma, il faut utiliser la directive `DROP TYPE… FORCE` afin de supprimer un type faisant référence à lui-même. Le script de destruction de la base est le suivant.

```
DROP TABLE Distance ;
DROP TABLE Aeroport ;
DROP TYPE distance_type ;
DROP TYPE aeroport_type
DROP TYPE collAero_type;
DROP TYPE aeroport_type;
DROP TYPE elt_collAero;
```

Classes-associations UML

Cette section décrit la traduction SQL3 des associations connectées à des classes-associations UML. Pour chaque association, il faut utiliser une des solutions précédemment étudiées en fonction de la nature de l'association (*plusieurs-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs*).

Considérons pour chacun des cas un exemple. Nous utiliserons arbitrairement une des solutions d'implantation SQL3, que nous avons énoncées précédemment, pour chaque association. D'autres schémas sont possibles en employant d'autres solutions SQL3 de traduction pour chaque association.

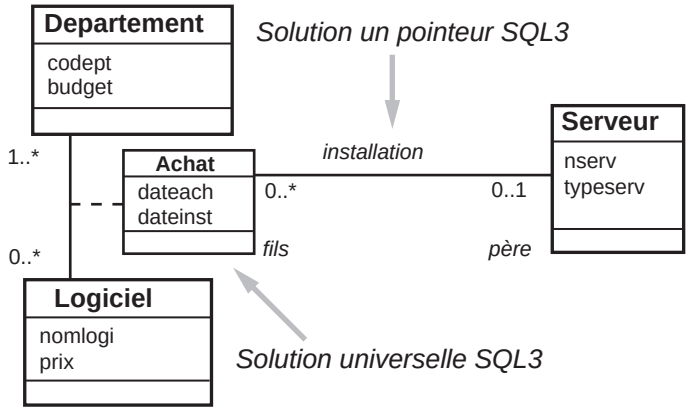
Classe-association un-à-plusieurs

Considérons des serveurs qui hébergent des logiciels achetés par des départements. Un logiciel acheté ne s'installe que sur un seul serveur.

Diagramme UML

La classe-association `Achat` est reliée à la classe `Serveur` par une association *un-à-plusieurs*.

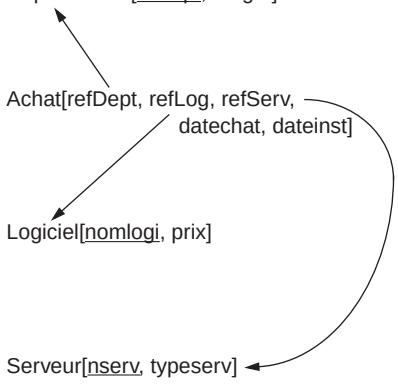
Figure 3-31 *Classe-association un-à-plusieurs*



Script SQL3

Supposons que nous ne privilégions aucun accès aux données, la solution universelle traduit la classe-association `Achat` avec deux références `refLog` et `refDept` et les attributs de l'association. Traduisons à l'aide de la deuxième solution l'association *un-à-plusieurs* `Installation`. Il en résulte une référence (`refServ`) dans le type *fils* (`Achat`) vers le type *père* (`Serveur`). La table associée inclut une clé étrangère sur la référence.

Tableau 3.30 Classe-association un-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)
	CREATE TYPE logiciel_type AS OBJECT (nomlogi CHAR(20), prix NUMBER) CREATE TYPE departement_type AS OBJECT (codept CHAR(6), budget NUMBER) CREATE TYPE serveur_type AS OBJECT (nserv NUMBER, typeserv CHAR(20)) CREATE TYPE achat_type AS OBJECT (refDept REF departement_type, refLog REF logiciel_type, **refServ REF serveur_type,** dateach DATE, dateinst DATE) CREATE TABLE Departement OF departement_type (CONSTRAINT pk_dept PRIMARY KEY(codept)) CREATE TABLE Serveur OF serveur_type (CONSTRAINT pk_serveur PRIMARY KEY(nserv)) CREATE TABLE Logiciel OF logiciel_type (CONSTRAINT pk_log PRIMARY KEY(nomlogi)) CREATE TABLE Achat OF achat_type (CONSTRAINT nn_refDept CHECK (refDept IS NOT NULL), CONSTRAINT fk_refDept refDept **REFERENCES** Departement CONSTRAINT nn_refLog CHECK (refLog IS NOT NULL), CONSTRAINT fk_refLog refLog **REFERENCES** Logiciel, CONSTRAINT nn_refServ CHECK (refServ IS NOT NULL), CONSTRAINT fk_refServ refServ **REFERENCES** Serveur)

Classe-association plusieurs-à-un

Supposons qu'un serveur n'héberge qu'un seul logiciel et qu'un logiciel acheté par un département puisse être installé sur différents serveurs.

Diagramme UML

La classe-association `Achat` est liée à la classe `Serveur` par une association *plusieurs-à-un*.

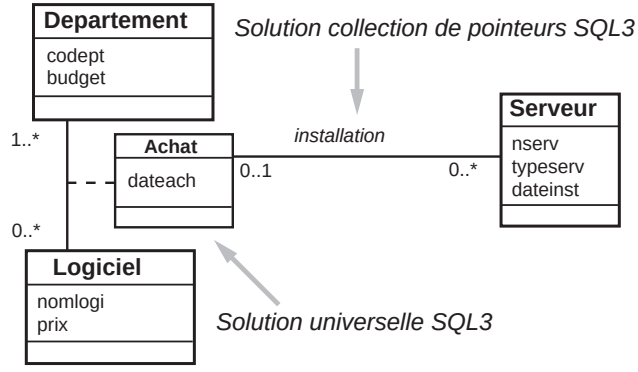


Figure 3-32 *Classe-association plusieurs-à-un*

Script SQL3

Utilisons la solution universelle pour traduire l'association `Achat` (en gras dans le script). Privilégions l'accès aux données par les logiciels pour l'association d'installation par une collection (`collServ`) de références (`refServ`). Cette collection est située dans le type *père* (`Achat`) et permet de référencer le type *fils* (`Serveur`).

Tableau 3.31 Classe-association plusieurs-à-un SQL3

Schéma logique	Script SQL3 (Oracle)
	`CREATE TYPE logiciel_type AS OBJECT (nomlogi CHAR(20), prix NUMBER)`
	`CREATE TYPE departement_type AS OBJECT (codept CHAR(6), budget NUMBER)`
Departement[codept, budget]	`CREATE TYPE serveur_type AS OBJECT (nserv NUMBER, typeserv CHAR(20), dateinst DATE)`
Achat[refDept, refLog, collServ{refServ}, datechat, dateinst]	`CREATE TYPE elt_coll_serv AS OBJECT (reServ REF serveur_type)`
	`CREATE TYPE coll_serv_type AS TABLE OF elt_coll_serv`
Logiciel[nomlogi, prix]	**`CREATE TYPE achat_type AS OBJECT`** `(refDept` **`REF`** `departement_type, refLog` **`REF`** `logiciel_type, dateach DATE,`
Serveur[nserv, typeserv]	`collServ coll_serv_type)`

Tableau 3.31 Classe-association plusieurs-à-un SQL3 (suite)

Schéma logique	Script SQL3 (Oracle)

	`CREATE TABLE Departement OF departement_type`
	`(CONSTRAINT pk_dept PRIMARY KEY(codept))`
	
	`CREATE TABLE Serveur OF serveur_type`
	`(CONSTRAINT pk_serveur PRIMARY KEY(nserv))`
	
	`CREATE TABLE Logiciel OF logiciel_type`
	`(CONSTRAINT pk_log PRIMARY KEY(nomlogi))`
	
	`CREATE TABLE Achat OF achat_type`
	`(CONSTRAINT nn_refDept CHECK`
	`(refDept IS NOT NULL),`
	`CONSTRAINT fk_refDept`
	`refDept` **`REFERENCES`** `Departement,`
	`CONSTRAINT nn_refLog CHECK`
	`(refLog IS NOT NULL),`
	`CONSTRAINT fk_refLog`
	`refLog` **`REFERENCES`** `Logiciel)`

	`NESTED TABLE collServ STORE AS tabServ`

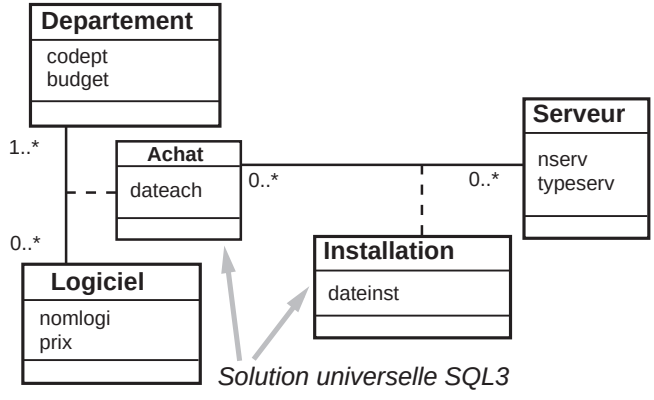
Classe-association plusieurs-à-plusieurs

Supposons qu'un serveur héberge plusieurs logiciels et qu'un logiciel acheté puisse être installé sur différents serveurs.

Diagramme UML

La classe-association `Achat` est liée à la classe `Serveur` par une association *plusieurs-à-plusieurs* qui se modélise par la classe-association `Installation`.

Figure 3-33 *Classe-association plusieurs-à-plusieurs*

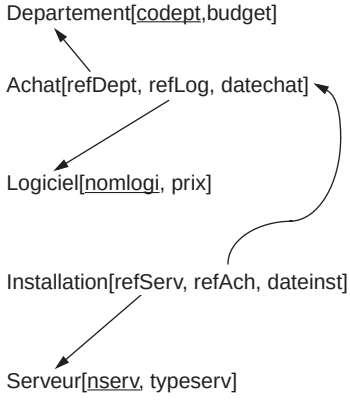


Script SQL3

Supposons que ne nous privilégions aucun accès aux données en utilisant la solution universelle pour traduire les deux classes-associations. Concernant `Achat`, nous obtenons deux références `refLog` et `refDept` avec l'attribut `dateach` (en surligné dans le script SQL3). Nous traduisons `Installation` de la même manière. Il en résulte deux références `refAchat` et `refServ` avec l'attribut `dateinst` (en gras dans le script).

Tableau 3.32 Classe-association plusieurs-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)

	`CREATE TYPE logiciel_type AS OBJECT`
	`(nomlogi CHAR(20), prix NUMBER)`
	
	`CREATE TYPE departement_type AS OBJECT`
	`(codept CHAR(6), budget NUMBER)`
	
	`CREATE TYPE serveur_type AS OBJECT`
	`(nserv NUMBER, typeserv CHAR(20))`
	
	`CREATE TYPE achat_type AS OBJECT`
	`(refDept REF departement_type,`
	`refLog REF logiciel_type, dateach DATE)`
	
	`CREATE TYPE installation_type AS OBJECT`
	`(refServ` **`REF`** `serveur_type,`
	`refAch` **`REF`** `achat_type, dateinst DATE)`
	
	`CREATE TABLE Departement OF departement_type`
	`(CONSTRAINT pk_dept PRIMARY KEY(codept))`
	
	`CREATE TABLE Serveur OF serveur_type`
	`(CONSTRAINT pk_serveur PRIMARY KEY(nserv))`
	
	`CREATE TABLE Logiciel OF logiciel_type`
	`(CONSTRAINT pk_log PRIMARY KEY(nomlogi))`
	
	`CREATE TABLE Achat OF achat_type`
	`(CONSTRAINT nn_refDept CHECK`
	`(refDept IS NOT NULL),`
	`CONSTRAINT fk_refDep`
	`refDep REFERENCES Departement`
	`CONSTRAINT nn_refLog CHECK`
	`(refLog IS NOT NULL),`
	`CONSTRAINT fk_refLog`
	`refLog REFERENCES Logiciel)`
	
	`CREATE TABLE Installation OF installation_type`
	`(CONSTRAINT nn_refServ CHECK`
	`(refServ IS NOT NULL),`
	`CONSTRAINT fk_refServ`
	`refServ` **`REFERENCES`** `Serveur,`
	`CONSTRAINT nn_refAch CHECK`
	`(refAch IS NOT NULL),`
	`CONSTRAINT fk_refAch`
	`refAch` **`REFERENCES`** `Achat)`

Transformation des associations d'héritage

SQL3 prend en compte l'héritage de types et l'héritage de tables. Nous ne parlons pas ici de décomposition pour traduire une association d'héritage, car la notion d'héritage est intrinsèque au modèle objet.

Héritage de types

En considérant l'exemple 2-48, le schéma SQL3 est composé des types `PNC_type` et `PNT_type` qui héritent du type `Personnel_type` à l'aide de la directive `UNDER`. Les tables `PNC` et `PNT` permettront de stocker des objets persistants. La table `Personnel` contiendra notamment les personnels n'étant ni PNT ni PNC (c'est un cas possible car l'héritage est sans contrainte).

Tableau 3.33 Héritage de types SQL3

Schéma logique	Script SQL3 (Oracle)
	`CREATE TYPE Personnel_type (numPers NUMBER, nomPers VARCHAR(20))`
PNC[indice, prime] ⇩	`CREATE TYPE PNC_type UNDER Personnel_type (indice NUMBER, prime NUMBER(8,2))`
Personnel[numPers, nomPers] ⇧	`CREATE TYPE PNT_type UNDER Personnel_type (brevet VARCHAR(10), validiteLicence DATE)`
PNT[brevet, validiteLicence]	`CREATE TABLE Personnel OF Personnel_type (CONSTRAINT pk_Personnel PRIMARY KEY(numPers))`
	`CREATE TABLE PNC OF PNC_type`
	`CREATE TABLE PNT OF PNT_type`

L'avantage de l'héritage de types par rapport à l'héritage de tables réside dans le fait qu'un type, une fois déclaré, peut entrer dans la composition d'un autre type ou peut permettre de définir plusieurs tables.

L'inconvénient de l'utilisation des types se situe au niveau de la modification de la structure des tables en exploitation (Oracle, tout en l'autorisant par `ALTER TYPE`, ne maîtrise pas encore très bien l'ajout ou la suppression de colonnes d'un type composant une table objet-relationnelle).

Héritage de tables

Dans le schéma SQL3 suivant, les tables `PNC` et `PNT` héritent de la table objet-relationnelle `Personnel` à l'aide de la directive `UNDER`.

Tableau 3.34 Héritage de tables SQL3

Schéma logique	Script SQL3
PNC[indice, prime] ⇩ Personnel[numPers, nomPers] ⇧ PNT[brevet, validiteLicence]	CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), CONSTRAINT pk_Personnel PRIMARY KEY(numPers)) CREATE TABLE PNC **UNDER** Personnel (indice NUMBER, prime NUMBER(8,2)) CREATE TABLE PNT **UNDER** Personnel (brevet VARCHAR(10), validiteLicence DATE)

Contraintes

Quelle que soit la contrainte d'une association d'héritage (partition, exclusivité et totalité) à programmer, le schéma SQL3 est identique. En revanche, chaque contrainte devra être programmée soit par des méthodes, soit avec les principes étudiés dans ce chapitre pour SQL2 (déclencheurs, procédures et contraintes de vérification).

Héritage multiple

De même que pour l'héritage simple, la traduction de l'héritage multiple sous SQL3 se programme à l'aide de la directive `UNDER`.

Le schéma SQL3 implémentant l'exemple 2-35 est composé de quatre types et quatre tables objet-relationnelles. Le type `Stagiaire_type` hérite simultanément des types `PNT_type` et `PNC_type`.

Tableau 3.35 Héritage multiple de types SQL3

Schéma logique	Script SQL3
PNC[indice, prime] Personnel[numPers, nomPers] PNT[brevet, validiteLicence] Stagiaire[dateStage]	CREATE TYPE Personnel_type (numPers NUMBER, nomPers VARCHAR(20)) CREATE TYPE PNC_type **UNDER** Personnel_type (indice NUMBER, prime NUMBER(8,2)) CREATE TYPE PNT_type **UNDER** Personnel_type (brevet VARCHAR(10), validiteLicence DATE) CREATE TYPE Stagiaire_type **UNDER PNC_type, PNT_type** (dateStage DATE) CREATE TABLE Personnel OF Personnel_type (CONSTRAINT pk_Personnel PRIMARY KEY(numPers)) CREATE TABLE PNC OF PNC_type CREATE TABLE PNT OF PNT_type CREATE TABLE Stagiaire OF Stagiaire_type

Héritage de tables

La clause UNDER concerne ici deux tables.

Tableau 3.36 Héritage multiple de tables SQL3

Schéma logique	Script SQL3

PNC[indice, prime]
Personnel[numPers, nomPers]
PNT[brevet, validiteLicence]
Stagiaire[dateStage]
	`CREATE TABLE Personnel`
	`(numPers NUMBER, nomPers VARCHAR(20),`
	`CONSTRAINT pk_Personnel PRIMARY KEY(numPers))`
	
	`CREATE TABLE PNC UNDER Personnel`
	`(indice NUMBER, prime NUMBER(8,2))`
	
	`CREATE TABLE PNT UNDER Personnel`
	`(brevet VARCHAR(10), validiteLicence DATE)`
	
	`CREATE TABLE Stagiaire_type`
	`UNDER PNC, PNT (dateStage DATE)`

Oracle comme IBM (et bien d'autres langages objet en commençant par Java) n'autorisent pas la programmation de l'héritage multiple.

Exercices

Exercice 3.1 Du logique à SQL2

Ecrire le script SQL2 (création des tables) du schéma relationnel extrait de l'exercice 2.4. Considérez les types de colonnes suivants :

- *a* et *g* `CHAR(12)`
- *b*, *h* et *i* `NUMBER`
- *e*, *f* `CHAR(40)`
- *c*, *d* et *j* `DATE`

Dans quel ordre ces créations doivent-elles être effectuées ?

Exercice 3.2 Du logique à SQL2

Traduisez les modèles logiques de l'exercice 2.9 (affrètement d'avions, Castanet Telecoms et voltige aérienne) en tables SQL2. Considérez les types de données suivants.

1. Affrètement d'avions

Ajouter les contraintes qui assurent que la capacité d'un avion et le nombre de passagers transportés sont compris entre 50 et 500 places.

attribut	type
comp	CHAR(8)
nomComp	CHAR(30)
typeAvion	CHAR(10)
npMax	NUMBER
nomAvion	CHAR(30)
immatriculation	CHAR(6)
capacite	NUMBER
dateVol	DATE
nbPax	NUMBER
cout	NUMBER

2. Castanet Telecoms

Ne traduisez pas la relation `Privileges` en une table relationnelle.

attribut	type
formule	CHAR(8)
prixParMois	NUMBER

attribut *(suite)*	type *(suite)*
heureSupp	NUMBER
numAbonne	CHAR(10)
nomAbonne	CHAR(30)
adresseAbonne	CHAR(40)
numTel	CHAR(14)
consoLocale	NUMBER
consoNationale	NUMBER
consoInternet	NUMBER
consoAutres	NUMBER
debitLigne	NUMBER
consoMobiles	NUMBER
numPortable	CHAR(14)
typeCombine	CHAR(20)
URL	CHAR(200)
adresseFourni	CHAR(40)
responsable	CHAR(30)
premierContact	DATE
finContrat	DATE
telPrefere	CHAR(14)
dureePreferee	NUMBER

3. Voltige aérienne

Ne traduisez pas la relation `Ordre` en une table relationnelle.

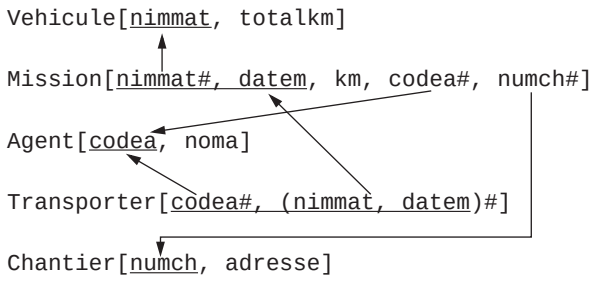
attribut	type
nclubVoltige	NUMBER
aerodromeBase	CHAR(40)
nomClub	CHAR(30)
ncomp	CHAR(8)
nomComp	CHAR(30)
pays	CHAR(20)
immatriculation	CHAR(6)
typeAvion	CHAR(20)
ninventeur	NUMBER
nomInventeur	CHAR(30)
typePilote	CHAR(5)
nfigure	NUMBER
nomFigure	CHAR(40)

attribut *(suite)*	type *(suite)*
noteMax	NUMBER
dateCreation	DATE
pageManuel	NUMBER
numeroChrono	NUMBER
dateVol	DATE
typeProgramme	CHAR(5)
note	NUMBER

Exercice 3.3 Associations d'agrégation

Traduisez avec SQL2 le schéma relationnel des associations d'agrégation de l'exercice 2.10. Considérez les types de données suivants :

attribut	type
nimmat	CHAR(10)
totalkm	NUMBER
datem	DATE
codea	CHAR(5)
noma	CHAR(40)
numch	NUMBER
adresse	CHAR(50)



Chapitre 4

Outils du marché : de la théorie à la pratique

Non mais t'as déjà vu ça ? En pleine paix. Y chante et puis crac, un bourre-pif ! Mais il est complètement fou ce mec ! Mais moi, les dingues j'les soigne. J'm'en vais lui faire une ordonnance, et une sévère ! J'vais lui montrer qui c'est Raoul. Aux quatre coins de Paris qu'on va le retrouver, éparpillé par petits bouts, façon puzzle. Moi quand on m'en fait trop, j'correctionne plus, j'dynamite, j'disperse, j'ventile...

Les Tontons flingueurs, B. Blier
G. Lautner, dialogues M. Audiard, 1963

Ce chapitre valide la démarche théorique de l'ouvrage en la comparant aux principales solutions informatiques du marché qui mettent en œuvre la notation UML et l'interconnexion à une base de données (le plus souvent par un pilote ODBC ou JDBC). Les 14 outils étudiés sont : Enterprise Architect, MagicDraw, MEGA Designer, ModelSphere, MyEclipse, Objecteering, Poseidon for UML, PowerAMC, Rational Rose Data Modeler, Together, Visio, Visual Paradigm, Visual UML et Win'Design. Le lecteur trouvera en annexe les adresses Internet de ces produits.

Sont exclus de ce comparatif, les outils qui ne prennent pas en compte UML pour l'instant, citons DB Designer, Database Design Studio, DeZign, AllFusion ERWin, xCase, CASE Studio et ER/Studio.

La partie consacrée aux bases de données n'est pas prépondérante pour la majorité des outils. D'autres fonctionnalités sont offertes en ce qui concerne la modélisation de processus métier BPM (*Business Process Models*) qui peuvent être importés ou exportés conformément au langage BPEL4WS (*Business Process Execution Language for Web Services*). Bon nombre d'entre eux fournissent un référentiel pour le contrôle des données métiers utiles aux architectes des systèmes d'information qui en seront les principaux utilisateurs. Les plus récents s'inscrivent davantage dans l'architecture MDA (*Model Driven Architecture*) en incluant le standard QVT (*Query View Transformation*) pour la transformation de modèles.

La majorité des outils proposent un processus de conception basé sur les différents niveaux du conceptuel au physique (*forward engineering*), un processus de rétroconception (*reverse engineering*) et un processus d'interéchange (*round-trip engineering*) entre chaque modèle. Les outils permettent également la génération de code en différents langages (SQL, PowerBuider, C++, C#, Java, XML, IDL-CORBA, Visual Basic, IHM ou site Web).

Seuls Together et MyEclipse sont totalement intégrés à Eclipse (Objecteering et MagicDraw proposent toutefois un plug-in).

Chaque logiciel sera évalué selon la qualité d'implémentation de différents diagrammes de classes incluant les critères suivants :

- associations binaires et *n*-aires, classes-associations et agrégations ;
- contraintes (partition, inclusion et relatives à l'héritage) ;
- héritage (décomposition au niveau logique et héritage multiple) ;
- rétroconception d'une base de données.

Pour chaque critère, un tableau illustre la mise en œuvre des outils avec une signalisation à t 248248rois états (satisfaisante : feu vert, moyenne : feu orange, insatisfaisante et absente : feu rouge). Certaines particularités intéressantes seront illustrées par des copies d'écran au fil des exemples. Le classement final, qui prend également en compte la robustesse et l'ergonomie de chaque logiciel, n'engage que moi naturellement.

Associations binaires

Les outils sont évalués ici sur la capacité à représenter, avec la notation UML, des associations binaires, de les transformer au niveau logique puis de générer un script SQL. La prise en compte des identifiants de classes, multiplicités, rôles, etc. est également étudiée.

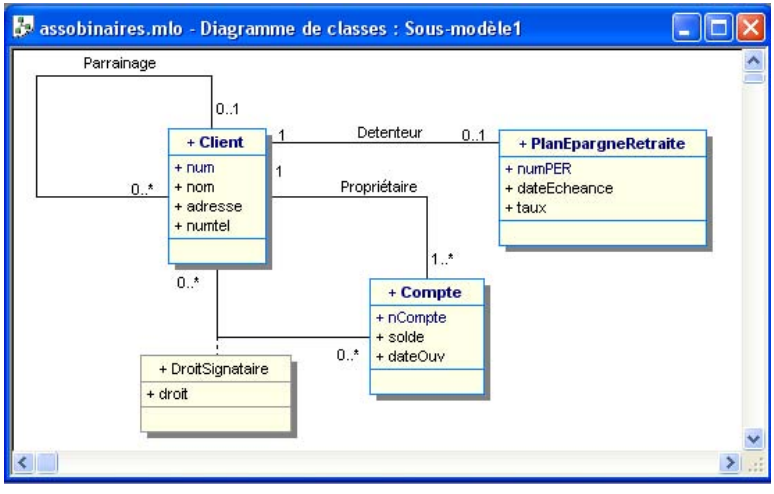


Figure 4-1 *Associations binaires (Win'Design)*

L'exemple 4-1 décrit une modélisation UML de comptes bancaires qui sont la propriété de clients. Un client peut parrainer d'autres clients. Un client n'a le droit de souscrire qu'à un seul plan d'épargne retraite. Enfin, un client peut être désigné comme signataire de comptes ne lui appartenant pas (un seul droit lui est alors affecté).

Niveau conceptuel

Les outils sont évalués sur les moyens mis en œuvre pour représenter une classe (avec ses attributs et son identifiant), une association binaire ou réflexive (nommage, désignation des rôles et des multiplicités).

Les copies d'écran suivantes illustrent l'implémentation des identifiants de classe selon Objecteering, MEGA, PowerAMC et Rational Rose (Win'Design le permet également).

Figure 4-2 *Définition d'un identifiant de classe (Objecteering)*

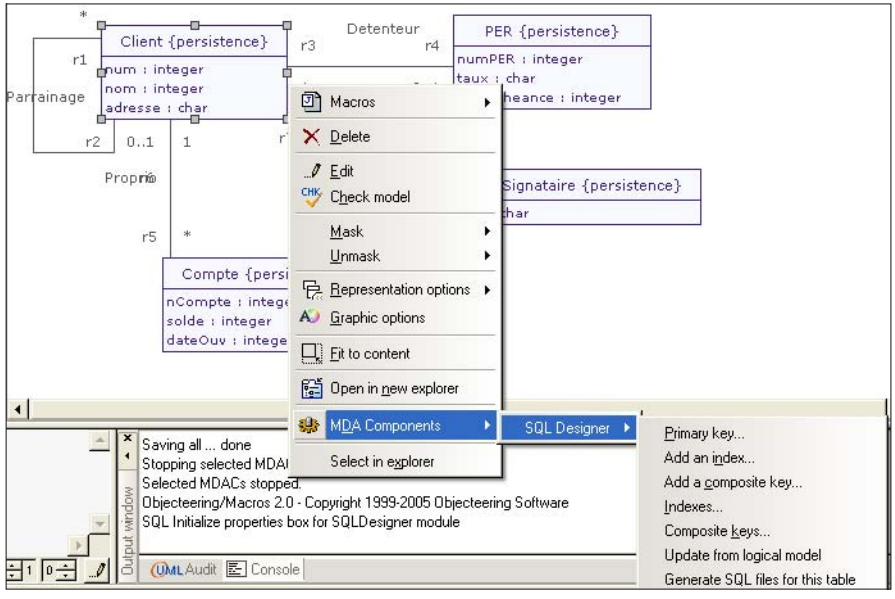


Figure 4-3 *Définition d'un identifiant de classe (MEGA)*