Références et héritage

- Contraintes référentielles pour assurer la consistance d'un schéma de données
- Utilisation (pervertie) de l'héritage objets pour faire du polymorphisme et assurer la cohérence d'un schéma de données
- Exemple, la base d'archives CEMAG (en cours)

Contraintes SQL (rappel)

Contraintes d'intégrité

Conditions imposées sur des attributs d'une table, que le moteur du SGBD doit vérifier lors de toute opération SQL modifiant des *tuples* d'une table, i.e. INSERT, UPDATE, DELETE.

Par exemple, on peut imposer qu'un attribut identificateur ne puisse jamais être non défini :

```
CREATE TABLE xxxxxx (
    ident     int4  NOT NULL,
    ...
);
```

ou encore qu'il ne puisse exister de doublons sur la valeur :

```
CREATE TABLE xxxxxx (
    ident     int4  UNIQUE,
    ...
);
```

NB: UNIQUE implique NOT NULL

Toute opération sur la table qui met en défaut la condition de contrainte sera rejetée, avec message d'erreur.

Références externes

Contraintes référentielles

On parle de référence externe lorsqu'un tuple d'une table contient qui attribut qui référence un autre attribut d'une autre table.
(Implémentation d'une relation d'ordre 1:N de Codd)

Par exemple, une base contient une table d'utilisateurs, identifiés par un code numérique (clé primaire) :

```
CREATE TABLE users_table (
    ident     int4  UNIQUE,
    ...
```

et une autre table d'objets, identifiés eux aussi par un code numérique. Chaque objet de la base doit appartenir à un utilisateur (créateur) :

```
CREATE TABLE cores_table (
    ident     int4  UNIQUE,
    owner     int4  REFERENCES users_table(ident),
    ...
```

Cohérence du schéma

Du fait de cette contrainte, il devient impossible de créer dans la table `cores_table` un tuple dont l'attribut `owner` ne soit pas un identificateur utilisateur valide (existant déjà dans la table), le SGBD rejetera la requête.

Références externes

Stratégies sur destruction

Si l'on détruit un tuple de la table utilisateurs, la cohérence pourrait être cassée. Le SGBD l'interdira a priori.

On pourra spécifier l'interdiction de la destruction (consistance) :

```
CREATE TABLE cores_table (
    ident     int4  UNIQUE,
    owner     int4  REFERENCES users_table(ident) ON DELETE RESTRICT,
    ...
```

ou bien la destruction automatique de TOUS les objets qui référençaient le tuple utilisateur détruit (cohérence) :

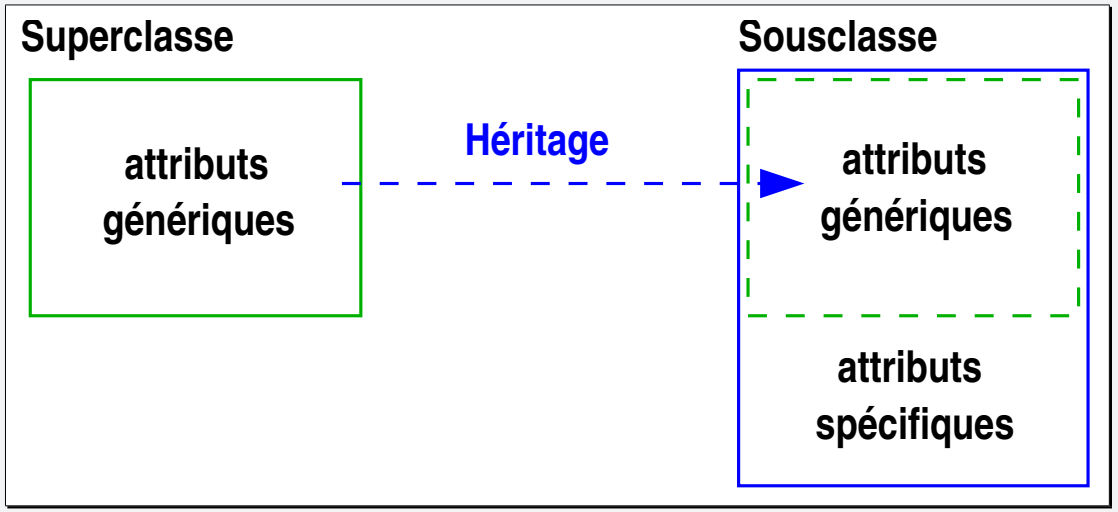
```
CREATE TABLE cores_table (
    ident     int4  UNIQUE,
    owner     int4  REFERENCES users_table(ident) ON DELETE CASCADE,
    ...
```

Choix de stratégie

Il n'y en a pas a priori. C'est la conception de l'application et du schéma qui doit poser la question (et y répondre). Dans tous les cas il sera impossible d'avoir une base comportant des *orphelins*.

- Schéma sécurisé

Héritage objets



L'héritage permet de construire une classe dérivée d'une autre en récupérant tous les attributs (et méthodes logicielles) de la classe ancêtre et en y ajoutant ses propres attributs.

Bases orientées objets

De manière similaire on pourra définir une table de tuples :

```
CREATE TABLE super_table (
    ...       -- attributs generiques
);
```

puis la dériver pour construire une autre table :

```
CREATE TABLE sous_table (
    ...       -- attributs specifiques
) INHERITS ( super_table );
```

Héritage dégénéré

Utilisation perverse

On parle d'héritage dégénéré lorsque la sous table ne comporte aucun attribut supplémentaire par rapport à la super table.

```
CREATE TABLE sous_table (
) INHERITS ( super_table );
```

Les tuples insérés dans la sous table sont :

- identiques à ceux de la super table,
- visibles (SELECT) par des requêtes sur la super table parmi les autres,
- seuls visibles par des requêtes sur la sous table.

Intérêt ?

Permet de structurer un groupe d'objets en sous-groupes sur lesquels on pourra spécifier des contraintes différentes.

➤ L'héritage n'importe pas les contraintes.

Exemple : la base CEMAG

Description

Une base d'archivage de résultats de simulation. Contient des objets scientifiques, fichiers de résultats, de post traitement, images, etc. ainsi que des objets *conteneurs* dont le seul rôle est d'assurer la structuration des données, e.g. projets, sous projets, codes de calcul, versions de codes, tâches, simulations, etc.

Sémantique du schéma

Ces objets conteneurs sont tous identiques mais on souhaite les organiser par groupes pour introduire un certain nombre de conditions qui ont un sens pour l'application :

- Tout projet / sous projet doit être rattaché à un autre projet,
- Tout code numérique doit être rattaché à un projet ou sous projet,
- Toute version de code doit être rattachée à un code et rien d'autre,
- etc.

On peut gérer tout ceci au niveau de l'application, on peut également (en plus) rigidifier le schéma au moment de sa construction en s'aidant du SGBD.

Exemple : la base CEMAG

Chaînage projets (et leur propriétaire)

```
CREATE TABLE cores_table (
    ident   int4  UNIQUE,
    owner   int4  REFERENCES users_table(ident) ON DELETE RESTRICT,
    croot   int4  REFERENCES cores_table(ident) ON DELETE RESTRICT,
    ...
```

L'identifiant `owner` doit référencer un propriétaire valide (référence externe vers une table des utilisateurs).

L'identifiant `croot` référence le core objet père et doit être une valeur valide dans la table cores_table.

Objet initial

Comme il est impossible de créer un tuple dont l'attribut ne soit pas un identifiant valide, l'initialisation de la table vide doit se faire explicitement par création d'un objet racine qui s'auto référence, par exemple :

```
INSERT INTO cores_table(ident, croot, ...)
    VALUES(1, 1, );
```

C'est possible parce que le SGBD vérifie les contraintes avant la validation de l'insertion qui vient d'être effectuée, donc le tuple existe potentiellement au moment du contrôle.

Exemple : la base CEMAG

Objets codes

Les conteneurs de type outils numériques sont créés dans une sous table dédiée, avec les mêmes contraintes :

```
CREATE TABLE tools_table (
    ident   int4  UNIQUE,
    owner   int4  REFERENCES users_table(ident) ON DELETE RESTRICT,
    croot   int4  REFERENCES cores_table(ident) ON DELETE RESTRICT,
) INHERITS ( cores_table );
```

NB: les attributs de même nom et même type ne sont pas dupliqués dans une sous classe mais fusionnés avec ceux de la super classe. On les répète ici car les contraintes ne sont pas héritables.

À quoi ça sert ?

À rien ! Les tuples de cette sous table sont exactement les mêmes que ceux de la super classe, même attributs, mêmes contraintes.

Le seul intérêt est de disposer d'un sous groupe avec un nom de table qui lui est propre.

Exemple : la base CEMAG

Objets versions exécutables de codes

On veut rattacher toutes les versions d'un même code à l'objet code lui-même, pas à un projet ou sous projet.

On référence la sous table précédente :

```
CREATE TABLE execs_table (
    ident   int4  UNIQUE,
    owner   int4  REFERENCES users_table(ident) ON DELETE RESTRICT,
    croot   int4  REFERENCES tools_table(ident) ON DELETE RESTRICT,
) INHERITS ( cores_table );
```

NB: cette table est également dérivée de la table ancêtre. La seule différence porte sur l'attribut `croot` qui doit correspondre à un identifiant de la table tools_table et non plus cores_table.

NB: tous ces objets restent des `cores` objets (et accessibles en SELECT depuis la super table). L'héritage dégénéré permet simplement d'implanter une forme de polymorphisme et de structurer le schéma.

Exemple : la base CEMAG

Objets tâches

Ces objets comportent deux références, un projet de rattachement (contexte scientifique) et une version de code numérique (contexte technique).

À cause du pointeur supplémentaire on aura là un véritable héritage avec attributs spécifiques :

```
CREATE TABLE tasks_table (
    ident   int4  UNIQUE,
    owner   int4  REFERENCES users_table(ident) ON DELETE RESTRICT,
    croot   int4  REFERENCES cores_table(ident) ON DELETE RESTRICT,
    --
    cexec   int4  REFERENCES execs_table(ident) ON DELETE RESTRICT,
) INHERITS ( cores_table );
```

Conclusions

- **Un outil gratuit**

  Les SGBD font ça très bien et le coût est nul (en terme de code) au niveau du développement de l'application, seule la construction initiale du schéma doit le prévoir.

- **Un schéma consistant**

  Avec l'héritage dégénéré on structure un ensemble de données a priori identiques en introduisant des liaisons sémantiques entre elles.

- **Un schéma cohérent et fiable**

  Le système de contraintes référentielles implique que tout objet existant dans la base est cohérent, pas d'orphelins, pas de références invalides.

- **Un niveau de sécurité supplémentaire**

  Les stratégies de cohérence en cas de destruction protègent contre les erreurs de programmation de l'application, ou les interventions manuelles intempestives sur la base. Toute requête ne respectant pas les contraintes est rejetée.

- **Une économie (parfois) au niveau programmation de l'application**

  Lorsqu'on extrait des objets de la base (SELECT), on peut éliminer des tests de validité sur les références puisque la simple existence d'un objet contraint assure que toutes ses références sont valides.

- **Polymorphisme**

  Toutes les requêtes génériques, applicables sur la super classe, permettent de manipuler l'ensemble des tuples quelque soit leur nature spécialisée, d'où une grosse économie d'opérations OR.

Compatibilités

Contraintes référentielles

Au standard SQL 99, implémentées dans DB2, Oracle, PostgreSQL, MySQL.

Bases orientées objets, héritage simple

Au standard SQL 99, implémenté dans DB2, Oracle. Implémenté dans PostgreSQL avec des écarts syntaxiques.

Bases orientées objets, héritage multiple

Pas au standard SQL 99, implémenté dans DB2, Oracle, PostgreSQL avec des syntaxes spécifiques.