Dénormalisation, Optimisation et modélisation

Anne-Cécile Caron, Anne Étien, Mikaël Monet, Sylvain Salvati

Outline

La normalisation en résumé

- Un des concepts de base de la modélisation des bases de données relationnelles.
- Permet de minimiser la redondance. Les données redondantes entraînent des anomalies sémantiques qui rendent la maintenance plus difficile.
  - Tout attribut est monovalué et dépend fonctionnellement de la clé. A partir d'une valeur de la clé, on est capable d'identifier de façon unique un tuple dans la table.
    Contre-exemples :

numF	**numLF**
132	45, 73, 26

ou

numF	**numLF**
132	45
132	73
132	26

La normalisation en résumé

- Un des concepts de base de la modélisation des bases de données relationnelles.
- Permet de minimiser la redondance. Les données redondantes entraînent des anomalies sémantiques qui rendent la maintenance plus difficile.
  - Tout attribut est monovalué et dépend fonctionnellement de la clé. A partir d'une valeur de la clé, on est capable d'identifier de façon unique un tuple dans la table.
    Contre-exemples :

numF	**numLF**
132	45, 73, 26

ou

numF	**numLF**
132	45
132	73
132	26

  - Tout attribut dépend de l'ensemble de la clé et non d'une sous partie uniquement. (Bien entendu ceci n'a de sens que si la clé est composée).
    Contre-ex : LigneFacture (numF, numLF, qte, numProd, numClient)

La normalisation en résumé

- Un des concepts de base de la modélisation des bases de données relationnelles.
- Permet de minimiser la redondance. Les données redondantes entraînent des anomalies sémantiques qui rendent la maintenance plus difficile.
  - Tout attribut est monovalué et dépend fonctionnellement de la clé. A partir d'une valeur de la clé, on est capable d'identifier de façon unique un tuple dans la table.
    Contre-exemples :

numF	**numLF**
132	45, 73, 26

ou

numF	**numLF**
132	45
132	73
132	26

  - Tout attribut dépend de l'ensemble de la clé et non d'une sous partie uniquement. (Bien entendu ceci n'a de sens que si la clé est composée).
    Contre-ex : LigneFacture (numF, numLF, qte, numProd, numClient)
  - Aucun attribut ne dépend fonctionnellement d'un attribut non clé.
    Contre-ex : LigneFacture (numF, numLF, qte, numProd, nomProd)

Anomalies dues au non respect de ces règles

Prenons une relation FACTURE qui n'est pas normalisée :

FACTURE(numf, numprod, nomproduit, quantite, numclient, nomclient, prenomclient, email, datecmde)

Cette table contient plusieurs lignes pour chaque facture payée par un client ; autant de lignes qu'il y a de produits différents achetés.

numclient	$\rightarrow$	nomclient, prenomclient, email
numprod	$\rightarrow$	nomproduit, prixunitaire
numf	$\rightarrow$	numclient, datef
numf, numprod	$\rightarrow$	quantite

Anomalies dues au non respect de ces règles

Prenons une relation `FACTURE` qui n'est pas normalisée :

`FACTURE(numf, numprod, nomproduit, quantite, numclient, nomclient, prenomclient, email, datecmde)`

Cette table contient plusieurs lignes pour chaque facture payée par un client ; autant de lignes qu'il y a de produits différents achetés.

$$
\begin{aligned}
\texttt{numclient} &\rightarrow \texttt{nomclient, prenomclient, email} \\
\texttt{numprod} &\rightarrow \texttt{nomproduit, prixunitaire} \\
\texttt{numf} &\rightarrow \texttt{numclient, datef} \\
\texttt{numf, numprod} &\rightarrow \texttt{quantite}
\end{aligned}
$$

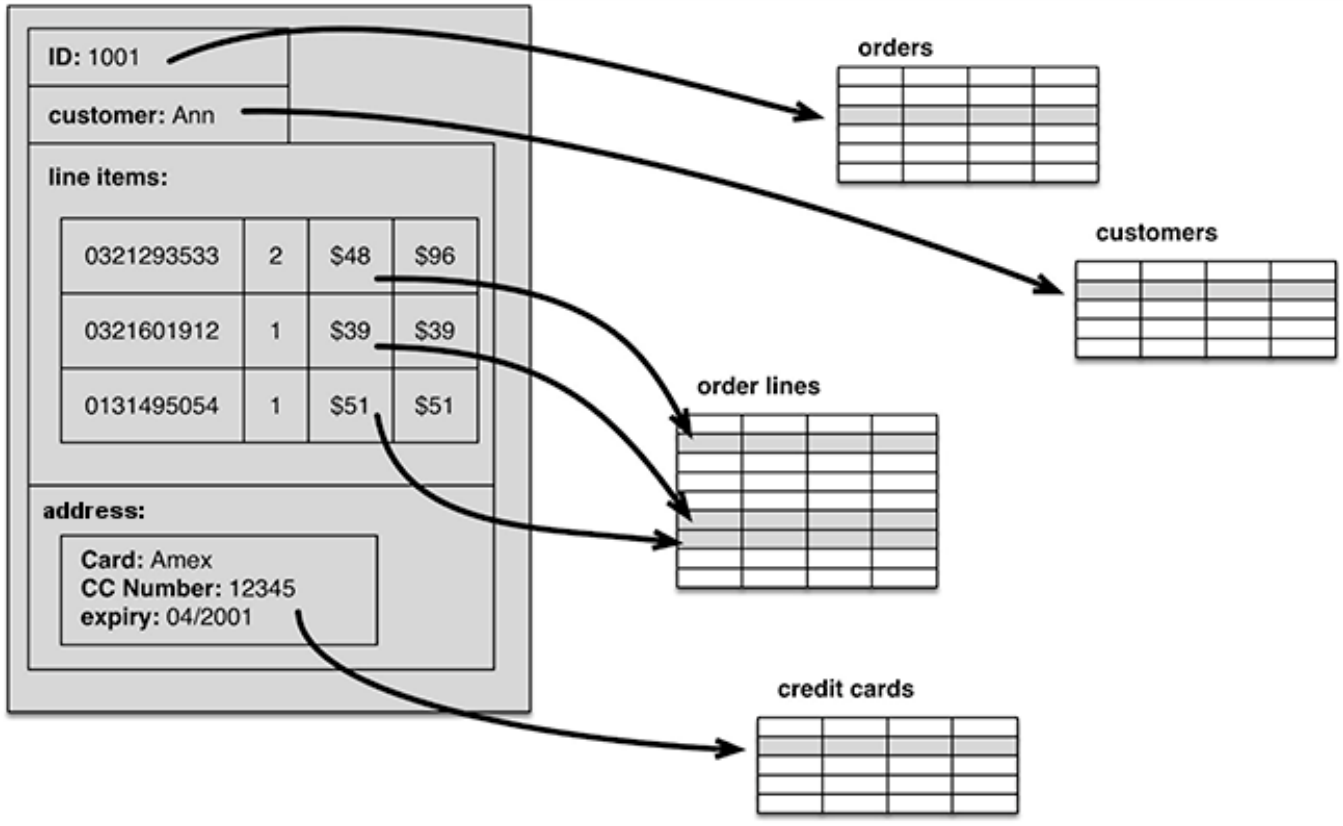
- La date de la facture ne dépend pas du produit acheté. Si l'on souhaite modifier la date d'une facture, il faut le faire pour toutes les lignes de cette facture. C'est ce qu'on appelle une anomalie de mise à jour.
- Le client ne dépend pas de la facture. Si on supprime toutes les factures d'un client, celui-ci disparaît du système d'information. C'est une anomalie de suppression.
- De même il n'est pas possible de faire apparaître des clients qui n'auraient pas encore payé de facture. C'est une anomalie d'insertion.

La solution : Normaliser

```
CLIENT(numclient, nomclient, prenomclient, email)
PRODUIT(numprod, nomproduit, prixunitaire)
FACTURE(numf, numclient, datef)
LIGNE_FACTURE(numf, numprod, quantite)
```

Cette normalisation permet de corriger les anomalies précédentes, mais oblige le moteur SQL à calculer des jointures.
C'est prix à payer : les mises à jour sont propres mais les lectures sont plus complexes.

Illustration : *Impedance Mismatch*



Les données sont sauvegardées sous forme de tuples alors que la structure des données dans l'application et en mémoire est plus complexe (par exemple des objets). Les ORM comme Hibernate font la traduction entre ces deux représentations MAIS besoin de décomposer / reconstruire un concept complexe.

Ici, la lecture d'un objet en apparence unitaire implique l'accès à plusieurs relations et le calcul de jointures.

Outline

La dénormalisation

Pour optimiser certaines requêtes, éviter des jointures ou des calculs répétitifs, ou gagner de la place, on peut être amené à dénormaliser.

- Attribut calculé : c'est un attribut dont la valeur est calculée à partir d'autres données dans la base. On introduit donc de la redondance. C'est un cas d'utilisation très fréquent de la dénormalisation.
- Attribut composé : les développeurs utilisent une chaîne de caractères, comme des valeurs séparées par des virgules, pour stocker un attribut multi-valué.
- Attribut répété sur plusieurs colonnes : plusieurs colonnes sont créées pour un attribut, par exemple pour les prénoms d'une personne. Certains peuvent n'avoir qu'un seul prénom d'autres 2, 4 ou même plus.
- Attribut transitif : ajouter dans une table un attribut d'une autre table pour éviter une jointure.
- God table : tous les attributs sont dans la même table ! C'est un peu comme utiliser un tableur pour stocker ses données.
- Clone tables : les développeurs clonent une table en la séparant horizontalement (en fonction des données) pour améliorer le passage à l'échelle sans utiliser les capacités du SGBD.

Attribut calculé

Attribut dont la valeur est calculée à partir d'autres données dans la base

- Dans l'exemple des factures, il pourrait être judicieux d'ajouter dans la table `FACTURE` une colonne `montant_total`, pour éviter de recalculer le montant à partir de `LIGNE_FACTURE.quantite` et `PRODUIT.prix_unitaire`.
- Avantage : on gagne du temps quand on recherche le montant d'une facture

```
select numf, datef, sum(quantite*prix_unitaire) as montant_total
from FACTURE
  join LIGNE_FACTURE using (numf)
  join PRODUIT using (numprod)
group by numf, datef ;
```

devient :

```
select numf, datef, montant_total
from FACTURE ;
```

- Inconvénient : il faudra modifier le montant si l'on modifie une ligne de la facture ou si l'on enlève/ajoute une ligne à la facture.

Attribut composé

attribut multi-valué, ou attribut dont la valeur est structurée.

- La norme SQL-99 introduit des types objets, et donc la possibilité de typer une colonne avec un objet ou une collection (vecteurs, tables imbriquées). Cette possibilité est peu utilisée dans les entreprises.
- Une alternative est d'utiliser une chaîne de caractères pour coder un objet complexe sous une forme sérialisée. Un exemple simple est d'utiliser des valeurs séparées par des virgules pour coder une liste, à la manière des fichiers CSV.
- L'arrivée de XML dans les années 2000 puis de JSON dans les années 2010 ont rendu plus fréquents les attributs composés. Au niveau de l'implémentation, les SGBD relationnels proposent des types pour répondre à la demande, mais la solution `VARCHAR` est toujours d'actualité.
- JSON et Postgres : Postgres propose 2 types, `json` et `jsonb`
  - Le type `json` stocke une copie exacte du texte, qu'il faut donc analyser à chaque traitement ;
  - Le type `jsonb` stocke une version binaire, ce qui permet un traitement plus efficace au prix d'une insertion plus lente.

Illustration : donnée JSON

```
CREATE TABLE netflix (
  show_id numeric primary key,
  tvshow_or_movie char(2), -- 'Mo' or 'TV'
  title varchar(100) not null,
  listed_in json not null default '[]' );

insert into netflix values( 80057969, 'Mo', 'Love',
'["Cult Movies","Dramas","Independent Movies"]' );

select title, json_array_length(listed_in) from netflix;

title       | json_array_length
------------+------------------
Love        |                 3
Océans      |                 2
...

select title, listed_in from netflix
where listed_in::jsonb @> '"Dramas"'::jsonb;

title            |                    listed_in
-----------------+-----------------------------------------------
Love             | ["Cult Movies","Dramas","Independent Movies"]
Black Snake Moan | ["Dramas","Independent Movies"]
...
```

Attribut transitif

Cette solution consiste en la création d'un lien de transitivité (ajout d'une clé étrangère) pour économiser une jointure avec une table intermédiaire.

`Enseignant(idens, nom, prenom, discipline, idbureau)`
`Bureau(idbureau, etage, surface, nbcoloc, idbat)`
`Batiment(idbat, nombat, campus, ville)`

En dénormalisant, on recopie l'identifiant du bâtiment dans la table `Enseignant` :

`Enseignant(idens, nom, prenom, discipline, idbureau, idbat)`

- Avantage : Si on veut la liste des enseignants avec le campus où les trouver, on n'a pas besoin de faire la jointure avec `Bureau`.
- Inconvénient : Si un enseignant déménage en changeant de bâtiment, on doit modifier à la fois la colonne `idbureau` et la colonne `idbat` de la table `Enseignant`.

God Table

C'est la table facture avec tous les attributs, présentée dans la diapo 4.
Évidemment cette solution extrême est rarement utilisée.

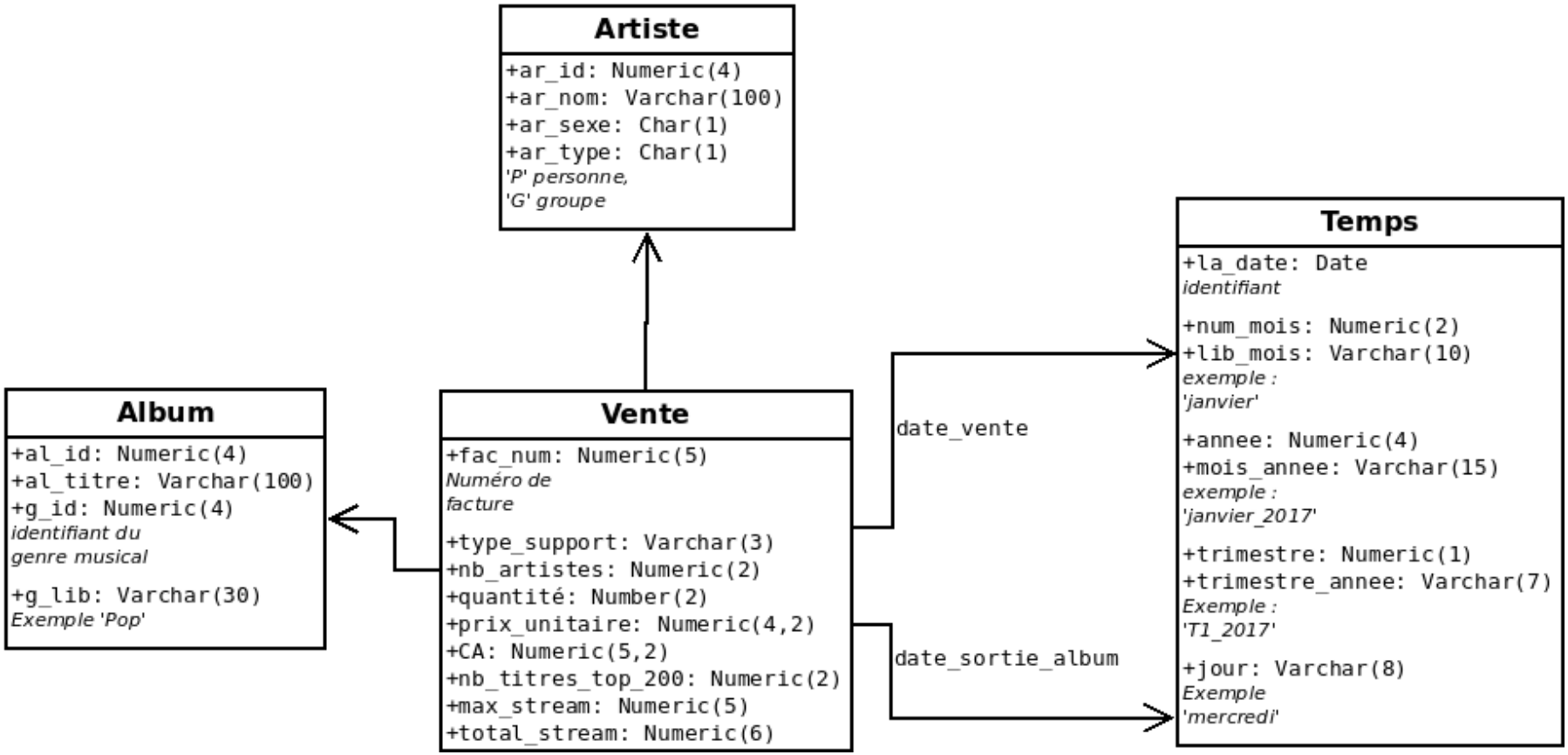
God Table

C'est la table facture avec tous les attributs, présentée dans la diapo 4.
Évidemment cette solution extrême est rarement utilisée.

Néanmoins, dans les entrepôts de données ou les tuples sont beaucoup lus et jamais modifiés (ils sont recopiés d'autres sources de données), les tables dites "de dimensions" sont le résultat de jointures ; la table centrale dite "de faits" peut aussi être construite par jointure(s).
Cette modélisation particulière est appelée modélisation en étoile.

Illustration : schéma en étoile

Ventes de disques (cf TP indexation), avec des artistes et des genres musicaux, ainsi que des données de Spotify.



- Une seule table pour un album et son genre, pourtant `Album` $\rightarrow$ `Genre`
- Un tuple de `Vente` représente la vente d'un disque, donc une ligne de facture. Et on a aussi des informations sur la facture, avec `Ligne_Facture` $\rightarrow$ `Facture`.
- Beaucoup d'attributs calculés, par exemple sur la table `Temps`

Conclusion

- La normalisation permet d'éviter des anomalies lorsqu'on modifie les données ;
- Pour des raisons de performance ou pour gagner de la place, on peut être amené à dénormaliser ;
- Souvent cette dénormalisation introduit de la redondance qu'il faut gérer pour maintenir la cohérence de la base (cf attributs calculés, ex1 TP) ;
- Les attributs multi-valués permettent une économie de place, moins de jointure, mais leur manipulation alourdit les requêtes avec l'usages de fonctions spécifiques (cf exemple avec le tableau JSON) ;
- La dénormalisation est très fréquente dans les bases qui sont beaucoup lues, mais pas/peu mises à jour (cas d'utilisation en informatique décisionnelle, cf exemple modélisation en étoile).
- Il faut toujours peser le pour et le contre : quelles sont les requêtes les plus fréquentes à optimiser, quels sont les mesures à prendre pour maintenir la cohérence, etc