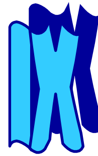

LA GESTION DES VUES

1. INTRODUCTION

Pourquoi des **vues** ? Elles permettent de réaliser, dans le monde des SGBD relationnels, le niveau externe des SGBD selon l'architecture ANSI/SPARC. Rappelons que le niveau externe propose à l'utilisateur une perception de la base plus proche de ses besoins, en termes de structures et de formats de données. De manière générale, les vues garantissent une meilleure indépendance logique des programmes par rapport aux données. En effet, le programme restera invariant aux modifications de schéma s'il accède à la base via une vue qui l'isole de celle-ci. Lors des modifications du schéma de la base, l'administrateur modifiera seulement la définition des vues, et les programmes d'application pourront continuer à travailler sans modification. Les vues ont aussi un rôle de sécurité : l'utilisateur ne peut accéder qu'aux données des vues auxquelles il a droit d'accès ; ainsi, les données en dehors de la vue sont protégées. De manière détournée, les vues permettent aussi certains contrôles d'intégrité lorsqu'elles sont utilisées pour les mises à jour : on dit alors qu'on effectue une mise à jour au travers d'une vue. De telles mises à jour sont très contrôlées, comme nous le verrons ci-dessous.

Ces dernières années, les vues ont trouvé de nouvelles applications. Elles permettent de définir des tables virtuelles correspondant aux besoins des programmes d'application en termes de données. Dans le monde du client-serveur, les vues constituent un élément essentiel d'optimisation de performance. Par exemple, la définition d'une vue jointure de deux tables, ou résultant d'un calcul d'agrégat, évite au client les risques de lire les tuples des tables et de faire le calcul de jointure ou d'agrégat sur le client : seules les données résultant du calcul de la vue sont exportées sur le site client. On laisse ainsi faire au serveur les calculs de jointure, agrégat, etc., qu'il sait en principe bien faire. Dans le monde des entrepôts de données et du décisionnel,

les vues peuvent être concrétisées. Elles permettent de réaliser par avance des cumuls ou synthèses plus sophistiqués de quantités extraites de la base selon plusieurs dimensions. Des mécanismes de mises à jour des vues concrètes lors des mises à jour des relations de base sont alors développés afin d'éviter le recalcul des cumuls.

Les vues ont donc une importance croissante dans les bases de données. En pratique, ce sont des relations virtuelles définies par des questions. Cette définition est stockée dans la métabase. Les vues sont interrogées comme des relations normales. Idéalement, elles devraient pouvoir être mises à jour comme des relations normales. Les vues concrètes sont calculées sur disques lors de leurs créations. Des mécanismes de mise à jour différentiels permettent le report efficace des mises à jour des relations de base. Toutes ces techniques sont aujourd'hui bien maîtrisées ; nous allons les présenter ci-dessous.

Afin d'illustrer les mécanismes de vues, nous utiliserons tout d'abord la base de données viticole classique composée des relations :

BUVEURS (NB, NOM, PRENOM, ADRESSE, TYPE)
VINS (NV, CRU, REGION, MILLESIME, DEGRE)
ABUS (NV, NB, DATE, QUANTITE).

décrivant respectivement les buveurs, les vins, et les consommations de vins quotidiennes. Comme habituellement, les clés des relations sont soulignées. Des vues typiques sont les vins de Bordeaux, les gros buveurs, les quantités de vins bues par crus, etc.

Pour des exemples plus avancées, intégrant notamment des agrégats complexes, nous utiliserons la base MAGASINS suivante :

VENTES(NUMV, NUMPRO, NUMFOU, DATE, QUANTITE, PRIX)
PRODUITS (NUMPRO, NOM, MARQUE, TYPE,PRIX)
FOURNISSEURS (NUMFOU, NOM, VILLE, REGION, TELEPHONE)

Des vues typiques sont les quantités de produits vendus par fournisseurs et par mois, les évolutions des quantités commandées par région, etc. La relation Ventes décrit les ventes journalières de produits. NUMPRO est la clé de produits et NUMFOU celle de fournisseurs. Dans une telle base de données, les faits de base sont les ventes, alors que produits et fournisseurs constituent des dimensions permettant d'explorer les ventes selon une ville ou une région par exemple.

Ce chapitre est organisé comme suit. Après cette introduction, la section 2 expose plus formellement le concept de vue, détaille le langage de définition et présente quelques exemples simples de vues. La section 3 développe les mécanismes d'interrogation de vues. La section 4 pose le problème de la mise à jour des vues et isole les cas simples tolérés par SQL. La section 5 traite des vues concrètes, notamment en vue des applications décisionnelles. En particulier, les techniques de report des mises à jour depuis les relations de base sur des vues concrètes avec agrégats sont étudiées. La conclusion évoque quelques autres extensions possibles du mécanisme de gestion des vues.

2. DEFINITION DES VUES

Dans cette partie, nous définissons tout d'abord précisément ce qu'est une **vue**, puis nous introduisons la syntaxe SQL pour définir une vue.

> **Notion IX.1 : Vue (*View*)**
>
> Une ou plusieurs tables virtuelles dont le schéma et le contenu sont dérivés de la base réelle par un ensemble de questions.

Une vue est donc un ensemble de relations déduites d'une bases de données, par composition des relations de la base. Le schéma de la vue est un schéma externe au sens ANSI/SPARC. Dans la norme SQL, la notion de vue a été réduite à une seule relation déduite. Une vue est donc finalement une table virtuelle calculable par une question.

La syntaxe générale de la commande SQL1 de création de vue est :

```
CREATE VIEW <NOM DE VUE> [ (LISTE D'ATTRIBUT)]
AS <QUESTION>
[WITH CHECK OPTION]
```

Le nom de vue est le nom de la table virtuelle correspondant à la vue, la liste des attributs définit les colonnes de la table virtuelle, la question permet de calculer les tuples peuplant la table virtuelle. Les colonnes du `SELECT` sont appliquées sur celles de la vue. Si les colonnes de la vue ne sont pas spécifiées, celle-ci hérite directement des colonnes du `SELECT` constituant la question.

La clause `WITH CHECK OPTION` permet de spécifier que les tuples insérés ou mis à jour via la vue doivent satisfaire aux conditions de la question définissant la vue. Ces conditions seront vérifiées après la mise à jour : le SGBD testera que les tuples insérés ou modifiés figurent bien parmi la réponse à la question, donc dans la vue. Ceci garantie que les tuples insérés ou modifiés via la vue lui appartiennent bien. Dans le cas contraire, la mise à jour est rejetée si la clause `WITH CHECK OPTION` est présente. Par exemple, si la vue possède des attributs d'une seule table et la question un critère de jointure avec une autre table, les tuples insérés dans la table correspondant à la vue devront joindre avec ceux de l'autre table. Ceci peut être utilisé pour forcer la vérification d'une contrainte référentielle lors d'une insertion via une vue. Nous étudierons plus en détail la justification de cette clause dans la partie traitant des mises à jour au travers de vues.

La suppression d'une vue s'effectue simplement par la commande :

```
DROP <NOM DE VUE>.
```

Cette commande permet de supprimer la définition de vue dans la métabase (aussi appelée catalogue) de la base. En principe, sauf comme nous le verrons ci-dessous, dans le cas des vues concrètes, une vue n'a pas d'existence physique : c'est une fenêtre dynamique (et déformante), non matérialisée sur les disques, par laquelle un utilisateur accède à la base. La destruction de vue n'a donc pas à se soucier de détruire des tuples.

Voici maintenant quelques exemples de définition de vues. Soit la base de données `Viticole` dont le schéma a été rappelé en introduction.

(V1) Les vins de Bordeaux :

```
CREATE VIEW VINSBORDEAUX (NV, CRU, MILL, DEGRÉ)
AS SELECT NV, CRU, MILLÉSIME, DEGRÉ
FROM VINS
WHERE RÉGION = "BORDELAIS".
```

Cette vue est simplement construite par une restriction suivie d'une projection de la table `Vins`. Chaque tuple de la vue est dérivé d'un tuple de la table `Vins`.

(V2) Les gros buveurs :

```
CREATE VIEW GROSBUVEURS
AS SELECT NB, NOM, PRÉNOM, ADRESSE
FROM BUVEURS B, ABUS A
WHERE B.NB = A.NB AND A.QUANTITÉ > 10
WITH CHECK OPTION
```

Un tuple figure dans la vue `GrosBuveurs` si le buveur correspondant a commis au moins un abus en quantité supérieure à 10 verres. C'est là une définition très large des gros buveurs. La définition de vue comporte une restriction et une jointure. La clause `WITH CHECK OPTION` précise que lors d'une insertion d'un buveur via la vue, on doit vérifier qu'il s'agit bien d'un gros buveur, c'est-à-dire que le buveur a déjà commis un abus de quantité supérieure à 10. Notez que ceci spécifie une règle d'intégrité référentielle à l'envers pour les gros buveurs, ce qui est paradoxal.

(V3) Les quantités de vins bues par crus :

```
CREATE VIEW VINSBUS (CRU, MILL, DEGRÉ, TOTAL)
AS SELECT CRU, MILLESIME, DEGRE, SUM(QUANTITE)
FROM VINS V, ABUS A
WHERE V.NV = A.NV
GROUP BY CRU.
```

Cette vue fait appel à une jointure (suivant une contrainte référentielle) suivie d'un calcul d'agrégat (la somme). Un tuple de la table `Vins` donne plusieurs tuples lors de la jointure (autant qu'il y a d'abus) ; ceux-ci sont ensuite regroupés selon le cru.

Pour la base `MAGASINS`, nous définirons seulement une vue (V4) documentant la table `VENTES` selon les « dimensions » `PRODUITS` et `FOURNISSEURS`, résultant de jointures sur clé de ces tables :

```
CREATE VIEW VENTEDOC (NUMV,NOMPRO,MARQUE,NOMFOU,VILLE,REGION, DATE,
      QUANTITE, PRIX) AS
SELECT V.NUMV,P.NOM,P.MARQUE,F.NOM,F.VILLE,F.REGION,V.DATE,
V.QUANTITE, V.PRIX
FROM VENTES V, PRODUITS P, FOURNISSEURS F
WHERE V.NUMPRO=P.NUMRO AND V.NUMFOU=F.NUMFOU
```

Nous verrons des vues plus complexes, notamment avec des agrégats ci-dessous.

La suppression des vues ci-dessus s'effectuera par les commandes :

```
DROP VINSBORDEAUX.
DROP GROSBUVEURS.
DROP VINSBUS.
DROP VENTESDOC.
```

3. INTERROGATION AU TRAVERS DE VUES

Du point de vue de l'utilisateur, l'interrogation au travers d'une vue s'effectue comme pour une table normale. Le rôle du serveur est alors d'enrichir la question avec la définition de vue qu'il retrouve dans la métabase. Pus formellement, la définition d'une vue V sur des relations R1, R2, ...Rn est une fonction V = F(R1, R2, ...Rn), où F est une expression de l'algèbre relationnelle étendue avec des agrégats. F est donc une expression de restriction, projection, jointure, union, différence et agrégats. Une question Q est exprimée sur la vue en SQL. Il s'agit aussi d'une fonction calculant la réponse R = Q(V), où Q est aussi une expression de l'algèbre relationnelle étendue. Calculer R à partir de la base revient donc à remplacer V dans la requête R = Q(V) par sa définition. On obtient alors la fonction composée :

R = Q(F(R1, R2, ...Rn)).

Ce mécanisme est illustré figure IX.1. La requête résultante peut alors être passée à l'optimiseur et le plan d'exécution correspondant peut être calculé. C'est ainsi que fonctionnent les SGBD : ils génèrent la requête composition de la définition de vue et de la requête usager, et traitent ensuite cette requête plus complexe, mais évaluable sur les relations de la base.

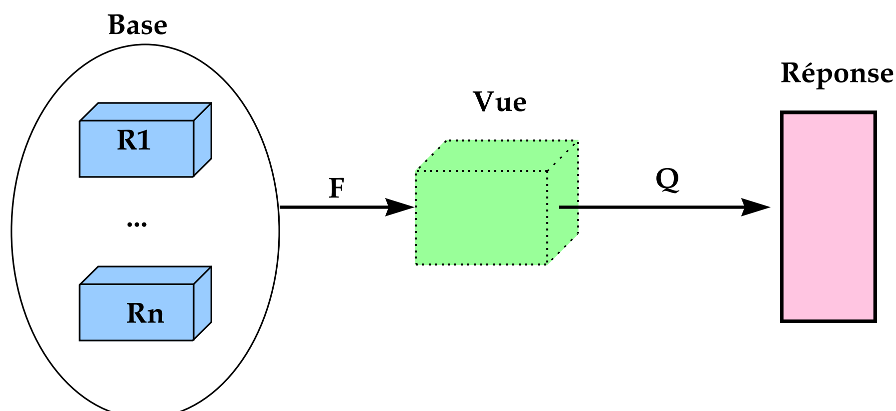

Figure IX.1 — Composition d'une question Q avec la définition de vue F

Deux techniques sont possibles pour réaliser la composition de la vue et de la requête utilisateur : la transformation de la requête source appelée **modification de question**, ou la transformation de l'arbre relationnelle, parfois appelée **concaténation d'arbre**.

> **Notion IX.2 : Modification de question (*Query modification*)**
>
> Mécanisme consistant à modifier une question en remplaçant certaines vues du `FROM` par les relations de base sources de ces vues et en enrichissant les conditions de la clause `WHERE` pour obtenir le résultat de la question initiale.

La modification de question est une technique inventée dans le projet Ingres à Berkeley [Stonebraker75]. Elle permet de remplacer les vues par leurs définitions lors des recherches ou d'ajouter des prédicats afin de vérifier des propriétés avant d'exécuter une requête. La figure IX.2 illustre cette technique pour la recherche des gros buveurs habitant à Versailles. Cette technique peut aussi être utilisée pour les mises à jour afin d'enrichir le critère pour vérifier par exemple la non-violation de contraintes d'intégrité.

(1) Question

```
SELECT NOM, PRENOM
FROM GROSBUVEURS
WHERE ADRESSE LIKE "VERSAILLES".
```

(2) Définition de vue

```
CREATE VIEW GROSBUVEURS
AS SELECT NB, NOM, PRENOM, ADRESSE
FROM BUVEURS B, ABUS A
WHERE B.NB = A.NB AND A.QUANTITE > 10.
```

(3) Question modifiée

```
SELECT NOM, PRENOM
FROM BUVEURS B, ABUS A
WHERE B.ADRESSE LIKE "VERSAILLES" AND B.NB=A.NB AND A.QUANTITE>10.
```

Figure IX.2 — Exemple de modification de question

> **Notion IX.3 : Concaténation d'arbres (*Tree concatenation*)**
>
> Mécanisme consistant à remplacer un nœud pendant dans un arbre relationnel par un autre arbre calculant le nœud remplacé.

La concaténation d'arbres est une technique inventée dans le fameux système R à San-José [Astrahan76]. La définition de vue se retrouve dans la métabase sous la forme d'un arbre relationnel. L'arbre résultant de l'analyse de la question est simplement connecté à celui définissant la vue. L'ensemble constitue un arbre qui représente la question enrichie, qui est alors passée à l'optimiseur. La figure IX.3 illustre ce mécanisme pour la recherche des gros buveurs habitant à Versailles.

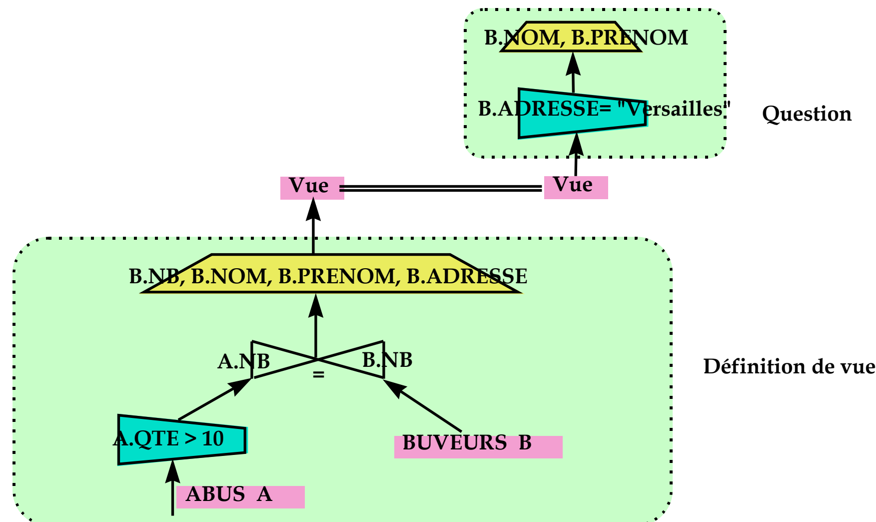

Figure IX.3 — Exemple de concaténation d'arbres

4. MISE A JOUR AU TRAVERS DE VUES

Dans cette section, nous examinons, d'un point de vue pratique puis théorique, le problème des mises à jour au travers des vues.

4.1 Vue mettable à jour

Le problème est de traduire une mise à jour (ou une insertion, ou une suppression) portant sur une vue en mise à jour sur les relations de la base. Toute vue n'est pas **mettable à jour**.

> **Notion IX.4 : Vue mettable à jour (*Updatable view*)**
>
> Vue comportant suffisamment d'attributs pour permettre un report des mises à jour dans la base sans ambiguïté.

De manière plus fine, une vue peut être mettable à jour en insertion, en suppression, ou en modification. Par exemple, la vue V1 définissant les vins de Bordeaux est totalement mettable à jour, c'est-à-dire que toute opération `INSERT`, `DELETE` ou `UPDATE` est reportable sur la base. Ajoutons à la vue V2 définissant les gros buveurs la quantité bue (`QUANTITÉ`) après le `SELECT`. Ceci pose problème lors d'une insertion : comment générer le numéro de vins (`NV`) et la date (`DATE`) de l'abus qui doit obligatoirement être inséré puisque `NB`, `NV`, `DATE` est clé de `ABUS` ? Si l'on obligeait l'existence de l'abus avant d'insérer le buveur il n'y aurait pas de problème. La clause `WITH CHECK OPTION` permet justement de vérifier l'existence de tels tuples. Malheureusement, sa présence simultanée à la contrainte référentielle `ABUS.NB REFERENCE BUVEURS` est impossible ! La suppression et la modification de tuples existants ne pose pas non plus de problème. La vue V3 est encore plus difficile à mettre à jour : il est impossible de déterminer les quantités élémentaires à partir de la somme ! En résumé, l'utilisation de certaines vues en mises à jour est problématique.

4.2 Approche pratique

En pratique, la plupart des systèmes résolvent le problème en restreignant les possibilités de mise à jour à des vues monotable : seuls les attributs d'une table de la base doivent apparaître dans la vue. En imposant de plus que la clé de la table de la base soit présente, il est possible de définir une stratégie de mise à jour simple. Lors d'une insertion, on insère simplement les nouveaux tuples dont la clé n'existe pas, avec des valeurs nulles pour les attributs inexistants. Lors d'une suppression, on supprime les tuples répondant au critère. Lors d'une modification, on modifie les tuples répondant au critère. La définition de vue peut référencer d'autres tables qui permettent de préciser les tuples de la vue. En théorie, il faut vérifier que les tuples insérés, supprimés ou modifiés appartiennent bien à la vue. En pratique, SQL n'effectue cette vérification que si elle a été demandée lors de la définition de vue par la `clause WITH CHECK OPTION`.

Restreindre la mise à jour à des vues monotables est beaucoup trop fort. On peut simplement étendre, au moins en insertion et en suppression, aux vues multitables comportant les clés des tables participantes. Les attributs non documentés sont alors remplacés par des valeurs nulles lors des insertions. Cette solution pratique génère cependant des bases de données avec de nombreuses valeurs nulles. Elle sera souvent interdite dans les systèmes commercialisés. Ceux-ci sont donc loin de permettre toutes les mises à jour théoriquement possibles, comme le montre la figure IX.4.

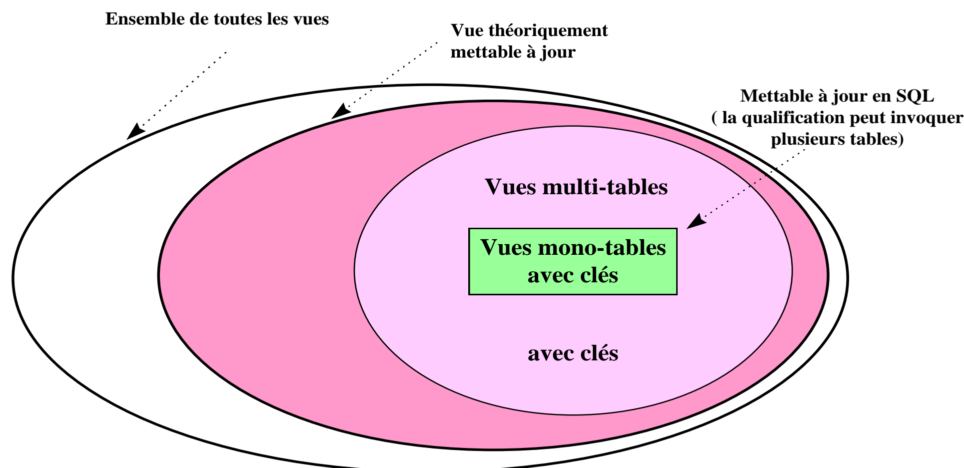

Figure IX.4 — Classification des vues selon les possibilités de mise à jour

4.3 Approche théorique

Le problème théorique est de définir une stratégie de report qui préserve la vue quelle que soit la mise à jour : si l'on calcule la vue et on lui applique la mise à jour, on doit obtenir le même résultat que si on applique les mises à jour à la base et on calcule ensuite la vue. L'équation `u(V(B)) = V(u'(B))` doit donc être vérifiée, en notant `B` la base, `v` le calcul de vue, `u` la mise à jour sur la vue et `u'` les mises à jour correspondantes sur la base. Autrement dit, le diagramme de la figure IX.5 doit commuter. D'autre part, il est évident qu'une bonne stratégie doit préserver le complément de la base, c'est-à-dire toutes les informations de la base qui ne figurent pas dans la vue [Bancilhon81]. Selon ces hypothèses raisonnables, le problème du calcul de u' n'a pas toujours une solution unique [Abiteboul91].

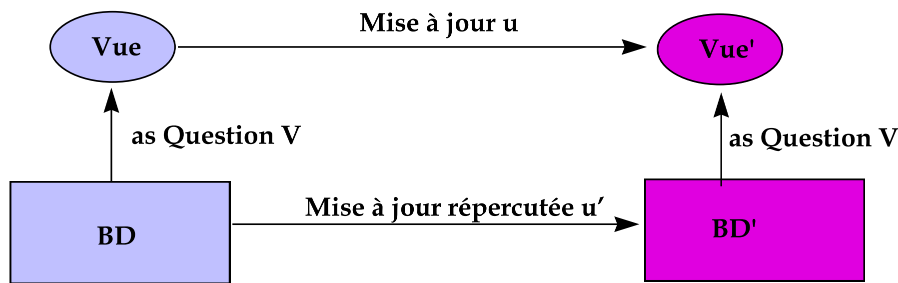

Figure IX.5 — Diagramme de mise à jour et dérivation de vue

Une approche intéressante a été proposée récemment dans [Bentayeb97]. L'idée est de définir l'inverse de chaque opération relationnelle. En effet, soit V = E(R1, R2, ...Rn) une définition de vue. E est une expression de l'algèbre relationnelle. Si l'on peut définir l'inverse de chaque opération de l'algèbre sur une relation, on pourra alors calculer $R1 \times R2... \times Rn = E^{-1}(V)$. La répercussion d'une mise à jour de V se fera simplement en unifiant Ri avec la projection de $E^{-1}(V)$ sur Ri. Le problème de l'inversion de l'algèbre relationnelle a aussi été étudié dans [Imielinski83]. Désignons par t un tuple de la vue V et par [t] une table composée de ce seul tuple. L'**image réciproque** d'un tuple t par E peut être vue comme un ensemble de tables T1, T2, ..., Tn telles que E (T1, T2, ..., Tn) = [t]. Les tables Ti peuvent posséder des attributs inconnus désignés par des variables x,y, etc. Elles ne doivent pas contenir de tuples inutiles pour composer t, donc pour tout tuple ti ∈ Ti, E (T1, ..., Ti-[ti], ..., Tn) ≠ [t]. Insérer le tuple t dans la vue V a alors pour effet de modifier les relations de base Ri en les unifiant avec les relations Ti. L'unification est une opération difficile, pas toujours possible, explicitée dans [Bentayeb97] sous forme d'algorithmes pour l'insertion et la suppression.

En résumé, la mise à jour au travers de vue est un problème complexe, bien étudié en théorie, mais dont les solutions pratiques sont réduites. Certains SGBD permettent des mises à jour de vues multitables en laissant l'administrateur définir la stratégie de report.

5. VUES CONCRETES ET DECISIONNEL

Dans cette section, nous abordons le problème des vues concrètes. Celles-ci sont particulièrement intéressantes dans les **entrepôts de données** (*data warehouse*), où elles facilitent l'analyse de données (*OLAP*) pour l'aide à la décision.

5.1 Définition

Une vue est en principe une fenêtre dynamique sur une base de données, dont une partie est instanciée lors d'une question. La concrétisation de la vue par le serveur peut être plus avantageuse si celle-ci est souvent utilisée et si les tables sources sont peu modifiées. Ainsi, certains serveurs supportent des **vues concrètes**.

Notion IX.5 : Vue concrète (*Concrete view*)

Table calculée à partir des tables de la base par une question et matérialisée sur disques par le SGBD.

Une vue concrète est calculée dès sa définition et mise à jour chaque fois qu'une transaction modifie la base sous-jacente. La mise à jour s'effectue si possible en différentiel, c'est-à-dire que seuls les tuples modifiés sont pris en compte pour calculer les modifications à apporter à la vue. Mais ceci n'est pas toujours possible. Dans le cas de vues définies par des sélections, projections et jointures (SPJ), le report différentiel des insertions est simple, celui des mises à jour de type suppression ou modification beaucoup plus difficile. Pour répercuter suppressions et mises à jour, il faut en général retrouver — au moins partiellement — les chaînes de dérivation qui ont permis de calculer les tuples de la vue concrète. Dans le cas général (vues avec différence, agrégat, etc.), les reports différentiels sont très difficiles, comme nous le verrons ci-dessous

Les vues concrètes sont définissables par une requête du type :

```
CREATE CONCRETE VIEW <SPECIFICATION DE VUE>.
```

Elles sont particulièrement intéressantes lorsqu'elles contiennent des agrégats, car elles permettent de mémoriser des résumés compacts des tables. Par exemple, il est possible de définir des vues concrètes avec agrégats de la table VENTES définie dans l'introduction :

```
VENTES(NUMV, NUMPRO, NUMFOU, DATE, QUANTITE, PRIX)
```

Les tables PRODUITS et FOURNISSEURS permettent de générer des exemples de vues avec jointures :

```
PRODUITS (NUMPRO, NOM, MARQUE, TYPE,PRIX)
FOURNISSEURS (NUMFOU, NOM, VILLE, REGION, TELEPHONE)
```

La vue suivante donne les ventes de produits par fournisseur et par jour :

```
CREATE CONCRETE VIEW VENTESPFD (NUMPRO,NUMFOU,DATE,COMPTE,QUANTOT)AS
SELECT NUMPRO,NUMFOU,DATE,COUNT(*)AS COMPTE,SUM(QUANTITE) AS QUANTOT
FROM VENTES
GROUP BY NUMPRO, NUMFOU, DATE.
```

La notation VENTESPFD signifie « ventes groupées par produits, fournisseurs et dates ». Une vue plus compacte éliminera par exemple les fournisseurs :

```
CREATE CONCRETE VIEW VENTESPD (NUMPRO,DATE,COMPTE,QUANTOT) AS
SELECT NUMPRO, DATE,COUNT(*)AS COMPTE,SUM(QUANTITE) AS QUANTOT
FROM VENTES
GROUP BY NUMPRO, DATE.
```

Notez que la vue VENTESPD peut être dérivée de la vue VENTESPFD par la définition suivante :

```
CREATE CONCRETE VIEW VENTESPD (NUMPRO, DATE, COMPTE,QUANTOT) AS
SELECT NUMPRO, DATE,SUM(COMPTE)AS COMPTE,SUM(QUANTOT) AS QUANTOT
FROM VENTESPFD
GROUP BY NUMFOU.
```

Il est aussi possible de dériver des vues concrètes avec agrégats par jointures avec les tables dimensions, par exemple en cumulant les ventes par région des fournisseurs :

```
CREATE CONCRETE VIEW VENTESPRD(NUMPRO,REGION,DATE,COMPTE,QUANTOT) AS
SELECT NUMPRO, DATE,COUNT(*)AS COMPTE,SUM(QUANTITE) AS QUANTOT
FROM VENTES V, FOURNISSEURS F
WHERE V.NUMFOU = F.NUMFOU
GROUP BY V.NUMPRO, F.REGION.
```

Toutes ces vues concrètes sont très utiles pour l'aide à la décision dans un contexte d'entrepôt de données, où l'on souhaite analyser les ventes selon différentes dimensions [Gray96].

5.2 Stratégie de report

Les vues concrètes doivent être mises à jour lors de la mise à jour des relations de la base. Un problème important est de déterminer une stratégie de report efficace des mises à jour effectuées sur les relations de base. Bien sûr, il est possible de recalculer complètement la vue après chaque mise à jour, mais ceci est inefficace.

Une meilleure stratégie consiste à maintenir la vue concrète de manière différentielle, ou si l'on préfère incrémentale. Soit une vue V = F(R1, R2, ...Rn) définie sur les relations R1, R2, ... Rn. Des mises à jour sont effectuées par exemple sur la relation Ri. Soient des insertions de tuples notées $\Delta^{+}Ri$ et des suppressions notées $\Delta^{-}Ri$. Une modification de tuples existants peut toujours se ramener à une suppression suivie d'une insertion. L'effet net des mises à jour sur Ri est donc $Ri = (Ri - \Delta^{-}Ri) \cup \Delta^{+}Ri$. Le problème est de calculer l'effet net sur V, qui peut être exprimé par des tuples supprimés, puis par des tuples insérés comme suit : $V = (V - \Delta^{-}V) \cup \Delta^{+}V$. Dans certains cas, les mises à jour à apporter sur V ($\Delta^{-}V$, $\Delta^{+}V$) peuvent être calculées à partir de celles apportées sur Ri ($\Delta^{-}Ri$, $\Delta^{+}Ri$). La vue V est alors qualifiée d'**auto-maintenable** [Tompa88, Gupta95]

> **Notion IX.6 : Vue auto-maintenable (*Self-maintenable view)***
>
> Vue concrète contenant suffisamment d'informations pour être mise à jour à partir des mises à jour des relations de base, sans accès aux tuples des relations de la base.

Cette notion est très importante dans les architectures distribuées (client-serveur, entrepôts de données, copies), où la vue est matérialisée sur un site différent de celui de la base. Elle est illustrée figure IX.6. Il est possible de distinguer l'auto-maintenabilité en insertion ou en suppression, c'est-à-dire lors des insertions ou suppressions dans une relation de base. L'auto-maintenabilité peut aussi être caractérisée pour chaque relation, et enfin généralisée au cas d'un groupe de vues [Huyn97].

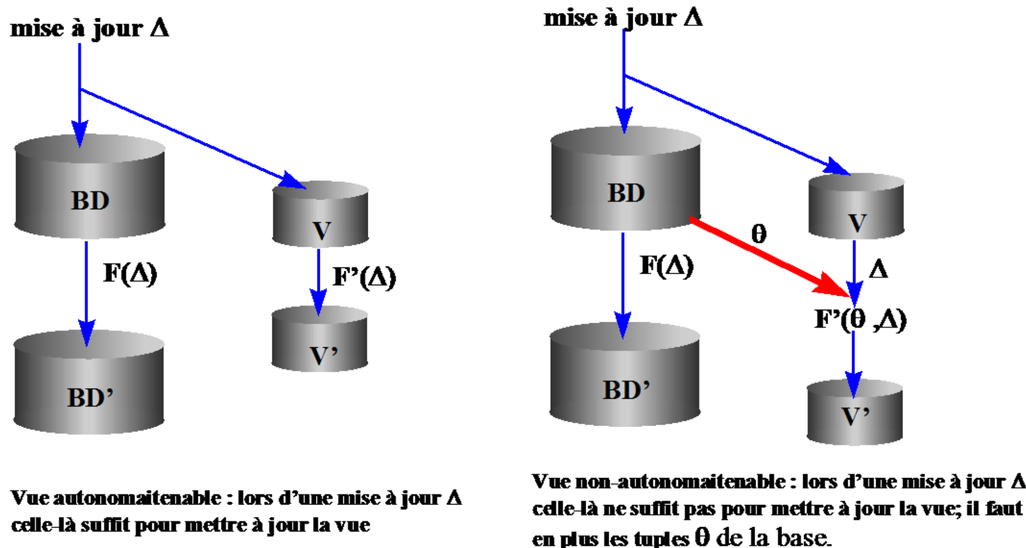

Figure IX.6 — Maintenance de vues avec ou sans accès à la base

En guise d'illustration, considérons la vue V1 des vins de Bordeaux définie ci-dessus comme une restriction de la table `Vins` : $V1 = \sigma_{REGION = "Bordelais"}(VINS)$. Cette vue est auto-maintenable. En effet, comme la restriction commute avec la différence, il est simple de calculer :

$$\Delta^{+}V1 = \sigma_{REGION = "Bordelais"}(\Delta^{+}VINS).$$

$$\Delta^{-}V1 = \sigma_{REGION = "Bordelais"}(\Delta^{-}VINS).$$

Par suite, les mises à jour à apporter ΔV1 sont calculables sans accéder à la table `VINS`, mais seulement à partir des mises à jour de `VINS`. Malheureusement, dès qu'une vue comporte des projections avec élimination de doubles, des jointures ou des différences, elle n'est généralement plus auto-maintenable [Fabret94]. Par exemple, dans le cas de projections sans double, lorsque l'on enlève un tuple d'une table source, on ne sait plus s'il faut l'enlever dans la vue car sa projection peut provenir d'un autre tuple. De même, pour calculer le tuple à enlever dans une vue jointure lors d'une suppression d'un tuple dans une table, il faut accéder à l'autre table pour trouver les tuples jointures. De manière général, savoir si une vue est auto-maintenable s'avère difficile, et même très difficile lorsque la vue contient des agrégats [Gupta96]. Ce dernier cas est examiné dans le paragraphe qui suit.

Dans le cas d'une vue calculée par sélections, projections et jointures (SPJ), les problèmes essentiels semblent provenir des jointures. Toute vue comportant toutes les informations de la base nécessaires à son calcul (attributs de projection et de jointure, tuples) est **auto-maintenable en insertion** ; en effet, la nouvelle instance de la vue V' est :

$$V' = F(R1,..., Ri \cup \Delta^{+}Ri,..., Rn) .$$

Comme F ne contient pas de différence, on a :

$$V' = F(R1,..., Ri,..., Rn) \cup F(R1,..., \Delta^{+}Ri,..., Rn) = V \cup F(R1,..., \Delta^{+}Ri,..., Rn)$$

d'où l'on déduit :

$\Delta V^+ = F (R1,..., \Delta^+Ri,..., Rn)$.

Si F (R1,..., Δ^+Ri,..., Rn) est calculable à partir de V et Δ^+Ri, la vue est auto-maintenable en insertion. Tel est le cas à condition que V contienne les attributs et les tuples de R1, ...Rn nécessaires à son calcul. Les jointures externes seules ne perdent pas de tuples. Donc, une vue calculée par des jointures externes et projections est généralement auto-maintenable en insertion.

Pour l'**auto-maintenabilité en suppression**, le problème est plus difficile encore. On ne peut distribuer F par rapport à Ri - Δ^-Ri, ce qui rend le raisonnement précédent caduc. Cependant, toute vue contenant les attributs de R1, ..., Rn nécessaires à son calcul et au moins une clé de chaque relation de base est auto-maintenable. En effet, l'origine des tuples de la vue peut être identifiée exactement par les clés, ce qui permet de reporter les suppressions.

Des structures annexes associées à une vue peuvent aussi être maintenues [Colby96, Colby97]. Les reports sur la vue concrète peuvent être différés. La vue devient alors un **cliché** (*snapshot*) [Adiba80]. Beaucoup de SGBD supportent les clichés.

> **Notion IX.7 : Cliché (*Snapshot*)**
>
> Vue concrète d'une base de données mise à jour périodiquement.

La gestion de structures différentielles associées au cliché, par exemple les mises à jour sur chaque relation dont il est dérivé (les ΔRi), peut permettre un accès intégré à la vue concrète à jour et conduit au développement d'algorithmes de mise à jour spécialisés [Jagadish97].

5.3 Le cas des agrégats

Le cas de vues avec agrégats est particulièrement important pour les systèmes décisionnels, comme nous allons le voir ci-dessous. Le problème de l'auto-maintenabilité des agrégats a été étudié dans [Gray96], qui propose de distinguer trois classes de fonctions : distributives, algébriques, et non régulières (*holistic*).

Les fonctions agrégatives distributives peuvent être calculées en partitionnant leurs entrées en ensembles disjoints, puis en appliquant la fonction à chacun, enfin en agrégeant les résultats partiels pour obtenir le résultat final. Une fonction distributive `AGG` vérifie donc la propriété :

```
AGG(v1,v2, ...vn) = AGG(AGG(v1, ...vi},AGG(vi+1,..vn)).
```

Parmi les fonctions standard de calcul d'agrégats, `SUM`, `MIN` et `MAX` sont distributives. La fonction `COUNT`, qui compte le nombre d'éléments sans éliminer les doubles, peut être considérée comme distributive en l'interprétant comme une somme de comptes partiels égaux à 1 pour des singletons ; par contre, `COUNT DISTINCT` n'est pas distributive car se pose le problème des doubles.

Les fonctions agrégatives algébriques peuvent être exprimées comme fonctions algébriques de fonctions distributives. Par exemple, la moyenne `AVG` est une fonction algébrique, car `AVG(v1, ...vn) = SUM(v1, ...vn) / COUNT(v1, ...vn)`. Pour le problème

de la maintenance des vues concrètes, les fonctions algébriques peuvent être remplacées par plusieurs fonctions distributives ; par exemple, une colonne AVG sera remplacée par deux colonnes SUM et COUNT.

Les fonctions non régulières ne peuvent être calculées par partitions. Des vues comportant de telles fonctions sont a priori non auto-maintenables.

La notion de vue auto-maintenable s'applique tout particulièrement aux vues résultant d'une agrégation d'une table de base. On parle alors d'**agrégats auto-maintenables**.

> **Notion IX.8 : Agrégats auto-maintenables (*Self-maintenable aggregate*)**
>
> Ensemble d'agrégats pouvant être calculés à partir des anciennes valeurs des fonctions d'agrégats et des mises à jour des données de base servant au calcul de la vue.

Comme pour les vues en général, on peut distinguer l'auto-maintenabilité en insertion et en suppression. [Gray96] a montré que toutes les fonctions d'agrégats distributives sont auto-maintenables en insertion. En général, ce n'est pas le cas en suppression.

Savoir quand éliminer un tuple de la vue pose problème. En introduisant un compteur du nombre de tuples source dans la base pour chaque tuple de la vue (COUNT(*)), il est facile de déterminer si un tuple doit être supprimé : on le supprime quand le compteur passe à 0. COUNT(*) est toujours auto-maintenable en insertion comme en suppression. Lorsque les valeurs nulles sont interdites dans l'attribut de base, SUM et COUNT(*) forment un ensemble auto-maintenables : SUM est maintenu par ajout ou retrait de la valeur, et COUNT(*) par ajout ou retrait de 1 ; COUNT(*) permet de savoir quand supprimer le tuple. En présence de valeurs nulles, il devient difficile de maintenir la somme. MIN et MAX sont aussi des fonctions non auto-maintenable en suppression. En effet, si on supprime la valeur minimale ou maximale, on ne peut recalculer le nouveau minimum ou maximum qu'à partir des valeurs figurant dans la base.

Par exemple, la vue :

```
CREATE CONCRETE VIEW VENTESPFD (NUMPRO,NUMFOU,DATE,COMPTE,QUANTOT)AS
SELECT NUMPRO,NUMFOU,COUNT(*)AS COMPTE,SUM(QUANTITE) AS QUANTOT
FROM VENTES
GROUP BY NUMPRO, NUMFOU
```

est composée d'agrégats auto-maintenables en insertion et en suppression, donc auto-maintenables à condition que Quantité ne puisse être nulle (c'est-à-dire de valeur inconnue) dans l'entrée d'une suppression : il faut alors aller rechercher sa valeur dans la base pour enlever la véritable quantité à la somme.

Par contre, la vue :

```
CREATE CONCRETE VIEW VENTESPFD (NUMPRO,NUMFOU,DATE,COMPTE,QUANTOT)AS
SELECT NUMPRO,NUMFOU,COUNT(*)AS COMPTE,MIN(QUANTITE) AS QUANMIN
FROM VENTES
GROUP BY NUMPRO, NUMFOU
```

est composée d'agrégats auto-maintenables en insertion mais pas en suppression (problème avec le `MIN`).

6. CONCLUSION

La gestion de vues en interrogation est un problème bien compris dans les SGBD relationnels depuis la fin des années 70. Nous avons introduit ci-dessus ses principes de base. Le report des mises à jour des vues sur la base est un problème plus difficile encore. Des solutions pratiques et générales restent à inventer. L'utilisation de déclencheurs sur les vues peut être une solution pour définir les stratégies de report.

La matérialisation des vues a connu une nouvelle activité ces dernières années, avec l'apparition des entrepôts de données. Les vues avec agrégats sont particulièrement importantes pour gérer des données résumées, en particulier le fameux cube de données très utile en décisionnel. Les techniques sont maintenant connues. Il reste à les mettre en œuvre efficacement dans les systèmes, qui gère pour l'instant peu de redondances et souvent des organisations physiques ad hoc pour le multidimensionnel. Ceci n'est plus vrai dans les entrepôts de données, qui utilisent souvent les vues concrètes pour faciliter les calculs d'agrégats [Bello98].

Les vues connaissent aussi un nouveau développement dans le monde objet avec l'apparition de vues orientées objets au-dessus de bases de données relationnelles par exemple. Nous aborderons cet aspect dans le cadre des SGBD objet ou objet-relationnel.

7. BIBLIOGRAPHIE

[Abiteboul91] Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley, 1995.

Comme son nom l'indique, ce livre traite des fondements des bases de données relationnelles et objet. Souvent fondé sur une approche logique ou algébrique, il reprend toute la théorie des bases de données, y compris le problème de la mise à jour au travers des vues, parfaitement bien analysé.

[Adiba80] Adiba M., Lindsay B., « Database Snapshots », *Intl. Conf. on Very Large Databases*, IEEE Ed., Montréal, Canada, Sept. 1980.

*Cet article introduit la notion de cliché (*snapshot*) et montre son intérêt, notamment dans les bases de données réparties. Il décrit aussi la réalisation effectuée dans le fameux système R*.*

[Adiba81] Adiba M., « Derived Relations : A Unified Mechanism for Views, Snapshots and Distributed Data », *Intl. Conf. on Very Large Databases,* IEEE Ed., Cannes, France, Sept. 1981.

L'auteur généralise les clichés aux relations dérivées, dont il montre l'intérêt dans les bases de données réparties. Il discute aussi les algorithmes de mise à jour.

[Astrahan76] Astrahan M. M., et. al., « System R : Relational Approach to Database Management », *ACM TODS, V1, N2*, Juin 1976.

L'article de référence sur le fameux System R. Il décrit aussi le mécanisme de concaténation d'arbres implémenté pour la première fois dans ce système.

[Bancilhon81] Bancilhon F., Spyratos N., « Update Semantics and Relational Views », *ACM TODS, V4, N6*, Dec. 1981, pp. 557-575.

Un article de référence en matière de mises à jour de vues. Les auteurs posent le problème en termes d'invariance de la vue suite aux mises à jour directes ou répercutées dans la base. Ils montrent qu'une condition suffisante de maintenabilité est la possibilité de définir des stratégies de report à compléments constants.

[Bello98] Bello R.G, Dias K., Downing A., Freenan J., Norcott D., Dun H., Witkowski A., Ziauddin M., « Materialized Views in Oracle », *Intl. Conf. on Very Large Databases,* Morgan & Kauffman Ed., New York, USA, Août 1998, pp. 659-664.

Cet article explique la gestion des vues matérialisées dans Oracle 8. Celles-ci sont utilisées pour les entrepôts de données et la réplication. Elles peuvent être rafraîchies à la fin des transactions, sur demande ou périodiquement. Les rafraîchissements par lots sont optimisés. L'optimisation des requêtes prend en compte les vues concrètes à l'aide d'une technique de réécriture basée sur un modèle de coût.

[Bentayeb97] Bentayeb F., Laurent D., « Inversion de l'algèbre relationnelle et mises à jour », *13e Journées Bases de Données Avancées* (BDA 97), Ed. INRIA, Grenoble, 1997, pp. 199-218.

Cet article propose une approche déterministe de la mise à jour au travers de vue, fondée sur la notion d'image réciproque. Des algorithmes d'inversion de l'algèbre sont proposés. Une stratégie de report avec stockage éventuel dans la vue concrétisée est étudiée.

[Chamberlin75] Chamberlin D., Gray J., Traiger I., « Views, Authorization and Locking in a Relational Database System », *Proc. National Computer Conf.*, 1975, pp. 425-430.

Cet article détaille les mécanismes de vue, d'autorisation et de verrouillage implantée dans System R.

[Colby96] Colby L.S., Griffin T., Libkin L., Mumick I., RTrickey H., "Algorithms for Deferred View Maintenance", *Proc. ACM SIGMOD* Int. Conf. on Management of Data, Montreal, USA, Mai 1996.

[Colby97] Colby L.S., Kawaguchi A., Lieuwen D.F, Mimick I.S., K. Ross, "Supporting Multiple View Maintenance Policies", *Proc. ACM SIGMOD* Int. Conf. on Management of Data, Tucson, USA, pp. 405-416, Mai 1997.

Ces deux articles explorent différentes stratégies pour la maintenance de vues concrètes. Les auteurs proposent notamment des algorithmes qui garantissent la cohérence dans des cas de reports des mises à jour en temps différés, avec divers niveaux et groupes de vues.

[Dayal82] Dayal U., Bernstein P., « On the Correct Translation of Update Operations on Relational Views », *ACM TODS*, V8, N3, Dept. 1982.

Cet article discute des stratégies de report de mise à jour au travers de vues et définit des critères de correction.

[Fabret94] Fabret F., *Optimisation du calcul incrémental dans les langages de règles pour bases de données*, Thèse, Université de Versailles SQ, Décembre 1994.

Cette thèse présente une caractérisation des vues concrètes auto-maintenables dans les bases de données. Les vues sont définies par projection, jointure, union, différence et restriction. Cette thèse propose aussi des algorithmes afin de maintenir des réseaux de vues pour accélérer le calcul des requêtes dans une BD déductive.

[Gray96] Gray J., Bosworth A., Layman A., Pirahesh H., "Datacube : A relational Aggregation Operator Generalizing Group-by, Cross-tab, and Sub-totals", *IEEE Int. Conf. on Data Engineering*, pp. 152-159, 1996.

Cet article introduit un nouvel opérateur d'agrégation, appelé Datacube, afin de calculer des agrégats multidimensionnels. Nous étudierons plus en détails les opérateurs des bases de données multidimensionnelles dans le cadre du chapitre sur les entrepôts de données.

[Gupta95] Gupta M., Mumick I.S., « Maintenance of Materialized Views : Problems, Techniques and Applications », *IEEE Data Engineering Bulletin, special Issue on Materialized Views and data Warehousing*, N18(2), Juin 1995.

Les auteurs introduisent les différents problèmes posés par la gestion de vues concrètes dans une bases de données. Ils proposent quelques solutions et montrent quelques applications possibles pour la gestion d'entrepôts de données performants.

[Gupta96] Gupta A., Jagadish H.V., Mumick I.S., « Data Integration Using Self-Maintainable Views », *Intl. Conf. on Extended Database Technology (EDBT),* LCNS N1057, Avignon, France, 1996, pp. 140-144.

Cet article définit précisément la notion de vue auto-maintenable. Les auteurs établissent des conditions suffisantes pour qu'une vue selection-projection-jointure soit auto-maintenable.

[Huyn97] Huyn N., « Multiple-View Self-Maintenance in Data Warehousing Environments » *Intl. Conf. on Very Large Databases,* Morgan & Kauffman Ed., Athens, Grèce, Août 1997, pp. 26-35.

Cet article étudie le problème de l'auto-maintenabilité d'un ensemble de vues. Il propose des algorithmes qui permettent de tester si un groupe de vues est auto-maintenable par génération de requêtes SQL. De plus, dans le cas où la réponse est positive, les algorithmes génèrent les programmes de mises à jour.

[Jagadish97] Jagadish H.V., Narayan P.P.S., Seshadri S., Sudarshan S., Kanneganti R., « Incremental Organization for Data Recording and Warehousing », *Intl. Conf. on Very Large Databases*, Morgan & Kauffman Ed., Athens, Greece, Août 1997, pp. 16-25.

Les auteurs proposent deux techniques, la première basée sur des séquences triées insérées dans un B-tree, la seconde sur une organisation par hachage, pour stocker les enregistrements lors de leur arrivée dans un entrepôt de données, avant leur intégration dans les vues matérialisées. Ces techniques précisément évaluées supportent des environnements concurrents avec reprises.

[Keller87] Keller M.A., « Comments on Bancilhon and Spyratos's Update Semantics and Relational Views », *ACM TODS*, V12, N3, Sept. 1987, pp. 521-523.

Cet article montre que la condition suffisante de maintenabilité qu'est la possibilité de définir des stratégies de report à compléments constants est trop forte, et qu'elle peut conduire à interdire des stratégies intéressantes.

[Kerhervé86] Kerhervé B., « Vues Relationnelles : Implémentation dans un SGBD centralisé et distribué », *Thèse de doctorat, Université de Paris VI*, Mars 1986.

Cette thèse décrit l'implémentation de vues réalisée dans le SGBD SABRE à l'INRIA. Un mécanisme de gestion de vues concrètes par manipulation de compteurs au niveau des opérateurs relationnels est aussi proposé.

[Nicolas83] Nicolas J-M., Yazdanian K., « An Outline of BDGen : A Deductive DBMS », *Woldwide IFIP Congress*, Paris, 1983.

Cet article introduit le système déductif BDGen réalisé au CERT à Toulouse. Il maintient des vues définies par des règles en utilisant des compteurs de raisons de présence d'un tuple.

[Stonebraker74] Stonebraker M.R., « A Functional View of Data Independance », *ACM SIGMOD Workshop on Data Description, Access and Control*, ACM Ed., mai 1974.

Un des premiers articles de Mike Stonebraker, l'un des pères du système INGRES. Il plaide ici pour l'introduction de vues assurant l'indépendance logique.

[Stonebraker75] Stonebraker M., « Implémentation of Integrity Constraints and Views by Query Modification », *Proc. ACM-SIGMOD* Intl. Conf. On Management of data, San José, CA 1975.

Cet article propose de modifier les questions au niveau du source par la définition de vues pour répondre aux requêtes. La technique est formalisée avec le langage QUEL d'INGRES, où elle a été implémentée.