INTEGRITE ET BD ACTIVES

1. INTRODUCTION

Un SGBD doit garantir la cohérence des données lors des mises à jour de la base. En effet, les données d'une base ne sont pas indépendantes, mais obéissent à des règles sémantiques appelées **contraintes d'intégrité**. Ces règles peuvent être déclarées explicitement et mémorisées dans le dictionnaire de données, ou plus discrètement implicites. Les transactions sont des groupes de mises à jour dépendantes qui font passer la base d'un état cohérent à un autre état cohérent. A la fin de chaque transaction, ou plus radicalement après chaque mise à jour, il est nécessaire de contrôler qu'aucune règle d'intégrité n'est violée.

Une contrainte d'intégrité peut spécifier l'égalité de deux données ; par exemple, un numéro de vins dans la table `VINS` doit être égal au numéro du même vin dans la table `ABUS`. De manière plus complexe, elle peut spécifier une assertion comportant de multiples données ; par exemple, la somme des avoirs des comptes doit rester égale à l'avoir de la banque. Nous étudions ci-dessous les différents types de contraintes supportées par le modèle relationnel. Quelle que soit la complexité de la contrainte, le problème est de rejeter les mises à jour qui la violent. Pour ce faire, différentes techniques sont possibles, fondées soit sur la prévention qui consiste à empêcher les mises à jour non valides de se produire, soit sur la détection impliquant de défaire les transactions incorrectes.

Une autre manière de protéger l'intégrité des données est l'utilisation de **déclencheurs** (en anglais, *triggers*). Ceux-ci permettent de déclencher une opération conséquente suite à une première opération sur la base. La forme générale d'un déclencheur est `ON <événement> IF <condition> THEN <action>`. L'événement est souvent une action de mise à jour de la base. La condition est un prédicat logique vrai ou faux. L'action peut permettre d'interdire la mise à jour (`ABORT`) ou de la compenser (`UPDATE`). Ainsi, en surveillant les mises à jour et en déclenchant des effets de bord, il est possible de maintenir l'intégrité d'une base.

Mieux, les déclencheurs permettent de modéliser au sein de la base de données le comportement réactif des applications. Les SGBD traditionnels sont passifs, en ce sens qu'ils exécutent des commandes de mises à jour et de recherche en provenance des applications. Avec des déclencheurs, ils deviennent actifs et sont capables de réagir à des événements externes. Par exemple, la surveillance d'un commerce électronique peut nécessiter le refus de vente à un client suspect, une demande d'approvisionnement en cas de rupture de stock, etc.. Tous ces événements peuvent être capturés directement par le SGBD avec des déclencheurs appropriés. On passe alors à la notion de **base de données actives**, qui peut comporter des règles avec conditions déclenchées par des événements composées de plusieurs sous-événements (par exemple, une conjonction d'événements simples). Une base de données active permet donc de déplacer le comportement réactif des applications dans le SGBD. Ceci nécessite la prise en compte d'un modèle de définition de connaissances et d'un modèle d'exécution de règles au sein du SGBD. Nous examinerons ces aspects dans la deuxième partie de ce chapitre.

Ce chapitre traite donc des règles d'intégrité et des bases de données actives. Après cette introduction, la section 2 examine les différents types de contraintes d'intégrité et résume le langage de définition de contraintes de SQL2. La section 3 introduit quelques techniques d'analyse (contrôle de cohérence, de non-redondance) de contraintes et quelques techniques de simplification : simplifications possibles compte tenu du type d'opération, différenciations en considérant les delta-relations, etc. La section 4 montre comment contrôler les contraintes lors des mises à jour : diverses techniques curatives ou préventives sont étudiées, pour aboutir à la technique préventive au vol souvent appliquée pour les contraintes simples exprimables en SQL. La section 5 introduit les notions de base de données active et de déclencheur, et analyse les composants d'un SGBD actif. La section 6 étudie plus en détail les déclencheurs et donne l'essentiel de la syntaxe SQL3, les déclencheurs n'apparaissant qu'à ce niveau dans la norme. La section 7 montre comment sont exécutés les déclencheurs dans un SGBD actif. Au-delà de SQL, elle soulève quelques problèmes épineux liés aux déclencheurs.

2. TYPOLOGIE DES CONTRAINTES D'INTEGRITE

Dans cette section, nous étudions les différents types de contraintes d'intégrité. Celles-ci sont classées selon leur utilisation à modéliser des relations statiques entre données, ou des relations dynamiques d'évolution des données. A un second niveau, nous énumérons les différents types de contraintes.

2.1 Contraintes structurelles

Une **contrainte structurelle** fait partie intégrante du modèle et s'applique sur les structures de base (table, colonne, ligne).

> **Notion VIII.1 : Contrainte structurelle (*Structural constraint*)**
>
> Contrainte d'intégrité spécifique à un modèle exprimant une propriété fondamentale d'une structure de données du modèle.

Les contraintes structurelles sont généralement statiques. Pour le modèle relationnel, elles permettent d'exprimer explicitement certaines propriétés des relations et des domaines des

attributs. Ces contraintes sont donc partie intégrante du modèle et ont déjà été introduites lors de sa présentation. Il est possible de distinguer les contraintes structurelles suivantes :

1. **Unicité de clé.** Elle permet de préciser les attributs clés d'une relation, c'est-à-dire un groupe d'attributs non nul dont la valeur permet de déterminer un tuple unique dans une table. Par exemple, la table `VINS` possède une clé unique `NV` ; la table `ABUS` possède une clé multiple (`NV, NB, DATE`).

2. **Contrainte référentielle.** Elle spécifie que toute valeur d'un groupe de colonnes d'une table doit figurer comme valeur de clé dans une autre table. Une telle contrainte représente une association obligatoire entre deux tables, la table référencée correspondant à l'entité, la table référençante à l'association. Par exemple, toute ligne de la table `ABUS` référencera un numéro de vin existant dans la table `VINS`, ou toute ligne de commande référencera un produit existant, etc.

3. **Contrainte de domaine.** Ce type de contrainte permet de restreindre la plage de valeurs d'un domaine. En général, un domaine est défini par un type et un éventuel domaine de variation spécifié par une contrainte de domaine. Une contrainte de domaine peut simplement préciser la liste des valeurs permises (définition en extension) ou une plage de valeurs (contrainte en intention). Par exemple, un cru sera choisi parmi {Volnay, Beaujolais, Chablis, Graves, Sancerre} ; une quantité de vin sera comprise entre 0 et 100.

4. **Contrainte de non nullité.** Une telle contrainte spécifie que la valeur d'un attribut doit être renseignée. Par exemple, le degré d'un vin ne pourra être nul, et devra donc être documenté lors de l'insertion d'un vin dans la base, ou après toute mise à jour.

Le choix des contraintes structurelles est effectué lors de la définition d'un modèle. Codd a par exemple retenu la notion de clé composée d'attributs visibles à l'utilisateur pour identifier les tuples dans une table. Ce choix est plutôt arbitraire. Le modèle objet a choisi d'utiliser des identifiants système appelés identifiants d'objets. Codd aurait pu retenir les identifiants de tuples invariants (TID) pour identifier les tuples. On aurait alors un modèle relationnel différent, mais ceci est une autre histoire. Au contraire, dans sa première version, le modèle relationnel n'introduisait pas les contraintes référentielles : elles ont été ajoutées en 1981 pour répondre aux critiques des tenants du modèle réseau, qui trouvaient que le modèle relationnel perdait la sémantique des associations [Date81].

2.2 Contraintes non structurelles

Les autres contraintes d'intégrité, non inhérentes au modèle de données, sont regroupées dans la classe des contraintes non structurelles. La plupart traitent plutôt de l'évolution des données suite aux mises à jour ; elles sont souvent appelées **contraintes de comportement**.

> **Notion VIII.2 : Contrainte de comportement (*Behavioral constraint*)**
>
> Contrainte d'intégrité exprimant une règle d'évolution que doivent vérifier les données lors des mises à jour.

Certaines contraintes structurelles peuvent aussi être qualifiées de contraintes de comportement (par exemple l'unicité de clé). Quoi qu'il en soit, par opposition aux contraintes

structurelles, les non structurelles ne font pas partie intégrante du modèle relationnel, et ne sont donc pas définies dans la commande `CREATE TABLE`. Elles sont définies par la commande additionnelle `CREATE ASSERTION`. Dans le cas du modèle relationnel, la plupart peuvent être exprimées sous forme d'assertions de la logique du premier ordre, éventuellement temporelles. On distingue en particulier :

1. **Les dépendances fonctionnelles**. Celles-ci expriment l'existence d'une fonction permettant de déterminer la valeur d'un groupe d'attributs à partir de celle d'un autre groupe. Comme nous le verrons dans le chapitre sur la conception, on dit que X -> Y (X détermine Y) si pour toute valeur de X il existe une valeur unique de Y associée. Par exemple, le cru détermine uniquement la région (dans une table de vins français).

2. **Les dépendances multivaluées**. Ce sont une généralisation des précédentes au cas de fonctions multivaluées. On dit que `X->> Y` (X multidétermine Y) dans une relation R si pour toute valeur de X il existe un ensemble de valeur de Y, et ceci indépendamment des valeurs des autres attributs Z de la relation R. Par exemple, dans une relation `BUVEURS (NOM, CRU, SPORT)` décrivant les vins bus (`CRU`) et les sports pratiqués (`SPORT`) par les buveurs, `NOM ->> CRU` indépendamment de `SPORT`.

3. **Les dépendances d'inclusion**. Elles permettent de spécifier que les valeurs d'un groupe de colonnes d'une table doivent rester incluses dans celles d'un groupe de colonnes d'une autre table. Elles généralisent donc les contraintes référentielles vues ci-dessus aux cas de colonnes quelconques. Par exemple, la colonne `VILLE` de la table `BUVEURS` doit rester incluse dans la colonne `VILLE` de la table `REGION`.

4. **Les contraintes temporelles**. Plus sophistiquées que les précédentes, elles font intervenir le temps. Elles permettent de comparer l'ancienne valeur d'un attribut à la nouvelle après mise à jour. On exprimera par exemple avec une telle contrainte le fait qu'un salaire ne peut que croître.

5. **Les contraintes équationnelles**. Il s'agit là de comparer deux expressions arithmétiques calculées à partir de données de la base et de forcer l'égalité ou une inégalité. La dimension temporelle peut être prise en compte en faisant intervenir des données avant et après mise à jour. Les calculs d'agrégats sont aussi possibles. Un exemple simple est une contrainte permettant d'exprimer, dans une base de gestion de stocks, le fait que la quantité en stock est égale à la somme des quantités achetées moins la somme des quantités vendues, et ce pour tout produit. Il est aussi possible d'exprimer des invariants en utilisant la dimension temps, par exemple le fait que l'avoir d'une banque reste le même après un transfert de fonds d'un compte à un autre.

Nous regroupons sous le terme **dépendances généralisées** les différents types de dépendances entre attributs (dépendances fonctionnelles, multivaluées et d'inclusion). Très utiles pour la conception des bases de données, elles permettent de mieux contrôler les redondances au sein des relations.

2.3 Expression des contraintes en SQL

SQL1 permet la création de contraintes lors de la création des tables, par la commande indiquée figure VIII.1. On retrouve les contraintes de non nullité, d'unicité de clé, référentielles et de domaines. Les contraintes mono-attributs peuvent être déclarées à chaque attribut, alors que les celles portant sur plusieurs attributs, appelées contraintes de relation, sont factorisées à la fin de la déclaration. La figure VIII.2 illustre de telles contraintes classiques pour la table `ABUS`.

```
CREATE TABLE <nom de table>
( {<Attribut> <Domaine> [<CONTRAINTE D'ATTRIBUT>]}+ )
        [<CONTRAINTE DE RELATION>]

<CONTRAINTE D'ATTRIBUT> ::=
                NOT NULL |
                UNIQUE | PRIMARY KEY
                REFERENCES <Relation> (<Attribut>) |
                CHECK <Condition>

<CONTRAINTE DE RELATION> ::=
                UNIQUE (<Attribut>+) | PRIMARY KEY (<Attribut>+) |
                FOREIGN KEY (<Attribut>+)
                        REFERENCES <Relation> (<Attribut>+) |
                CHECK <Condition>
```

Figure VIII.1 — Création de table et contrainte d'intégrité en SQL1

```
CREATE TABLE ABUS (
                NB INT NOT NULL,
                NV INT NOT NULL REFERENCES VINS(NV),
                DATE DEC(6) CHECK BETWEEN 010180 AND 311299,
                QUANTITE SMALLINT DEFAULT 1,
                PRIMARY KEY(NB, NV, DATE),
                FOREIGN KEY NB REFERENCES BUVEURS,
                CHECK (QUANTITE BETWEEN 1 AND 100) )
```

Figure VIII.2 — Exemple de création de table avec contrainte en SQL1

SQL2 étend d'une part les contraintes attachées à une table et permet de déclarer d'autres contraintes par une commande séparée `CREATE ASSERTION`. L'extension essentielle au niveau du `CREATE TABLE` porte sur les contraintes référentielles. Il devient possible de répercuter certaines mises à jour de la relation référencée. La nouvelle syntaxe est donnée

figure VIII.3. La clause ON DELETE indique l'action que doit exécuter le système dans la table dépendante lors d'une suppression dans la table maître. NO ACTION ne provoque aucune action, et a donc un effet identique à l'absence de la clause comme en SQL1. CASCADE signifie qu'il faut enlever les tuples correspondant de la table dépendante. SET DEFAULT indique qu'il faut remplacer la clé étrangère des tuples correspondant de la table dépendante par la valeur par défaut qui doit être déclarée au niveau de l'attribut. SET NULL a un effet identique, mais cette fois avec la valeur nulle. La clause ON UPDATE indique comment le système doit modifier la clé étrangère dans la table dépendante lors de la mise à jour d'une clé primaire dans la table maître. Les effets sont identiques au ON DELETE, à ceci près que CASCADE provoque la modification de la clé étrangère de la même manière que la clé primaire.

De même pour la clause ON UPDATE lors d'une mise à jour de la clé référencée dans la table maître.

FOREIGN KEY (<Attribut>+)
REFERENCES <Relation> (<Attribut>+)
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]

Figure VIII.3 — Contrainte référentielle en SQL2

La syntaxe de la nouvelle clause de définition de contraintes est résumée figure VIII.4. Une telle contrainte peut être vérifiée immédiatement après chaque mise à jour précisée (clause AFTER), ou en fin de transaction (clause BEFORE COMMIT). La condition peut porter sur une table entière (option FOR), ou sur chaque ligne de la table (option FOR EACH ROW OF).

CREATE ASSERTION <nom de contrainte>
[{ **BEFORE COMMIT |**
AFTER {INSERT|DELETE|UPDATE[OF(Attribut+)]} **ON** <Relation> }...]
CHECK <Condition>
[FOR [EACH ROW OF] <Relation>]

Figure VIII.4 — Création de contrainte indépendante en SQL2

Voici quelques exemples. L'assertion suivante permet de vérifier d'une manière un peu détournée que chaque vin à bien un degré supérieur à 10 :

```
CREATE ASSERTION MINDEGRÉ
BEFORE COMMIT
CHECK (SELECT MIN(DEGRÉ) FROM VINS > 10)
FOR VINS ;
```

Celle qui suit vérifie que la quantité totale bue reste inférieure à 100 pour chaque buveur :

```
CREATE ASSERTION SOMMEQUANTITÉBUE
BEFORE COMMIT
CHECK (SELECT SUM(QUANTITE) FROM ABUS GROUP BY NB) < 100
FOR ABUS.
```

En supposant que la table VINS possède un attribut QUALITÉ, on pourra par exemple vérifier que chaque vin de qualité supérieure a au moins dix tuples d'ABUS le référençant. Une telle contrainte nécessitera d'insérer les ABUS d'abord et de reporter la vérification de contrainte référentielle au COMMIT, ce qui peut être fait par la clause BEFORE COMMIT.

```
CREATE ASSERTION QUALITÉSUP
AFTER INSERT ON VINS
CHECK ((SELECT COUNT(*)
         FROM ABUS A, VINS V
         WHERE A.NV = V.NV
         AND V.QUALITE = "SUPERIEURE") > 10).
```

On peut donc ainsi écrire des contraintes très complexes, difficiles à vérifier pour le système.

3. ANALYSE DES CONTRAINTES D'INTEGRITE

Les contraintes d'intégrité définies par un administrateur de données sont créées en SQL. Avant d'être stockées dans la méta-base, elles doivent être analysées et contrôlées sur la base si celle-ci n'est pas vide. L'analyse doit mettre les contraintes sous une forme interne facilement exploitable, et aussi répondre aux questions suivantes :

1. Les contraintes sont-elles cohérentes entre elles ?
2. Ne sont-elles pas redondantes ?
3. Quelles contraintes doit-on vérifier suite à une insertion, une suppression, une mise à jour d'une table ?
4. Est-il possible de simplifier les contraintes pour faciliter le contrôle ?

Ces différents points sont examinés ci-dessous.

3.1 Test de cohérence et de non-redondance

Soit I = {I1, I2, ...In} un ensemble de contraintes. Existe-t-il une base de données capable de satisfaire toutes ces contraintes ? Si oui, on dira que l'ensemble de contraintes est **cohérent**.

> **Notion VIII.3 : Cohérence de contraintes (*Constraint consistency*)**
>
> Ensemble de contraintes non contradictoires, pouvant en conséquence être satisfait par au moins une base de données.

Par exemple :

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ > 12 ;
```

et :

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ < 11 ;
```

sont deux contraintes contradictoires, un vin ne pouvant être de degré à la fois supérieur à 12 et inférieur à 11.

Si les contraintes sont exprimables en logique du premier ordre, il est possible d'utiliser une méthode de preuve pour tester la cohérence des contraintes. De nos jours, aucun SGBD n'est capable de vérifier la cohérence d'un ensemble de contraintes (sauf peut-être les trop rares SGBD déductifs).

Étant donné un ensemble de contraintes, il est aussi possible que certaines contraintes puissent être déduites des autres, donc soient **redondantes**.

> **Notion VIII.4 : Contraintes redondantes (*Redundant constraints*)**
>
> Ensemble de contraintes dont l'une au moins peut être déduite des autres.

Par exemple :

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ > 12 ;
```

et :

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ > 11 ;
```

sont deux contraintes redondantes, la seconde pouvant être réduite de la première. Là encore, si les contraintes sont exprimables en logique du premier ordre, il est possible d'utiliser une méthode de preuve pour tester leur non-redondance. En cas de redondance, il n'est pas simple de déterminer quelle contrainte éliminer. Le problème est de trouver un ensemble minimal de contraintes à vérifier permettant de démontrer que toutes les contraintes sont satisfaites. L'ensemble retenu doit être optimal du point de vue du temps de vérification, ce qui implique l'utilisation d'une fonction de coût. De nos jours, aucun SGBD (sauf peut-être les trop rares SGBD déductifs) n'est capable de vérifier la non-redondance d'un ensemble de contraintes, et encore moins de déterminer un ensemble minimal de contraintes. Cela n'est pas très grave car les contraintes non structurelles restent peu utilisées.

3.2 Simplification opérationnelle

Certaines contraintes d'intégrité ne peuvent être violées que par certains types de mise à jour sur une relation donnée. Par exemple, l'unicité de clé dans une relation `R` ne peut être violée par une suppression sur `R`. Pour éviter des contrôles inutiles, il est important d'identifier quel type de mise à jour peut violer une contrainte donnée. SQL distingue les opérations d'insertion (`INSERT`), de suppression (`DELETE`) et de mise à jour (`UPDATE`). Il est alors intéressant de marquer une contrainte générale `I` avec des étiquettes (`R`,`U`), `R` indiquant les relations pouvant

être violées et U le type de mise à jour associé. Par exemple, l'unicité de clé K sur une relation R sera étiquetée (R,INSERT) et (R, UPDATE).

Des règles générales d'étiquetage peuvent être simplement énoncées :

1. Toute contrainte affirmant l'existence d'un tuple dans une relation R doit être étiquetée (R, DELETE).
2. Toute contrainte vraie pour tous les tuples d'une relation R doit être étiquetée (R,INSERT).
3. Toute contrainte étiquetée (R,DELETE) ou (R,INSERT) doit être étiquetée (R,MODIFY).

Soit par exemple une contrainte référentielle de R vers S. Celle-ci affirme que pour tout tuple de R il doit exister un tuple de S vérifiant R.A = S.K. Un tuple de S doit exister, d'où l'étiquette (S, DELETE). Tout tuple de R doit vérifier la contrainte, d'où l'étiquette (R, INSERT). Il faut donc ajouter les étiquettes (S, MODIFY) et (R, MODIFY). Ces petites manipulations peuvent être plus formellement définies en utilisant le calcul relationnel de tuples avec quantificateurs que nous verrons dans le contexte des bases de données déductives.

3.3 Simplification différentielle

Les contraintes d'intégrité sont vérifiées après chaque transaction ayant modifiée la base. Ainsi, une transaction transforme une base de données d'état cohérent en état cohérent (voir figure VIII.5).

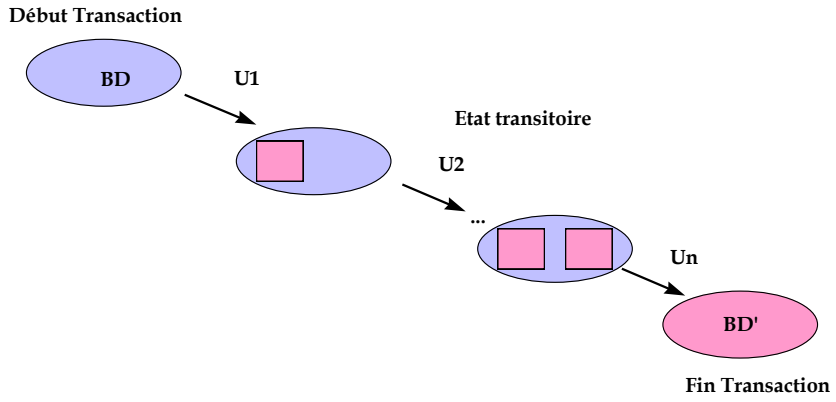

Figure VIII.5 — Modification d'une base de données par une transaction

Lors de la vérification d'une contrainte d'intégrité, il est possible de profiter du fait que la contrainte était vérifiée en début de transaction, avant mise à jour. Dans le meilleur des cas, seules les données mises à jour doivent être vérifiées. Plus généralement, il est souvent possible de générer une forme simplifiée d'une contrainte d'intégrité qu'il suffit de vérifier sur la base après mise à jour pour garantir la satisfaction de la contrainte.

Cherchons donc à exprimer une contrainte d'intégrité par rapport aux modifications apportées à la base afin de la simplifier par prise en compte de la véracité de la contrainte avant modifications. Toute modification d'une relation R est soit une insertion, soit une suppression,

soit une mise à jour pouvant être modélisée par une suppression suivie d'une insertion du même tuple modifié. Considérons une transaction `t` modifiant une relation `R`. Notons R^+ les tuples insérés et R^- les tuples supprimés dans `R`. La transaction `t` fait passer `R` de l'état `R` à l'état `Rt` comme suit : $Rt := (R - R^-) \cup R^+$. Considérons une contrainte d'intégrité `I` portant sur la relation `R`. Avant mise à jour, la contrainte est vraie sur `R`, ce que nous noterons $R \models I$ (`R` satisfait `I`). Après mise à jour, il faut vérifier que `Rt` satisfait `I`, soit $((R - R^-) \cup R^+) \models I$. Dans certains cas de contrainte et de transaction comportant un seul type de mise à jour (Insertion, Suppression ou Mise à jour), cette dernière forme peut être simplifiée par l'introduction de **tests différentiels** [Simon87].

> **Notion VIII.5 : Test différentiel (*Differential test*)**
>
> Contrainte d'intégrité simplifiée à vérifier après une opération sur `R` afin de garantir la satisfaction de la contrainte après application de l'opération.

Par exemple, considérons une contrainte de domaine telle que `DEGRE > 10` et une transaction d'insertion dans la table `VINS`. Il est clair que seuls les nouveaux vins insérés doivent être vérifiés. Dans ce cas $R^- = \varnothing$. On en déduit que $((R - R^-) \cup R^+) \models I$ est équivalent à $(R \cup R^+) \models I$. La contrainte devant être vérifiée pour chaque tuple, sa vérification commute avec l'union, et il suffit donc vérifier que $R^+ \models I$. Ceci est d'ailleurs vrai lors d'une insertion pour toute contrainte vérifiée par chaque tuple.

De manière générale, l'établissement de tests différentiels est possibles pour les différentes contraintes du modèle relationnel étudiées ci-dessus dans la section 2. Le tableau de la figure VIII.6 donne quelques tests différentiels à évaluer pour les opérations d'insertion, de suppression et de différence.

Opération → **Type de contrainte ↓**	**Insertion**	**Suppression**	**Mise à jour**
Clé primaire K de R	Les clés de R^+ sont uniques et ne figurent pas dans R.	Pas de vérification	Les clés de R^+ sont uniques et ne figurent pas dans R-R^-.
Clé étrangère **A de R Ref K de S**	Les tuples de R^+ référencent un tuple de S.	R : Pas de vérification S : Les clés K de S^- ne figurent pas dans A de R.	Les tuples de R^+ référencent un tuple de S.
Domaine A de R	Domaine A sur R^+	Pas de vérification	Domaine A sur R^+
Non nullité	Non-nullité sur R^+	Pas de vérification	Non-nullité sur R^+
Dépendance fonctionnelle A->B	Pour tout tuple t de R^+, s'il existe u de R tel que t.A = u.A alors t.B = u.B	Pas de vérification	Pas de forme simplifiée
Contrainte temporelle sur attribut	Pas de vérification	Pas de vérification	Vérifier les tuples de R^+ par rapport à ceux de R^-

Figure VIII.6 — Quelques tests différentiels de contraintes typiques

4. CONTROLE DES CONTRAINTES D'INTEGRITE

Pour garder la base cohérente, le SGBD doit vérifier que les contraintes d'intégrité sont satisfaites à chaque fin de transaction. En pratique, cette opération s'accomplit à chaque mise à jour en SQL. Cette approche oblige à ordonner les mises à jour en cas de contraintes référentielles : il faut par exemple insérer dans la relation maître avant d'insérer dans la relation dépendante les tuples liés. Le problème essentiel étant les performances, on pousse d'ailleurs la vérification avant la mise à jour, ce qui peut s'effectuer par des tests appropriés, comme nous allons le voir.

4.1 Méthode de détection

Une **méthode de détection** consiste simplement à évaluer la contrainte d'intégrité, éventuellement simplifiée, sur la base après exécution de la transaction. Pour améliorer les performances, la détection recherche en général les tuples ne satisfaisant pas la contrainte.

> **Notion VIII.6 : Méthode de détection (*Detection method*)**
>
> Méthode consistant à retrouver les tuples ne satisfaisant pas une contrainte dans la base après une mise à jour, et à rejeter la mise à jour s'il en existe.

Ainsi, soit I une contrainte d'intégrité mettant en jeu des relations R1, R2, ...Rn dans une base de données. Une méthode de détection naïve lors d'une mise à jour consiste à exécuter la requête :

```
SELECT COUNT(*)
FROM R1, R2, ...RN
WHERE NOT (I)
```

et à défaire la mise à jour si le résultat n'est pas 0.

En utilisant les stratégies de simplification vues ci-dessus, il est possible d'améliorer la méthode en remplaçant la requête par une requête plus élaborée mettant en jeu les relations différentielles. Plus généralement, il est possible de remplacer la requête vérifiant qu'aucun tuple ne viole la contrainte par un **post-test** pour une opération de modification donnée (`INSERT`, `DELETE` ou `UPDATE`).

> **Notion VIII.7 : Post-test (*Posttest*)**
>
> Test appliqué à la base après mise à jour permettant de déterminer si une contrainte d'intégrité I est violée.

Un post-test différentiel prendra en compte le fait que la contrainte était vérifiée avant la modification. Il portera sur les relations différentielles R^- et R^+. Par exemple, pour une contrainte de domaine et une insertion, un post-test possible consistera à vérifier simplement que les tuples insérés satisfont la condition :

```
(SELECT COUNT(*)
FROM R+
WHERE NOT (I)) = 0.
```

4.2 Méthode de prévention

Une méthode efficace souvent appliquée consiste à empêcher les modifications invalides. Une telle méthode est appelée **méthode de prévention**.

> **Notion VIII.8 : Méthode de prévention (*Prevention Method*)**
>
> Méthode consistant à empêcher les modifications de la base qui violeraient une quelconque contrainte d'intégrité.

Pour cela, un test avant mise à jour garantissant l'intégrité de la base si elle est mise à jour est appliqué. Un tel test est appelé un **pré-test**.

> **Notion VIII.9 : Pré-test (*Pretest*)**
>
> Test appliqué à la base avant une mise à jour permettant de garantir la non-violation d'une contrainte d'intégrité par la mise à jour.

Un pré-test peut éventuellement modifier la mise à jour en la restreignant aux tuples conservant l'intégrité de la base. La détermination d'un pré-test peut être effectuée en transformant la contrainte d'intégrité en lui appliquant l'inverse de la mise à jour. Plus précisément, soit u une mise à jour sur une relation R et I(R) une contrainte d'intégrité portant sur R. La contrainte d'intégrité modifiée I(u(R)) est l'image de I obtenue en remplaçant R par l'effet de u sur R. Par exemple, soient la contrainte d'intégrité (Pour tout `VINS : DEGRE < 15`) et la mise à jour u :

```
UPDATE VINS
SET DEGRE = DEGRE + 1.
```

La contrainte d'intégrité modifiée est (Pour tout `VINS : DEGRE+1 < 15`), puisque l'effet de la mise à jour est de remplacer `DEGRE` par `DEGRE+1`.

A partir d'une contrainte d'intégrité modifiée `I(u(R))`, il est possible de générer un pré-test en vérifiant simplement que la requête suivante a une réponse égale à 0 :

```
SELECT COUNT(*)
FROM R
WHERE NOT (I(U(R)).
```

Dans le cas des vins de degré inférieur à 15, on obtient :

```
SELECT COUNT(*)
FROM VINS
WHERE (DEGRE + 1) ≥ 15
```

Ceci permet donc la mise à jour si aucun vin n'a un degré supérieur ou égal à 14. En effet, dans ce cas aucun vin ne pourra avoir un degré supérieur à 15 après mise à jour.

Une méthode de prévention plus connue est la modification de requêtes [Stonebraker75] appliquée dans INGRES. Elle améliore la méthode précédente en intégrant le pré-test à la requête de mise à jour, ce qui permet de restreindre cette mise à jour aux seuls tuples respectant la contrainte d'intégrité après mise à jour. Bien que définie en QUEL, la méthode est transposable en SQL. Soit la mise à jour générique :

```
UPDATE R
SET R = U(R)
WHERE Q.
```

Soit donc I(R) une contrainte d'intégrité sur R et I(u(R)) la contrainte modifiée par la mise à jour inverse, comme vu ci-dessus. La mise à jour est alors transformée en :

```
UPDATE R
SET R = U(R)
WHERE Q AND I(U(R)).
```

Par exemple, dans le cas des vins de degré inférieur à 15, on exécutera la mise à jour modifiée :

```
UPDATE VINS
SET DEGRE = DEGRE+1
WHERE DEGRE < 14,
```

ce qui fait que seuls les vins de degré inférieur à 14 seront incrémentés. La contrainte ne sera pas violée, mais la sémantique de la mise à jour sera quelque peu changée ....

L'optimisation des pré-tests peut être plus poussée et prendre en compte la sémantique des mises à jour. Par exemple, si la mise à jour décroît le degré d'un vins, il n'est pas nécessaire d'ajouter un pré-test pour vérifier que le degré ne dépassera pas 15 ! Plus généralement, si la mise à jour est intégrée à un programme, par exemple en C/SQL, il est possible de bénéficier de la sémantique du programme pour élaborer un pré-test. Une technique pour élaborer des pré-tests en PASCAL/SQL a été proposée dans [Gardarin79]. L'idée est d'utiliser les axiomes de Hoare définissant la sémantique de PASCAL pour pousser les contraintes d'intégrité écrites en logique du premier ordre depuis la fin d'une transaction vers le début, en laissant ainsi au début de chaque mise à jour les pré-tests nécessaires.

Dans le cas de contrainte avec agrégats, les pré-tests constituent une des rares méthodes efficaces de contrôle. L'idée simple développée dans [Bernstein80] est de gérer dans la méta-base des agrégats redondants. Par exemple, si la moyenne des salaires dans une relation `EMPLOYES` doit rester inférieure à 20.000 F, on gérera cette moyenne (notée `MOYENNE`) ainsi que le nombre d'employés (noté `COMPTE`) dans la méta-base. Un pré-test simple lors de l'insertion d'un nouvel employé consistera alors à vérifier que `(MOYENNE*COMPTE+NOUVEAU SALAIRE)/(COMPTE+1) < 2000`. De même, pour une contrainte spécifiant que toute valeur d'une colonne A doit rester inférieure à toute valeur d'une colonne B, on pourra garder le minimum de B dans la méta-base. Lors d'une mise à jour de A, un pré-test efficace consistera simplement à vérifier que la nouvelle valeur de A reste inférieure au minimum de B.

Bien que les mises à jour ne soient pas effectuées lors de l'évaluation des pré-tests, une méthode intermédiaire a été appliquée dans le système SABRE à l'INRIA [Simon87], basée sur des pré-tests différentiels. Elle comporte deux étapes :

1. Préparer la mise à jour en construisant les relations R^+ et R^-, comme vu ci-dessus.
2. Pour chaque contrainte menacée, appliquer un pré-test différentiel pour contrôler la validité de la mise à jour.

C'est seulement à la fin de la transaction que R^- et R^+ sont appliquées à la relation R.

Pour chaque contrainte d'intégrité et chaque type de mise à jour, un pré-test différentiel était élaboré. Pour les cas usuels, ces pré-tests correspondent à peu près à ceux du tableau de la figure VIII.6. Pour les cas plus complexe, une méthode systématique de différentiation d'expression logique était appliquée [Simon87].

4.3 Contrôles au vol des contraintes simples

La plupart des systèmes implémentent une version des contraintes possibles en SQL2 réduite à l'unicité de clé, aux contraintes référentielles, et aux contraintes de domaines de type liste ou plage de valeurs, ou comparaison avec une autre colonne de la même table (`CHECK <condition>`, où la condition peut être toute condition SQL sans variable). Ces contraintes sont relativement simples à vérifier, bien que les actions possibles en cas de violation d'une contrainte référentielle (`SET NULL`, `SET DEFAULT`, ou `CASCADE`) impliquent une complexité que nous verrons ci-dessous. Elles peuvent pour la plupart être vérifiées au vol, c'est-à-dire lors de la mise à jour du tuple, ou plutôt juste avant.

La méthode de contrôle généralement employée consiste à effectuer une prévention au vol en employant un pré-test à chaque modification de tuple. Cette méthode est efficace car elle réduit au maximum les entrée-sorties nécessaires. Nous examinons ci-dessous les pré-tests mis en œuvre dans chacun des cas.

4.3.1 Unicité de clé

Tout SGBD gère en général un index sur les clés primaires. Un pré-test simple consiste à vérifier que la nouvelle clé ne figure pas dans l'index. Ce pré-test est effectué lors de l'insertion d'un tuple ou de la modification d'une clé dans la table, en général d'ailleurs juste avant mise à jour de l'index.

4.3.2 Contrainte référentielle

Du fait que deux tables, la table maître et la table dépendante, sont mises en jeu par une contrainte référentielle, quatre types de modifications nécessitent des vérifications, comme vu ci-dessus :

1. **Insertion dans la table dépendante.** La colonne référencée dans la table maître étant une clé primaire, il existe un index. Un pré-test simple consiste donc à vérifier l'existence dans cet index de la valeur de la colonne référençante à insérer.

2. **Mise à jour de la colonne référençante dans la table dépendante.** Le pré-test est identique au précédent pour la nouvelle valeur de la colonne.

3. **Suppression dans la table maître**. Le pré-test consiste à vérifier qu'il n'existe pas de tuple contenant la valeur de clé à supprimer dans la colonne référençante. Si le pré-test est faux, une complexité importante surgit du fait de la multitude des actions prévues

dans ce cas en SQL2. Il faut en effet soit rejeter la mise à jour, soit modifier voire supprimer les tuples de la table dépendante correspondant à cette valeur de clé. Ceci peut nécessiter d'autres contrôles d'intégrité, source de complexité examinée plus loin.

4. **Modification de clé dans la table maître**. Le pré-test est identique au précédent pour la valeur de clé avant modification.

Finalement, sauf pour les suppressions de clé dans la table maître, les vérifications sont simples. Elles peuvent devenir très complexes en cas de suppression en cascade le long de plusieurs contraintes référentielles.

4.3.3 Contrainte de domaine

Pour les contraintes de type `CHECK <condition>` avec une condition simple, il suffit de vérifier avant d'appliquer la mise à jour que la valeur qui serait donnée à l'attribut après mise à jour vérifie la condition. Ce pré-test est simple et peut être effectué au vol.

4.4 Interaction entre contraintes

Un problème difficile est celui posé par les interactions possibles entre contraintes. Il serait souhaitable que quel que soit l'ordre d'évaluation des contraintes ou de mise à jour de colonnes, une mise à jour donne le même effet. Ceci n'est pas simple à réaliser, notamment en cas de cascade de contraintes référentielles. Les interactions avec les mises à jour au travers des vues avec option de contrôle (`WITH CHECK OPTION`) sont aussi une source d'interaction [Cochrane96]. Afin de garantir le déterminisme des mises à jour, une sémantique de type point fixe doit être appliquée. Elle consiste à appliquer une procédure de contrôle comportant les étapes suivantes :

1. Évaluer tous les pré-tests des modifications directement issues de l'ordre original.

2. Si l'un au moins n'est pas vérifié, accomplir toute les modifications à cascader (`SET NULL`, `SET DEFAULT` ou `DELETE`) et déclencher récursivement les contrôles induits par ces modifications.

3. Évaluer à nouveau tous les pré-tests des modifications directement issues de l'ordre original. Si tous sont vérifiés, exécuter la mise à jour, sinon rejeter la mise à jour et défaire toutes les modifications faites à tous les niveaux.

Cette procédure récursive permet de se prémunir contre les interactions entre contraintes référentielles, mais aussi avec les contraintes de non-nullité, de domaine, etc. Elle est certes un peu complexe, ce qui démontre finalement la difficulté de traiter efficacement les contraintes exprimables en SQL2.

5. SGBD ACTIFS ET DECLENCHEURS

Dans cette section, nous précisons ce qu'est un **SGBD actif**, puis ce qu'est un **déclencheur**. Enfin nous étudions une architecture type pour un SGBD actif.

5.1 Objectifs

La notion de **SGBD actif** s'oppose à celle de SGBD passif, qui subit sans réagir des opérations de modification et interrogation de données.

> **Notion VIII.10 : SGBD actif (*Active DBMS*)**
>
> SGBD capable de réagir à des événements afin de contrôler l'intégrité, gérer des redondances, autoriser ou interdire des accès, alerter des utilisateurs, et plus généralement gérer le comportement réactif des applications..

La notion de **base de données active** va permettre de déplacer une partie de la sémantique des applications au sein du SGBD. Dans sa forme la plus simple, une BD active répercute les effets de mises à jour sur certaines tables vers d'autres tables. Un SGBD actif peut donc réagir par exemple lors d'opérations illicites, mais aussi lors d'opérations licites, parfois à un instant donné, et cela sous certaines conditions. Comment réagit-il ? Il pourra déclencher une opération subséquente à un événement donné (par exemple, une mise à jour), interdire une opération en annulant la transaction qui l'a demandée, ou encore envoyer un message à l'application, voire sur Internet.

5.2 Types de règles

Les SGBD actifs ont intégré de manière procédurale les règles de production de l'intelligence artificielle. Une règle de production est une construction de la forme :

```
IF <Condition sur BD> THEN <Action sur BD>.
```

Une règle de production permet d'agir sur une base de données lorsque la condition de la règle est satisfaite : l'action est alors exécutée et change en général l'état de la base. Le système de contrôle choisit quelle règle appliquer, jusqu'à saturation d'une condition de terminaison, ou jusqu'au point de saturation où plus aucune règle ne peut modifier l'état de la base : on a alors atteint un point fixe.

Cette idée, qui est aussi à la source des bases de données déductives comme nous le verrons plus tard, a été reprise dans les SGBD relationnels en ajoutant aux règles un contrôle procédural : chaque règle sera appliquée suite à un **événement**. Les règles deviennent alors des **déclencheurs** ou **règles ECA** (Événement - Condition - Action).

> **Notion VIII.11 : Déclencheur (*Trigger*)**
>
> Règle dont l'évaluation de type procédural est déclenchée par un événement, généralement exprimée sous la forme d'un triplet Événement - Condition – Action : `WHEN` <Événement> `IF` <Condition sur BD> `THEN` <Action sur BD>.

Lorsque l'événement se produit, la condition est évaluée sur la base. Si elle est vraie, l'action est effectuée. Nous préciserons ultérieurement ce qu'est un événement, une condition et une action. Disons pour l'instant que l'événement est souvent une modification de la base, la condition un prédicat vrai ou faux, et l'action un programme de mise à jour. La condition est optionnelle ; sans condition, on obtient un déclencheur dégénéré événement-action (EA). Il est

a remarquer que, dans ce dernier cas, la condition peut toujours être testée dans le programme constituant l'action, mais celui-ci est alors déclenché plus souvent et inutilement.

Le modèle d'exécution des déclencheurs est très variable dans les SGBD, mais il a été proposé une définition standard pour SQL [Cochrane96]. Malheureusement,, les *triggers* apparaissent seulement au niveau de SQL3. Dans la suite, nous nous appuierons sur ce modèle, mais il faut savoir que les systèmes ne le respectent en général guère. Pour comprendre la sémantique des déclencheurs dans un SGBD actif, il faut distinguer la prise en compte des événements, la détermination des règles candidates à l'exécution, le choix d'une règle si plusieurs sont candidates, l'exécution d'une règle qui comporte l'évaluation de la condition puis l'exécution de l'action.

La façon dont les règles sont exécutées dans les SGBD actifs n'est pas standard. La sémantique d'une BD active est donc souvent difficile à percevoir. Pour cela, un SGBD actif se doit de répondre aux questions suivantes :

1. Quand prendre en compte un événement ? Ce peut être dès son apparition, ou seulement lorsqu'une règle n'est pas en cours d'exécution ; dans ce dernier cas, il ne sera pas possible d'interrompre l'exécution d'une règle pour en exécuter une plus prioritaire.

2. Quand exécuter les règles ? Ce peut dès l'apparition de l'événement, ou plus tard lors d'un retour au système par exemple.

3. Comment choisir une règle lorsque plusieurs sont candidates à l'exécution ? Des mécanismes de priorité simples (par exemple, un niveau de priorité de 1 à 10) peuvent être mis en œuvre.

4. Quelle est la force du lien condition-action ? Autrement dit, doit-on exécuter l'action dès que la condition a été évaluée ou peut-on attendre ?

5. Lorsqu'une règle est exécutée et produit des événements provoquant le déclenchement d'autres règles, doit-on se dérouter ou attendre la fin de l'exécution de toutes les règles actives ou seulement de celle en cours d'exécution ?

Toutes ces questions ne sont pas indépendantes. Les réponses apportées fixent la sémantique du SGBD actif et conduisent à des résultats différents. Dans la suite, nous utilisons la sémantique décrite dans [Cochrane96], proposée pour SQL3. Par exemple, ce modèle procède en profondeur d'abord : il interrompt la règle en cours d'exécution à chaque nouvel événement.

5.3 Composants d'un SGBD actif

Les SGBD actifs ne sont donc pas standard. Cependant, la figure VIII.7 présente une architecture typique pour un SGBD actif. Les composants essentiels sont les suivants :

1. L'**analyseur de règles** permet de saisir les règles en forme externe (souvent textuelle), de les analyser et de les ranger en format interne dans le dictionnaire de règles.

2. Le **moteur de règles** coordonne l'exécution des règles suite aux événements. Il reçoit les événements primitifs et composites et détermine les règles candidates à l'exécution. Une règle devient alors active. Parmi les règles actives, le moteur de règle choisi la plus prioritaire et lance sont exécution, qui comporte donc l'évaluation de la condition, puis l'exécution de l'action si la condition est satisfaite. Le moteur de règles gère aussi le contexte des règles actives, c'est-à-dire mémorise les variables qui doivent être maintenues et passées par exemple de la condition à l'action.

3. Le **moniteur d'événements** détecte les événements primitifs et composites. Il demande au moteur de règles l'exécution des règles déclenchées par ces événements.

4. Sur demande du moteur de règles, l'**évaluateur de conditions** évalue les conditions des règles actives, éventuellement en demandant l'exécution des recherches au SGBD. Il détermine si la condition est vraie et retourne cette information au moteur de règles.

5. L'**exécuteur d'actions** exécute les actions des règles actives dont les conditions ont été préalablement vérifiées. Il invoque pour ce faire le SGBD.

6. Le **dictionnaire de règles** est la partie de la méta-base du SGBD qui contient la définition des règles en format interne.

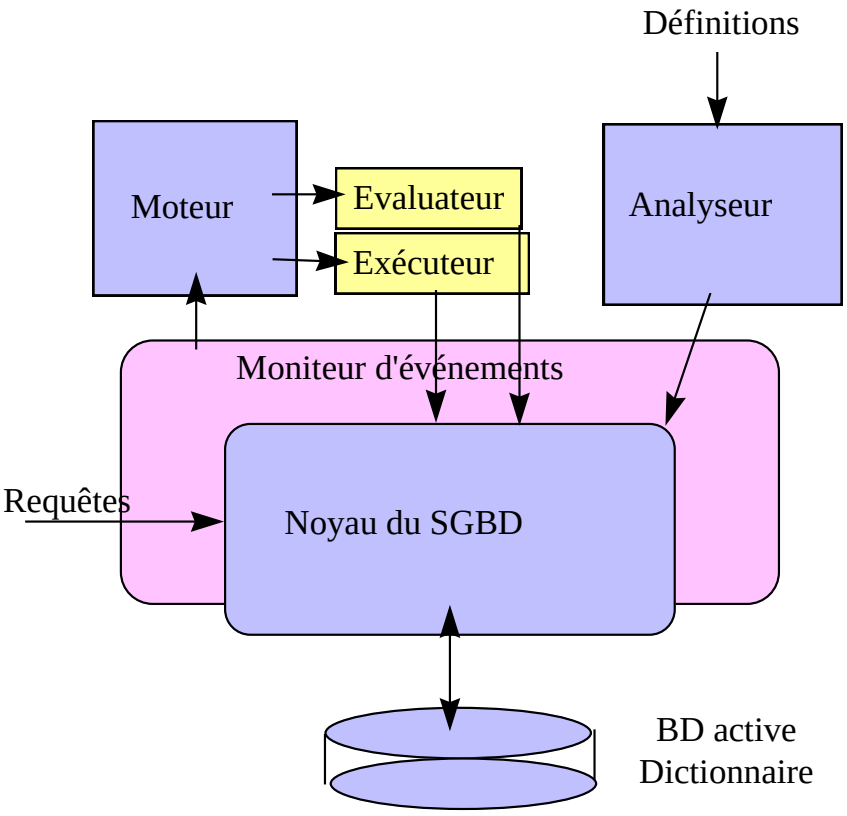

Figure VIII.7 — Architecture d'un SGBD actif

Deux approches sont possibles pour un SGBD actif [Llirbat97] : l'approche intégrée lie intimement les composants spécifiques du SGBD actif au SGBD, alors que l'approche sur-couche construit au-dessus d'un SGBD les composants nécessaires. L'approche intégrée est

souvent plus efficace, mais elle ne permet pas de réaliser des systèmes de déclencheurs portables, indépendants du SGBD natif (il en existe bien peu).

6. LA DEFINITION DES DECLENCHEURS

Dans cette section, nous définissons plus précisément les éléments constituant un déclencheur. Puis, nous donnons la syntaxe permettant de définir les déclencheurs en SQL3.

6.1 Les événements

> **Notion VIII.12 : Événement (*Event*)**
>
> Signal instantané apparaissant à un point spécifique dans le temps, externe ou interne, détecté par le SGBD.

Un **événement** correspond donc à l'apparition d'un point d'intérêt dans le temps. La spécification d'un événement nécessite la définition d'un type. Un type d'événement peut correspondre au début (`BEFORE`) ou à la fin (`AFTER`) d'une recherche (`SELECT`), d'une mise à jour (`UPDATE`, `DELETE`, `INSERT`) ou d'une transaction (`BEGIN`, `COMMIT`, `ABORT`), à l'écoulement d'un délai, au passage d'une horodate, etc. Un type permet de définir un ensemble potentiellement infini d'instances d'événements, simplement appelé événement. Une instance d'événement se produit donc à un instant donné. Á un événement sont souvent associés des paramètres : une valeur d'une variable, un objet, une colonne de table, etc. Le **contexte d'un événement** est une structure de données nommée contenant les données nécessaires à l'évaluation des règles déclenchées par cet événement, en particulier les paramètres de l'événement. Parmi les événements, on distingue les événements simples (ou primitifs) et les événements composés.

6.1.1 Événement simple

Dans un SGBD, les événements simples (encore appelés primitifs) particulièrement considérés sont l'appel d'une modification de données (`BEFORE UPDATE`, `BEFORE INSERT`, `BEFORE DELETE`), la terminaison d'une modification de données (`AFTER UPDATE`, `AFTER INSERT`, `AFTER DELETE`), l'appel d'une recherche de données (`BEFORE SELECT`), la fin d'une recherche de données (`AFTER SELECT`), le début, la validation, l'annulation d'une transaction (`BEGIN`, `COMMIT`, `ABORT`), un événement temporel absolu (`AT TIMES` <Heure>) ou relatif (`IN TIMES` <Delta>), ou tout simplement un événement utilisateur (avant ou après exécution de procédure : `BEFORE` <procedure> ou `AFTER` <procedure>). Une typologie simplifiée des différents événements primitifs apparaît figure VIII.8. D'autres sont détectables, par exemple les erreurs.

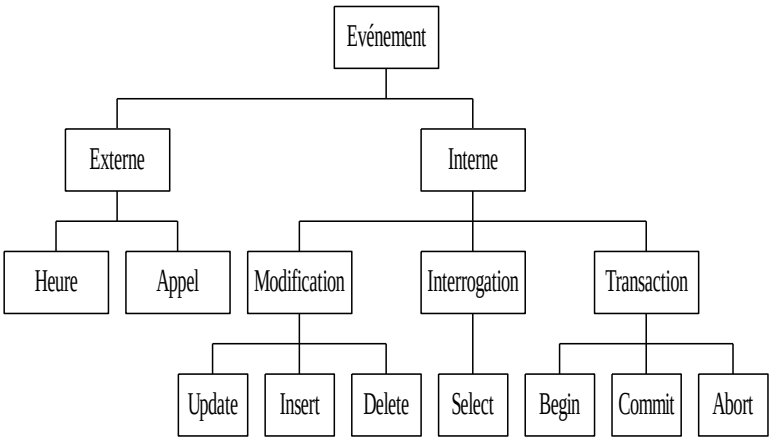

Figure VIII.8 — Typologie des événements primitifs

6.1.2 Événement composé

Un événement composé est une composition d'événements simples. Celle-ci est effectuée par des constructeurs spécifiques, tels le `OU` et le `ET` logique. On obtient ainsi des expressions logiques simples et parenthésables d'événements. (`A OU B`) se produit lorsque l'un ou l'autre des événements est vrai. Alors que les événements primitifs ne sont pas mémorisé après leur production, le problème devient plus difficile pour les événements composés. Par exemple, si A a été signalé, (`A OU B`) aussi. Que doit-on faire si `B` se produit ? Faut-il signaler à nouveau (`A OU B`) ? Le problème n'est pas simple et la solution retenue peut changer complètement la sémantique d'une application.

Au-delà des constructeurs logiques `ET` et `OU`, il est possible de faire intervenir des constructeurs temporels tels que la séquence (`A PUIS B`) qui est signalée si `B` suit `A`, la répétition (`N FOIS A`) déclenchée si `A` se produit `N` fois, ou la production ou la non production pendant un intervalle de temps (`A IN` <intervalle>, `A NOT IN` <Intervalle>). Tout ceci complique la gestion des événements, mais facilite la prise en compte de la logique temporelle des applications. On pourra par exemple écrire des événements du style ((`1H AFTER UPDATE`) `OR` (`AFTER INSERT THEN BEFORE DELETE`) `AND` (`5 TIMES BEFORE INSERT`) `AND` (`COMMIT NOT IN [10H, 12H]`)), dont la signification est un peu complexe. Heureusement, peu de SGBD excepté quelques prototypes de recherche utilisent des événements composés [Gatziu92].

6.2 La condition

La **condition** est une expression logique de prédicats portant sur les variables de contexte d'exécution de la règle ou/et sur la base de données. Peu de systèmes permettent des requêtes complètes au niveau de la condition.

Notion VIII.13 : Condition de déclencheur (*Trigger condition*)

Qualification (aussi appelée condition de recherche, ou *search condition*) portant sur les variables de contexte et/ou la base et donnant en résultat une valeur booléenne.

Si l'on souhaite tester l'existence de tuples, on utilisera les prédicats EXIST et NOT EXIST, ou le compte de tuples résultats (COUNT). La condition est en général optionnelle au niveau des déclencheurs : sans condition, on a simplement une règle événement-action.

6.3 L'action

L'**action** est une procédure exécutée lorsque la condition est vérifiée.

Notion VIII.14 : Action de déclencheur (*Trigger action*)

Procédure exécutée lorsque la règle est déclenchée.

Ce peut être une seule requête ou une séquence de requêtes SQL, une procédure stockée agissant sur la base écrite en L4G ou dans un L3G intégrant SQL (C/SQL par exemple), ou enfin une opération sur transaction (`ABORT`, `COMMIT`). L'action peut utiliser les paramètres du contexte. Elle peut être exécutée une seule fois suite à l'événement ou pour chaque ligne satisfaisant la condition, comme nous le verrons plus en détail ci-dessous.

6.4 Expression en SQL3

Avec SQL3, tous les objets (règles, contraintes, etc.) sont nommés, donc en particulier les déclencheurs. Les événement sont simples et déclenchés par les requêtes de modification INSERT, UPDATE, et DELETE. L'événement peut être déclenché soit avant exécution de la modification (BEFORE), soit après (AFTER). Deux granularités sont possibles afin de contrôler l'évaluation de la condition et l'exécution éventuelle de l'action : les granularités ligne ou requête. Dans le premier cas, l'action est exécutée pour chaque ligne modifiée satisfaisant la condition, dans le second elle est exécutée une seule fois pour l'événement. L'action peut référencer les valeurs avant et après mise à jour (clause `REFERENCING ... AS ...`). Elle peut même remplacer complètement la modification dont l'appel a provoqué l'événement (clause `INSTEAD OF`).

6.4.1 Syntaxe

La figure VIII.9 donne la syntaxe résumée de la requête de création de déclencheur en SQL3. La sémantique est explicitée dans le paragraphe qui suit.

CREATE TRIGGER <nom>

// événement (avec paramètres)

{BEFORE | AFTER | INSTEAD OF}

{INSERT | DELETE | UPDATE [OF <liste de colonnes>]**}**

ON <table> [**ORDER** <valeur>]

[REFERENCING{NEW|OLD|NEW_TABLE|OLD_TABLE} AS <nom>]...

// condition

(WHEN (<condition de recherche SQL>)

// action

<Procédure SQL3>

// granularité

[FOR EACH {ROW | STATEMENT}])

Figure VIII.9 — Syntaxe de la commande de création de déclencheur

6.4.2 Sémantique

L'événement de déclenchement est une mise à jour (`UPDATE`), une insertion (`INSERT`) ou une suppression (`DELETE`) dans une table. En cas de mise à jour, un paramètre optionnel permet de préciser une liste de colonnes afin de réduire la portée de l'événement. L'événement n'est pas instantané et a un effet. Afin de préciser l'instant d'activation de la règle, trois options sont disponibles : avant (`BEFORE`), après (`AFTER`) la modification, ou à la place de (`INSTEAD OF`). On revient donc là à des événements instantanés. L'option "avant" est très utile pour contrôler les paramètres et les données d'entrées avant une modification. L'option "après" permet plutôt de déclencher des traitements applicatifs consécutifs à une mise à jour. L'option "à la place de" permet de remplacer un ordre par un autre, par exemple pour des raisons de sécurité.

Un ordre optionnel est spécifié pour préciser les priorités lorsque plusieurs déclencheurs sont définis sur une même table. La déclaration `REFERENCING NEW AS <nom>` définit une variable de nom <nom> contenant la valeur de la dernière ligne modifiée dans la base par l'événement. De même, `REFERENCING OLD AS <nom>` définit une variable de nom <nom> contenant la valeur de la même ligne avant l'événement. Des tables différentielles contenant les valeurs avant et après l'événement sont définies par les options `NEW_TABLE` et `OLD_TABLE`. Les déclencheurs sur `INSERT` ne peuvent que déclarer les nouvelles valeurs. Les déclencheurs sur `DELETE` ne peuvent que déclarer les anciennes.

Chaque déclencheur spécifie en option une action gardée par une condition. La condition est un critère de recherche SQL pouvant porter sur les variables de contexte ou sur la base via des requêtes imbriquées. L'action est une procédure contenant une séquence d'ordre SQL. Condition et action peuvent donc interroger la base de données et le contexte du déclencheur (par exemple, manipuler les variables et les tables différentielles). Condition et action lisent ou modifient les valeurs de la base avant mise à jour dans un déclencheur avant (`BEFORE`), après mise à jour dans un déclencheur après (`AFTER`).

La granularité d'un déclencheur est soit ligne (`FOR EACH ROW`), soit requête (`FOR EACH STATEMENT`). Dans le premier cas, il est exécuté pour chaque ligne modifiée (0 fois si aucune ligne n'est modifiée), alors que dans le second, il l'est une seule fois pour la requête.

6.5 Quelques exemples

6.5.1 Contrôle d'intégrité

Cet exemple porte sur la table des `VINS`. Le premier déclencheur de la figure VIII.10 contrôle l'existence d'un vin lors de l'ajout d'un abus. Le second cascade la suppression d'un vin, c'est-à-dire supprime les abus correspondant au vin supprimé.

```
// Ajout d'un abus
CREATE TRIGGER InsertAbus
BEFORE INSERT ON ABUS
REFERENCING NEW AS N
(WHEN (NOT EXIST (SELECT * FROM Vins WHERE NV=N.NV ))
ABORT TRANSACTION
FOR EACH ROW ) ;

// Suppression d'un vin
CREATE TRIGGER DeleteVins
BEFORE DELETE ON VINS
REFERENCING OLD AS O
(DELETE FROM ABUS WHERE NV = O.NV
FOR EACH ROW ) ;
```

Figure VIII.10 — Contrôle d'intégrité référentielle

Le déclencheur de la figure VIII.11 est associé à la table des employés, de schéma :

EMPLOYE (ID Int, Nom Varchar, Salaire Float).

Il effectue un contrôle d'intégrité temporelle : le salaire ne peut que croître.

```
CREATE TRIGGER SalaireCroissant
BEFORE UPDATE OF Salaire ON EMPLOYE
REFERENCING OLD AS O, NEW AS N
(WHEN O.Salaire > N.Salaire
SIGNAL.SQLState '7005' ("Les salaires ne peuvent décroître')
FOR EACH ROW);
```

Figure VIII.11 — Contrôle d'intégrité temporelle

6.5.2 Mise à jour automatique de colonnes

Le déclencheur de la figure VIII.12 est déclenché lors de la mise à jour de la table

PRODUITS (NP Int, NF Int, Coût Real, Auteur String, DateMaj Date).

Il positionne les attributs `Auteur` et `DateMaj` aux valeurs courantes de la transaction pour lequel il est exécuté.

```
CREATE TRIGGER SetAuteurDate
BEFORE UPDATE ON PRODUITS
REFERENCING NEW_TABLE AS N
(UPDATE N
SET N.Auteur = USER, N.DateMaj = CURRENT DATE);
```

Figure VIII.12 — Complément de mise à jour

Le déclencheur de la figure VIII.13 est aussi associé à la table `PRODUITS`. Il crée automatiquement la clé lors de l'insertion d'un tuple. Attention : si vous insérez plusieurs tuples ensemble, vous obtenez plusieurs fois la même clé ! Il faut alors écrire le programme en L4G et travailler sur chaque ligne.

```
CREATE TRIGGER SetCléVins
BEFORE INSERT ON PRODUITS
REFERENCING NEW_TABLE AS N
(UPDATE N
        SET N.NP = SELECT COUNT(*) +1 FROM PRODUITS );
```

Figure VIII.13 — Génération automatique de clé

6.5.3 Gestion de données agrégatives

Le déclencheur de la figure VIII.14 est associé à la table `EMPLOYE` vue ci-dessus permettant de gérer le salaire des employés. Il répercute les mises à jour du salaire sur la table de cumul des augmentations de schéma : `CUMUL (ID int, Augmentation float).`

```
CREATE TRIGGER CumulSal
AFTER UPDATE OF salaire ON EMPLOYE
REFERENCING OLD AS a, NEW AS n
(UPDATE CUMUL SET Augmentation = Augmentation +
        n.salaire - a.salaire WHERE ID = a.ID
FOR EACH ROW ) ;
```

Figure VIII.14 — Gestion de données agrégative

7. EXECUTION DES DECLENCHEURS

Nous examinons ici brièvement les mécanismes nécessaires au sein d'un moteur de règles afin de cordonner l'exécution des règles, en plus bien sûr de la gestion du contexte des événements. Ces mécanismes sont difficiles du fait de la sémantique plutôt complexe des déclencheurs, basée sur une application jusqu'à saturation (point fixe) de règles de mise à jour. Nous utilisons ici la sémantique de référence de SQL3 [Cochrane96].

7.1 Procédure générale

La procédure générale d'exécution d'une modification (`UPDATE`, `INSERT` ou `DELETE`) doit prendre en compte les déclencheurs et la vérification des contraintes d'intégrité. Les deux peuvent interagir. La procédure doit distinguer les actions à effectuer avant l'exécution de l'opération de modification ayant déclenché la règle de celle exécutée après. Il faut exécuter avant les déclencheurs avec option `BEFORE` et après ceux avec option `AFTER`. Les contrôles d'intégrité, généralement accomplis par des pré-tests, seront aussi effectués avant. Dans tous les cas, la modification doit être préparée afin de rendre accessibles les valeurs avant et après mise à jour. La figure VIII.15 résume l'enchaînement des contrôles lors de l'exécution d'une mise à jour. La figure VIII.16 montre comment est exécutée une règle, selon l'option. Cette procédure est au cœur du moteur de règles.

```
// Exécution d'une modification d'une relation R
Modification(R) {
 Préparer les mises à jour de R dans R+ et R- ;
// Exécuter les déclencheur avant mise à jour (BEFORE)
 For each "déclencheur BEFORE d" do { Exécuter(d.Règle) } ;

// Effectuer les contrôles d'intégrité, puis la mise à jour
 If (Not Integre) then ABORT TRANSACTION ;
// effectuer la mise à jour
 R = (R - R-) ∪ R+ ;

// Exécuter les déclencheurs après mise à jour (AFTER)
 For each "déclencheur AFTER" do { Exécuter(d.Règle) } ;
} ;
```

Figure VIII.15 — Exécution d'une modification

```
// Exécution d'une règle WHEN Condition Action FOR EACH <Option>
Exécuter(Condition, Action, Option) {
// Appliquer à chaque tuple si option ligne
 if Option = "FOR EACH ROW" then
        { For each t de ΔR = R+ ∪ R- do {
                if (Condition(t) = True) then Exécuter (Action(t)) ; }
 if Option = "FOR EACH STATEMENT" then {
                if (Condition(ΔR) = True) then Exécuter (Action(ΔR) ; }
} ;
```

Figure VIII.16 — Exécution d'une règle

7.2 Priorités et imbrications

En fait, suite à un événement, plusieurs règles peuvent être candidates à l'exécution. Dans l'algorithme de la figure VIII.15, nous avons choisi d'exécuter les déclencheurs avant puis après dans un ordre quelconque (boucle `FOR EACH`). Malheureusement, l'ordre peut influer sur le résultat. Par exemple, un déclencheur peut défaire une mise à jour exécutée par un autre. Ceci ne se produirait pas si on exécutait les déclencheurs dans un ordre différent ! Il y a donc là un problème de sémantique.

Avec SQL3, il est prévu une priorité affectée soit par l'utilisateur (clause `ORDER`), soit selon l'ordre de définition des déclencheurs. Cette priorité doit donc être respectée : les boucles `For each` de la figure VIII.15 doivent choisir les déclencheurs en respectant les priorités.

Plus complexe, l'exécution d'une règle peut entraîner des mises à jour, qui peuvent à leur tour impliquer l'exécution de déclencheurs sur la même relation ou sur une autre. Dans ce cas, les contextes d'exécution des modifications sont empilés, comme lors de l'appel de sous-programmes. A chaque fin d'exécution d'une modification, un dépilage est nécessaire afin de revenir à la modification précédente. Deux problèmes se posent cependant, que nous examinons successivement.

Le premier problème tient aux mises à jour cumulatives. Si la modification porte sur la même relation R, de nouveaux tuples sont insérés dans R^+ et R^- par la modification imbriquée. On peut soit empiler les versions de R^+ et R^-, soit ne s'intéresser qu'à l'**effet net** de l'ensemble des mises à jour. La sémantique des déclencheurs sera alors différente.

> **Notion VIII.15 : Effet net (*Net effect*)**
>
> Impact cumulé d'un ensemble de mise à jour en termes de relations différentielles R^+ (tuples à insérer) et R^- (tuples à supprimer).

L'effet net est une notion importante aussi pour les transactions qui sont composées de plusieurs mises à jour. Par exemple, une transaction qui insère puis supprime un tuple a un effet net vide.

Le second problème concerne les risques de bouclage. En effet, il peut exister des boucles de déclencheurs qui ne s'arrêtent pas. En effet, une mise à jour peut impliquer une nouvelle mise à jour, et ainsi de suite. Le nombre de relations étant fini, la boucle reviendra forcément plusieurs fois sur une même relation. C'est un critère simple détectable par l'analyseur de déclencheurs. Cependant, ceci limite fortement l'usage des déclencheurs. Par exemple, on ne peut écrite un déclencheur qui en cas de baisse de votre salaire déclenche une autre règle pour vous donner une prime compensatrice ! Une solution plus sophistiquée consiste à détecter les boucles à l'exécution en limitant le nombre de déclencheurs activés lors d'une mise à jour. Une approche plus sophistiquée encore consiste à prendre en compte l'effet net d'un ensemble de déclencheurs successifs et à vérifier qu'il change de manière continue.

7.3 Couplage à la gestion de transactions

Jusque-là, nous avons supposé que les déclencheurs étaient exécutés lors de chaque modification, pour le compte de la transaction qui effectue la modification. En fait, différents types de liaisons avec les transactions sont possibles, notamment pour les déclencheurs après, pour les conditions et/ou les actions. On distingue :

1. Le **mode d'exécution** qui précise quand exécuter le déclencheur. Ce peut être immédiat, c'est-à-dire dès que l'événement se produit (`IMMEDIAT`), ou différé en fin de transaction (`DEFERRED`).

2. Le **mode de couplage** qui précise si le déclencheur doit être exécuté dans la même transaction (`COUPLED`) ou dans une transaction indépendante (`DECOUPLED`).

3. Le **mode de synchronisation** qui dans le cas d'une transaction indépendante précise si elle est synchrone (après) ou asynchrone (en parallèle) (`SYNCHRONOUS`) avec la transaction initiatrice.

Tout ceci complique la sémantique des déclencheurs. Dans le cas de du mode différé, seul l'effet net de la transaction sera pris en compte par les déclencheurs. Le cas de transaction découplée asynchrone devient complexe puisque les deux transactions peuvent interagir via les mises à jour. L'une, pourquoi pas la transaction initiatrice, peut être reprise alors que l'autre ne l'est pas. Ainsi, le déclencheur peut être exécuté sans que la transaction déclenchante ne le soit ! Les SGBD actuels évitent ce mode de synchronisation et d'ailleurs plus radicalement le mode découplé.

8. CONCLUSION

Un système de règles supporte au minimum des événements simples de mise à jour. C'est le cas de SQL3 et des SGBD relationnels du commerce. Il est aussi possible de considérer les événements de recherche (`SELECT`) comme dans Illustra [Stonebraker90]. Seuls des systèmes de recherche gèrent des événements composés, avec OU, ET, séquences et intervalles de temps. On citera par exemple HIPAC [Dayal88] et SAMOS [Gatziu92].

Le standard SQL3 supporte des conditions et actions exécutées avant le traitement naturel de l'opération initiatrice, ou après, ou à sa place. Il permet aussi de différer l'exécution de certains déclencheurs en fin de transaction. Par contre, les déclencheurs sont exécutés pour le compte

de la transaction déclenchante. Ils ne peuvent être découplés. Le découplage pose beaucoup de problèmes de sémantique et a été partiellement exploré dans des projets de recherche comme SAMOS.

Les déclencheurs sont très utiles dans les SGBD. Ils correspondent cependant à une vision procédurale des règles. De ce fait, la sémantique est souvent obscure. La conception de bases de données actives efficaces reste un problème difficile.

9. BIBLIOGRAPHIE

[Agrawal91] Agrawal R., Cochrane R.J., Linsay B., « On Maintaining Priorities in a Production Rule System », *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, Morgan Kaufman Ed., pp. 479-487, Sept. 1991.

Cet article rapporte sur l'expérimentation du mécanisme de priorité entre règles implémenté à IBM dans le projet Starburst. Il montre l'intérêt du mécanisme.

[Bernstein80] Bernstein P., Blaustein B., Clarke E..M., « Fast Maintenance of semantic Integrity Assertions Using Redundant Aggregate Data », *Proc. 6th Int. Conf. on Very Large Data Bases*, Montreal, Canada, Morgan Kaufman Ed., Oct. 1991.

Les auteurs furent les premiers à proposer la maintenance d'agrégats redondants pour faciliter la vérification des contraintes d'intégrité lors des mises à jour. Depuis, ces techniques se sont répandues sous la forme de vues concrètes.

[Ceri90] Ceri S., Widom J., « Deriving Production Rules for Constraint Maintenance» , *Proc. 16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman Ed., pp. 566-577, Aug. 1990.

Les auteurs proposent des algorithmes pour générer des déclencheurs permettant la vérification automatique de règles d'intégrité lors des mises à jour. De tels algorithmes pourraient être intégrés à un compilateur de définitions de contraintes d'intégrité.

[Cochrane96] Cocherane R., Pirahesh H., Mattos N., Integrating triggers and Declarative Constraints in SQL Database Systems, *Proc. 16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman Ed., pp. 566-577, Aug. 1990.

L'article de référence pour la sémantique des déclencheurs en SQL3. Après un rappel des contraintes d'intégrité existant en SQL2, l'article montre comment on exécute les déclencheurs en absence de contraintes, puis les interactions entre ces deux mécanismes. Il définit ensuite une sémantique de point fixe pour les déclencheurs.

[Date81] Date C.J., « Referential Integrity» , *Proc. 7th Intl. Conf. on Very Large Data Bases*, Cannes, France, IEEE Ed., Sept. 1981.

L'article de base qui a introduit les contraintes référentielles. Celles-ci étaient intégrées au relationnel pour répondre aux attaques de perte de sémantique du modèle par rapport aux modèle type réseau ou entité-association.

[Dayal88] Dayal U., Blaunstein B., Buchmann A., Chakravavarthy S., Hsu M., Ladin R., McCarthy D., Rosenthal A., Sarin S., Carey M. Livny M., Jauhari J., « The HiPAC Project : Combining Active databases and Timing Constraints », *SIGMOD Record V°17, N°1*, Mars 1988.

Un des premiers projets à étudier un système de déclencheurs avec contraintes temporelles. Le système HiPAC fut un précurseur en matière de déclencheurs.

[Dittrich95] K.R., Gatziu S., Geppert A., « The Active Database Management System Manifesto», *Proc. 2nd Int. Workshop on Rules in Databas Systems*, Athens, Greece, Sept. 1995.

Ce manifesto se veut l'équivalent pour les bases de données actives du manifesto des bases de données objet. Il définit précisément les fonctionnalités que doit supporter un SGBD actif.

[Eswaran75] Eswaran K.P., Chamberlin D.D., « Functional Specifications of a Subsystem for Database Integrity», *Proc. 1st Intl. Conf. on Very Large Data Bases*, Framingham, Mass., pp. 48-67, Sept. 1975.

La description détaillée du premier sous-système d'intégrité réalisé. Ce travail a été effectué dans le cadre du System R à IBM.

[Eswaran76] Eswaran K.P., Chamberlin D.D., « Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System », *IBM Research report RJ 1820*, IBM Research Lab., San José, California, Août 76.

La description détaillée du premier sous-système de déclencheurs réalisé. Ce travail a été effectué dans le cadre du System R à IBM.

[Gardarin79] Gardarin G., Melkanoff M., « Proving Consistency of Database Transactions» , *Proc. 5th Int. Conf. on Very Large Data Bases*, Rio de Janeiro, Brésil, Sept. 1979.

Cet article propose une méthode pour générer automatiquement des pré-tests dans des programmes écrits en PASCAL/SQL. La méthode est basée sur une technique de preuve d'assertions en utilisant la sémantique de Hoare.

[Hammer78] Hammer M., Sarin S., « Efficient Monitoring of Database Assertions » , *Proc. ACM SIGMOD Int. Conf. On Management of Data*, 1978.

Cet article propose des techniques de simplification de contraintes d'intégrité basées sur l'étude logique de ces contraintes.

[Horowitz92] Horowith B., « A Runtime Excecution Model for referential Integrity Maintenance », *Proc. 8th Intl. Conf. on Data Engineering*, Tempe, Arizona, pp. 546-556, 1992.

Les auteurs décrivent un prototype réalisant efficacement la vérification des contraintes référentielles lors de l'exécution des mises à jour.

[Llirbat97] Llirbat F., Fabret F., Simon E., « Eliminating Costly Redundant Computations from SQL Trigger Executions » *Proc. ACM SIGMOD Int. Conf.* on Management of data, Tucson, Arizona, pp. 428-439, Juin 1997.

Cet article développe une nouvelle technique d'optimisation des déclencheurs en SQL. Cette technique, au coeur de la thèse de F. Llirbat, permet d'extraire les invariants dans les boucles et de les mémoriser afin d'éviter les recalculs. Le papier développe aussi un modèle pour décrire les programmes, les déclencheurs, et leurs interactions.

[Simon87] Simon E., Valduriez P., « Design and Analysis of a Relational Integrity Subsystem », *MCC Technical report Number DB-015-87*, Jan. 1987.

Ce rapport résume la thèse d'Eric Simon (Paris VI, 1987) et décrit le système de contrôle d'intégrité du SGBD SABRE réalisé à l'INRIA. Ce système est basé sur des post-tests différentiels.

[Stonebraker75] Stonebraker M., « Implementation of integrity Constraints and Views by Query Modification », *Proc. ACM SIGMOD*, San José, California, 1975.

Cet article présente le premier sous-système de gestion de vues et d'intégrité d'Ingres, basé sur la modification de questions. Celle-ci est utilisée afin de générer des pré-tests intégrés aux requêtes modifiées.

[Stonebraker90] M. Stonebraker, Jhingran A., Goh J., Potamianos S., « On Rules, Procedures, Caching, and Views in Database Systems» , *Proc. ACM SIGMOD Int. Conf. on Management of Data,* Atlantic City, New Jersey, pp. 281-290, Juin 1990.

Suite aux expériences d'Ingres et Postgres, les auteurs commentent les mécanismes de gestion de vues, caches, procédures et règles dans les SGBD. Ils proposent d'intégrer tous ces composants autour du mécanisme de contrôle de concurrence, en étendant les types de verrous possibles. Les idées originales développées furent implémentées au cœur de Postgres.

[Widom91] Widom J, Cochrane R., Lindsay B., « Implementing Set-oriented Production Rules as an Extension to Starburst », *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, Morgan Kaufman Ed., pp. 275-285, Sept. 1991.

Cet article décrit l'implémentation des règles réalisée dans Starburst au centre de recherche d'IBM en Californie. Ce prototype fut à la source du système de triggers aujourd'hui supporté par DB2.

[Widom96] Widom J., Ceri S., *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan-Kaufmann, San Fransisco, California, 1996.

Ce livre de synthèse sur les bases de données actives présente les concepts de base, différents prototypes et de multiples expériences réalisées sur le sujet.