

Capitolul 14. Tabele temporare și virtuale

În capitolul anterior am prezentat câteva exemple de tabele create și populate prin SELECT-uri mai mult sau mai puțin complicate. O bună parte dintre acestea ne-au folosit la stocarea temporară a unor rezultate de formă tabelară, rezultate ce ar putea constitui sursa unor rapoarte sau chiar a unor controale din formulare. Pentru rapoarte sau popularea controalelor de tip listă, combo, mult mai nimerită este folosirea tabelelor temporare și/sau virtuale. Acestea, spre deosebire de tabelele „obișnuite”, nu ocupă spațiu pe disc, decât pe parcursul unei sesiuni de lucru, sau chiar mai puțin, întrucât în schema bazei se stochează permanent doar structura lor, nu și conținutul.

14.1. Tabele temporare

Tabelele temporare se crează prin aceeași comandă - CREATE TABLE cu cele două variante majore - listă de attribute și/sau restricții sau consultări (SELECT). Există două tipuri de tabele temporare¹, *locale* și *globale*. Conținutul ambelor tipuri nu este vizibil decât utilizatorului curent (sesiunii curente, pentru că între sesiunile de lucru conținutul unei tabele temporare poate fi stocat doar prin copierea într-o tabelă obișnuită). Gulutzan și Pelzer le numesc *tabele-dependente-de-sesiune*.²

Tabelele temporare locale sunt cele mai „invizibile”, conținutul lor fiind disponibil doar înăuntrul aceluiași modul de cod executat în cadrul unei sesiuni de lucru. Doar cele globale pot fi „transmise” între modulele executate într-o aceeași sesiune de lucru, însă și aici trebuie avut grijă, pentru că tabelele temporare pot *păstra* conținutul lor la finalizarea unei tranzacții (ON COMMIT PRESERVE ROWS), dar îl și pot pierde automat (ON COMMIT DELETE ROWS). Cele două clauze funcționează și pentru tabelele virtuale locale.

Prima utilizare majoră a tabelelor temporare țintește simplificarea interogărilor. Ne referim, pentru început, la frustrarea pe care au trăit-o utilizatorii de PostgreSQL în paragraful 9.6 dedicat expresiilor tabelă (WITH... SELECT...), paragraf în care numai PostgreSQL-iștii au stat pe tușă. Să luăm problema atât de

¹ [Melton & Simon 2002], pp.100-102,

² [Gulutzan & Pelzer 1999], p.350

îndrăgită: Care este județul în care berea s-a vândut cel mai bine ? Folosind o tabelă temporară, și în acest dialect lucrurile pot fi simplificate sensibil:

```
CREATE LOCAL TEMPORARY TABLE judete_bere AS
  (SELECT Judet, SUM(Cantitate*PretUnit*(1+ProcTVA)) AS Vnz_Bere
   FROM judete j
        INNER JOIN coduri_postale cp ON j.Jud=cp.Jud
        INNER JOIN clienti c ON cp.CodPost=c.CodPost
        INNER JOIN facturi f ON c.CodCl=f.CodCl
        INNER JOIN liniifact lf ON f.NrFact= lf.NrFact
        INNER JOIN produse p ON lf.CodPr=p.CodPr
   WHERE Grupa='Bere'
   GROUP BY Judet);

SELECT * FROM judete_bere
WHERE Vnz_Bere = (SELECT MAX(Vnz_Bere) FROM judete_bere);
```

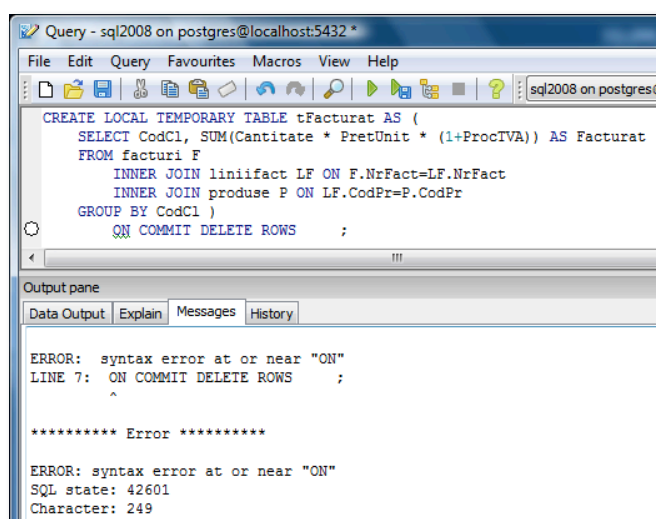


Figura 14.1. O (altă) tulburare de personalitate a PostgreSQL-ului

Practic, expresia-tabelă a devenit tabelă temporară. Acum, dacă tot a venit vremea confidențelor, trebuie să vă mărturisesc că PostgreSQL-ul „iese din frontul” standardului, deoarece:

- PostgreSQL-ul nu păstrează definiția tabelelor temporare între sesiuni;
- Nu face diferențierea dintre tabelele temporare globale și locale;
- Dacă, în lipsa clauzei ON COMMIT, standardul prevede ștergerea liniilor, în PostgreSQL este invers – când lipsește ON COMMIT conținutul se păstrează;

- Clauza ON COMMIT are, pe lângă cele două opțiuni standard PRESERVE ROWS și DELETE ROWS, clauza DROP care semnalizează că, la finalul tranzacției, tabela urmează să fie ștearsă.

În plus (de fapt, în minus!), nu am reușit să folosesc clauza ON COMMIT, mesajul de eroare fiind, de fiecare dată de genul celui din figura 14.1³.

Care sunt cei mai mari trei 3 datornici ?

În varianta „scripturală” PostgreSQL pe care o vom înșira mai jos vom crea trei tabele temporare locale/globale: *tFACTURAT* care conține vânzările pentru fiecare client, *tINCASAT* ce conține totalul încasărilor de la fiecare client și *tDE_INCASAT*, cu restul de plată al fiecărui client:

```
CREATE LOCAL TEMPORARY TABLE tFacturat AS (
    SELECT CodCl, SUM(Cantitate * PretUnit * (1+ProcTVA)) AS Facturat
    FROM facturi F
        INNER JOIN liniifact LF ON F.NrFact=LF.NrFact
        INNER JOIN produse P ON LF.CodPr=P.CodPr
    GROUP BY CodCl ;
CREATE LOCAL TEMPORARY TABLE tIncasat AS
    SELECT CodCl, SUM(Transa) AS Incasat
    FROM facturi F INNER JOIN incasfact I ON F.NrFact=I.NrFact
    GROUP BY CodCl ;
CREATE LOCAL TEMPORARY TABLE tDe_INCASAT AS
    SELECT DenCl, tFACTURAT.CodCl, Facturat,
        COALESCE(Incasat,0) AS Incasat,
        Facturat - COALESCE(Incasat,0) AS De_Incasat
    FROM tFACTURAT
    INNER JOIN clienti ON tFacturat.CodCl=clienti.CodCl
    LEFT OUTER JOIN Tincasat
        ON tFACTURAT.CodCl=tINCASAT.CodCl ;
```

Interogarea finală face apel doar la ultima tabelă temporară:

```
SELECT * FROM tDe_INCASAT WHERE De_Incasat >=
    (SELECT MAX(De_Incasat) FROM tDe_INCASAT WHERE De_Incasat <
        (SELECT MAX(De_Incasat) FROM tDe_INCASAT WHERE De_Incasat <
```

³ Vezi și documentația PostgreSQL la pagina: <http://www.postgresql.org/docs/8.3/interactive/sql-createtable.html>

```
(SELECT MAX(De_Incasat) FROM tDe_INCASAT)
) ) ;
```

Siliți fiind de sintaxa rigidă a funcției CONNECTBY(), în paragraful 12.5 am creat o tabelă virtuală – *PERSONAL2_MODIF01*. Puteam, la fel de bine, folosi și o tabelă temporară în locul celei virtuale. Analog simplificăm obținerea traseelor Iași-Focșani (și distanței fiecărei rute – vezi figura 12.33), dacă recurgem la o tabelă temporară – X.

```
CREATE LOCAL TEMPORARY TABLE x AS
SELECT ierarhie.Loc2 AS Loc1, ierarhie.Loc1 AS Loc2, distanta, ierarhie.Cale
FROM ( SELECT *
      FROM CONNECTBY('distanțe', 'Loc2', 'Loc1', 'Iasi', 0, '**')
      AS t2 (Loc1 TEXT, Loc2 TEXT, LEVEL INT, cale text)
    ) ierarhie, distanțe d
WHERE ierarhie.Loc2 =d.Loc1 AND ierarhie.Loc1=d.Loc2 ;

SELECT x1.cale, SUM(x2.distanta) AS Km
FROM x x1, x x2
WHERE x1.cale LIKE 'Iasi%Focsani' AND x1.cale LIKE x2.cale || '%'
GROUP BY x1.cale
ORDER BY Km
```

În capitolul 17 vom vedea la lucru tabelele temporare PostgreSQL, dar în altă ipostază – cea de substitut al variabilelor publice.

Și în Oracle apar câteva diferențe față de „evangheliile” SQL în materie de tabele temporare. Poate cea mai importantă este că în Oracle nu pot fi create tabele temporare locale, ci numai globale. Încercăm să simplificăm interogarea din paragraful 13.1 care acum produce rezultatul din figura 13.4:

```
CREATE GLOBAL TEMPORARY TABLE tab ON COMMIT PRESERVE ROWS AS
SELECT SYS_CONNECT_BY_PATH (Loc1, '**') || '**' || Loc2 AS Traseu,
      LEVEL AS Nivel, Loc1, Loc2,
      EXTRACT (HOUR FROM Durata) * 60 + EXTRACT (MINUTE
      FROM durata) AS Durata_Min, ROWNUM AS Ord
FROM distanțe d START WITH Loc1='Iasi'
CONNECT BY PRIOR Loc2 = Loc1 AND Loc1<>'Focsani' AND Level < 20
ORDER BY Ord ;

SELECT 10001 AS IdCursa, 1 AS "StatieNr", 'Iasi' AS "Statie",
      TIMESTAMP'2008-04-01 09:05:00' AS "Data/Ora"
FROM dual
UNION
SELECT 10001, t2.Nivel + 1, t2.Loc2, TIMESTAMP'2008-04-01 09:05:00' +
      TO_DSINTERVAL ('0 ' || FLOOR(
```

```

(SELECT SUM(Durata_Min) FROM tab WHERE t2.Traseu LIKE Traseu || '%' )
/ 60) || ':' ||
MOD ( (SELECT SUM( Durata_Min ) FROM tab WHERE t2.Traseu
LIKE Traseu || '%' ), 60)
|| ':00')
FROM tab t1 INNER JOIN tab t2 ON t1.Traseu LIKE '**Iasi**Vaslui%Focsani'
AND t1.traseu LIKE t2.Traseu || '%'
ORDER BY 1,2

```

Interesant este și modul în care ștergem o tabelă temporară în Oracle. Astfel, comanda DROP fără o „pregătire” prealabilă generează mesajul de eroare din figura 14.2.

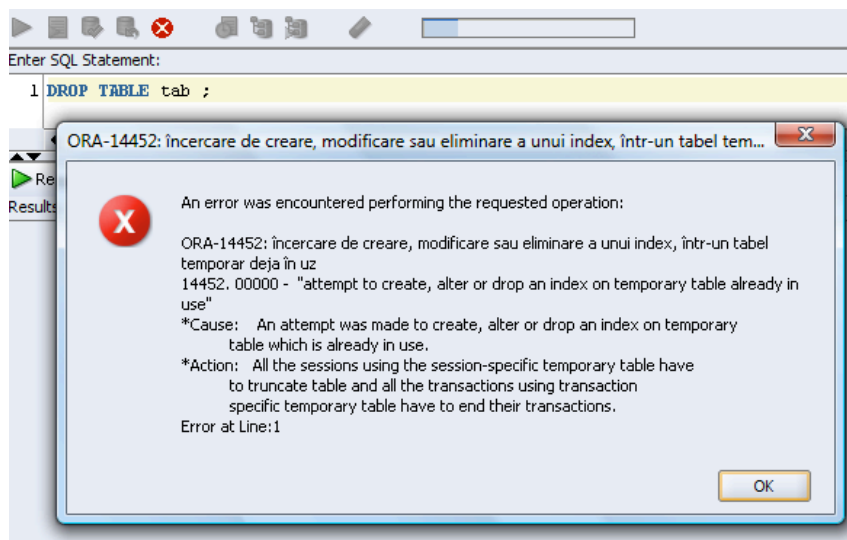


Figura 14.2. Tentativă eșuată de ștergere a unei tabele temporare în Oracle

Din mesaj ar rezulta că tabela temporară ar trebui, mai întâi, ștearsă cu TRUNC. Într-adevăr, prin succesiunea celor două comenzi:

```
TRUNCATE TABLE tab ;
```

```
DROP TABLE tab ;
```

ștergerea decurge fără probleme.

DB2 face (ca și SQL Server, dar în altă direcție) opinie separată în materie de tabele temporare. În DB2 o tabelă temporară este definită la nivel de sesiune, însă descrierea sa nu apare în dicționarul de date (catalogul de sistem). De aceea, o tabelă temporară nu poate fi partajată cu alte sesiuni. La terminarea sesiunii, orice urmă a tabelului temporar dispăre. Comanda nu este CREATE TABLE, ci DECLARE GLOBAL TEMPORARY TABLE care este însă destul de limitată. Copiem atributele tabelului CURSE din paragraful 13.1 în tabela temporară tCURSE:

```

DECLARE GLOBAL TEMPORARY TABLE tCurse AS
(
    SELECT 1001 AS IdCursa, CAST (' ' AS VARCHAR(200)) AS Traseu,
    CAST ('2008-04-01 09:05:00' AS TIMESTAMP) AS DataOra_Plecare
FROM sysibm.sysdummy1 )
DEFINITION ONLY ;

```

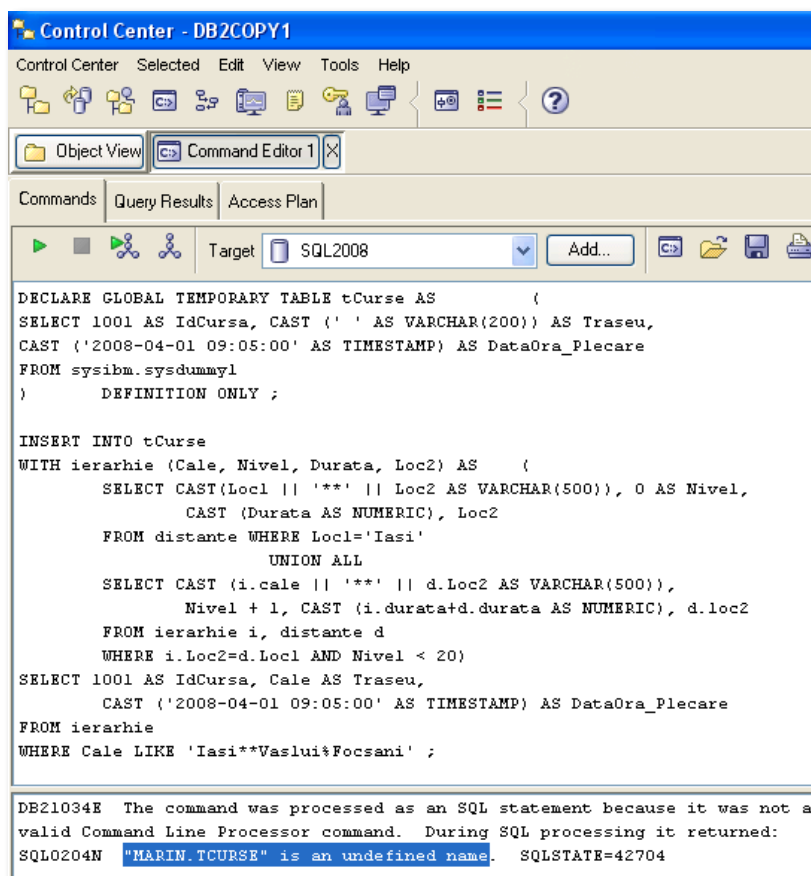


Figura 14.3. Tabele extrem de temporare în DB2

În schimb, când încercăm să populăm tabela temporară cu INSERT-ul din paragraful 13.3.1 avem parte de un mesaj destul de ciudat, potrivit căruia *tCURSE* este un obiect necunoscut (un fel de OZN) - vezi figura 14.3. Explicația este că DB2-ul uită extrem de repede definiția tablei temporare. Firește că vă, întrebați, ca și mine, la ce bune tabelele temporare în DB2. Răspunsul va fi oferit în următoarea ediție a cărții (care, dacă ar urma trendul acesteia, ar apărea prin 2015).

Și SQL Server este departe de sintaxa din standardul SQL. Cele două tipuri de tabele temporare sunt specificate prin simboluri # plasate înaintea numelui. Astfel, dacă, la creare, numele tablei este prefixat de un singur simbol #, atunci tabela temporară este una locală, vizibilă deci numai în sesiunea curentă. Dacă prefixul

numelui tabelului este format din două semne # (##), atunci tabela temporară este globală. Reluăm exemplul celor trei datornici prezentat și pentru PostgreSQL:

```
SELECT CodCl, SUM(Cantitate * PretUnit * (1+ProcTVA)) AS Facturat
INTO #tFacturat
FROM facturi F INNER JOIN liniifact LF ON F.NrFact=LF.NrFact
      INNER JOIN produse P ON LF.CodPr=P.CodPr
GROUP BY CodCl ;

      SELECT CodCl, SUM(Transa) AS Incasat
      INTO #tIncasat
      FROM facturi F INNER JOIN incasfact I ON F.NrFact=I.NrFact
      GROUP BY CodCl ;

SELECT DenCl, #tFacturat.CodCl, Facturat, COALESCE(Incasat,0) AS Incasat,
      Facturat - COALESCE(Incasat,0) AS De_Incasat
INTO #tDe_INCASAT
FROM #tFacturat
      INNER JOIN clienti ON #tFacturat.CodCl=clienti.CodCl
      LEFT OUTER JOIN #tIncasat ON #tFacturat.CodCl = #tIncasat.CodCl ;

SELECT * FROM #tDe_INCASAT WHERE De_Incasat >=
(SELECT MAX(De_Incasat) FROM #tDe_INCASAT WHERE De_Incasat <
      (SELECT MAX(De_Incasat) FROM #tDe_INCASAT WHERE De_Incasat <
            (SELECT MAX(De_Incasat) FROM #tDe_INCASAT)
      ) ) ;
```

14.2. Tabele virtuale în interogări

Ca și cele temporare, tabelele virtuale (*view*-urile) sunt construite prin subconsultări aplicate asupra tabelului de bază și/sau altor tabele temporare sau *view*-uri. Tot similar celor temporare, numai definiția (structura) este persistentă și publică (accesibilă altor sesiuni/utilizatori), conținutul nu ! Există însă și diferențe majore. O tabelă virtuală nu are conținut propriu-zis, fiind „instanțiată” doar la invocarea acesteia (invocare ce înseamnă, de fapt, execuția interogării-definiției). Dar cea mai importantă deosebire este că numai o parte dintre *view*-uri sunt actualizabile, iar modificarea lor se propagă automat în tabelele sursă. Inserarea, modificarea sau ștergerea unei linii dintr-un *view actualizabil* se va traduce, în fapt, în inserarea, modificarea sau ștergerea de linii în/din una sau mai multe dintre tabelele „sursă”.

Avantajele tabelului virtual sunt multiple:

- ușurarea interogărilor, prin salvarea rezultatelor intermediare;
- construirea unor surse de date pentru rapoarte și controale ale formularelor (liste, combo-uri sau chiar grid-uri);
- facilitatea vizualizării informațiilor „presărate” în multe tabele;

- mai bună securitate a datelor; anumitor categorii de utilizatori, în locul accesului la tabelele bazei, li se poate acorda acces exclusiv la view-uri în care pot vedea datele pe care sunt autorizați să le consulte/modifice.

Standardul SQL-92 a introdus view-urile ca tabele virtuale ce sunt materializate la invocarea numelui lor. Materializarea înseamnă execuția frazei SELECT ce constituie definiția tabelului virtual și popularea cu înregistrările-rezultat ale interogării. Formatul simplist al comenzii de creare a unei tabele virtuale este:

```
CREATE VIEW <nume-tabelă-virtuală> [<listă-coloane>] AS
    <frază SELECT>
    [WITH [<clauză de niveluri>] CHECK OPTION]
```

unde <clauză de niveluri> ::= CASCADED | LOCAL.

Odată creată tabela virtuală, definiția sa este salvată în schema bazei. Ulterior, ori de câte ori este necesar, la deschiderea/reîmprospătarea tabelului virtual, aceasta este (re) populată cu înregistrări extrase din cele ale tabelelor propriu-zise ce apar în clauza FROM a interogării. O tabelă virtuală poate fi deci privită și ca o expresie de subconsultare a unei tabele persistente, stocată în bază și invocată prin numele său.

Potrivit standardelor SQL, unei tabele virtuale nu i se pot asocia indecși și nici defini restricții, însă unele SGBD-uri optimizează lucrul cu *view*-urile folosind indecșii tabelului persistente. Numele tabelului virtual este unic, și nu se poate auto-referi, deși un *view* poate fi creat pe baza unei combinații de tabele persistente și/sau alte *view*-uri.

Ștergerea unei tabele virtuale din schema bazei de date se realizează prin comanda DROP VIEW:

```
DROP VIEW <nume-tabelă-virtuală> <comportament>
```

unde <comportament> ::= CASCADE | RESTRICT. Prin clauza CASCADE, se șterg atât tabela virtuală curentă, cât și toate cele create pe baza acesteia, în timp ce RESTRICT interzice operațiunea, atâta timp cât există măcar un view construit pe baza tabelului virtual curent.

De regulă, formatul general al comenzii de creare a tabelului virtual este mai generos decât cel al unei tabele temporare. Spre exemplu, în DB2 la definirea unei tabele temporare nu putem include o expresie tabelă, în timp ce la o tabelă virtuală da. Comanda următoare definește o tabelă virtuală, NUMERE_NATURALE, ce va conține numere naturale (atributul Numar) de la 0 la 9999:

```
CREATE VIEW numere_naturale AS
    WITH temp1 (Numar) AS (
        VALUES (0) UNION ALL
        SELECT Numar + 1 FROM temp1 WHERE Numar < 10000
    )
    SELECT * FROM temp1;
```

Reformulăm, astfel, o problemă pentru care am formulat soluția în paragraful 12.6: Care sunt numerele de facturi nefolosite (în tabela FACTURI)?

```
SELECT Numar AS Numar_nefolosit
```



```

FROM (SELECT Numar FROM numere_naturale WHERE Numar BETWEEN
      (SELECT MIN(NrFact) FROM facturi) AND
      (SELECT MAX(NrFact) FROM facturi )
      ) LEFT OUTER JOIN facturi f ON Numar = NrFact
WHERE f.NrFact IS NULL
ORDER BY 1

```

Tabelele virtuale sunt preferabile celor temporare și datorită modului de reîmprospătare a conținutului. În cadrul aceleiași sesiuni, o tabelă temporară globală nu-și schimbă conținutul (dacă a fost definită cu clauza ON COMMIT PRESERVE ROWS). Dacă sesiunea este lungă, este probabil ca tabelele din care provine tabela temporară să fi fost actualizate. Modificările sunt reflectate automat la următoarea invocare a *view*-ului, în timp ce tabela temporară va fi reîmprospătată abia la următoarea inițializare (următoarea sesiune).

The screenshot shows an SQL*Plus session with the following SQL statements and results:

```

CREATE GLOBAL TEMPORARY TABLE tFacturatCl ON COMMIT PRESERVE ROWS AS (
SELECT DenCl, SUM(Cantitate * PretUnit * (1+ProcTVA)) AS Facturat
FROM clienti C INNER JOIN facturi F ON C.CodCl=F.CodCl
INNER JOIN liniifact LF ON F.NrFact=LF.NrFact
INNER JOIN produse P ON LF.CodPr=P.CodPr
GROUP BY DenCl ) ;
SELECT * FROM tFacturatCl ORDER BY 1;

CREATE VIEW vFacturatCl AS (
SELECT DenCl, SUM(Cantitate * PretUnit * (1+ProcTVA)) AS Facturat
FROM clienti C INNER JOIN facturi F ON C.CodCl=F.CodCl
INNER JOIN liniifact LF ON F.NrFact=LF.NrFact
INNER JOIN produse P ON LF.CodPr=P.CodPr
GROUP BY DenCl )
WITH READ ONLY ;
SELECT * FROM vFacturatCl ORDER BY 1;

INSERT INTO facturi (nrfact, datafact, codcl)
VALUES (3120, DATE'2007-10-17', 1001);
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
VALUES (3120, 1, 1, 50, 1000) ;
INSERT INTO liniifact (nrfact, linie, codpr, cantitate, pretunit)
VALUES (3120, 2, 2, 75, 1050) ;

SELECT * FROM tFacturatCl ORDER BY 1;
SELECT * FROM vFacturatCl ORDER BY 1;

```

The results show two tables: DENCL and FACTURAT. The top table shows the state before the insert, and the bottom table shows the state after the insert. The view vFacturatCl is updated immediately, while the temporary table tFacturatCl remains unchanged.

DENCL	FACTURAT
Client 1 SRL	40163327
Client 2 SA	956475
Client 3 SRL	46421835,5
Client 4	28437328,5
Client 5 SRL	1143758
Client 6 SA	18065119,5
Client 7 SRL	1064385

DENCL	FACTURAT
Client 1 SRL	40308664,5
Client 2 SA	956475
Client 3 SRL	46421835,5
Client 4	28437328,5
Client 5 SRL	1143758
Client 6 SA	18065119,5
Client 7 SRL	1064385

Figura 14.4. Actualizarea dinamică a tabelor virtuale din tabelele-sursă

Figura 14.4 prezintă o sesiune în care sunt create o tabelă temporară (*tFacturatCl*) și una virtuală (*vFacturatCl*) care, inițial, au conținut identic. În aceeași sesiune, se adaugă o factură (3120) cu două linii pentru *Clientul 1 SQL*. După inserare, conținutul tabelii temporare este identic celui de după crearea sa, în timp ce prima linie din tabela virtuală se împrospătează cu noua valoare a atributului *Facturat* pentru *Client 1 SRL*.

14.3. Probleme ale actualizărilor tabelelor sursă pe baza modificării tabelelor virtuale

Am văzut în figura 14.4 că o tabelă virtuală se împrăștează automat cu modificările operate în tabellele-Sursă. Ei, bine, în unele cazuri și reciproca este valabilă, adică o serie de *view*-uri sunt modificabile (d.p.d.v. al conținutului), iar modificările se propagă în tabela sau tabellele sursă. Numai că aici există o serie întreagă de restricții. Pentru *vFacturatCl* discuția se simplifică din start deoarece la creare s-a folosit opțiunea READ ONLY, ceea ce înseamnă că orice tentativă de inserare, modificare și ștergere va fi automat tratată cu dușmănie.

Pentru a fi actualizabilă, fiecare înregistrare a unei tabelle virtuale trebuie să poată fi asociată unei singure linii dintr-o tabelă sursă. Astfel, modificarea unei linii din *view* poate fi propagată fără probleme de ambiguitate. Firește, o tabelă derivată de genul:

```
CREATE VIEW vJudete1 AS
SELECT * FROM Judete
WHERE regiune = 'Moldova' OR regiune = 'Dobrogea'
```

nu va crea probleme insolubile la actualizare. În schimb, deși cu un conținut identic *vJudete1*, tabela virtuală *vJudete2*, creată prin comanda care urmează, nu este actualizabilă nici în SQL, nici în majoritatea SGBD-urilor importante.

```
CREATE VIEW vJudete2 AS
SELECT * FROM Judete WHERE regiune = 'Moldova'
UNION
SELECT * FROM Judete WHERE regiune = 'Dobrogea'
```

Pentru a intra în câteva detalii, să imaginăm o tabelă derivată denumită INCA-SARI_CLIENTI – figura 14.5 – ce conține documentul de încasare, suma încasată (corespunzătoare documentului) și clientul care a efectuat plata. Comanda de creare a tabellei virtuale este:

```
CREATE VIEW vINCASARI_CLIENTI AS (
SELECT DenCl, CodDoc, NrDoc, DataDoc, SUM(Transa) AS SumaPlatita
FROM clienti c
INNER JOIN facturi f ON c.CodCl = f.CodCl
INNER JOIN incasfact incf ON f.NrFact = incf.NrFact
INNER JOIN incasari inc ON incf.CodInc = inc.CodInc
GROUP BY DenCl, CodDoc, NrDoc, DataDoc
);
```

DENCL	CODDOC	NRDOC	DATADOC	SUMAPLATITA
Client 1 SRL OP	333	09-08-2007	117069	
Client 1 SRL CEC	444	10-08-2007	9754	
Client 1 SRL OP	111	10-08-2007	155971	
Client 2 SA OP	555	10-08-2007	106275	
Client 1 SRL OP	666	11-08-2007	3696	
Client 5 SRL CHIT	222	15-08-2007	125516	

Figura 14.5. O tabelă virtuală

Utilitatea unei asemenea relații virtuale poate fi certificată de un angajat al compartimentului financiar care se ocupă, printre altele, și cu evidența încasărilor de la clienți. Pe cât de utilă, pe atât de problematică devine INCASARI_CLIENTI atunci când se pune problema actualizării sale și propagării modificărilor în tabelele de bază din care provine, CLIENTI, FACTURI, INCASARI și INCASFACT. Iată câteva dintre probleme:

- modificarea atributului *SumaPlatită* este imposibil de operat în tabela de bază, INCASFACT. O linie din tabela virtuală corespunde uneia sau mai multor linii din tabela de bază (depinde câte facturi sunt achitate prin documentul de plată respectiv). Dacă pentru ordinul de plată 111 din 10 august 2007 se dorește modificarea (corecția) sumei din 155 971 în 156 000, nu se poate cunoaște tranșa și factura unde s-a comis eroarea și, normal, unde trebuie operată modificarea.
- modificarea celui de-al patrulea tuplu din (*Client 2 SA, OP, 555, 10-Aug-2007, 106275*) în (*Client 5 SRL, OP, 555, 10-Aug-2007, 106275*), adică modificarea clientului pentru Ordinul de plată nr. 555 din 10.08.2007 poate fi operată în trei moduri în tabele de bază:
 - fie se modifică în FACTURI pentru factura 1113 valoarea atributului *CodCl* din 1002 în 1005;
 - fie se modifică în linia din INCASFACT codul încasării (*CodInc*) din 1238 în 1235;
 - fie se modifică în INCASFACT valoarea atributului *NrFact* din 1113 în 1112, păstrându-se codul încasării neschimbat;
 chiar dacă nu la fel de plauzibile, cele trei variante generează o “stare” de confuzie pentru SGBD.
- inserarea unei linii în *view* este o acțiune temerară, dar fără prea mulți sorți de izbândă: oricare ar fi cele patru tabele sursă în care s-ar face inserarea, cel puțin un atribut important (ce nu poate avea valori NULL) nu are valori specificate, astfel încât se încalcă una dintre restricții (de entitate sau comportament). Spre exemplu, convenim că inserarea trebuie să se propage numai în INCASFACT. Nu se cunoaște însă numărul facturii încasate. Cum *NrFact* este un component al cheii primare a tabelului, este clar că operațiunea va fi interzisă de SGBD.
- ștergerea unei linii din tabela virtuală ridică problema identificării liniei sau liniilor dintr-una sau mai multe tabele de bază.

În plus, sursele unei tabele virtuale pot fi atât tabele propriu-zise, cât și alte tabele virtuale, situație în care vorbim de mai multe niveluri ale tabelelor virtuale. Spre exemplu, pentru a construi o tabelă virtuală *vCLIENTI* în care pot fi modificate: denumirea clientului, adresa sa, denumirea localității în care își are sediul și numele județului, sunt necesare următoarele view-uri:

```
CREATE VIEW vJudete AS
    SELECT * FROM Judete ;
CREATE VIEW vCoduri_Postale AS
    SELECT CodPost, Loc, vJudete.Jud, Judet, Regiune
    FROM coduri_postale cp INNER JOIN vJUDETE ON cp.Jud = vJUDETE.Jud ;
CREATE VIEW vClienti AS
    SELECT CodCl, DenCl, Adresa, vCoduri_Postale.CodPost, Loc, Jud,
        Judet, Regiune
    FROM clienti INNER JOIN vCoduri_Postale
        ON clienti.CodPost = vCoduri_Postale.CodPost ;
```

Tabelele virtuale trebuie să includă toate coloanele cheilor primare/alternative ale tabelelor de bază. O linie dintr-o tabelă virtuală trebuie să corespundă unei singure linii din tabela de bază. Toate coloanele neincluse în *view* trebuie să permită valori NULL sau să prezinte valori implicite. Altminteri, inserarea unei linii într-o tabelă derivată ar fi imposibilă. Nu poate fi actualizată o tabelă virtuală creată prin fraze SELECT în care apar:

- clauze GROUP BY / HAVING,
- funcții agregat,
- coloane calculate,
- operatorii: UNION, INTERSECT, EXCEPT (MINUS),
- SELECT DISTINCT.

SQL:1999 definește câțiva termeni legați de tabelele virtuale sau consultările-argument ale comenzilor de actualizare⁴:

- potențial actualizabil;
- actualizabil;
- banal-actualizabil;
- inserabil-în.

⁴ [Melton & Simon 2002], pp.111-112, 283-284

O specificație de interogare (care poate fi expresia de definire a unui view, sau o expresie ce urmează comenzii UPDATE – vezi paragraful 13.3.4) este *potențial actualizabilă* dacă și numai dacă⁵:

- nu se folosește clauza DISTINCT;
- toate coloanele din clauza SELECT apar în listă o singură dată;
- nu se folosesc clauze GROUP BY/HAVING.

Dacă specificația de interogare este potențial actualizabilă, iar în clauza FROM apare o singură tabelă, atunci toate coloanele din această tabelă care apar în specificație sunt *actualizabile*. Dacă specificația este potențial actualizabilă, însă în clauza FROM apar două sau mai multe tabele, atunci o coloană din specificație este actualizabilă numai dacă:

- provine dintr-o singură tabelă dintre cele din clauza FROM;
- între tabela respectivă și specificația de interogare există o relație de corespondență a unicității, adică tabela își păstrează în interogare cheia primară sau cheile alternative.

Altfel spus, valoarea unui atribut dintr-o tabelă virtuală este actualizabilă doar dacă aceasta corespunde unei singure valori dintr-o tabelă sursă. O specificație (expresie) de interogare este actualizabilă dacă măcar una dintre coloane este actualizabilă, și *banal-actualizabilă* dacă:

- este actualizabilă,
- clauza FROM conține o singură tabelă, și
- toate coloanele sale sunt actualizabile.

Termenul (destul de nenorocit) *inserabil-în* desemnează o specificație/expresie actualizabilă, dacă toate tabelele din clauza FROM acceptă adăugări de linii (INSERT-uri) și dacă expresia nu include operatorii UNION, INTERSECT și EXCEPT.

SGBD-urile actuale prezintă câteva diferențe în materie de reguli privind actualizarea tabelelor virtuale. Cel mai flexibil mecanism este, însă, cel al declanșatoarelor de tip *INSTEAD OF*, despre care vom discuta pe scurt în capitolul 17.

14.3.1. Tabele virtuale în DB2

Documentația IBM și bibliografia DB2⁶ delimitează tabelele virtuale DB2 în funcție de operațiunile pe care le acceptă, după cum urmează:

- tabele virtuale ce permit ștergerea (*deletable*);

⁵ [Melton & Simon 2002], pp.111-112, 284

⁶ Vezi spre exemplu [Baklarz & Zikopoulos 2008]

- tabele virtuale ce permit modificarea (*updatable*);
- tabele virtuale ce permit inserarea (*insertable*);
- tabele virtuale ce permit doar citirea, non-modificabile (*read-only*);

Cele din prima categorie acceptă comanda DELETE, comandă pe baza căreia ștergerea se propagă într-o tabelă-sursă. Pentru a putea propaga ștergerea, fraza SELECT de creare a tabelii virtuale nu trebuie să conțină clauze precum DISTINCT, GROUP BY/HAVING, VALUES, UNION, INTERSECT, EXCEPT, iar fiecare linie din *view* corespunde unei singure linii dintr-o tabelă sursă. Singurul operator ansamblist acceptabil în unele situații este UNION ALL. Un view este *actualizabil* (*updatable*) dacă, în plus, are măcar un atribut al cărui valoare poate fi modificată și *inserabil* (*insertable*) dacă toate coloanele sale sunt *actualizabile*.

Astfel, *vClienti*, creată ceva mai sus, nu permite ștergerea unei linii – vezi figura 14.6. Cum nici măcar ștergerile nu sunt admise, cu atât mai puțin modificările și inserările.

Tabela virtuală *vLiniiFact7001*, fiind definită printr-o selecție aplicată unei singure tabelă – LINIIFACT:

```
CREATE VIEW vLiniifact7001 AS
```

```
    SELECT NrFact, Linie, CodPr, Cantitate, PretUnit
    FROM liniifact WHERE NrFact=7001 ;
```

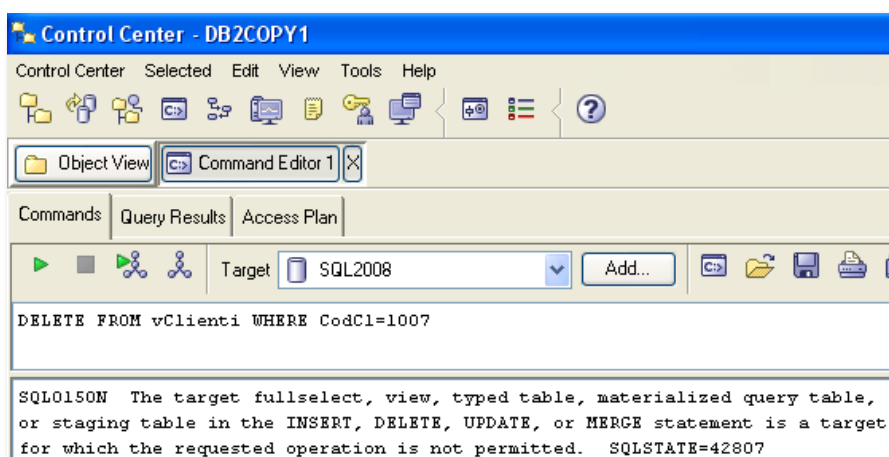


Figura 14.6. O tabelă virtuală din care nu pot fi șterse linii

acceptă orice gen de modificare, fiind *inserabilă*, suportă oricare dintre comenzile următoare:

```
DELETE FROM vLiniifact7001 WHERE Linie=3 ;
```

```
UPDATE vLiniifact7001 SET Cantitate = 100 WHERE Linie=2 ;
```

```
INSERT INTO vLiniifact7001 VALUES (7001, 3, 4, 30, 730) ;
```

Nici tabela virtuală *vLiniifact7001_7002* nu acceptă ștergerea de înregistrări, deși operatorul folosit este UNION ALL:

```
CREATE VIEW vLiniifact7001_7002 AS
```

```
SELECT * FROM liniifact WHERE NrFact=7001  
UNION ALL
```

```
SELECT * FROM liniifact WHERE NrFact=7002 ;
```

Comanda următoare se va solda cu un mesaj de eroare similar celui din figura 14.6, deoarece, pentru a funcționa, UNION ALL trebuie să conecteze tabele-sursă diferite:

```
DELETE FROM vLiniifact7001_7002 WHERE NrFact=7001 AND Linie=3
```

14.3.2. Tabele virtuale în Oracle

Sintaxa Oracle pentru definirea tabelelor virtuale este un pic mai generoasă decât cea din DB2. Dacă în DB2, pentru modificarea frazei SELECT ce definește un view aveam nevoie de o combinație DROP VIEW/CREATE VIEW, în Oracle se poate folosi CREATE OR REPLACE VIEW. De asemenea, în DB2, o tabelă virtuală era READ ONLY în funcție de modul său de definire. În Oracle se poate folosi clauza READ ONLY pentru a bloca modificarea conținutului unui *view* actualizabil.

```
CREATE OR REPLACE VIEW vLiniifact7001 AS
```

```
SELECT NrFact, Linie, CodPr, Cantitate, PretUnit  
FROM liniifact WHERE NrFact=7001 ;
```

Regulile privind actualizarea conținutului unei tabele virtuale în vederea propagării în tabelele sursă sunt apropiate de cele din DB2, așa că nu mai insistăm.

14.3.3. Tabele virtuale în PostgreSQL

PostgreSQL are, probabil, cea mai săracă sintaxă (dintre cele patru dialecte) în materie de CREATE VIEW:

```
CREATE VIEW vJudete AS
```

```
SELECT * FROM judete ORDER BY Judet;
```

După creare, un *view* este destul de apatic în PostgreSQL. Astfel, dacă se încearcă inserarea unei noi înregistrări corespunzătoare județului Galați, mesajul este cel din figura 14.7. Asta înseamnă că toate tabelele virtuale sunt READ ONLY în PostgreSQL. Totuși, din explicații deducem că trebuie să apelăm la mecanismul de reguli.

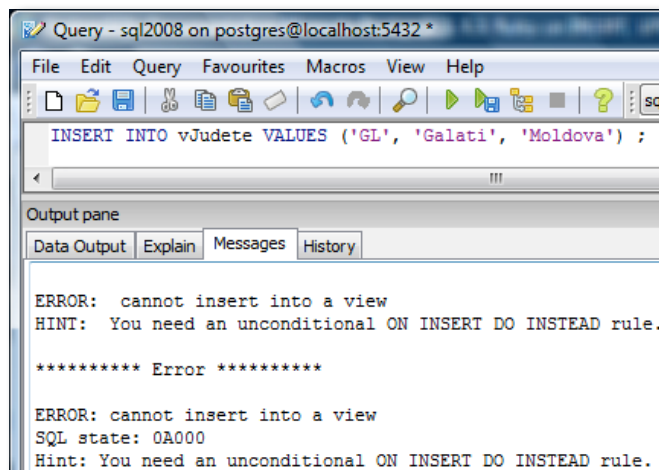


Figura 14.7. Probleme PostgreSQL chiar și la inserări într-o tabelă virtuală banal- actualizabilă

Acest mecanism de reguli prefațează destul de bine scurta discuție din capitolul 17 dedicată declanșatoarelor de tip **INSTEAD OF**. Tabela virtuală *vJudete* este creată dintr-o singură tabelă, **JUDETE**, așa că inserarea unei linii în view *trebuie* să atragă după sine adăugarea înregistrării respective în tabelă:

```
CREATE RULE vJudete_ins AS ON INSERT TO vJudete DO INSTEAD
```

```
INSERT INTO Judete VALUES (NEW.Jud, NEW.Judet, NEW.Regione) ;
```

Numele fiecărui atribut este prefixat în clauza **VALUES** cu *NEW.*, ceea ce semnalizează că este vorba despre valorile atributelor de pe noua linie a tabelii virtuale. Acum comanda **INSERT** funcționează:

```
INSERT INTO vJudete VALUES ('GL', 'Galati', 'Moldova') ;
```

La modificare unei înregistrări, vom folosi și clauza *OLD.* pentru a califica valoarea dinaintea eventualei modificări a atributului cheie primară – **Jud**.

```
CREATE RULE vJudete_upd AS ON UPDATE TO vJudete DO INSTEAD
```

```
UPDATE judete
```

```
SET Jud = NEW.Jud, Judet = NEW.Judet, Regione = NEW.Regione
```

```
WHERE Jud = OLD.Jud;
```

Redactarea regulii pentru ștergere devine acum o treabă simplă:

```
CREATE RULE vJudete_del AS ON DELETE TO vJudete
```

```
DO INSTEAD DELETE FROM judete WHERE Jud = OLD.Jud;
```

Firește, o modificare a unei valori din *view* care încalcă restricții definite în tabela de bază, va genera un mesaj de eroare – vezi figura 14.8.

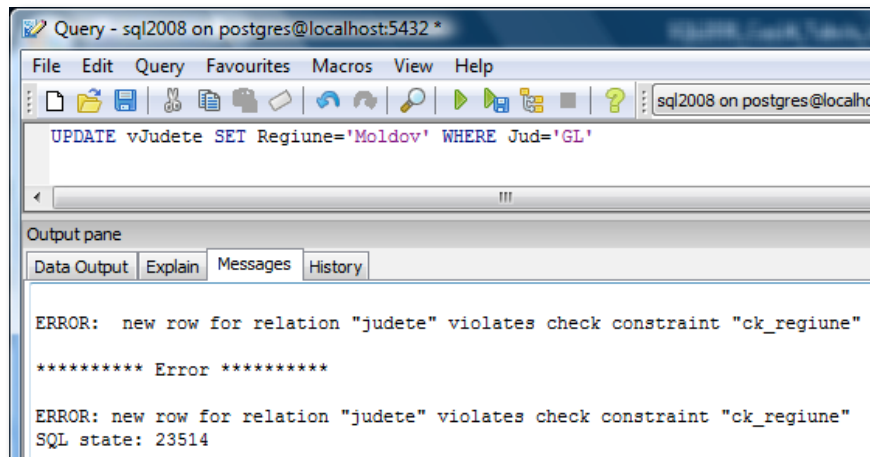


Figura 14.8. Încălcarea unei restricții la inserarea într-o tabelă virtuală

Mecanismul regulilor ne permite să încercăm rezolvarea problemelor de actualizare chiar și pentru tabele virtuale cu două sau mai multe tabele sursă jonctionate. Iată cazul view-ului *vFacturi*:

CREATE VIEW vFacturi AS

```
SELECT lf.NrFact, DataFact, DenCl, Linie, lf.CodPr, DenPr, UM,
       ProcTVA, Grupa,
       Cantitate, PretUnit, Cantitate * PretUnit AS ValFaraTVALinie,
       Cantitate * PretUnit * ProcTVA AS TVALinie,
       Cantitate * PretUnit * (1+ProcTVA) AS ValTotLinie
FROM liniifact lf
      INNER JOIN produse p ON lf.CodPr=p.CodPr
      INNER JOIN facturi f ON lf.NrFact=f.NrFact
      INNER JOIN clienti c ON f.CodCl=c.CodCl
ORDER BY lf.NrFact, Linie;
```

Deși sunt nu mai puțin de patru tabele sursă, tabela „cea mai de jos” este LINIIFACT, așa că suntem tentați să scriem regula de inserare astfel:

```
CREATE OR REPLACE RULE vFacturi_ins AS
ON INSERT TO vFacturi          DO INSTEAD
INSERT INTO liniifact VALUES ( NEW.NrFact, NEW.Linie,
                               NEW.CodPr, NEW.Cantitate, NEW.PretUnit );
```

Regula funcționează dacă adăugăm o linie unei facturi existente, ca de exemplu a cincea linie din factura 3119:

```
INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit)
VALUES (3119, 5, 6, 1000, 1200);
```

Dar însă adăugăm o linie într-o factură nouă - să zicem factura cu nr. 3120 -, firește că vom încălca restricția referențială - vezi figura 14.9.

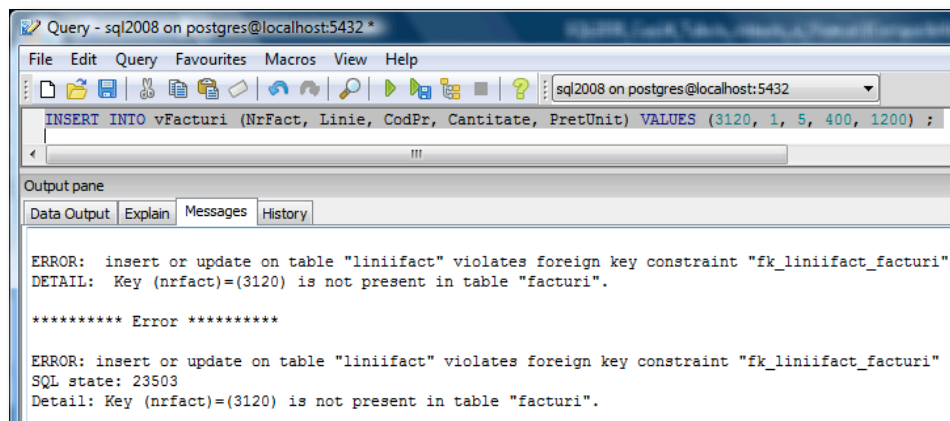


Figura 14.9. Încălcarea restricției referențiale la inserarea într-un view cu mai multe tabele-sursă

Avem soluții și pentru acest gen de probleme. Să redefinim tabela virtuală, preluând și codul clientului:

DROP VIEW vFacturi ;

CREATE VIEW vFacturi AS

```
SELECT lf.NrFact, DataFact, DenCl, f.CodCl, Linie, lf.CodPr, DenPr,
       UM, ProcTVA, Grupa,
       Cantitate, PretUnit, Cantitate * PretUnit AS ValFaraTVALinie,
       Cantitate * PretUnit * ProcTVA AS TVALinie,
       Cantitate * PretUnit * (1+ProcTVA) AS ValTotLinie
FROM liniifact lf
      INNER JOIN produse p ON lf.CodPr=p.CodPr
      INNER JOIN facturi f ON lf.NrFact=f.NrFact
      INNER JOIN clienti c ON f.CodCl=c.CodCl
ORDER BY lf.NrFact, Linie;
```

Problema cea mai delicată la rescrierea regulii de inserare este că un INSERT în *vFacturi* poate să se propage în tabele sursă în mai multe moduri:

- inserarea unei linii numai în tabela LINIIFACT;
- inserare a câte o linie în tabelele LINIIFACT și FACTURI;
- inserare a câte o linie în tabelele LINIIFACT și PRODUSE;
- inserare a câte o linie în tabelele LINIIFACT, FACTURI și PRODUSE.

Din fericire, putem formula mai multe reguli pentru aceeași operațiune (inserare), astfel:

DROP RULE IF EXISTS vFacturi_ins ON vFacturi ;

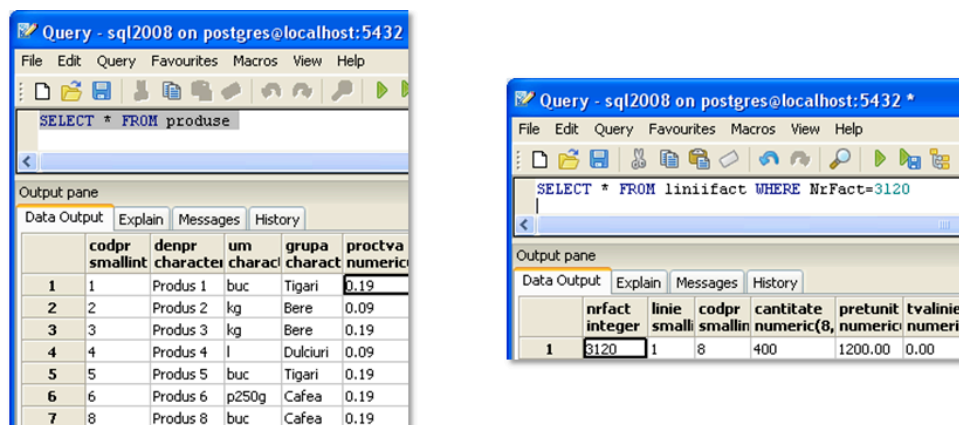
CREATE OR REPLACE RULE vFacturi_ins1 AS ON INSERT TO vFacturi

```
WHERE NOT EXISTS (SELECT 1 FROM facturi WHERE NrFact=NEW.NrFact
DO INSTEAD
```

```

INSERT INTO facturi VALUES (NEW.NrFact, NEW.DataFact, NEW.CodCl) ;
CREATE OR REPLACE RULE vFacturi_ins2 AS ON INSERT TO vFacturi
WHERE NOT EXISTS (SELECT 1 FROM produse WHERE CodPr=NEW.CodPr)
DO INSTEAD
INSERT INTO produse VALUES (NEW.CodPr, NEW.DenPr, NEW.UM,
NEW.Grupa, NEW.ProcTVA) ;
CREATE OR REPLACE RULE vFacturi_ins3 AS ON INSERT TO vFacturi
DO INSTEAD
INSERT INTO liniifact VALUES ( NEW.NrFact, NEW.Linie, NEW.CodPr,
NEW.Cantitate, NEW.PretUnit) ;

```



Left Screenshot: Query - sql2008 on postgres@localhost:5432

Query: `SELECT * FROM produse`

	codpr smallint	denpr character	um character	grupa character	proctva numeric
1	1	Produs 1	buc	Tigari	0.19
2	2	Produs 2	kg	Bere	0.09
3	3	Produs 3	kg	Bere	0.19
4	4	Produs 4	l	Dulciuri	0.09
5	5	Produs 5	buc	Tigari	0.19
6	6	Produs 6	p250g	Cafea	0.19
7	8	Produs 8	buc	Cafea	0.19

Right Screenshot: Query - sql2008 on postgres@localhost:5432 *

Query: `SELECT * FROM liniifact WHERE NrFact=3120`

	nrfact integer	linie smallint	codpr smallint	cantitate numeric(8, numeric)	pretunit numeric	tvalinie numeric
1	3120	1	8	400	1200.00	0.00

Figura 14.10. Propagarea inserării în tabelele PRODUSE și LINIIFACT

Testarea funcționalității o vom face printr-un INSERT în care vom adăuga o linie dintr-o factură nouă și un produs la fel de nou:

```

INSERT INTO vFacturi (NrFact, Linie, CodPr, Cantitate, PretUnit, DataFact,
CodCl, DenPr, UM, Grupa, ProcTVA)
VALUES (3120, 1, 8, 400, 1200, DATE'2007-10-01', 1001, 'Produs 8', 'buc', 'Cafea', 0.19) ;

```

Figura 14.10 demonstrează că s-au făcut inserările în cele trei tabele (în tabela FACTURI inserarea este implicită, altminteri restricția referențială ar fi fost încălcată).

14.3.4. Tabele virtuale în SQL Server

SQL Server este comparabil în materie de tabele virtuale cu dialectele din DB2 și Oracle. Comenzile rulate în DB2 descrise în paragraful 14.3.1 sunt funcționale și în SQL Server, așa că nu mai insistăm.

14.4. Restricții în tabele virtuale

După cum am văzut la începutul paragrafului 14.2, formatul general al comenzii CREATE VIEW are și o clauză importantă pe care o vom discuta pe scurt în continuare – WITH CHECK OPTION. Această clauză permite ca la orice inserare sau modificare, înregistrările din tabela virtuală să respecte predicatul formulat în clauza WHERE. *vJudeteMoldova* conține înregistrările tabelii JUDEȚE pentru care valoarea atributului Regiune este *Moldova*:

```
CREATE VIEW vJudeteMoldova AS
```

```
SELECT * FROM judete WHERE Regiune='Moldova' WITH CHECK OPTION
```

Comanda INSERT următoare:

```
INSERT INTO vJudeteMoldova VALUES ('BC', 'Bacau', 'Moldova')
```

se execută, fără probleme, ceea ce nu este valabil și pentru (vezi figura 14.11):

```
INSERT INTO vJudeteMoldova VALUES ('AR', 'Arad', 'Banat')
```

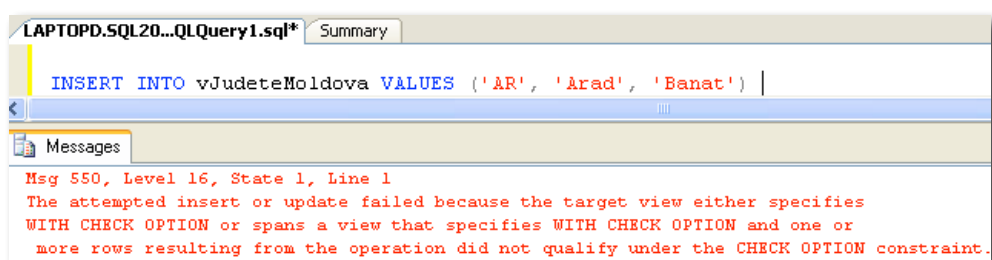


Figura 14.11. Blocarea unei inserări de către clauza WITH CHECK OPTION

Dacă sursa unei tabele virtuale este tot o tabelă virtuală care, la rândul său, este tot o tabelă virtuală s.a.m.d., clauza CHECK OPTION poate folosi opțiunea CASCADED pentru a verifica dacă inserarea/modificarea respectă predicatele din clauza WHERE ale tuturor *view*-urilor (de pe toate nivelurile) sau LOCAL pentru a verifica numai respectarea predicatului din clauza WHERE a tabelii virtuale curente. Dintre cele patru dialecte SQL, numai PostgreSQL nu are implementată clauza WITH CHECK OPTION, așa că și pentru acest gen de restricții sunt necesare declanșatoare sau reguli.

Standardele SQL nu prevăd declararea de restricții „clasice” (cheie primară, cheie alternativă, restricție referențială etc.) într-o tabelă virtuală. Nici dialectele DB2, SQL Server și PostgreSQL. Oracle permite trei tipuri de restricții declarabile pentru un *view*: chei primare, chei alternative și restricții referențiale.