**I. CONCEPTE ALE BAZELOR DE DATE RELAŢIONALE**

**1.1 Definiţii**

**1.2 Niveluri de abstractizare a datelor**

**1.3 Componente ale bazelor de date relaţionale**

**1.4 Proiectarea bazelor de date relaţionale. Etape. Normalizarea bazelor de date**

**1.1 Definiţii**

**Baze de date**

O bază de date este un fisier care contine o colecţie de informaţii interrelaţionate gestionate ca o singură unitate. Această definiţie este intenţionat foarte largă, deoarece există mari diferenţe între concepţiile diferiţilor producători care pun la dispoziţie sisteme de baze de date. De exemplu, Oracle Corporation defineşte o bază de date ca fiind o colecţie de fişiere fizice gestionate de o singură instanţă (copie) a produsului software pentru baze de date, în timp ce Microsoft defineşte o bază de date SQL Server ca fiind o colecţie de date şi alte obiecte. Un *obiect* al bazei de date este o structură de date denumită stocată în baza de date, cum ar fi un tabel, o vizualizare sau un index.

**Sisteme de gestiune a bazelor de date**

Un *sistem de gestionare a bazei de date (DBMS - dotabase management system)* este un produs software furnizat de producătorul bazei de date. Produse software precum Microsoft Access, Microsoft SQL Server, Oracle Database, Sybase, DB2, INGRES, MySQL şi PostgreSQL fac parte din categoria DBMS sau, mai corect, *DBMS relaţionale (RDBMS).*

Sistemul DBMS pune la dispoziţie toate serviciile de bază necesare pentru organizarea şi întreţinerea bazei de date, inclusiv următoarele:

• Transferarea datelor în şi din fişierele fizice de date, în funcţie de cerinţe.

• Gestionarea accesului concurenţial la date al mai multor utilizatori, inclusiv prevenirea conflictelor care ar putea fi cauzate de actualizările simultane.

• Gestionarea tranzacţiilor, astfel încât toate modificările făcute asupra bazei de date printr-o

tranzacţie să fie executate ca o singură unitate. Cu alte cuvinte, dacă tranzacţia reuşeşte, toate modificările efectuate de tranzacţie sunt înregistrate în baza de date; dacă tranzacţia eşuează, nici una dintre modificări nu este înregistrată în baza de date. Totuşi, unele sisteme RDBMS nu asigură suportul pentru tranzacţii.

• Acceptă un *limbaj de interogare,* care reprezintă sistemul de comenzi folosit de utilizator pentru

a obţine date din baza de date. SQL este principalul limbaj folosit pentru sistemele DBMS

relaţionale.

• Funcţii pentru salvarea bazei de date şi pentru refacerea bazei de date în urma erorilor.

• Mecanisme de securitate pentru împiedicarea accesului neautorizat la date şi modificarea acestora.

**Baze de date relaţionale**

O *bază de date relaţională* este o bază de date care respectă modelul relaţional, dezvoltat de Dr. E.

F. Codd. Modelul relaţional prezintă datele sub forma familiarelor tabele bidimensionale, similar cu o foaie de calcul tabelar Excel. Spre deosebire de o foaie de calcul tabelar, nu este obligatoriu ca datele să fie stocate într-o formă tabelară, iar modelul permite şi combinarea tabelelor *(crearea uniunilor (joining),* în terminologia relaţională) pentru formarea vizualizărilor, care sunt prezentate tot ca tabele bidimensionale. Flexibilitatea extraordinară a bazelor de date relaţionale este dată de posibilitatea de a folosi tabelele independent sau în combinaţii, fără nici o ierarhie sau secvenţă predefinită în care trebuie să se facă accesul la date.

**1.2 Niveluri de abstractizare a datelor**

Într-un sistem informatic ce utilizează baze de date, organizarea datelor poate fi analizată din mai multe puncte de vedere şi pe diferite niveluri. De obicei, abordarea se face pe trei niveluri: intern, conceptual şi extern (fig. 1.1).

- **Nivelul fizic (intern)**

Structura datelor este descrisă foarte detaliat, fiind accesibilă numai specialiştilor (ingineri de sistem, programatori în limbaje de asamblare sau alte limbaje apropiate de „maşină"). Cele două părţi principale ale bazei la acest nivel sunt:

1. un set de programe care interacţionează cu sistemul de operare pentru îmbunătăţirea managementului bazei de date;

2. fişierele stocate în memoria externă a calculatorului.

Fişierele ce conţin datele propriu-zise sunt alcătuite din articole sau înregistrări cu format comun. La acest nivel, structura bazei de date se concretizează în *schema internă*.

- **Nivelul conceptual (global)**

Este nivelul imediat superior celui fizic, datele fiind privite prin prisma semanticii lor; interesează conţinutul lor efectiv, ca şi relaţiile care le leagă de alte date. Reprezintă primul nivel de abstractizare a lumii reale observate. Obiectivul acestui nivel îl constituie modelarea realităţii considerate, asigurându-se independenţa bazei faţă de orice restricţie tehnologică sau echipament anume. Întreaga bază de date este descrisă prin intermediul unui număr restrâns de structuri. Toţi utilizatorii îşi exprimă nevoile de date la nivel conceptual, prezentându-le administratorului bazei de date, acesta fiind cel care are o viziune globală necesară satisfacerii tuturor cerinţelor informaţionale. La acest nivel, structura bazei de date se concretizează în *schema conceptuală*.

N i v e l

e x te r n G r u p 1 … … . G r u p n

N i v e l

c o ncep t u al S c hem a c o ncep t u al a

N i v e l S c h e m a in te r n a in te r n

M e d i u l d e s t o car e

**Fig. 1.1 Niveluri de abstractizare a datelor**

- **Nivelul extern**

Este ultimul nivel de abstractizare la care poate fi descrisă o bază de date. Structurile de la nivelul conceptual sunt relativ simple, însă volumul lor poate fi deconcertant. Dacă la nivelul conceptual baza de date este abordată în ansamblul ei, în practică un utilizator sau un grup de utilizatori lucrează numai cu o porţiune specifică a bazei, în funcţie de departamentul în care îşi desfăşoară activitatea şi ce atribuţii au. Simplificarea interacţiunii utilizatori – bază de date, precum şi creşterea securităţii bazei de date sunt deziderate ale unui nivel superior de abstractizare, care este nivelul extern. Astfel, structura bazei de date se prezintă sub diferite machete, referite uneori şi *ca sub- scheme*, scheme externe sau imagini (view-uri), în funcţie de nevoile fiecărui utilizator sau grup de utilizatori.

***Observaţii:***

Este importantă această organizare pe trei niveluri pentru că explică conceptul de independenţă a datelor, prin posibilitatea de modificare a sistemului bazei de date la orice nivel fără a avea influenţă la nivelele superioare. Independenţa datelor se poate defini în două moduri, ce sunt aferente nivelelor conceptual şi intern.

Prin ***independenţa logică*** se înţelege capacitatea schimbării schemei conceptuale, fără a atrage după sine schimbări in schema externă sau în programele de aplicaţii. Este posibilă schimbarea schemei conceptuale prin expandarea bazei de date ca urmare a adăugării de noi tipuri de înregistrări sau a datelor însăşi, sau prin reducerea bazei de date ca urmare a reducerii înregistrărilor.

***Independenţa fizică*** este reprezentată prin capacitatea de schimbare a schemei interne fără schimbarea schemei conceptuale sau externe. Schimbarea schemei conceptuale poate surveni ca urmare a reorganizării fizice a unor fişiere, prin crearea de noi structuri de acces menite să asigure accesul eficient la date.

Accesul utilizatorului la informaţiile din baza de date este posibil numai prin intermediul sistemului de gestiune a bazei de date (SGBD).

**1.3 Componente ale bazelor de date relaţionale**

**1. Tabele**

Unitatea primară de stocare a datelor într-o bază de date relaţională este ***tabelul****,* care este o structură bidimensională compusă din rânduri şi coloane. Fiecare tabel reprezintă o *entitate,* ceea ce înseamnă o persoană, un loc, un lucru sau un eveniment care trebuie să fie reprezentat în baza de date, cum ar fi un client, un cont bancar sau o tranzacţie bancară. Fiecare rând al tabelului reprezintă o apariţie a entităţii.

**2. Relaţii**

*Relaţiile* reprezintă asocierile dintre tabelele bazelor de date relaţionale. Deşi fiecare tabel relaţional poate exista independent, esenţa bazelor de date este tocmai stocarea informaţiilor între care există legături. De exemplu, pe lângă filmele propriu-zise, se pot stoca şi informaţii despre categoriile folosite de magazin pentru organizarea inventarelor de filme. În acelaşi timp, puteţi stoca şi informaţii despre copiile fiecărui film, inclusiv data la care a fost primită copia şi formatul acesteia, cum ar DVD sau VHS. Prin folosirea relaţiilor, se pot asocia tabelele înrudite, într-un mod formal, uşor de folosit astfel încât să combinăm date din tabele multiple în aceeaşi interogare a bazei de date, dar păstrând flexibilitatea de a include numai informaţiile care îl interesează pe utilizator. Posibilitatea de a selecta

din baza de date numai informaţiile care ne interesează ne permite să ajustăm informaţiile din baza de date în funcţie de cerinţele specifice ale persoanelor sau aplicaţiilor care au acces la baza de date.

Figura 1.2 prezintă patru tabele din baza de date a magazinului de produse video şi relaţiile dintre acestea, într-un format cunoscut sub numele de diagramă de relaţii a entităţilor (ERD - Entity Relationship Diagram). Diagramele ERD ne pun la dispoziţie o modalitate de prezentare a proiectului general al unei baze de date relaţionale, într-un format uşor de înţeles pentru utilizatorii bazei de date, indiferent dacă au sau nu cunoştinţe tehnice. Fiecare dreptunghi din diagramă reprezintă un tabel relaţional, cu numele tabelului scris deasupra liniei orizontale şi coloanele tabelului enumerate pe

verticală, în porţiunea principală a dreptunghiului.

**MPAA\_RATING** MPAA\_RATING\_CODE (pk) MPAA\_RATING\_DESCRIPTION

**MOVIE** MOVIE\_ID (pk) MOVIE\_GENRE\_CODE (fk1) MPAA\_RATING\_CODE (fk2) MOVIE\_TITLE RETAIL\_PRICE\_VHS RETAIL\_PRICE\_DVD

YEAR PRODUCED

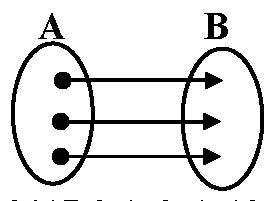
**MOVIE\_GENRE** MOVIE\_GENRE\_CODE (pk) MOVIE\_GENRE\_DESCRIPTI ON

**MOVIE\_COPY** MOVIE\_ID (pk, fk) COPY\_NUMBER (pk) DATE\_ACQUIRED DATE\_SOLD MEDIA\_FORMAT

**Fig. 1.2 Diagrama ERD a bazei de date pentru magazinul de produse video (prezentare parţială)**

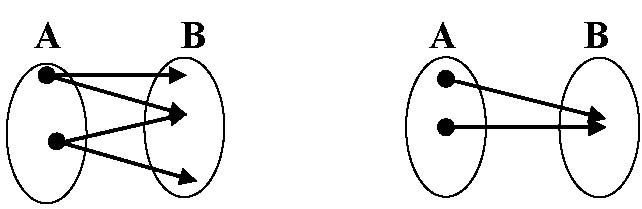
În funcţie de numărul de elemente, între care se stabilesc relaţii, aparţinând celor două colecţii, aceste relaţii pot fi de tip unu la unu, unu la mulţi şi mulţi la mulţi.

**Relaţiile de tipul 1**→**1 (unu la unu)**, care presupun că unui membru din colecţia A îi corespunde un singur membru din colecţia B.



**Relaţie de tip 1 la 1**

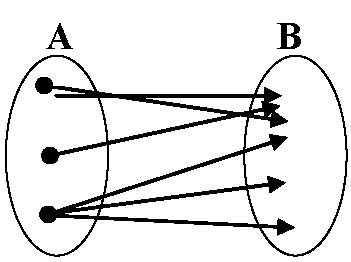
**Relaţiile de tipul 1**→**m sau m**→**1 (unu la mulţi sau mulţi la unu)**, care presupun că unui membru din prima entitate A îi corespund mai mulţi membri din a doua entitate B; astfel de relaţii se mai numesc şi relaţii ierarhice.



**a) b)**

**Relaţie de tip 1 la m (a) şi m la 1 (b)**

**Relaţiile de tipul m**→**m (mulţi la mulţi)**, în care unui membru din entitatea A îi corespund mai multe date din colecţia B şi mai multor date din colecţia A îi corespunde o singură dată din colecţia B.



**Relaţie de tip m la m**

Relaţii de tip mulţi la mulţi se mai numesc şi relaţii de tip reţea. O relaţie mulţi la mulţi se va descompune întotdeauna în două relaţii, o relaţie tip unu la mulţi şi respectiv o a doua relaţie de tip mulţi la unu prin intermediul unei entităţi de legătură.

Fiecare relaţie este prezentată în diagrama ERD ca o linie ce conectează două tabele. Cele două capete ale liniei arată *cardinalitatea maximă* a relaţiei, respectiv numărul maxim de rânduri dintr-un tabel care pot fi asociate cu un rând dat din tabelul aflat la celălalt capăt al relaţiei.

Cardinalitatea maximă poate fi:

• *unu* (caz în care linia nu are nici un simbol special la capăt) sau

• *mai multe* (caz în care linia se termină cu un simbol numit *picior de cioară (crow'sfoot) -* linia se împarte la capăt în trei segmente).

În apropiere de capătul liniei se află un alt simbol, care arată *cardinalitatea minimă,* adică

numărul minim de rânduri dintr-un tabel care poate fi asociat cu tabelul de la celălalt capăt al relaţiei.

Cardinalitatea minimă poate fi *zero,* indicată printr-un cerc desenat pe linie, sau *unu,* indicată

printr-o liniuţă care taie linia relaţiei.

De exemplu, relaţia dintre tabelele MPAA\_RATING şi MOVIE din figura 1-2 este o relaţie de tip un*u-la-mai-mulţi,* ceea ce înseamnă că fiecare rând din tabelul MPAA\_RATING (tabelul din partea

„unu, numit şi tabel *părinte")* poate fi asociat cu mai multe rânduri din tabelul MOVIE (tabelul din partea „mai mulţi" a relaţiei, numit şi tabel *copil'"),* dar fiecare rând din tabelul MOVIE poate fi asociat cu un singur rând din tabelul MPAA\_RATING.

Relaţia este logică, deoarece orice film lansat în SUA poate avea o singură categorie MPAA, dar o categorie poate fi asociată mai multor filme diferite. Este adevărat ca filmele sunt uneori „cenzurate" pentru a putea fi încadrate în diferite categorii, dar această problemă este rezolvată uşor, prin tratarea diferitelor versiuni ale aceluiaşi film ca şi cum ar fi filme diferite, la fel cum facem atunci când un film este reluat cu alţi actori. Este foarte important să ţineţi seama de aceste lucruri, deoarece bazele de date relaţionale acceptă numai relaţiile de tip unu-la-mai-mulţi sau unu-la-unu.

Toate relaţiile din fig. 1.2 sunt obligatorii în partea „unu" şi opţionale în partea „mai mulţi", aceasta fiind cea mai frecvent folosită formă de relaţie. Dacă ne uităm din nou la relaţia dintre tabelele MPAA\_RATING şi MOVIE, aceasta înseamnă că fiecare rând din tabelul MOVIE *trebuie* să aibă un rând corespondent în tabelul MPAA\_RATING, dar nu este obligatoriu ca fiecare rând din tabelul MPAA\_RATING să aibă asociat un rând din tabelul MOVIE. Dacă vreţi să permiteţi ca inventarul de filme al magazinului să conţină titluri care nu au asociată o categorie MPAA, liniuţa de la capătul dinspre tabelul MPAA\_RATING al liniei care reprezintă relaţia cu tabelul MOVIE va fi înlocuită de un cerc. Deşi sunt relativ frecvent întâlnite cazurile în care partea „unu" a unei relaţii nu este obligatorie, este foarte neobişnuit să aveţi o relaţie în care să fie obligatorie partea „mai mulţi" a relaţiei, ceea ce ar însemna că tabelul părinte trebuie să aibă în orice moment cel puţin un „copil" în baza de date.

**Relaţiile sunt implementate folosind coloane corespondente din cele două tabele participante. În diagrama ERD, coloana sau coloanele subliniate din fiecare tabel, având în dreapta notaţia „pk", reprezintă cheia primară (primary key), adică o coloană sau un set de coloane care identifică în mod unic fiecare rând dintr-un tabel.**

Un tabel poate avea o singură cheie primară. Totuşi, o cheie primară poate fi compusă din mai multe coloane, dacă aceasta este calea de formare a unei chei unice. Dacă o cheie primară este folosită într-un alt tabel pentru stabilirea unei relaţii, poartă numele de cheie externă.

Cheile primare şi cheile externe sunt blocuri de construcţie fundamentale ale modelului relaţional, deoarece stabilesc relaţii şi permit crearea legăturilor între date, atunci când este necesar. Trebuie să înţelegeţi acest concept pentru a putea înţelege cum funcţionează bazele de date relaţionale.

**3. Restricţii**

O restricţie este o regulă specificată pentru un obiect al bazei de date (de obicei, un tabel sau o coloană), având rolul de a limita într-un mod oarecare domeniul de valori permise pentru obiectul respectiv al bazei de date

După ce sunt specificate, restricţiile sunt impuse automat de sistemul DBMS şi nu pot fi ocolite decât dacă o persoană autorizată le dezactivează sau le şterge (le elimină). Fiecare restricţie primeşte un nume unic, astfel încât să poată fi referită în mesajele de eroare şi în comenzile folosite ulterior în baza de date. Este recomandabil ca proiectanţii bazei de date să denumească restricţiile, deoarece numele generate automat de baza de date nu sunt foarte descriptive.

Există mai multe tipuri de restricţii pentru baze de date:

• ***Restricţia NOT NULL***. Poate fi plasată pe o coloană pentru a împiedica folosirea valorilor nule. O valoare nulă (null) este o modalitate specială prin care sistemul RDBMS tratează valoarea unei coloane pentru a indica faptul că valoarea coloanei respective nu este cunoscută. O valoare nulă nu este acelaşi lucru cu un spaţiu liber, un şir vid sau valoarea zero — este o valoare specială care nu este egală cu nimic altceva.

• ***Restricţia cheie primară (primary key)****.* Definită pe coloana (coloanele) cheie primară ale unui tabel pentru a garanta că valorile cheie primară sunt întotdeauna unice în întreg tabelul. Atunci când cheia primară este definită pe mai multe coloane, combinaţia valorilor acelor coloane trebuie să fie unică în tabel - o coloană care reprezintă doar o parte a cheii primare poate conţine valori duplicate în tabel. Restricţiile cheie primară sunt aproape întotdeauna implementate de RDBMS prin folosirea unui index. Indexul este un tip special de obiect al bazei de date care permite efectuarea căutărilor rapide în valorile coloanei. Atunci când în

asigura că ***pk*** a noului rând nu este deja folosită în tabel şi, dacă se întâmplă acest lucru, respinge cererea de inserare. Căutarea în indexuri se face mult mai repede decât căutarea în tabel; ca urmare, indexarea cheii primare este esenţială pentru orice tabel, indiferent de dimensiunea acestuia, astfel încât căutarea cheilor duplicate la fiecare inserare să nu ducă la o reducere semnificativă a performanţelor. O caracteristică suplimentară a cheilor primare este faptul că nu pot fi definite decât pe coloane pentru care a fost definită şi restricţia NOT NULL.

• ***Restricţia de unicitate (unique)****.* Definită pe o coloană sau un set de coloane care trebuie să conţină valori unice în cadrul tabelului. Ca şi în cazul cheilor primare, sistemul RDBMS foloseşte aproape întotdeauna un index ca modalitate de impunere eficientă a restricţiei, Totuşi, spre deosebire de cheile primare, un tabel poate avea definite mai multe restricţii de unicitate, iar coloanele care participă la o restricţie de unicitate pot conţine (în cele mai multe sisteme RDBMS) şi valori nule.

• ***Restricţia referenţială*** (numita uneori *restricţie de integritate referenţială).*

O restricţie care impune o relaţie între două tabele dintr-o bază de date relaţională. Prin

„impunere„ se înţelege că sistemul RDBMS se asigură întotdeauna, în mod automat, că fiecărei valori a cheii externe îi corespunde o valoare a cheii primare în tabelul părinte. Pe scurt, restricţia referenţială garantează că relaţia dintre cele două tabele şi valorile corespondente ale cheii primare şi cheii externe îşi păstrează logica în orice moment.

• ***Restricţia CHECK.*** Foloseşte o instrucţiune logică simplă (scrisă în SQL) pentru a valida valoarea unei coloane. Rezultatul instrucţiunii trebuie să fie o valoare logică de adevărat (true) sau fals (false), astfel încât un rezultat adevărat să permită inserarea în tabel a valorii coloanei, iar un rezultat fals să ducă la rejectarea valorii coloanei, cu mesajul de eroare corespunzător.

**4.Vizualizări**

O vizualizare (view) este o interogare stocată în baza de date care pune la dispoziţia utilizatorului un subset personalizat al datelor din unul sau mai multe tabele ale bazei de date. Cu alte cuvinte, o vizualizare este un tabel virtual, deoarece arată ca un tabel şi, în cele mai multe privinţe, se comportă ca un tabel, dar nu stochează date (nu este stocată decât interogarea SQL care defineşte vizualizarea). Vizualizările au mai multe funcţii utile:

• Maschează coloanele pe care utilizatorul nu este nevoie să le vadă (sau nu-i este permis să le vadă).

• Maschează rândurile pe care utilizatorul nu este nevoie să le vadă (sau nu-i este permis să le vadă).

• Maschează operaţiile complexe efectuate în baza de date, cum ar fi uniunile de tabele (respectiv combinarea coloanelor din tabele multiple într-o singură interogare a bazei de date).

• Îmbunătăţesc performanţele interogărilor (în unele sisteme RDBMS precum Microsoft SQL Server).

**1.4 Proiectarea bazelor de date relaţionale. Etape. Normalizarea bazelor de date**

Proiectarea unei baze de date este o activitate laborioasă şi necesită parcurgerea următoarelor etape:

• formularea problemei;

• analiza cerinţelor informaţionale şi definirea datelor de ieşire şi a datelor de intrare;

• definirea tabelelor, a structurii acestora şi a relaţiilor dintre tabele;

• optimizarea structurii bazei de date.

Odată ce acest proces a fost finalizat se continuă cu:

• proiectarea procedurilor tehnologice, pentru prelucrarea bazei de date;

• elaborarea programelor;

• testarea programelor;

• definitivarea documentaţiei.

Toate aceste activităţi necesită, pentru proiectele reale complexe, o muncă în echipă pe baza unei metodologii riguroase, cunoscută ca metodologia de analiză şi proiectare a sistemelor informatice. În cadrul unui sistem informatic baza de date reprezintă elementul central în jurul căruia se concentrează celelalte componente ale sistemului.

**Formularea problemei** presupune stabilirea obiectivelor aplicaţiei informatice care asigură actualizarea şi exploatarea bazei de date în concordanţă cu cerinţele managementului activităţii economice pentru care este proiectată baza de date. Obiectivele unei aplicaţii informatice sunt legate de asigurarea informaţională a desfăşurării proceselor decizionale specifice actului de conducere. Deci, noi trebuie să ne gândim că, prin existenţa unei baze de date, să asigurăm fondul de informaţii, într-o structură şi de o calitate corespunzătoare cu cerinţele managementului firmei. Baza de date trebuie să permită atât obţinerea unor informaţii de detaliu, elementare, cât şi calculul şi prezentarea unor indicatori sintetici, agregaţi. Dacă am lua doar două obiective: reducerea costurilor şi creşterea productivităţii muncii într-o firmă, atunci o bază de date trebuie să furnizeze informaţii despre consumul factorilor de producţie, costurile medii şi globale, despre personalul muncitor şi producţia realizată, despre cheltuielile salariale etc. Aceste informaţii vor servi conducerii la identificarea căilor de reducere a costurilor şi adoptarea celor mai adecvate măsuri pentru reducerea acestor costuri. După aplicarea măsurilor în practică, informaţiile stocate în baza de date trebuie să permită de data aceasta şi o analiză comparată a costurilor noi cu cele vechi, de exemplu, o analiză a dinamicii costurilor pe baza indicilor statistici. Am ales acest mic exemplu didactic pentru a accentua încă o dată complexitatea procesului de proiectare a bazei de date.

**Analiza cerinţelor informaţionale**, pornind de la obiectivele formulate anterior, se concentrează

asupra a două probleme:

• indicatorii, rapoartele, listele şi datele de ieşire care trebuie obţinute;

• datele de intrare necesare pentru obţinerea datelor de ieşire.

Acestea sunt cerinţele informaţionale care înglobează atât cerinţele pentru datele de intrare pe baza cărora se creează şi se actualizează baza de date, cât şi cerinţele pentru datele de ieşire folosite pentru urmărirea, controlul şi dirijarea activităţii economice. **Datele de intrare se culeg, de regulă, din documentele primare** care circulă în cadrul fluxului informaţional al firmei. Datele finale se vor integra în ansamblul de rapoarte, liste, situaţii cu rezultate pe care le furnizează sistemul informaţional

compartimentelor de conducere. Pentru indicatorii incluşi în rapoartele finale, în general pentru oricare din indicatorii de ieşire, trebuie să fie foarte clar modul în care sunt obţinuţi prin prelucrarea datelor de intrare. În consecinţă, acolo unde este cazul, se precizează algoritmii de calcul, regulile de totalizare, sau alte reguli de obţinere a fiecărei coloane, sau totaluri din rapoartele finale.

Aceasta are ca punct de plecare inventarierea câmpurilor prezente în situaţiile finale şi apoi gruparea lor în tabele. Gruparea câmpurilor pe tabele se realizează prin diverse metode. Dintre aceste metode, două sunt cele mai utilizate:

• analiza concordanţei IEŞIRI – INTRĂRI;

• analiza semnificaţiei semantice a datelor.

*Analiza concordanţei IEŞIRI – INTRĂRI* este o tehnică specifică proiectării sistemelor informatice care identifică documentele primare din care se preiau datele de intrare folosite în calculul datelor de ieşire. Aceste documente vor constitui sursele de creare şi actualizare a tabelelor bazei de date.

Tehnica este utilă în cazul în care proiectantul bazei de date este familiarizat cu sistemul informaţional existent.

Un indicator prezent într-o situaţie finală se poate obţine astfel:

• prin preluarea directă din documentul primar, respectiv din tabelul bazei de date;

• prin aplicarea unui algoritm de calcul.

Tabelele se compun prin gruparea câmpurilor pe principiul apartenenţei acestora la anumite documente primare care circulă în cadrul sistemului informaţional.

*Analiza semantică* se practică atunci când proiectantul bazei de date nu are la dispoziţie un set de documente primare şi apare în cazul sistemelor informaţionale care se proiectează pentru firmele noi.

**Definirea tabelelor şi relaţiilor dintre tabele** este etapa următoare în proiectarea bazei de date. Analiza cerinţelor informaţionale şi a proceselor de prelucrare va conduce la identificarea datelor ce vor trebui stocate şi care vor alcătui tabelele bazei de date. O tabelă va păstra datele fie despre toate caracteristicile unei colecţii de date, fie numai pentru o parte dintre aceste caracteristici. Aici intervine spiritul analitic al proiectantului. Structura unei tabele este reprezentată de lista câmpurilor asociate tabelei împreună cu descrierea atributelor fiecărui câmp (natură, lungime, număr de zecimale etc.). În structura unui tabel se regăsesc următoarele categorii de câmpuri:

• câmpuri de identificare (chei primare şi chei condiţionate);

• câmpuri tip dată calendaristică;

• câmpuri cantitativ-valorice;

• câmpuri de legătură cu alte tabele;

• câmpuri de stare care păstrează informaţii privind ultimele operaţii de prelucrare care au fost efectuate pe înregistrările din tabel.

Relaţiile dintre tabele se caracterizează prin plasarea unor câmpuri comune în structura fiecăruia dintre tabelele aflate în relaţie directă. Pe baza acestor câmpuri, chei, fiecare sistem de gestiune a bazelor de date îşi construieşte un mecanism propriu de accesare a înregistrărilor de date. Aceste mecanisme sunt transparente pentru utilizatorul obişnuit. Totuşi este bine de reţinut că nu orice câmp poate fi folosit la stabilirea unei relaţii, a unei legături între două tabele. Numai câmpurile de tip cheie candidat, care au proprietatea de a identifica în mod unic o înregistrare dintr-o tabelă, pot fi folosite în acest scop. Cheile candidate se mai numesc şi indecşi. **Operaţia prin care se construieşte sistemul de legături pentru ordonarea în vederea regăsirii înregistrărilor într-o tabelă se numeşte indexare**. Prin indexare, fiecărei tabele principale de date i se va asocia o tabelă index corespunzătoare cheilor deindexare.

**Optimizarea structurii bazei de date** este un proces prin care se urmăreşte:

• reducerea redundanţei datelor;

• eliminarea anomaliilor de actualizare.

*Reducerea redundanţei datelor* până la un nivel minim şi controlat urmăreşte eliminarea duplicării inutile a unor câmpuri în mai multe tabele sau eliminarea câmpurilor obţinute prin calcul pe baza câmpurilor atomice. Un anumit nivel de redundanţă, însă, trebuie admis pentru a nu denatura realitatea reflectată de date. De exemplu, câmpul **VALOAREA\_CONTRACTULUI**, se calculează după relaţia: **VALOAREA\_CONTRACTULUI=CANTITATE\*PRET**

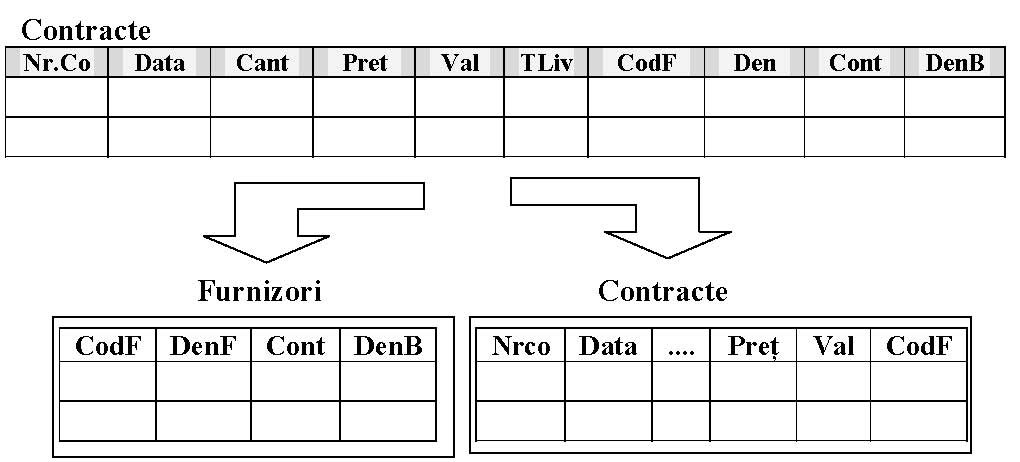
Nu este recomandată eliminarea acestui câmp pe considerentul că el se obţine automat prin

calcul.

***De exemplu***, în cazul în care după un anumit interval de timp, preţurile suportă o majorare

globală, cum se practică foarte des, atunci automat se vor modifica şi valorile contractelor încheiate anterior datei de majorare a preţurilor ori acest lucru nu este corect, contractul odată perfectat nu-şi poate modifica preţul convenit prin negociere.

*Anomaliile de actualizare* se referă la anomaliile de ştergere, respectiv de modificare. De exemplu, se consideră o firmă care derulează lunar mii de contracte de aprovizionare, pentru un nomenclator foarte mare de produse agroalimentare, dar care operează numai cu câţiva furnizori. Dacă în tabela **CONTRACTE** sunt incluse alături de codul furnizorului şi denumirea furnizorului, contul său bancar şi denumirea băncii, atunci va apărea următoarea anomalie de actualizare, în cazul schimbării băncii şi a contului bancar de către furnizor. Acest lucru necesită parcurgerea tuturor înregistrărilor în care există aceste valori şi actualizarea acestora (figura 1.4).



**Fig. 1.4 Eliminarea anomaliilor de actualizare**

Soluţia este de a crea o tabelă separat, care are în structură: codul furnizorului, denumirea, contul şi banca acestuia, în tabela **CONTRACTE** păstrându-se doar codul furnizorului ca element de legătură, ceea ce previne pierderea de informaţii prin spargerea unui tabel în două. În acest fel modificarea se realizează numai asupra unei singure înregistrări.

În 1972, Dr. E. F. Codd, părintele bazelor de date relaţionale, şi-a dat seama că tabelele relaţionale care îndeplinesc anumite criterii pun mai puţine probleme la inserarea, actualizarea sau ştergerea datelor. Ca urmare, a pus la punct un set de reguli care trebuie respectate (organizate în trei

„forme normale") şi un proces numit normalizare, care este o tehnică pentru producerea unui set de relaţii (termenul folosit de Dr. Codd pentru tabele) cu proprietăţile dorite.

**Necesitatea normalizării**

Figura 1.5 prezintă tabelul **MOVIE** fără normalizare, aşa cum ar arăta dacă toate informaţiile despre filme ar fi colectate într-un singur tabel. Acest exemplu va fi folosit pentru ilustrarea procesului de normalizare.

În general, numele coloanelor din tabelele relaţionale folosesc liniuţe de subliniere pentru separarea cuvintelor. În discuţia despre normalizare am eliminat aceste liniuţe din figuri, pentru a face textul mai uşor de citit.

Există trei probleme care pot apărea în tabelele fără normalizare din bazele de date relaţionale. Scopul procesului de normalizare este de a elimina aceste probleme (anomalii) din proiectul bazei de date. Acestea sunt:

- Anomalia de inserare

- Anomalia de ştergere

- Anomalia de actualizare

**Anomalia de inserare**

*Anomalia de inserare* se referă la o situaţie în care nu puteţi insera date în baza de date din cauza unei dependenţe artificiale dintre coloanele unui tabel.

**Anomalia de ştergere**

*Anomalia de ştergere* este inversul anomaliei de inserare. Se referă la situaţia în care ştergerea unor date duce la pierderea neintenţionată a altor date.

**Anomalia de actualizare**

*Anomalia de actualizare* se referă la o situaţie în care actualizarea unei singure valori necesită actualizarea mai multor rânduri. Un alt pericol legat de această anomalie este faptul că stocarea unor date redundante poate duce la posibilitatea de a actualiza numai o parte a copiilor respectivelor date, ceea ce ar avea ca rezultat apariţia inconsecvenţelor în baza de date.

**Aplicarea procesului de normalizare**

De obicei, normalizarea începe de la mijloacele de redare a datelor care sunt (sau vor fi) prezentate utilizatorilor, cum ar fi pagini web, ecrane ale aplicaţiilor, rapoarte şi aşa mai departe. Colectiv, acestea sunt numite *vizualizări de utilizator (user views).* Poate părea ciudat la prima vedere, dar este ceva obişnuit ca proiectarea unui sistem de prelucrare a datelor să înceapă de la rezultatele pe care le va vedea utilizatorul, parcurgând apoi drumul înapoi către mijloacele folosite pentru obţinerea rezultatelor dorite.

În timpul proiectării bazei de date, procesul de normalizare este aplicat fiecărei vizualizări, iar rezultatul este un set de relaţii normalizate care pot fi apoi direct implementate ca tabele ale bazei de date relaţionale. Procesul în sine este destul de simplu, iar regulile nu sunt foarte dificile. Totuşi, stăpânirea procesului de normalizare cere timp şi exerciţiu, în special deoarece impune proiectantului să se gândească într-un mod conceptual la datele şi relaţiile pe care intenţionează sa le folosească.

În timpul normalizării, consideraţi că fiecare vizualizare este o relaţie. Cu alte cuvinte, conceptualizaţi fiecare vizualizare ca şi cum ar fi deja un tabel bidimensional - un mod de lucru pentru care aveţi nevoie de experienţă.

Reţineţi că scopul procesului de normalizare este eliminarea anomaliilor de inserare, actualizare

şi ştergere. Procesul determină crearea unui număr mai mare de relaţii decât aţi avea într-un model fără

normalizare. Relaţiile suplimentare sunt necesare pentru eliminarea anomaliilor, dar împărţirea datelor în mai multe relaţii face ca extragerea datelor stocate să fie puţin mai dificilă. De fapt, sacrificaţi o parte din performanţele de extragere a datelor şi din uşurinţa utilizării pentru ca operaţiile de inserare, actualizare şi ştergere să fie mai simple.

**Alegerea unul identificator unic**

Primul pas al procesului de normalizare constă în alegerea unui *identificator unic (unique identifier),* care este un atribut (o coloană) sau un set de atribute care identifică în mod unic fiecare rând de date dintr-o relaţie.

Identificatorul unic va deveni ulterior cheia primară a tabelului creat din relaţia normalizată. Pentru normalizare, *este obligatoriu ca* fiecare relaţie să aibă un identificator unic, în multe cazuri, puteţi găsi un atribut care identifică în mod unic datele din fiecare rând al relaţiei pe care vreţi să o normalizaţi. Atunci când nu puteţi găsi un singur atribut care să poată fi folosit ca identificator unic, este posibil să găsiţi mai multe atribute care pot fi concatenate (combinate) pentru a forma un identificator unic. Atunci când identificatoarele unice sunt formate din atribute multiple, fiecare atribut rămâne pe propria lui coloană - nu faceţi decât să definiţi un identificator unic format din mai multe coloane.

În foarte puţine cazuri, într-o relaţie nu există un set rezonabil de atribute care să poată fi folosit ca identificator unic. Atunci când se întâmplă acest lucru, trebuie să inventaţi un identificator unic, deseori cu valori atribuite secvenţial sau aleatoriu pe măsură ce noile rânduri de date sunt adăugate în tabelul bazei de date. Această tehnică este sursa unor identificatoare unice, precum numărul de asigurări sociale folosit în Statele Unite, numerele de identificare ale angajaţilor, numerele de înmatriculare ale maşinilor sau CNP.

**Prima formă normală: eliminarea datelor repetate**

O relaţie este în *prima formă normală* atunci când nu conţine atribute cu valori multiple (atribute multivaloare), adică atribute care au mai multe valori pentru acelaşi rând de date. Într-o relaţie, orice intersecţie a unui rând cu o coloană trebuie să conţină *cel mult* o valoare pentru ca relaţia să fie în prima formă normală.

Pentru transformarea relaţiilor ne-normalizate în prima formă normală, trebuie mutate atributele multivaloare şi grupurile repetitive în noi relaţii.

Procedura de mutare a unui atribut multivaloare sau a unui grup repetitiv într-o nouă relaţie constă în următoarele etape:

1. Creaţi o nouă relaţie, cu un nume sugestiv. Deseori, este bine să includeţi numele relaţiei originale, parţial sau în întregime, în numele noii relaţii.

2. Copiaţi identificatorul unic din prima relaţie în noua relaţie. Datele depind de acest identificator în relaţia originală, aşa că trebuie să depindă de aceeaşi cheie şi în noua relaţie. Identificatorul copiat va deveni cheie externă în noua relaţie.

3. Mutaţi grupul repetitiv sau atributul multivaloare în noua relaţie.

4. Formaţi un identificator unic în noua relaţie, adăugând atribute la identificatorul unic copiat din relaţia originală. Ca întotdeauna, asiguraţi-vă ca identificatorul unic nou format conţine numai numărul minim de atribute necesar pentru a-1 face unic. Dacă mutaţi un atribut multivaloare, care, în esenţă, este un grup repetitiv cu un singur atribut, este adăugat atributul respectiv pentru formarea identificatorului unic. Poate părea ciudat la prima vedere, dar identificatorul unic copiat din relaţia originală nu este doar o cheie externă, ci, de obicei, şi o parte a identificatorului unic (cheia primară) a

noii relaţii. Acest lucru este absolut normal. De asemenea, este perfect acceptabil să avem o relaţie în care toate atributele fac parte din identificatorul unic (adică nu există atribute care să nu facă parte din cheie).

5. Opţional, puteţi să înlocuiţi cheia primară cu un singur atribut surogat pentru cheie. Dacă faceţi acest lucru, trebuie să păstraţi şi atributele care compun cheia primară naturală, formată la paşii 2 şi 4.

**A doua formă normală: eliminarea dependenţelor parţiale**

Înainte de a explora a doua formă normală, trebuie să înţelegem conceptul *de dependenţă funcţională.*

*Pentru* această definiţie, vom folosi două atribute arbitrare, inteligent denumite „A" şi „B". Atributul B este *dependent funcţional* de atributul A dacă în nici un moment nu există mai mult de o valoare a atributului B asociată cu o valoare dată a atributului A.

Se spune că o relaţie este în *a doua formă normala* dacă îndeplineşte următoarele criterii:

• Relaţia este în prima formă normală.

• Toate atributele non-cheie sunt dependente funcţional de identificatorul unic

(cheia primară), *luat ca întreg.*

În esenţă, amestecăm atribute care descriu în aceeaşi relaţie două lucruri (entităţi) diferite (deşi înrudite) din lumea reală. Nici nu e de mirare că am obţinut un asemenea haos. A doua formă normală ne va ajuta să rezolvăm problemele.

A doua formă normală se aplică numai relaţiilor care au identificatoare unice concatenate (adică formate din atribute multiple). Într-o relaţie care are un singur atribut ca identificator unic, este imposibil ca un alt atribut să depindă de o parte a identificatorului unic, deoarece acesta, fiind format dintr-un singur atribut, nu are părţi componente. **Ca urmare, orice relaţie în prima formă normală care are cheia primară formată dintr-un singur atribut este *automat în* a doua formă normală.**

**A treia formă normală: eliminarea dependenţelor tranzitive**

Pentru a înţelege a treia formă normală, trebuie să înţelegem mai întâi conceptul de dependenţă tranzitivă.

Despre un atribut care depinde de un atribut care nu este identificator unic (cheie primară) a relaţiei se spune că *este dependent tranzitiv.*

Se spune că o relaţie este în *a treia formă normală* dacă îndeplineşte următoarele două criterii:

• Relaţia este în a doua formă normală.

• Nu există dependenţe tranzitive (cu alte cuvinte, toate atributele non-cheie depind

*numai* de identificatorul unic).

Pentru a aduce la a treia formă normală o relaţie aflată în a doua formă normală, mutăm atributele dependente tranzitiv în relaţii în care depind numai de cheia primară Avem grijă să lăsăm atributul de care depind acestea în relaţia originală, cu rolul de cheie externă. Va trebui apoi să reconstruim vizualizarea originală printr-o uniune. Ca efect secundar, toate atributele uşor de calculat sunt eliminate ca încălcări ale criteriilor celei de-a treia forme normale.De exemplu, într-o bază de date pentru vânzări, Suma Totală este obţinută înmulţind Cantitatea Cumpărată cu Preţul Unitar; aşa cum se observă cu uşurinţă, Suma Totală este dependentă de Cantitatea Cumpărată şi de Preţul Unitar. Presupunând că toate cele trei atribute sunt dependente de identificatorul unic al relaţiei care le conţine, este uşor de văzut că Suma Totală (rezultatul calculat)este, de fapt, *dependentă tranzitiv* de celelalte două atribute.

**Limbajul SQL(  *Structured Query Language )***

**Caracteristici generale**

Limbajul SQL este un limbaj declarativ; utilizatorul descrie informaţiile pe care vrea să le obţină în urma interogării, fără a preciza algoritmii necesari pentru obţinerea rezultatelor dorite. În 1986 SQL a devenit standard ANSI (SQL-86). În ultimii ani, comitetele ANSI şi ISO pentru standardizarea limbajului SQL au adăugat noi facilităţi pentru gestiunea bazelor de date orientate pe obiecte şi obiect-relaţional. S-a ajuns astfel la o versiune SQL3, pe cale de a deveni un standard în domeniu. Aceste facilităţi se pot structura:

• facilităţi orientate obiect ce propun definirea la nivel de utilizator a tipurilor de date abstracte;

• structuri de control specifice: IF, FOR, WHILE ce permit transformarea SQL-ului într-un limbaj de sine stătător;

• facilităţi de comunicare în reţea;

• facilităţi de prelucrare distribuită;

• facilităţi multi-media, înglobate în modulul Multi - Media SQL.

**Tipuri de instrucţiuni**

În funcţie de rolul lor în manipularea datelor şi tranzacţiilor, instrucţiunile SQL, pot fi grupate astfel:

**a)** instrucţiuni de definire a datelor care permit descrierea structurii bazei de date;

**b)** instrucţiuni de manipulare a datelor în sensul adăugării, modificării şi ştergerii înregistrărilor;

**c)** instrucţiuni de selecţie a datelor care permit consultarea bazei de date;

**d)** instrucţiuni de procesare a tranzacţiilor care privesc unităţile logice de prelucrare şi constituie în fapt, operaţii multiple de manipulare a datelor;

**e)** instrucţiuni de control al cursorului;

**f)** instrucţiuni privind controlul accesului la date.

**Instrucţiuni pentru selectarea datelor- formatul frazei SELECT**

SELECT [ALL/DISTINCT/DISTINCTROW] listă atribute

FROM listă-tabele

[WHERE criteriu-de-căutare]

[GROUP BY criteriu-de-grupare] [HAVING criteriu-de-grupare]

[ORDER BY criteriu-de-ordonare [ASC|DESC]];

Unde:

lista-atribute specifică atributele ale căror valori vor fi returnate; FROM listă-tabele specifică tabelele din care se vor extrage datele;

WHERE permite prin precizarea unei expresii exprimarea criteriului de selecţie;

ORDER BY precizează atributul după care se va face ordonarea;

GROUP BY folosit pentru a partiţiona o tabelă în grupuri acordând acestora valori pe un atribut sau listă de atribute;

HAVING specifică criterii de selecţie pe grupuri de tupluri.

Toate rindurile tabelei *Produse*:

**SELECT DISTINCTROW** \* **FROM** PRODUSE;

Mumele şi preţul produselor având preţul mai mare de 10000:

**SELECT DISTINCTROW** [DEN PROD], PREŢ **FROM** PRODUSE **WHERE** PREŢ>10000;

În cazul în care doriţi să cunoaşteţi denumirea produselor cu preţuri între 19000 şi 110000 lei puteţi fie să

introduceţi o expresie compusă în clauza WHERE, fie să utilizaţi operatorul **between**:

**SELECT** [DEN PROD] **FROM** PRODUSE **WHERE** PREŢ >=19000 AND PREŢ=<110000;

sau:

**SELECT** [DEN PROD] FROM PRODUSE **WHERE** PREŢ BETWEEN 19000 AND 110000;

În cazul în care doriţi să cunoaşteţi produsele ale căror preţuri sunt fie 110000, fie 150000, în clauza

WHERE se va folosi operatorul **IN**:

**SELECT** [DEN PROD] **FROM** PRODUSE **WHERE** PREŢ IN (110000,150000);

Poate fi utilă solicitarea ordonării rindurilor după calitate şi preţ, pornind de la preţul cel mai mare din grupa de calitate respectivă:

**SELECT DISTINCTROW** \* **FROM** PRODUSE **ORDER BY** CALITATE, PREŢ DESC;

În cazul în care se doreşte aflarea preţului maxim, minim şi mediu al produselor, blocul de cereri va utiliza funcţiile corespunzătoare:

**SELECT DISTINCTROW** MAX(PREŢ) AS [PREŢ MAX], MIN(PREŢ) AS [PREŢ MIN], AVG(PREŢ) AS [PREŢ MEDIU]

**FROM** PRODUSE;

Prin clauza AS**,** în exemplul de mai sus, am putut modifica eticheta coloanelor ce returnează valoarea funcţiilor specificate.

Dacă doriţi să stabiliţi preţul maxim al produselor grupându-le pe calităţi utilizaţi în blocul de cerere clauza

GROUP BY:

**SELECT DISTINCTROW** MAX(PREŢ) AS [PREŢ MAXIM], CALITATE

**FROM** PRODUSE

**GROUP BY** CALITATE;

De cele mai multe ori, însă, este necesar să extragem date stocate în mai multe tabele. Să presupunem că dorim să cunoaştem în ce cantităţi am livrat produse clienţilor şi care este valoarea facturată a fiecărui produs:

**SELECT DISTINCTROW** PRODFACT.[NR FACT], PRODUSE.[DEN PROD], PRODFACT.[CANT FACT], PRODUSE.PREŢ, PRODUSE.PREŢ\*PRODFACT.[CANT FACT] AS VALOARE

**FROM** PRODUSE, PRODFACT

**WHERE** PRODUSE.[COD PRODUS]=PRODFACT.[COD PRODUS];

Observaţi în acest bloc de cerere utilizarea calificării atributelor (precedarea numelui lor de numele tabelei căreia îi aparţin) şi precizarea în clauza FROM a numelor celor două tabele sursă. Criteriul de compunere al celor două tabele este precizat în clauza WHERE.

**Interogări cu subinterogări**

Realizarea unor căutări mai complexe în baza de date presupune construirea unor subcereri în cadrul cererii iniţiale. Să presupunem că dorim să cunoaştem care sunt produsele cu preţul mai mare decât al merelor:

**SELECT DISTINCTROW** [DEN PROD], PREŢ

**FROM** PRODUSE

**WHERE** PREŢ > (**SELECT** PREŢ **FROM** PRODUSE **WHERE** [DEN PROD]=’MERE’);

***Utilizarea predicatelor ALL, ANY, EXIST***

Domeniul de obţinere a rezultatelor unei subinterogări poate fi influenţat prin precizarea unuia din cuvintele cheie: ALL, ANY şi respectiv EXIST.

 **ALL**

Se preiau rezultatele subinterogării şi, dacă acestea îndeplinesc condiţia cerută, se returnează valoarea logică

True.

*Exemplu* :

Dacă dorim să cunoaştem numele produselor de calitatea întâi cu preţul mai mic decât al tuturor produselor de calitatea a 2-a cererea va avea următoarea formă:

**SELECT** [DEN PROD] **FROM** PRODUSE

**WHERE** CALITATEA=’1’AND PREŢ< **ALL** (**SELECT** PREŢ **FROM** PRODUSE **WHERE** CALITATEA=’2’);

 **ANY**

Are în vedere compararea valorii de ieşire a subinterogării cu fiecare înregistrare din interogarea externă. Dacă pentru fiecare înregistrare din interogare există un rezultat al subinterogării, se va returna valoarea logică True. Spre deosebire de clauza IN poate fi folosit cu diverşi operatori relaţionali. Cuvântul cheie SOME are acelaşi rol şi caracteristici ca ANY.

*Exemplu*:

Să se returneze lista cărţilor din bibliotecă care aparţin unuia din domeniile abordate de editura ABC în anul

2000:

**SELECT** TITLUL\_CARTII, DOMENIUL, ANUL\_APARITIEI, PRET

**FROM** CĂRŢI

**WHERE** DOMENIUL= ANY (**SELECT** DOMENIUL **FROM** CĂRŢI **WHERE** EDITURA=”ABC” AND [ANUL APARIŢIEI]=2000);

Înlocuirea lui ANY cu SOME produce un rezultat identic:

**SELECT** TITLUL\_CARTII, DOMENIUL, ANUL\_APARITIEI, PREŢ

**FROM** CARTI

**WHERE** DOMENIUL= **SOME** (**SELECT** DOMENIUL **FROM** CARTI **WHERE** DOMENIUL ="INFORMATICA");



**EXISTS**

Foloseşte subinterogarea ca pe o condiţie, analizând setul de rezultate al acesteia şi returnând valoarea False dacă nu există nici o ieşire. Se poate astfel verifica existenţa anumitor înregistrări şi controla ansamblul răspunsurilor date de interogare.

*Exemplu*:

Să se verifice dacă în cadrul bibliotecii există şi cărţi apărute înainte de anul 2001:

**SELECT** TITLUL\_CĂRŢII, ANUL\_APARIŢIEI

**FROM** CĂRŢI

**WHERE EXISTS** (**SELECT** \* **FROM** CĂRŢI **WHERE** ANUL\_APARIŢIEI < 2001);

Fiecare dintre cuvintele cheie ANY, ALL şi EXISTS poate să fie utilizat pentru restricţionarea interogărilor şi în forma cu negaţie – adică prin folosirea operatorului logic NOT.

***Interogări folosind operatorul UNION***

Când utilizatorul doreşte să vadă rezultatele mai multor interogări SELECT în acelaşi timp, prin combinarea ieşirilor lor, poate utiliza UNION:

SELECT lista\_campuri FROM tabela1

UNION SELECT listă\_campuri FROM tabela2

[GROUP BY camp\_de\_grupare] [HAVING criteriul\_de\_agregare]

[UNION SELECT listă\_campuri FROM tabela3

[GROUP BY camp\_de\_grupare ] [HAVING criteriul\_de\_grupare]]

[UNION...]

[ORDER BY camp\_criteriu\_de\_sortare];

Există mai multe restricţii pentru instrucţiunile care generează interogări UNION şi anume: numărul de câmpuri din lista de câmpuri asociată fiecărei instrucţiuni SELECT şi UNION SELECT trebuie să fie acelaşi; este permisă doar o dată utilizarea clauzei ORDER BY, după ultima instrucţiune UNION SELECT; secvenţa de nume din fiecare listă de câmpuri trebuie să corespundă unor intrări identice. Când se foloseşte UNION, automat se vor elimina duplicatele ce apar în urma combinării. In cazul folosirii domeniului ALL se vor lua în considerare şi valorile duplicate.

*Exemple*:

a) Care a fost media salariilor colaboratorilor studenţi şi ingineri ?

**SELECT** CATEGORIE, AVG(SALARIU) AS MEDIE **FROM** COLABORATORI

**GROUP BY** CATEGORIE

**HAVING** CATEGORIE = "STUDENT"

**UNION SELECT** CATEGORIE, AVG(SALARIU) AS MEDIE **FROM** COLABORATORI

**GROUP** BY CATEGORIE

**HAVING** CATEGORIE = "ING";

b) Fie tabelele *Clienţi* (cod, nume, localitate) şi *Furnizori* (cod, nume, localitate). Prin utilizarea operatorului

UNION putem obţine lista comună a clienţilor şi furnizorilor:

**SELECT** \* **FROM** CLIENŢI

**UNION**

**SELECT** \* **FROM** FURNIZORI

**ORDER BY** NUME;

**Instrucţiuni pentru actualizarea bazei de date**

Pentru inserarea unui tuplu într-o tabelă se utilizează comanda INSERT:

INSERT INTO nume-tabela [(nume-atribut,…)] {VALUES(valoare,…)|cerere}

Lista de atribute se specifică în cazul în care realizările din clauza VALUES nu corespund ordinii tuplurilor din tabelă. Dacă tuplul/tuplurile inserate sunt preluate dintr-o altă tabelă se va specifica comanda Select.

*Exemple*: Se introduce un nou tuplu în tabela *Produse* (având în structura sa atributele: cod produs, denumire, unit. măsură, preţ, calitate) :

**INSERT INTO** PRODUSE **VALUES** (‘345’,’PERE’,’KG’,12000,’1’);

sau:

Se introduc tupluri în tabela *Produse* sursa de date constituind-o tabela *Noutăţi* (care prezintă aceeaşi structură cu tabela *Produse*):

**INSERT INTO** PRODUSE

**SELECT** \* **FROM** NOUTĂTI;

Actualizarea datelor dintr-o tabelă se realizează prin comanda UPDATE care prezintă următoarea sintaxă:

**UPDATE** nume-tabela

**SET** {nume-atribut=expresie,…| (nume-atribut=subcerere),…} [**WHERE** condiţie];

Noua realizare a atributului actualizat se precizează prin expresie sau este returnat de subcererea precizată. Clauza WHERE permite selectarea tuplurilor afectate de actualizare.

*Exemplu*:

Se majorează cu 1000 preţul produselor de categoria a doua :

**UPDATE** PRODUSE

**SET** PREŢ=PREŢ+500

**WHERE** CALITATE=”2”;

**Gestiunea view-urilor**

Un view este o tabelă virtuală, care nu conţine date, dar îi conferă utilizatorului posibilitatea de vizualizare a datelor din baza de date dintr-o anumită perspectivă. Pentru definirea unui view se foloseşte o cerere SQL.

View-urile pot fi folosite în următoarele scopuri:

• limitează accesul unor categorii de utilizatori la anumite date, mai mult sau mai puţin confidenţiale;

• permite simplificarea viziunii utilizatorilor asupra bazei de date;

• asigură securitatea şi confidenţialitatea datelor. Fiecărui utilizator al bazei de date îi sunt asignate anumite drepturi (GRANT/REVOKE) ceea ce determină o limitare a accesului la baza de date, la obiectele bazei de date şi la efectuarea unor operaţii asupra obiectelor bazei de date.

Asupra unui view se pot efectua ulterior aceleaşi operaţii ca şi asupra unei tabele, dar cu deosebirea că view- ul nu conţine date, ci este o reflectare dinamică a unei tabele. Astfel, o modificare a datelor dintr-o tabelă este automat reflectată în view-urile definite pe acea tabelă. Un view se recreează de fiecare dată când este apelat, pe baza definiţiei sale stocate în dicţionarul bazei de date.

Operaţiile aplicate asupra view-urilor sunt:

• crearea unui view cu comanda CREATE VIEW;

• interogarea view-urilor - prin folosirea unei fraze SELECT;

• actualizarea datelor dintr-un view folosind comanda UPDATE;

• adăugarea de noi date folosind comanda INSERT;

• ştergerea datelor dintr-un view cu comanda DELETE.

**Crearea unui view** - permite definirea unei "ferestre" prin care se pot consulta datele stocate în tabele.

CREATE VIEW nume-view [<lista atribute>] AS SELECT secvenţa-select

[WITH CHECK OPTION]

Se poate observa în sintaxa comenzii că utilizatorului i se permite:

• specificarea denumirii view-ului şi opţional a denumirii atributelor din view, în cazul în care se doreşte redenumirea atributelor specificate în instrucţiunea SELECT;

• specificarea interogării (fraza SELECT);

• specificarea opţională a unei condiţii suplimentare impuse view-ului, astfel încât să poată fi realizată

actualizarea sau inserarea datelor în view (prin utilizarea clauzei WITH CHECK OPTION).

Definirea unui view este stocată în dicţionarul de date al sistemului. Odată definit view-ul, drepturile de acces ale utilizatorilor decurg pe de o parte, din privilegiile acordate utilizatorilor asupra relaţiilor pentru care este definit view-ul şi, pe de altă parte, din posibilitatea de actualizare a view-ului.

Crearea unui view simplu, care va permite selectarea unei mulţimi de tupluri dintr-o tabelă, se realizează prin comanda:

**CREATE** VIEW FZ1 AS

**SELECT** CODFZ, DENFZ, ADRESA **FROM** FURNIZOR **WHERE** ADRESA="BUCURESTI";

View created.

Sursa de date o reprezintă tabela *Furnizor* iar rezultatul returnat constă în afişarea codului, denumirii şi adresei furnizorilor bucureşteni. Când un view este creat, e afişat un mesaj "View Created". Ulterior, view-ul poate fi vizualizat ca orice tabelă:

**SELECT** \* **FROM** FZ1;

unde FZ1 reprezintă view-ul creat prin comanda anterioară.

Crearea unui view complex presupune utilizarea unor funcţii agregat în fraza Select:

**CREATE VIEW** STOC\_3

(DEPOZIT, STOC\_MEDIU, STOC\_MAX, STOC\_MIN) AS

**SELECT** COD\_DEP, AVG(CANTITATE), MAX(CANTITATE), MIN(CANTITATE) **FROM** STOCURI

**GROUP BY** COD\_DEP;

În acest caz coloanele view-ului au primit nume distincte de numele atributelor existente în tabela *Stocuri*, atribuirea valorilor făcându-se în ordinea definirii atributelor în view.

În următorul exemplu se exemplifică folosirea mai multor fraze SELECT imbricate, rezultatul returnat reprezentând clienţii care au solicitat comenzi depozitelor alimentare:

**CREATE VIEW** CLIENT\_A

AS **SELECT** \* **FROM** CLIENT

**WHERE** CODCLI **IN** (**SELECT** CODCLI **FROM** COMANDA

**WHERE** CODDEP **IN** (**SELECT** CODDEP **FROM** DEPOZIT

**WHERE** DENDEP LIKE "\*ALIM\*"));

**Utilizarea opţiunii WITH CHECK OPTION**

Această opţiune asigură faptul că în cazul unei operaţii INSERT sau UPDATE - nu va fi adăugat sau actualizat nici un tuplu care nu se încadrează în condiţiile specificate la definirea view-ului.

*Exemplu*:

**CREATE VIEW** SAL1

AS **SELECT** MARCA, NUME, SAL **FROM** SALARIAT

**WHERE** SAL BETWEEN 1500000 AND 3000000

**WITH CHECK OPTION**;

Asupra view-urilor pot fi efectuate aceleaşi operaţii ca asupra tabelelor pe baza cărora au fost definite, cu unele restricţii:

***Interogarea view-urilor***

*Restricţii pentru interogarea view-rilor:*

a) Clauza WHERE nu poate conţine un atribut rezultat din aplicarea unei funcţii agregat.

**CREATE VIEW** DEP1(DEPOZIT, TOTAL)

AS **SELECT** COD\_DEP, SUM(CANTITATE) **FROM** STOCURI **GROUP BY** COD\_DEP;

Nu se poate specifica atributul TOTAL al vizualizării DEP1 în clauza WHERE a unei interogări de tipul:

**SELECT** \* **FROM** DEP1 **WHERE** TOTAL>300;

Atributul TOTAL poate fi apelat numai în clauza ORDER BY.

**SELECT** DEPOZIT, TOTAL **FROM** DEP1 **ORDER BY** TOTAL;

b) Un atribut al view-ului rezultat în urma aplicării unei funcţii agregat, nu poate fi utilizată ulterior ca argument al unei alte funcţii agregat. Nu va putea fi executată o cerere de forma :

**SELECT** AVG(TOTAL) **FROM** DEP1;

**Realizarea operatorilor relaţionali folosind limbajul SQL**

**REUNIUNEA** a două tabele R şi S (RUS) prezentând aceeaşi structură, unde R este formată din n tupluri şi S este formată din m tupluri, are ca rezultat o a treia tabelă, T având aceeaşi structură cu a tabelelor sursă şi conţinând m+n tupluri.

Fie cele două tabele *Clienti* şi *ClientiNoi* (tabela *ClientiNoi* cuprinde noii clienţi creaţi în cursul zilei curente) definite cu aceeaşi structură. Operatorul **reuniune** se realizează în Access prin două cereri de acţiune: o cerere de tip *make*-*table* prin care se creează noua tabelă (numită T în cadrul definiţiei reuniunii) *TotiClientii*, pe baza tabelei *Clienti*, şi o cerere de acţiune de tip *append*-*query* pe baza tabelei *ClientiNoi*.

Comenzile SQL corespunzătoare celor două cereri de acţiune sunt:

**SELECT** CLIENTI.\* **INTO** TOTICLIENTII **FROM** CLIENTI;

**INSERT INTO** TOTICLIENTII

**SELECT** CLIENTINOI.\* **FROM** CLIENTINOI;

O a doua soluţie ar consta în utilizarea operatorului **union** care va returna toate tuplurile din cele două tabele sursă, *Clienţi* şi *ClientiNoi*, şi apoi generarea tabelei *TotiClientii* care să stocheze toate tuplurile returnate de operatorul **union**. Frazele SELECT corespunzătoare sunt:

**SELECT** \* **FROM** CLIENTI

**UNION**

**SELECT** \* **FROM** CLIENTINOI;

**SELECT UNION**.\* **INTO** TOTICLIENTII **FROM** [UNION];

**INTERSECŢIA** a două tabele R şi S (R∩S) prezentând aceeaşi structură, generează o a treia tabelă, T (cu aceeaşi structură), conţinând tuplurile identice aparţinând atât lui R cât şi lui S.

Pentru exemplificare vom utiliza tabelele *Clienti* şi *ClientiDebitori* care prezintă aceeaşi structură. Implementarea operatorului **intersecţie** se face printr-o frază SELECT în care se realizează un **inner join** între cele două tabele sursă, *Clienti* şi *ClientiDebitori*:

**SELECT** Client.\*

**FROM** Clienti **INNER JOIN** ClientiDebitori **ON** Clienti.CodClient=ClientiDebitori.CodClient;

Pentru crearea tabelei *T* se poate executa apoi o cerere de acţiune de tip make table având drept sursă de date cererea construită prin fraza select prezentată mai sus (cerere salvată cu numele *Query1)*:

**SELECT** Query1.\* INTO T **FROM** Query1;

**DIFERENŢA** dintre două tabele R şi S având aceeaşi structură (R-S), este reprezentată de tabela T

cuprinzând mulţimea tuplurilor aparţinând lui R dar neaparţinând lui S.

Implementarea operatorului **diferenţă** se realizează construind o cerere de căutare a tuplurilor din tabela

*Clienti* fără corespondent în tabela *ClientiDebitori*. Fraza SQL corespunzătoare este:

**SELECT DISTINCTROW** [CLIENTI].[CodClient], [CLIENTI].[NumeClient], [CLIENTI].[Localitate], [CLIENTI].[Adresa], [CLIENTI].[Telefon]

**FROM** CLIENTI **LEFT JOIN** ClientiDebitori **ON** [CLIENTI].[CodClient] = [ClientiDebitori].[CodClient]

**WHERE** [ClientiDebitori].[CodClient] Is Null;

Pentru crearea tabelei *T* se poate construi apoi o cerere de acţiune de tip make table având ca sursă de date cererea prezentată mai sus.

**PRODUSUL CARTEZIAN** a două tabele R şi S (RxS) este reprezentat de tabela T stocând mulţimea perechilor obţinute prin concatenarea înregistrărilor aparţinând lui R cu cele aparţinând lui S.

Pentru exemplificare vom folosi tabela *Clienti* şi tabela *Produse*. Vom construi o cerere de selecţie având drept sursă de date cele două tabele (nelegate între ele) şi vom coborî în grila de proiectare în rândul FIELD atributele NumeClient din tabela *Client* şi Denumire din tabela *Produse*. Fraza SELECT corespunzătoare acestei cereri de selecţie este:

**SELECT** CLIENTI.NUMECLIENT, PRODUSE.DENUMIRE **FROM** CLIENTI, PRODUSE;

Tabela *T* se poate crea apoi printr-o cerere de acţiune de tip make table.

**SELECŢIA** aplicată asupra unei tabele R stocând n tupluri generează o nouă tabelă R1, prezentând aceeaşi structură, dar stocând m tupluri (m<=n).

Acest operator se implementează prin construirea unei cereri de selecţie conţinând pe rândul **criteria** din grila de proiectare criteriul de selecţie dorit. Să presupunem că dorim să extragem din tabela *Clienti* toate informaţiile privind clienţii bucureşteni. Fraza select corespunzătoare acestei cereri de selecţie este:

**SELECT** CLIENTI.\*

**FROM** CLIENTI

**WHERE** CLIENTI.Localitate="Bucureşti";

**PROIECŢIA** aplicată asupra unei tabele R prezentând gradul n va genera o nouă tabelă R1 prezentând gradul m (m<=n) şi stocând acelaşi număr de tupluri ca şi R.

Şi de această dată implementarea operatorului se va realiza prin construirea unei cereri de selecţie dar în rândul **fields** al grilei de proiectare se vor coborî doar o parte din atributele aparţinând structurii tabelei sursă de date.

*Exemplu*: Dacă dorim să cunoaştem numele şi adresa clienţilor, se va construi o cerere de selecţie pe tabela

*Clienti* prezentând următorul conţinut:

**SELECT** CLIENTI.NumeClient, CLIENTI.Adresa **FROM** CLIENTI;

Considerăm necesar să subliniem faptul că frazele select permit utilizarea mai multor tabele drept sursă de date (în definiţia operatorilor algebrei relaţionale sursa de date este reprezentată de o singură tabelă) şi combinarea operatorilor de selecţie şi proiecţie. Proiecţia se realizează prin precizarea listei atributelor pentru care se întorc valori, iar selecţia prin utilizarea clauzei WHERE care specifică criteriul de selecţie.

**COMPUNEREA (JOIN)** se aplică asupra a două tabele R şi S care prezintă în structura lor un atribut comun (numit atribut de joncţiune) şi generează o nouă tabelă T cu o structură complexă formată implicit din mulţimea atributelor tabelelor sursă.

Operatorul de compunere se implementează prin fraza SELECT atunci când sursa de date este reprezentată de mai multe tabele. Cel mai utilizat operator de compunere este **inner join** care presupune ca valorile atributelor de compunere din cele două tabele sursă să prezinte realizări comune (egale).

**SELECT** CLIENTI.NumeClient, COMENZI.NrComanda, COMENZI.DataComanda

**FROM** CLIENTI **INNER JOIN** COMENZI **ON** CLIENTI.CodClient = COMENZI.CodClient;

În exemplul prezentat mai sus cererea de selecţie returnează numele clienţilor (din tabela *Clienti*), numărul şi data comenzilor din tabela *Comenz*i, operatorul de compunere fiind un **inner join**. Pe lângă **inner join** frazele SELECT pot cuprinde şi operatori de compunere de tip: **left join**, **right join**, **self join, theta non- equal**.

**DECLANŞATOARE**

Un declanşator (trigger) este o procedur**ă** care este executat**ă** în mod implicit când asupra tabelului asociat se execută o comandă **insert**, **update** sau **delete**. Declanşatoarele sunt medii prin care SQL ofer**ă** programatorilor de aplicaţii şi proiectantilor de BD să asigure integritatea BD.

Declanşatoarele sunt utile deoarece impun reguli fară a fi cuprinse în aplicaţiile utilizatorilor. Proiectantul declanşatorului specifică şi momentul când trebuie executat în raport cu instrucţiunile LMD.SQL Server ia în seama regulile şi valorile implicite înainte ca informaţiile să fie scrise în BD.

Acestea reprezintă pre-filtre care pot să impiedice manipulările de date, prin controlarea activităţilor din cadrul BD. Un declanşator poate fi şi un post-filtru care se execută după ce modificarea datelor a trecut de reguli.

Declanşatorii permit să executăm o procedură rezidentă ori de câte ori este executată o instrucţiune **INSERT, UPDATE** sau **DELETE** asupra unui tabel predefinit. Declanşatorul este o procedură specială stocată care:

- generează automat anumite valori ce derivă din valorile coloanelor;

- determină respectarea restricţiilor şi privilegiilor ;

- face posibilă jurnalizarea transparentă a evenimentelor;

- strânge informaţii statistice în legătură cu accesarea tabelelor.

Un declanşator este o procedură stocată specială care este executată de SQL Server la efectuarea unei operaţii de inserare, modificare sau stergere. Pentru că declanşatorii sunt executaţi după efectuarea operaţiei de inserare, modificare sau ştergere ei reprezintă un fel de ultim cuvânt la acestea. Dacă un declanşator respinge cererea, modificarea informaţiilor este refuzată, iar aplicaţia care a iniţiat operaţia deţine un mesaj de eroare. Cea mai simplă utilizare este determinată de impunerea regulilor de integritate în BD.

**Observaţie:** Deoarece declanşatoarele sunt executate după regulile de integritate dacă acestea nu sunt trecute atunci declanşatorul nu este executat. Pentru ca un declanşator să fie executat trebuie ca operaţia în cauză să nu fi eşuat.

**Componentele unui declanşator**

Un declanşator are trei componente:

- o instrucţiune de declarare –aceasta specifică instrucţiunile SQL care activează

declanşatorul;

- o restricţie de declanşare –specifică condiţia care trebui să fie adevarată pentru ca declanşatorul să fie activat;

- acţiunea declanşatorului care specifică blocul de instrucţiuni care trebuie executate.

Pentru a crea un declanşator trebuie să fiţi posesorul BD. Atunci când se adaugă un declanşator pentru o coloana, linie sau tabel se schimbă uneori şi modul de accesare şi modul cum interacţionează alte obiecte cu acesta. La folosirea obiectelor declanşator se presupune respectarea condiţiilor:

- Declanşatoarele nu pot fi create pentru tabele temporare,

- Declanşatoarele trebuie să fie create numai pentru tabele din baza de date curentă,

- La eliminarea unui tabel, toate obicetele declanşator asociate cu acest tabel sunt,

eliminate automat împreună cu tabelul.

Crearea unui declanşator se realizează cu instrucţiunea:

**CREATE TRIGGER** nume\_declanşator

**ON** { *Nume\_ tabel* | *Nume\_ view* }

**[ WITH ENCRYPTION** ]

{

{ **{ FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }**

**[ WITH APPEND ]**

**[ NOT FOR REPLICATION ] AS**

**[ { IF UPDATE (** *coloana* **)**

[ **{ AND | OR } UPDATE (** *coloana* **)** ] [ ...*n* ]

**| IF ( COLUMNS\_UPDATED ( )** { *bitwise\_operator* } *updated\_bitmask* **)**

{ operator de comparare } *coloana\_bitmask* [ ...*n* ]

} ]

**Instrucţiuni\_SQL [ ...*n* ]**

}

}

Sau următoarea formă mai simplă

**CREATE TRIGGER** [posesor.]nume\_declanşator

**ON [**posesor**.]**nume\_tabela

**FOR {**INSERT, UPDATE, DELETE **} [WITH ENCRYPTION]**

**AS**

Instrucţiuni **SQL**

**Nume\_declanşator** este numele declanşatorului (trigger-ului). Un nume de declanşator trebuie să respecte regulile pentru identificatori şi trebuie să fie unic încadrul unei baze de date.

**Nume\_ tabel | Nume\_ view** este numele tabelului sau vederii în care declanşatorul este

executat.

INSERT, UPDATE, DELETE acestea definesc scopul declanşatorului şi specifică

operaţiile care iniţiază declanşatorul.

WITH ENCRYPTION –această opţiune este utilizată pentru a împiedica ca utilizatorii să citescă definiţia declanşatorului după încărcarea lui pe server. SQL Server stochează definiţia unui declanşator în fişierul **Syscomment**. Criptează intrările în tabela **syscomments** care conţine textul comenzii CREATE TRIGGER.

**AFTER** specifică faptul că declanşatorul este executat doar când toate operaţiile din enunţul SQL al declanşatorului au fost execuate cu succes. Toate acţiunile referenţiale în cascadă şi verificările de constrângere deasemenea trebuie să reusească înainte de executarea acestui trigger. Folosirea clauzei AFTER este identică cu folosirea clauzei

FOR utilizată mai mult în ultimele versiuni de SQL Server. Declanşatorii AFTER nu pot fi definiţi pentru vederi.

**INSTEAD OF** Specifică faptul că declanşatorul este executat în locul enunţului SQL

declanşat, astfel suprapunând operaţiile din enunţul declanşatorului.

Acest tip de declanşator poate fi definit pentru un tabel sau pentru o vedere când se realizează; INSERT, UPDATE sau DELETE.

Este posibil să definim vederi unde fiecare vedere să aibă propriul declanşator

INSTEAD OF.

{ [DELETE] [**,**] [INSERT] [**,**] [UPDATE] } Sunt cuvinte cheie care specifică pentru ce operaţie are loc declanşatorul. Este obligatorie apariţia cel puţin a unei opţiuni.

Pentru declanşatori INSTEAD OF, opţiunea **DELETE nu este permisă** pentru tabelele care au legături referenţiale ce specifică o acţiune încascadă pentru opţiunea ON DELETE. Asemănător, opţiunea UPDATE nu este permisă pe tabele care au legături referenţiale ce specifică o acţiune încascadă pentru opţiunea ON UPDATE. ***Instrucţiuni\_sql*** Reprezintă condiţia (condiţiile) şi acţiunea (acţiunile) declanşatorului. Acţiunile declanşatorului sunt scrise în Transact-SQL şi pot cuprinde oricâte instrucţini sau comenzi Transact-SQL.

Câteva tabele speciale, de care am mai amintit, sunt folosite în instrucţiunea CREATE TRIGGER şi anume: **inserted** şi **deleted**.

**IF UPDATE (*coloana*)**Testează acţiunea unei comenzi INSERT sau UPDATE pentru coloana specificată şi nu este folosit pentru DELETE. Mai multe coloane pot fi specificate. Deoarece numele tabelului este specificat după clauza ON, nu este nevoie să includem numele tabelului înainte de numele coloanei în clauza IF UPDATE. Pentru a testa o adăugare sau o modificare pentru mai multe coloane, specificăm separat clauza UPDATE(*coloana)* după ce am scris-o pe prima. IF UPDATE va returna valoarea TRUE în acţiunea INSERT deoarece coloanele vor fi explicit sau implicit (NULL) inserate.

**Nota** Clauza IF UPDATE **(***coloana***)** functionează identic cu IF, IF...ELSE sau WHILE

poate folosi şi blocul BEGIN...END.

UPDATE(*coloana*) poate fi folosit oriunde în corpul declanşatorului.

***Coloana*** este numele unei coloane pentru testarea acţiunii INSERT sau UPDATE. Această coloană poate avea orice tip de date SQL Server.Oricum coloanele calculate nu pot fi utilizate în acest context.

IF **(**COLOANAS\_UPDATED**())** Testează, dacă într-un declanşator INSERT sau UPDATE,

coloanele menţionate au fost inserate respectiv modificate.

COLOANAS\_UPDATED returnează un tip **varbinary** (tip binar variabil) ce indică ce coloane din tabel au fost inserate sau modificate.

Funcţia COLOANAS\_UPDATED returnează biţii în ordine de la stanga la dreapta, cu cel mai

puţin semnificativ bit în partea cea mai din stânga. Cel mai din stânga bit reprezentând prima coloana din tabel, următorul bit reprezentând cea de-a două coloană etc. COLOANAS\_UPDATED returnează mai mulţi octeţi pentru declanşatorul care a fost creat conţinând mai mult de 8 coloane, cu cel mai puţin semnificativ byte în partea din stânga.

COLOANAS\_UPDATED va returna TRUE pentru toate coloanele din acţiunea INSERT deoarece coloanele au fost inserate cu valori explicite sau cu valori implicite (NULL). COLOANAS\_UPDATED poate fi folosit oriunde în corpul declanşatorului.

*bitwise\_operator* Operaţor pe bit folosit în comparări.

*Updated\_bitmask* Este o mască de tip întreg pentru acele coloanele care tocmai au fost modificate sau inserate. De exemplu, tabela **t1** conţine coloanele **C1**, **C2**, **C3**, **C4**, şi **C5**. Pentru a verifica dacă coloanele **C2**, **C3**, şi **C4** sunt toate modificate (tabela **t1** având un declanşator UPDATE), specifică valoarea 14 (01110). Pentru a verifica dacă doar coloana **C2** este modificată, specifică valoarea 2(00010).

***Operaţor* de comparare**.Este unul din operatorii de comparare. Folosiţi semnul egal (=) pentru a verifica dacă toate coloanele specificate în ***updated\_bitmask*** sunt realmente modificate.Folosiţi semnul mai mare (>) pentru a verifica dacă oricare sau câteva din coloanele specificate în *updated\_bitmask* sunt modificate.

*Coloana\_bitmask* Este o mască de tip întreg pentru acele coloane pentru care verificăm dacă ele au fost modificate sau inserate.

SQL Server permite declanşatori multiplii ce pot fi creaţi pentru fiecare eveniment de modificare

de date (DELETE, INSERT, or UPDATE). De exemplu, dacă este executat CREATE TRIGGER FOR UPDATE pentru un tabel care deja are un declanşator UPDATE, atunci este creat un declanşator adiţional de tip UPDATE.

Exemplul 1 Declanşator ce afisează pe ecran un mesaj când se adaugă sau se modifică date din tabela

Catalog

**IF EXISTS (SELECT name**

**FROM sysobjects**

**WHERE name = 'declanş\_mesaj ' AND type = 'TR') DROP TRIGGER declanş\_mesaj**

**GO**

**CREATE TRIGGER** declanş\_mesaj

**ON** Catalog

**FOR INSERT, UPDATE**

**AS RAISERROR** ('S-a executat un INSERT sau un UPDATE', 16, 1)

**GO**

**INSERT INTO** Catalog VALUES ('101','5',7,'11-17-2006')

**GO**

**UPDATE** Catalog

**SET** Nota=5

**WHERE** NrLeg='101' **and** Cod\_disciplina='5'

Exemplul 2 **Acest declanşator afisează pe ecran anumite mesaje pentru oricine încearcă să adauge sau să modifice date din tabelul Catalog**

**IF EXISTS (SELECT name**

**FROM sysobjects**

**WHERE name = ' declan1' AND type = 'TR') DROP TRIGGER declan1**

**GO**

**CREATE TRIGGER** declan1

**ON** Catalog

**FOR INSERT, UPDATE AS**

**DECLARE**

@@nota integer,

@@Nr\_leg varchar(5)

**SELECT** @@Nr\_leg = i.NrLeg, @@nota = i.Nota

**FROM** Catalog C, i**nserted** i

**WHERE** C.NrLeg = i.NrLeg

**IF** (@@nota <5 )and(@@Nr\_leg like '101')

**BEGIN**

**RAISERROR** ('Studentul cu NrLeg=101 are nota <5', 16, 1)

**END ELSE BEGIN**

**RAISERRO**R ('Studentul cu NrLeg=%s are note de %d ',

16, 1, @@Nr\_leg, @@nota)

**END**

**go**

**INSERT INTO Catalog VALUES ('101','4',6,'11-06-2006')**

**go**

**INSERT INTO Catalog VALUES ('101','4',2,'12-07-2006') Go**

**UPDATE Catalog**

**SET Nota=5**

**WHERE NrLeg='101' and Cod\_materie='4' go**

**Exemplul 3 Declanşator pentru ştergeri şi modificări întabelul Catalog**

IF EXISTS (SELECT name

FROM sysobjects

WHERE name = 'dec1' AND type = 'TR') DROP TRIGGER dec1

GO

**CREATE TRIGGER** dec1

**ON** Catalog

**FOR INSERT, UPDATE AS**

**DECLARE**

**@@nota integer,**

**@@Nr\_leg varchar(5)**

**SELECT @@Nr\_leg = i.NrLeg, @@nota = i.Nota**

**FROM Catalog C, inserted i**

**WHERE C.NrLeg = i.NrLeg**

**IF (@@nota <5 )AND(@@Nr\_leg like '101') BEGIN**

**RAISERROR ('Studentul cu NrLeg=111 are Catalog <6', 16, 1)**

ROLLBACK TRANSACTION END

ELSE BEGIN

RAISERROR ('Studentul cu NrLeg=%s are Catalog de %d ',

16, 1, @@Nr\_leg, @@nota)

**ROLLBACK TRANSACTION END**

**go**

**INSERT INTO Catalog VALUES ('101','5',6,'11-10-2006') Go**

**INSERT INTO Catalog VALUES ('101','5',2,'12-01-2006')**

**go**

**UPDATE Catalog SET Nota=7 WHERE NrLeg='101' AND Cod\_disciplina='4'**

**go**

**Exemplul 4 Declanşator ce nu permite introducerea unor date nevalide în Catalog**

IF EXISTS (SELECT name

FROM sysobjects

WHERE name = ' dec2' AND type = 'TR') DROP TRIGGER dec2

GO

CREATE TRIGGER dec2

ON Catalog

FOR INSERT, UPDATE AS

DECLARE @nota integer

SELECT @nota=Catalog.nota from Catalog, inserted WHERE Catalog.NrLeg=inserted.NrLeg

IF(@nota<=0) or (@nota>10) BEGIN

RAISERROR ('Nota invalida', 16, 1) ROLLBACK TRANSACTION

END

go

INSERT INTO Catalog VALUES ('102','4',-2,'12-01-2006')

go

SELECT \* FROM Catalog WHERE NrLeg='102'

**Exemplul 5 Declanşator ce adaugă 1 punct la orice notă din tabelul Catalog**

IF EXISTS (SELECT name

FROM sysobjects

WHERE name = ' dec3' AND type = 'TR') DROP TRIGGER dec3

GO

CREATE TRIGGER dec3

ON Catalog

AFTER INSERT AS

UPDATE Catalog

SET Catalog.nota = Catalog.nota + 1

FROM inserted

WHERE Catalog.NrLeg = inserted.NrLeg go

INSERT INTO Catalog VALUES ('103','4',4,'12-01-2006')

go

SELECT \* FROM Catalog WHERE NrLeg='105'

**Exemplul 6 Declanşator inlocuieste o opera**ţ**ie INSERT cu operaţia UPDATE din enunţul SQL al declanşatorului astfel dacă se doreste inserarea unei Catalog pentru un student la o anumită materie se va face o modificarea adică se adaugă 1 punct la nota introdusă anterior la acea materie pentru studentul dat în tabelul Catalog**

IF EXISTS (SELECT name FROM sysobjects

WHERE name = 'dec3' AND type = 'TR') DROP TRIGGER dec3

CREATE TRIGGER dec3

ON Catalog

INSTEAD OF INSERT AS

UPDATE Catalog

SET Catalog.nota = Catalog.nota+1

FROM inserted

WHERE Catalog.NrLeg = inserted.NrLeg go

INSERT INTO Catalog VALUES ('102','4',4,'12-01-2006')

go

SELECT \* FROM Catalog WHERE NrLeg='102'

**Exemplul7 Crearea unui declanşator ce afisează un mesaj dacă s-a modificat Nota şi un alt mesaj**

**încaz contrar**

IF EXISTS (SELECT name FROM sysobjects

WHERE name = 'dec4' ÂND type = 'TR') DROP TRIGGER dec4

GO

CREATE TRIGGER dec4

ON Catalog FOR UPDATE AS

IF UPDATE(Nota)

RAISERROR ('S-a modificat campul Nota',16,1) ELSE

RAISERROR ('S-a modificat alt camp al tabelei CATALOG',16,1)

go

UPDATE Catalog SET Nota=10 WHERE NrLeg='105' go

SELECT \* FROM Catalog WHERE NrLeg='105' go

UPDATE Catalog SET Cod\_materie='3' WHERE NrLeg='105' go

**Exemplul 8 Crearea unui declanşator care afisează un mesaj dacă s-au modificat Nota şi**

**Cod\_disciplina şi ‘\*\*\*\*\*’ încaz contrar**

IF EXISTS (SELECT name

FROM sysobjects

WHERE name = 'dec5' AND type = 'TR') DROP TRIGGER dec5

GO

CREATE TRIGGER dec5

ON Catalog FOR UPDATE AS

IF UPDATE(Nota) AND UPDATE(Cod\_diciplina)

RAISERROR('S-au modificat atributele Nota şi

Cod\_disciplina',16,1) ELSE

RAISERROR ('\*\*\*\*\*',16,1)

go

UPDATE Catalog SET Nota=10 WHERE NrLeg='105' go

SELECT \* FROM Catalog WHERE NrLeg='105'

go

UPDATE Catalog SET Cod\_diciplina ='3', Nota=8 WHERE NrLeg='105'

**Exemplul 9 *Crearea unui declanşator care afisează un mesaj dacă s-au modificat atributele Nota sau Cod\_disciplina şi ‘\*.\*.\*’ în caz contrar.***

IF EXISTS (SELECT name

FROM sysobjects

WHERE name = 'dec6' AND type = 'TR') DROP TRIGGER dec6

GO

CREATE TRIGGER dec6

ON Catalog

FOR UPDATE

AS

IF (COLOANAS\_UPDATED()&6)>0

RAISERROR ('S-au modificat atrib. Nota sau Cod\_materie',16,1) ELSE

RAISERROR ('\*.\*.\*',16,1)

go

UPDATE Catalog SET Nota=10 WHERE NrLeg='105' go

SELECT \* FROM Catalog WHERE NrLeg='105' go

UPDATE Catalog SET Cod\_materie='3', Nota=8 WHERE NrLeg='105' go

UPDATE Catalog SET Data='3-3-2002' WHERE NrLeg='105'

**Exemplul 9 *Crearea unui declanşator care verifica dacă au fost modificate printr-o comanda***

***UPDATE coloanele 3, 6 şi 9 din tabelul Student***

IF EXISTS (SELECT name FROM sysobjects WHERE name = 'dec7' AND type = 'TR') DROP TRIGGER dec7

GO

CREATE TRIGGER dec7

ON Student

FOR UPDATE AS

IF ( (SUBSTRING(COLOANAS\_UPDATED(),1,1)=power(2,(3-1))--coloana

3+ power(2,(6-1))) --coloana 6

AND (SUBSTRING(COLOANAS\_UPDATED(),2,1)=power(2,(1-1)))--coloana 9 (coloana 1 din cel de-al II-lea octet )

PRINT 'Coloanele 3, 6 şi 9 au fost modificate'

GO

UPDATE Student

SET Initiala='O', Data\_nastere='12-12-1980', NrLeg\_sot='105' WHERE NrLeg='105'

GO

SELECT \* FROM Student WHERE NrLeg='105'

**Exemplul 10 Crearea unui declanşator pentru DELETE**

IF EXISTS (SELECT name FROM sysobjects

WHERE name = 'sterge\_stud' ÂND type = 'TR') DROP TRIGGER sterge\_stud

GO

CREATE TRIGGER sterge\_stud

ON Student FOR DELETE AS

DECLARE @NL varchar(5) SELECT @NL=deleted.NrLeg

FROM deleted

IF (cast(@NL as integer)>0) PRINT 'S-a Sters'

ELSE BEGIN

PRINT 'Acest NrLeg nu exista'

ROLLBACK TRANSACTION END

GO

DELETE FROM Student WHERE NrLeg='17'

**Modificarea unui declanşator** se face cu comanda ALTER TRIGGER care are sintaxa:

ALTER TRIGGER *nume\_declan*

ON **(** *table* | view **)**

[ WITH ENCRYPTION ]

{

{ **(** FOR | AFTER | INSTEAD OF **)** { [ DELETE ] [ **,** ] [ INSERT ] [ **,** ] [ UPDATE ] } [ NOT FOR REPLICATION ]

AS

*sql\_statement* [ ...*n* ]

}

|

{ **(** FOR | AFTER | INSTEAD OF **)** { [ INSERT ] [ **,** ] [ UPDATE ] }

[ NOT FOR REPLICATION ] AS

{ IF UPDATE **(** *coloana* **)**

[ { AND | OR } UPDATE **(** *coloana* **)** ] [ ...*n* ]

| IF **(** COLOANAS\_UPDATED **( )** { *bitwise\_operaţor* } *updated\_bitmask* **)**

{ *comparison\_operaţor* } *coloana\_bitmask* [ ...*n* ]

}

*sql\_statement* [ ...*n* ]

}

}

**Ştergerea unui declanşator** se face cu comanda:

DROP TRIGGER { *nume\_declan* } [ **,**...*n* ]

Un declanşator poate să conţină oricâte instrucţiuni cu condiţia să fie încadrate într-un bloc Begin…END. La executarea unui declanşator SQL serverul accesează o tabelă specială cu datele ce au determinat rularea declanşatorului.Aceasta tabelă este **INSERTED** pentru operaţiile INSERT şi UPDATE şi respectiv **DELETED** pentru operaţiile DELETE şi UPDATE. Deoarece un declanşator este executat întotdeauna după operaţie, liniile din INSERTED reprezintă întotdeauna o dublură a uneia sau a mai multor înregistrări din tabela de bază.

**Restricţii de utilizare a declanşatorilor**

SQL-Serverul impune anumite restricţii asupra tipurilor de instrucţiuni care pot fi executate în cadrul unui declanşator. Majoritatea restricţiilor sunt determinate de imposibilitatea de anulare a consecinţelor unei operaţii insert, update sau delete.

În corpul unui declanşator nu poate apare nici una din instrucţiunile:

- CREATE DATABASE, CREATE TABLE INDEX, PROCEDURE, DEFAULT, RULE, TRIGGER şi VIEW.

- toate instrucţiunile DROP;

- instrucţiunile de modificare a obiectelor din bază ALTER TABLE, ALTER DATABASE şi TRUNCATE TABLE;

- UPDATE STATISTICS;

- Operaţii de încărcare a BD LOAD DATABASE şi LOAD TRANZACTION;

- Toate instrucţiunile care realizează modificări fizice pe disc:DISK.

- Crearea de fişiere temporare fie implicita fie prin SELECT;

- Nu se poate crea un declanşator pentru o vedere ci numai pentru tabelă de bază care defineşte vederea.

Manipularea unor coloane care conţin bloc nu determină execuţia unui declanşator.

**Tipuri de declanşatoare**

Atunci când se creează un declanşator se specifică de câte ori se execută. Există 2 tipuri de declanşatoare:

-**Declanşator linie –**care se execută ori de câte ori tabelul este afectat de instrucţiunea declanşatoare. Dacă o instrucţiune UPDATE actualizează 20 de linii, declanşatorul se execută de 20 de ori. Dacă instrucţiunea nu afecteză nici o linie declanşatorul nu este executat.

**-Declanşatorul instrucţiune** este executat o singură dată pentru instrucţiunea declanşatoare. De exemplu dacă este executată o instrucţiune UPDATE care actualizează

10 linii este executat o singură dată

Declanşatorele INSERT şi UPDATE sunt utile pentru ca impun restricţiile referenţiale şi asigură validarea datelor înainte de a le scrie în tabele. De obicei se foloseşte pentru a verifică dacă datele urmărite satisfac anumite criterii. La declanşatoare se apelează numai când criteriile de integritate sunt foarte complete. Declanşatorul de mai jos este activat ori de câte ori se înserează sau se modifică o înregistrare în tabelul vânzări.

**CREATE TRIGGER** ins\_vanzari

**ON** vanzari

**FOR INSERT,UPDATE AS**

**DECLARE** @zidinluna tinyint

**SELECT** @zidinluna=DataPart(Day, iData\_comenzi)

**FROM** vanzari v, Inserted i

**WHERE** v.star\_id=i.star\_id **AND** v.nr\_com=i.nr\_com **AND** v.denumire=i.denumire

**IF** @zidinluna>15

**Begin ROLLBACK END**

**GO**

**CREATE TRIGGER** elimart

**ON** articole **FOR DELETE AS**

**FOR EACH** row

**WHERE (**pretart**)<100**

**DELETE END**

**CREATE TRIGGER** ins.art

**ON** articol **FOR INSERT FOR EACH** row **BEGIN**

**INSERT INTO** art

**END**

Se poate crea declanşatorul care să reacţioneze dacă se modifică doar o coloană. Instrucţiunea IF update poate fi utilizată într-un declanşator pentru a decide dacă se conţinuă execuţia acestuia.

IF UPDATE (preţ )AND (@@row COUNT=1) BEGIN

-

- END

In acest caz blocul se execută numai dacă s-a modificat coloana pret .Modificarea unei coloane nu înseamnă neapărat schimbarea valorii ei.

**Declanşatoare pentru stergere**

Se folosesc pentru a preveni ştergerea acolo unde acestea ar afecta integritatea datelor (Este exemplul cheilor străine pentru alte tabele. Al doilea este ştergerea unui articol în cascadă care să elimine înregistrările copie ale unei înregistrări particulare )

**CREATE TRIGGER** sterg

**ON** vanzări **FOR DELETE AS**

**IF** @@row COUNT>1 /\*verifica nr de linii afectate impiedicând

stergerea a mai mult de o linie\*/

**BEGIN ROLLBACK**

**RAISERROR(‘**puteti sterge o singură instrucţiune la un moment dat**’,16, 10)**

**END**

**DECLARE @**stare\_id char(4)

**SELECT** @stare\_id=s.stare-id

**FROM** vanzări v deleted s

**IF EXIST (SELECT \* FROM** vanzări

**WHERE** stare\_id=@s.stare\_id**)**

**BEGIN ROLLBACK**

**RAISERROR(‘**nu poate fi sters**’,16,10)**

**END GO**

**xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**

**GESTIUNEA TRANZACŢIILOR**

Conceptul de ***gestiune a tranzacţiilor*** se referă la problematica menţinerii într-o stare consistentă a bazei de date în condiţiile în care accesul la date se face într-un regim concurent şi este posibilă apariţia unor defecte. În consecinţă se disting două domenii de sine stătătoare în cadrul problematicii generale a gestiunii tranzacţiilor :

**Controlul concurenţei**

Se ocupă cu mecanismele de sincronizare a acceselor astfel încât să fie menţinută integritatea bazei de date. Atunci când controlul concurenţei este realizat prin mecanismele de blocare (care la ora actuală sunt cele mai răspândite) mai apare o problemă, colaterală, şi anume aceea a interblocărilor. Datorită importanţei sale problema ***gestiuni interblocărilor*** este de multe ori tratată ca o problematică de sine stătătoare a gestiunii tranzacţiilor.

**Rezistenţa la defecte**

Se referă la tehnicile prin care se asigură atât toleranţa sistemului faţă de apariţia unor defecte, cât şi capacitatea de recuperare a acestuia, adică posibilitatea de revenire la o stare consistentă în urma apariţiei unui defect care a cauzat intrarea lui într-o stare inconsistentă.

**Definiţia conceptului de tranzacţie**

Prin controlul concurenţei şi rezistenţa la defecte se urmăreşte asigurarea **consistenţei** şi **siguranţei** bazei de date. O bază de date este într-o stare consistentă dacă respectă toate constrângerile de integritate a datelor definite asupra sa. În timpul operaţiilor de adăugare, ştergere şi modificare, baza de date trece dintr-o stare în alta. Evident, starea rezultată după orice prelucrare asupra bazei de date trebuie să fie o stare consistentă, chiar dacă în timpul prelucrării baza de date s-a aflat temporar într-o stare inconsistentă.

Siguranţa bazei de date se referă la toleranţa acesteia faţă de defecte şi la capacitatea de recuperare după apariţia unui defect.

**O tranzacţie** este o unitate logică de prelucrare care asigură consistenţa şi siguranţa bazei de date. În principiu, orice execuţie a unui program se poate considera o tranzacţie dacă baza de date este într-o stare consistentă atât înainte cât şi după execuţia sa. Consistenţa bazei de date este garantă independent de faptul că :

1. tranzacţia a fost executată în mod concurent cu alte tranzacţii ;
2. au apărut defecte în timpul execuţiei tranzacţiei.

În general, o tranzacţie constă dintr-o secvenţă de operaţii de citire şi scriere a bazei de date, la care se adaugă o serie de operaţii de calcul. Baza de date poate fi într-o stare temporar inconsistentă în timpul executării tranzacţiei dar trebuie să fie în stări consistente atât înainte cât şi după execuţia acesteia.

Tranzacţiile ar trebui să conţină doar acele comenzi DML care realizează o singură modificare asupra datelor. De exemplu un transfer de fonduri (să spunem 1000$) între două conturi ar trebui să implice un debit al unui cont de 1000$ şi un credit al altui cont de 1000$. Ambele acţiuni ar trebui să se încheie cu succes sau să dea eroare împreună. Creditul nu ar trebui executat fără debit.

***Condiţii de terminare a tranzacţiilor***

O tranzacţie nu se termină întotdeauna cu succes totuşi orice tranzacţie trebuie să se termine, indiferent de situaţia existentă (chiar şi în cazul unor defecte). Dacă tranzacţia reuşeşte să execute cu succes toate operaţiile prevăzute, atunci aceasta se va termina printr-o operaţie de **validare** (*commit*). În schimb, dacă dintr-un motiv sau altul tranzacţia nu reuşeşte să-şi execute complet operaţiile prevăzute, atunci se va termina printr-o operaţie de **abortare** (*abort* sau *rollback*). Motivele pentru care o tranzacţie se abortează sunt  numeroase, ele pot fi interne tranzacţiei sau externe acesteia (ex. : detectarea de către SGBD a unei situaţii de interblocare). În cazul abortării, execuţia tranzacţiei este oprită iar efectele tuturor operaţiilor pe care le-a executat până în acel moment sunt anulate astfel încât baza de date revine la starea de dinaintea lansării tranzacţiei.

Comanda de validare a unei tranzacţii are dublu rol :

indică SGBD-ului momentul de la care efectele tranzacţiei pot fi reflectate în baza de date şi devin vizibile altor tranzacţii ;

marchează momentul începând de la care efectele tranzacţiei nu mai pot fi anulate (tranzacţia nu se mai poate aborta) şi modificările efectuate în baza de dte devin permanente.

Operaţia de validare este vitală în cazul sistemelor concurente, deci acolo unde este posibilă executarea în acelaşi timp a mai multor tranzacţii care accesează aceeaşi bază de date. Prin validare se pot preveni o serie de fenomene nedorite cum este abortarea în cascadă a tranzacţiilor.

Să presupunem că o tranzacţie T este abortată după ce a efectuat una sau mai multe operaţii de actualizare a bazei de date. În acest caz datele alterate de către tranzacţia T vor fi readuse la valorile pe care le-au avut înainte de a fi modificate de aceasta. Este însă posibil ca unele dintre tranzacţiile executate în mod concurent cu tranzacţia T să fi accesat aceste date înainte de abortarea lui T. Aceste tranzacţii vor trebui să fie, la rândul lor, abortate deoarece au avut acces la date inconsistente din baza de date iar rezultatele produse de ele pot fi compromise. Acest efect se poate propaga în continuare şi asupra altor tranzacţii, pe un număr nedefinit de nivele, conducând la **abortarea în cascadă** a tranzacţiilor. Fenomenul este cunoscut în literatura de specialitate sub numele de **efect domino**.

Dacă se foloseşte un mecanism de validare care respectă cele două reguli de mai sus, atunci apariţia fenomenului de abortare în cascadă devine imposibilă. Într-adevăr, conform primei reguli, nici o tranzacţie nu va putea accesa datele modificate de tranzacţia T decât după validarea acesteia. Pe de altă parte, conform regulii a doua, după validarea sa, tranzacţia T nu mai poate fi abortată, deci nu poate declanşa un lanţ de abortări în cascadă.

Validarea unei tranzacţii marchează, din punct de vedere logic, terminarea acesteia. Validarea nu se poate face înainte ca operaţiile specificate prin codul tranzacţiei să fie executate integral şi înainte ca tranzacţia să  ajungă într-o stare începând de la care există certitudinea că nu mai poate fi abortată.

Până în momentul validării, actualizările efectuate de tranzacţie sunt invizibile alror tranzacţii, au caracter tentativ şi pot fi oricând revocate (odată cu abortarea tranzacţiei). După validare actualizările se înscriu cu caracter permanent în baza de date şi devin irevocabile. După validare nu maie ste posibilă abortatrea tranzacţiilor.

***Proprietăţile tranzacţiilor***

Prin definţie, tranzacţiile sunt unităţi de execuţie care garantează consistenţa şi siguranţa bazei de date. Pentru aceasta orice tranzacţie trebuie să satisfacă un set de patru condiţii sintetizate în literatura de specialitate prin acronimul **ACID** – *atomicitate, consistenţă, izolare, durabilitate*.

**Atomicitatea**

 Se referă la faptul că o tranzacţie este considerată o unitate elementară de prelucrare. Aceasta înseamnă că execuţia unei tranzacţii se face după regula « totul sau nimic », adică ori sunt executate toate operaţiile din tranzacţie ori nu se execută nimic. Dacă o tranzacţie este întreruptă datorită unor cauze oarecare, atunci îi revine SGBD-ului sarcina de a asigura, într-un fel sau altul, terminarea tranzacţiei. După eliminarea cauzei care a dus la întreruperea tranzacţiei, în funcţie de stadiul de execuţie în care s-a aflat aceasta în momentul apariţiei întreruperii, SGBD-ul poate proceda într-unul dintre modurile următoare :

fie completează operaţiile rămase neexecutate din cadrul tranzacţiei, terminând tranzacţia cu succes

fie anulează toate efectele operaţiilor executate până în momentul întreruperii, terminând tranzacţia prin abortare.

**Consistenţa**

 Consistenţa unei tranzacţii constă pur şi simplu în corectitudinea ei. Orice tranzacţie, dacă este executată independent, trebuie să menţină consistenţa bazei de date. Altfel spus, o tranzacţie este un program corect care transformă baza de date dintr-o stare consistentă într-o altă stare consistentă a sa. Prin consistenţa bazei de date înţelegem satisfacerea tuturor constrângerilor de integritate, explicite sau implicite, cum ar fi :

unicitatea cheilor primare ;

integritatea referenţială ;

orice predicat exprimat în sens de constrângere de integritate asupra bazei de date.

Bineînţeles că este de neconceput verificarea tuturor acestor condiţii după executarea fiecărei tranzacţii. De aceea unicul criteriu pentru stabilirea proprietăţii de consistenţă a unei tranzacţii rămâne corectitudinea sa din punct de vedere logic. Spre deosebire de celelalte proprietăţi din complexul ACID, care sunt asigurate de către sistem, proprietatea de consistenţă a tranzacţiei cade în sarcina programatorului de aplicaţii. De remarcat faprul că stările intermediare prin care trece baza de date în timpul execuţiei unei tranzacţii nu sunt neapărat consistente.

**Izolarea**

             Se referă la proprietatea oricărei tranzacţii de a avea acces doar la stările consistente ale bazei de date. Aceasta înseamnă că modificările efectuate de către o tranzacţie sunt inaccesible altor tranzacţii concurente până în momentul validării acesteia. Prin proprietatea de izolare se creează iluzia că fiecare tranzacţie este executată singură în sistem. Utilizatorul care  a lansat o tranzacţie nu va percepe în nicu un fel (cel puţin în ceea ce priveşte rezultatele) faptul că alte tranzacţii sunt executate în acelaşi timp. Izolarea tranzacţiilor este asigurată prin algoritmi de control al concurenţei.

            Proprietatea de izolare este importantă deoarece elimină fenomenul de abortare în cascadă a tranzacţiilor (efect domino). Într-adevăr dacă rezultatele incomplete ale unei tranzacţii ar fi vizibile altor tranzacţii înainte de validarea acesteia şi dacă se întâmplă ca această tranzacţie să aborteze, atunci toate tranzacţiile care au accesat rezultatele incomplete vor trebui abortate. Aceasta poate conduce la abortarea altor tranzacţii ş.a.m.d. rezultând efectul domino.

**Durabilitatea**

 Durabilitatea unei tranzacţii este proprietatea prin care se garantează faprul că, odată tranzacţia validată, rezultatele sale devin permanente şi sunt înscrise în baza de date. Chiar dacă după momentul validării apare un defect care împiedică înscrierea normală a rezultatelor tranzacţiei în baza de date, acestea vor fi trecute în baza de date după reluarea activităţii. Rezultatele tranzacţiilor validate vor supravieţui oricărei căderi de sistem.

Mecanismul prin care re realizează proprietatea de durabilitate are la bază conceptul de jurnal. Jurnalul este un fişier secvenţial în care sunt înregistrate toate operaţiile efectuate de tranzacţiile din sistem. Jurnalul conţine istoria evoluţiei întegului sistem. El este folosit la reluarea activităţii de către procedurile de recuperare care vor completa eventualele operaţii neterminate ale tranzacţiilor care au fost validate înainte de apariţia defectului.

***Controlul tranzacţiilor cu instrucţiuni SQL***

     Există două clase de tranzacţii. Tranzacţii DML – care conţin un număr oarecare de blocuri DML şi pe care SGBD-ul (ex ORACLE) le tratează ca o singură entitate sau o singură unitate logică de lucru, şi tranzactii DDL  care conţin un singur bloc DDL.

O tranzacţie nouă este lansată fie imediat după conectarea la serverul de baze de date (de exemplu, printr-o sesiune SQL\*Plus) fie după o comandă care a încheiat tranzacţia precedentă (execuţia unui COMMIT sau ROLLBACK), când este întâlnit primul bloc executabil DML sau DDL. O tranzacţie se termină în una din următoarele situaţii :

     \* Întâlnirea comenzior COMMIT/ROLLBACK

     \* Sfârşitul comenzii DDL

     \* Apariţia anumitor erori (DEADLOCK)

     \* Întâlnirea comenzii EXIT – iesire din SQL\*Plus

     \* Apariţia unei erori de sistem

Un bloc DDL este executat automat şi de aceea implicit încheie o tranzacţie. După încheierea unei tranzacţii, următorul bloc executabil SQL va lansa automat următoarea tranzacţie.

*Erorile de Sistem*

            Când o tranzacţie este întreruptă de o eroare serioasă, de exemplu o eroare de sistem, întreaga tranzacţie este anulată. Aceasta previne erorile datorate modificărilor nedorite asupra datelor, şi realizează întoarcerea tabelelor la stările de după ultimul COMMIT. În acest fel SQL protejează integritatea tabelelor. Anularea automata este cauzată cel mai des de către o eroare de sistem, ca de exemplu o resetare a sistemului sau o cădere de tensiune. Erorile de tastare a comenzilor, ca de exemplu tastarea gresită a unor nume de coloane sau încercările de a realiza operaţii neautorizate asupra tabelelor altor utilizatori, nu întrerup tranzacţia şi nu realizează anularea automată. Aceasta se datorează faptului că aceste erori sunt detectate în cursul compilării (când un bloc SQL este scanat şi verificat), şi nu în timpul execuţiei.

            Următoarele instrucţiuni SQL sunt utilizate când apar finalizări (commit) sau refaceri (rollback) :

     \* COMMIT[WORK]

     \* SAVEPOINT *nume\_savepoint*

     \* ROLLBACK[WORK] to [SAVEPOINT] *nume\_savepoint*;

 De notat ca atât COMMIT cât şi ROLLBACK sunt instrucţiuni  SQL.

 Cele trei instrucţiuni SQL utilizate pentru controlul tranzacţiilor sunt explicate mai jos:

*COMMIT[WORK]*

  Sintaxa : COMMIT[WORK];

     \* Permanentizează schimbările din tranzacţia curentă

     \* Şterge toate punctele de salvare (savepoint) din tranzacţie

     \* Termină tranzacţia

     \* Eliberează toate blocările (Lock) tranzacţiei

     \* Cuvantul cheie WORK este opţional

     Utilizatorul trebuie să expliciteze sfârşitul tranzacţiei în programul aplicaţie utilizând COMMIT (sau ROLLBACK). Dacă nu se finalizează explicit tranzacţia şi programul se termină anormal, ultima tranzacţie executată va fi anulată.

     Finalizări implicite (commit) apar în următoarele situaţii :

          + înainte de o comandă DDL

          + după o comandă DDL

          + la închiderea normală a unei baze de date

Dacă introduceţi un bloc DDL după câteva blocuri DML, blocul DDL cauzează apariţia unui commit înaintea propriei execuţii, incheind tranzacţia curenta. Apoi, dacă blocul DDL este executat până la capăt, este şi înregistrat (commit).

*SAVEPOINT*

 Sintaxa : SAVEPOINT *nume\_savepoint*

        Exemplu :

                SAVEPOINT *terminare\_actualizari*

     \* Poate fi utilizat pentru a împărţi o tranzacţie în bucăţi mai mici

     \* Punctele de salvare (savepoints) permit utilizatorului să reţină toată munca sa la orice moment din timp, cu opţiunea de a înregistra mai târziu totul, a anula totul sau o parte din ea.

       Astfel, pentru o tranzacţie lungă, se pot salva părţi din ea, pe măsura execuţiei, la sfârşit înregistrându-se sau refăcându-se conţinutul iniţial. La apariţia unei erori nu trebuie executat din nou fiecare bloc.

     \* La crearea unui nou punct de salvare cu acelaşi nume ca al unuia dinainte, primul punct este şters.

     \* Numărul maxim de puncte de salvare pentru un proces utilizator este implicit 5. Această limită poate fi schimbată.

*ROLLBACK[WORK] to [SAVEPOINT] nume\_punct\_salvare*

     \* Instructiunea ROLLBACK este utilizata pentru a reface starea bazei de date.

     \* Cuvantul cheie "work" este opţional. Întoarcerea la un punct de salvare este de asemenea opţionala.

     \* Dacă se utilizează ROLLBACK fără clauza TO SAVEPOINT, atunci :

          + se termină tranzacţia

          + se anulează modificările din tranzacţia curentă

          + se şterg toate punctele de salvare din tranzacţie

          + se eliberează blocările tranzacţiei

*Întoarcerea la Nivel de Bloc*

             O parte a unei tranzacţii poate fi anulată. Dacă un singur bloc DML dă eroare, ORACLE va întoarce înapoi doar acel bloc. Această facilitate este cunoscută ca STATEMENT LEVEL ROLLBACK. Întoarcerea la nivel de bloc constă în faptul că dacă un singur segment DML dă eroare la execuţia unei tranzacţii, efectul lui este anulat, dar schimbările realizate de precedentul bloc DML în tranzactie nu vor fi anulate şi pot fi înscrise (COMMIT) sau întoarse (ROLLBACK) explicit de către utilizator.

            Dacă blocul este unul de tip DDL, înscrierea (commit) care precede imediat acest bloc nu este anulată (schimbările au fost făcute deja permanente). ORACLE realizează întoarcerea la nivel de bloc prin crearea unui punct de salvare implicit înainte de executarea fiecarei comenzi DML. Utilizatorul nu poate referi acest punct de salvare în mod direct.

            Astfel, dacă vă întoarceţi la un punct de salvare, atunci:

     \* se anulează o parte din tranzacţie ;

     \* se reţine punctul de salvare pentru întoarcere, dar se pierd toate celelalte puncte create după punctul de salvare respectiv ;

     \* se eliberează toate blocările de tabele şi linii.

*Întoarceri implicite*

              Întoarcerile implicite apar când se întâlnesc terminări anormale ale execuţiei (de exemplu când se întrerupe un proces utilizator). Întoarcerile implicite la nivel de bloc apar la eroarea de execuţie a unui bloc.

             Este recomandat ca tranzacţiile să se termine explicit utilizând COMMIT[WORK] ori ROLLBACK[WORK].

            Următorul exemplu ilustrează utilizarea unui punct de salvare şi a instrucţiunilor ROLLBACK şi COMMIT.

         INSERT INTO OFFICES VALUES

            ( 23, ‘LAS VEGAS’,’WESTERN’,null, null, 0 );

         SAVEPOINT insert\_done;

         UPDATE SALESREPS

            SET REP\_OFFICE = 23, MANAGER=106;

         ROLLBACK TO insert\_done ( modificările sunt abandonate );

         UPDATE SALESREPS

            SET REP\_OFFICE = 23, MANAGER=106

            WHERE  EMPL\_NUM=102 ; ( revizuim comanda UPDATE )

         UPDATE OFFICES

            SET MGR=102

            WHERE OFFICE=23 ;

         COMMIT;

*AUTOCOMMIT*

COMMIT sau ROLLBACK pot fi date manual sau automat, prin utilizarea opţiunii AUTOCOMMIT a comenzii SET. Opţiunea AUTOCOMMIT controlează   când schimbările dintr-o bază de date sunt făcute permanente.

 Există două setări :

             COMANDA + DESCRIEREA

*SET AUTO[COMMIT] ON*

 COMMIT este utilizat automat la fiecare INSERT, UPDATE sau DELETE

*SET AUTO[COMMIT] OFF*

COMMIT poate fi utilizată de utilizator explicit. De asemenea, COMMIT se execută după execuţia comenzilor DROP, ALTER, CREATE sau la ieşirea din SQL\*Plus. ROLLBACK poate fi executat explicit de către utilizator pentru refacerea bazei de date.

            De reţinut că SET este o comandă SQL\*Plus.

**Controlul concurenţei**

La sistemele în care o bază de date este accesată simultan de către mai mulţi utilizatori apar situaţii de coflict datorate accesului concurent la datele care constituie o resursă comună.

***Consistenţa la citire***

            Utilizatorii bazelor de date fac două tipuri de accesări asupra bazelor de date:

Operaţii de citire ( SELECT )

Operaţii de scriere ( INSERT, UPDATE, DELETE )

Cititorului şi scriitorului unei baze de date trebuie să i se garanteze o vedere consistentă asupra bazei de date. Cititorii nu trebuie să vizualizeze o dată care este în curs de modificare iar scriitorii trebuie să fie siguri că schimbările dintr-o bază de date sunt făcute într-un mod consistent : schimbările făcute de un scriitor să nu distrugă sau să intre în conflict cu schimbările pe care le face un alt scriitor.

             Scopul consistenţei la citire este acela de a asigura faptul că fiecare utilizator vede data ca fiind cea de la ultimul COMMIT, inainte ca o operatie DML să înceapă. Consistenţa la citire este implementată prin ţinerea unor copii parţiale ale bazei de date în segmente de întoarcere (ROLLBACK). Când de execută operaţii de scriere într-o bază de date, ORACLE va face o copie a datelor înainte de schimbare şi o va scrie într-un segment de întoarcere. Toţi cititorii, exceptându-i pe cei care au facut schimbările, încă mai văd baza de date care exista înainte ca schimbările să fie făcute – ei văd segmentul de întoarcere de fapt.

            Oricum, înainte ca schimbările să fie făcute permanente în baza de date, doar utilizatorul care modifică datele poate să vadă baza de date cu alteraţiile încorporate. Toţi ceilalţi văd baza de date nemodificată (fereastra din segmentul de întoarcere ). Aceasta garantează citirea unor date consistente care nu fac subiectul unor modificari în curs.

            Când execuţia unui bloc DML se încheie (commit), schimbările făcute în baza de date devin vizibile oricarui utilizator care execută SELECT. Modificările sunt făcute ‘universale’ şi acum toţi utilizatorii văd baza de date cu modificarile încorporate. Spaţiul ocupat de către ‘vechile’ date în segmentul de întoarcere este eliberat pentru a fi reutilizat.

Dacă tranzacţia este anulată (ROLLBACK), atunci toate schimbările sunt ‘anulate’ :

Versiunea veche (‘originală’) a bazei de date aflată în segmentul de întoarcere este scrisă înapoi (‘recuperată’) în baza de date.

Toţi utilizatorii văd baza de date existentă înainte de începerea tranzacţiei.

***Tranzacţii de citire***

Implicit, modelul consistent al ORACLE DBMS garantează cărezultatul execuţiei unui bloc este consistent. Înanumite situaţii se poate dori efectuarea unor interogărimultiple, asupra datelor din mai multe tabele şi se doreşteasigurarea  consistenţei datelor. Adică, rezultateledin orice tabelă sunt consistente în timp în raport curezultatele din orice altă tabelă.

            Comanda SQL : SET TRANSACTION READ ONLY este utilizată pentru a începe o tranzactie de citire exclusiv.

            Consistenţa la citire pe care READ ONLY o furnizează este implementată în acelaşi fel cu consistenţa la nivel de bloc – utilizând segmente de întoarcere. Fiecare bloc vede implicit o fereastră consistentă a datelor la momentul începerii blocului. Această facilitate este foarte folositoare pentru rapoarte care efectuează mai multe interogări asupra mai multor tabele, în timp ce alţi utilizatori actualizează aceleaşi tabele.

            Observaţii :

SET TRANSACTION READ ONLY este utilizată pentru a începe o tranzacţie doar de citire.

Sunt permise doar cereri ( blocuri SELECT ). Comenzile DML nu sunt permise. SELECT FOR UPDATE va genera o eroare.

O instrucţiune COMMIT, ROLLBACK, sau un bloc DDL va termina tranzacţia de citire ( de reţinut că blocurile DDL generează implicit suprascrieri – COMMIT ). În cazul blocurilor DDL, nu este dată nici o indicaţie referitoare la faptul că tranzacţia se termină implicit.

În timpul tranzacţiei de citire, toate cererile se referă la aceeaşi copie a bazei de date ( schimbările sunt efectuate înainte ca tranzacţia de citire să înceapa).

 Alţi utilizatori pot continua să citească sau să modifice datele.

             Urmatoarele instrucţiuni pot fi rulate pentru a extrage datele din tabelele SALESREPS şi OFFICES.

         COMMIT;

        SET TRANSACTION READ ONLY;

         SELECT EMPL\_NUM, NAME, REP\_OFFICE, TITLE

            FROM SALESREPS;

         SELECT OFFICE, CITY, REGION

            FROM OFFICES;

         COMMIT;

Ultimul COMMIT este necesar pentru a termina explicit tranzacţia de citire.

***Niveluri de izolare a tranzacţiilor***

**Serializarea**

Serializarea presupune derularea tranzacţiilor fără nici o posibilitate de intercalare a operaţiilor lor, fiind regula care impune programarea acestora în unităţi atomice. Acest lucru înseamnă că, o dată ce o tranzacţie a devenit activă, prin execuţia primei operaţii de citire sau scriere, tranzacţiile iniţiate de alţi utilizatori nu pot fi derulate (sunt blocate în stare de aşteptare) până când cea dintâi se încheie.

Totuşi, nici un sistem de baze de date nu impune o planificare atât de strictă a operaţiilor tranzacţionale, din următorul motiv: intercalarea operaţiilor tranzacţionale oferă oportunităţi deosebite de îmbunătăţire a performanţelor în folosirea concurenţială a resurselor, lucru care se poate determina prin numărul de tranzacţii încheiate într-o anumită perioadă de timp.

O astfel de abordare trebuie totuşi să respecte principiul seriabilizării, în caz contrar fiind ameninţată integritatea datelor. Cu alte cuvinte, chiar în condiţii concurenţiale, modul de derulare a tranzacţiilor trebuie să fie echivalent cu derularea serială ca unităţi atomice a acestora, adică situaţia în care fiecare tranzacţie s-ar fi derulat în ordine, după ce precedenta s-a încheiat complet.

Pentru a obţine un grad de concurenţă mai ridicat s-a recurs la „ slăbirea ” principiului seriabilizării pentru a scadea numărul de tranzacţii blocate în starea de aşteptare. Consecinţa mai puţin dorită, dar acceptabilă în anumite condiţii, va fi scăderea gradului de izolare a tranzacţiilor, crescând astfel posibilitatea executării operaţiilor tranzacţionale într-o ordine neechivalent serializabilă.

**REPLICAREA**

Un fenomen interesant în informatica actuală - şi cu precădere în domeniul bazelor de date - îl reprezintă manifestarea pregnantă a două tendinţe contradictorii: centralizarea şi distribuirea. Ambele tendinţe sunt animate de intenţia de a obţine anumite avantaje şi, desigur, preţul acestor avantaje îl reprezintă anumite dezavantaje. Dacă ne referim la baze de date, avantajul centralizării constă în posibilităţile mult mai simple de a asigura integritatea datelor şi consistenţa prelucrărilor, în timp ce descentralizarea aduce un spor de disponibilitate. Performanţa de ansamblu este un aspect care, în cele din urmă, "arbitrează" criteriile care se cer urmărite pentru un echilibru cât mai apropiat de optim între centralizare şi descentralizare.

Replicarea este o tehnologie care, în foarte multe situaţii, reprezintă un compromis acceptabil în această direcţie. Replicarea a devenit din ce în ce mai populară pe măsură ce posibilităţile de comunicaţii pentru date au devenit mai bogate, mai ales odată cu dezvoltarea Internet-ului..

**Replicarea** este un proces care constă în realizarea şi distribuirea unor copii ale datelor. Distribuirea acestor copii (numite "replici") are ca scop procesarea datelor în mod local.

**Definiţie**

Revenind acum la definiţie, am putea să spunem că replicarea este o tehnologie care permite ca informaţiile provenind de la una sau mai multe surse să fie distribuite către una sau mai multe ţinte şi, în plus, ca modificările intervenite în surse să fie propagate în mod consistent către ţintele corespunzătoare.

În primul rând, "sursele" şi "ţintele" sunt, în cazul nostru, baze de date. Vom restrânge chiar mai mult domeniul, referindu-ne în cele ce urmează doar la baze de date relaţionale. Este însă bine să reţinem că replicarea este o noţiune mai generală, fiind prezentă în domenii diverse, cum ar fi administrarea documentelor, mesagerie electronică, Internet etc.

Faptul că atât sursele cât şi ţintele pot fi multiple permite stabilirea unor relaţii diverse între acestea, ansamblul acestor relaţii definind ceea ce se numeşte de obicei "topologia replicării". În scenariile evocate mai sus, avem de-a face cu topologii ierarhice în primele două scenarii ("one-to-many" în cazul (a), "many-to-one" în cel de-al doilea caz) şi o topologie de tip reţea ("many-to-many") în ultimul scenariu (c).

Un alt aspect important este faptul că replicarea nu implică, de obicei, întreaga bază de date (caz în care avem de-a face cu o replicare totală). Cazul cel mai comun îl reprezintă replicarea unei tabele, dar adesea se recurge la replicarea unor submulţimi a datelor unei tabele sau a mai multor tabele, ceea ce ridică şi mai mult nivelul de complexitate a replicării.

Procesul prin care se asigură capturarea, propagarea şi reproducerea la ţinte a actualizărilor din surse se numeşte **sincronizare**, fiind elementul central al întregii tehnologii. Sincronizarea asigură caracterul dinamic al întregului proces. Un termen alternativ pentru sincronizare este "**reîmprospătare**" (refresh), deşi în anumite contexte sensul său este mai limitat (se referă doar la anumite tipuri de sincronizare).

O ultimă precizare se referă la obiectul sincronizării. Din cele prezentate până acum s-ar putea deduce că sincronizarea se referă exclusiv la date. Cu toate acestea, expresia "propagarea actualizărilor" sugerează faptul că sincronizarea poate fi privită şi dintr-o perspectivă procedurală: este posibil ca ceea ce se transmite de la sursă către ţintă să nu fie de fapt noile valori ce trebuie memorate, ci chiar operaţiile de actualizare aplicate asupra sursei, astfel încât acestea să fie aplicate şi asupra replicilor. Cu alte cuvinte, replicarea nu se referă doar la distribuirea datelor, ci şi la distribuirea prelucrărilor. Aşa cum vom vedea în continuare, în practică se utilizează o combinaţie de tehnici.

**Sincron şi asincron**

O primă diviziune a tehnicilor de replicare se bazează pe momentul în care se realizează sincronizarea. Din acest punct de vedere, replicarea poate fi sincronă sau asincronă.

Replicarea sincronă implică faptul că o actualizare în sursă trebuie propagată şi aplicată imediat în toate replicile. Acest proces se realizează de obicei pe baza unui protocol numit "comitere în două faze" (two-phase commit) şi se referă la tranzacţia care realizează actualizarea. În prima fază, sursa (un server de baze de date) va solicita ţintelor (alte sisteme de baze de date) să se pregătească pentru efectuarea tranzacţiei şi va aştepta confirmările din partea acestora. Când toate ţintele sunt pregătite, sursa le va solicita comiterea tranzacţiei şi va aştepta ca fiecare ţintă să raporteze succesul. Când toate ţintele au comis cu succes tranzacţia, sursa declară tranzacţia încheiată şi consemnează acest lucru în jurnal (log). În cazul în care într-una dintre faze cel puţin o ţintă nu poate comite tranzacţia, orice modificări produse deja sunt anulate (rollback).

Este evident că replicarea sincronă impune un două cerinţe esenţiale: reţeaua (fie ea LAN sau WAN) trebuie să funcţioneze şi fiecare bază de date ţintă să poată să comită tranzacţia care realizează actualizarea. Aceste cerinţe pot fi îndeplinite doar în anumite medii, cu anumite costuri, iar balanţa este înclinată în favoarea integrităţii procesării şi în defavoarea disponibilităţii. Anumite aplicaţii, cum ar fi de exemplu cele financiare sau cele de rezervare a locurilor, sunt bine deservite de această tehnică.

Care este diferenţa între replicarea sincronă şi tranzacţiile distribuite? Este greu de trasat o linie de demarcaţie clară. În principiu, o tranzacţie distribuită nu lucrează (în mod deliberat) cu copii ale datelor, ci cu date care sunt distribuite în mai multe locaţii. Astfel, este posibil ca prelucrarea unei comenzi de la un client să implice atât date ale serviciului de vânzări (situat la o anumită locaţie) cât şi date ale serviciului financiar (situat într-o altă locaţie). Din punctul de vedere al aplicaţiei, localizarea datelor este un proces transparent.

În cazul replicării sincrone, un scenariu tipic este următorul: O tranzacţie T încearcă să actualizeze (printre altele) nişte date care reprezintă sursa unei replicări. Pentru ca această actualizare să se propage imediat, sistemul generează o tranzacţie specială P, care să actualizeze (independent de T) replicile datelor respective. Comiterea cu succes a tranzacţiei T este condiţionată de comiterea tranzacţiei P. Să remarcăm că T poate fi tranzacţie simplă (locală), în timp ce P este o tranzacţie distribuită (care este însă transparentă din perspectiva aplicaţiei).

Replicarea asincronă se bazează pe un mecanism de tip store and forward (adesea se foloseşte şi termenul message based replication). Informaţiile despre actualizările produse în sursa replicării sunt stocate într-o coadă de aşteptare, de unde sunt apoi trimise către locaţiile ţintelor, care se sincronizează pentru a reflecta starea sursei. Acest proces implică o anumită întârziere, care poate fi de câteva secunde sau de câteva zile, în funcţie de configurarea sistemului şi de disponibilitatea sistemelor implicate. În schimb, avantajele pot fi considerabile din punct de vedere al disponibilităţii (operarea într-o locaţie nu este afectată de indisponibilitatea unei alte locaţii) şi a costurilor legate de transferul informaţiilor (nu este nevoie de o conexiune permanentă).

Deoarece replicarea sincronă este adesea asimilată cu bazele de date distribuite şi implică un nivel de complexitate foarte ridicat, ceea ce se traduce în costuri adesea foarte mari (atât pentru comunicaţii cât şi pentru proiectare şi exploatare), replicarea asincronă - mult mai practică şi mai ieftină - a devenit alternativa cea mai populară în ultima vreme. În plus, replicarea sincronă este inaplicabilă într-o întreagă clasă de aplicaţii, din ce în ce mai răspândită, care implică utilizatori mobili (ca în cazul celui de-al doilea scenariu evocat). Din aceste motive, în foarte multe lucrări şi articole, termenul "replicare" se referă implicit la replicarea asincronă. În cele ce urmează mă voi referi în mod preponderent la replicarea asincronă.

**Tranzacţional sau nu**

Înainte de a detalia mai departe mecanismele care stau la baza replicării, merită să insistăm puţin asupra unor aspecte legate de consistenţa de ansamblu a unei baze de date distribuite prin replicare.

Pentru aceasta, trebuie să revenim puţin la noţiunea de tranzacţie . Implementarea acestora se face în majoritatea sistemelor comerciale prin mecanisme de blocare (lock - încuietoare, zăvor) care, pe baza unui protocol bine stabilit, interzic accesul altor tranzacţii la datele asupra cărora acţionează deja o tranzacţie. Blocarea este însă un mecanism care funcţionează în mod sincron, deci este inaplicabil în cazul replicării asincrone. Aceasta înseamnă că există oricând posibilitatea ca efectele unei tranzacţii să nu fie vizibile într-una sau mai multe replici, ceea ce conduce la faptul că, cel puţin între două sincronizări consecutive, ansamblul bazei de date (considerând aici şi replicile) să fie inconsistent.

Să considerăm scenariul (a) şi să ne imaginăm că se produce o tranzacţie care modifică preţul produsului P de la p1 la p2. Teoretic, tranzacţia poate fi considerată încheiată abia atunci când modificarea a fost efectuată în tabela "părinte" (de la sediul central) şi în toate replicile sale. Replicarea fiind asincronă, filialele vor continua să lucreze cu preţul p1 până la sincronizare. Mai mult, replicile nu se sincronizează simultan, deci este posibil ca anumite filiale să lucreze cu un preţ în timp ce altele lucrează cu alt preţ. Iar dacă această inconsistenţă nu pare suficient de "flagrantă", să ne imaginăm ce se întâmplă în situaţia în care produsul P nu mai este disponibil pentru vânzare şi este şters din baza de date centrală.

Există mai multe variante de a trata astfel de situaţii, în funcţie de posibilităţile tehnice şi, mai ales, de activitatea modelată (business rules).

O primă posibilitate este de a accepta posibilitatea de a avea stări inconsistente. Dacă firma consideră că nu-i grav ca într-un interval rezonabil de timp (să zicem, 2-3 ore) produsul să fie vândut cu preţuri diferite de diverse filiale atunci, o replicare non-tranzacţională este acceptabilă. Aceasta înseamnă că modificările sunt transmise către replici fără nici o informaţie despre tranzacţiile care le-au produs (se foloseşte şi termenul de replicare linie cu linie - row by row replication).

O variantă "forte" este de a combina replicarea asincronă cu replicarea sincronă. Anumite operaţii critice (în cazul cărora compromisul este inacceptabil) ar putea fi forţate la o replicare sincronă. De pildă, la ştergerea unui produs din nomenclatorul central, baza de date va încerca să utilizeze canalele de comunicaţie disponibile pentru a lua legătura cu toate filialele şi a aplica protocolul two phase commit, chiar dacă în acest fel tranzacţia în cauză va dura câteva minute (sau chiar ore). Desigur, trebuie să existe posibilităţile tehnice (măcar conexiuni dial-up) şi să fie acceptabil din perspectiva activităţii respective. Dacă, de exemplu, se tranzacţionează intens iar baza de date nu permite blocarea la nivel de linie ci doar la nivel de tabelă, s-ar putea ca o astfel de tranzacţie lungă să oprească toată activitatea firmei.

În fine, o altă variantă o reprezintă "tranzacţiile întârziate". Aceasta înseamnă că replicarea va ţine seama nu doar de modificările în sursă, ci şi de tranzacţiile care le-au produs. Deocamdată, este suficient de menţionat că se transmit către ţinte doar modificările produse de tranzacţii deja comise în sursă şi că aceste modificări sunt trimise în ordinea în care s-au produs, însoţite de mărci de timp (timestamps). Este vorba, în principiu, de un fel de replicare a tranzacţiilor. E importat de notat faptul că se recurge la o serializare a actualizărilor. Dacă, de pildă, în sursă o tranzacţie T1 a modificat preţul produsului P de la p la p+1, după care o tranzacţie T2 a modificat preţul aceluiaşi produs la p\*2, este esenţial ca aceste modificări să fie efectuate în ţintă în aceeaşi ordine.

Există şi alte modalităţi de tratare a acestor probleme, dar nu există nici o garanţie că ansamblul de date va fi în permanenţă consistent. Este un compromis obligatoriu pe care replicarea asincronă îl impune. Vom vedea pe parcurs şi care sunt tehnicile prin care se pot obţine variante cât mai avantajoase.

**Unidirecţional sau bidirecţional**

Cea mai simplă formă de replicare o reprezintă replicarea unidirecţională. Altfel spus, replica este disponibilă doar pentru citire. Deoarece o astfel de replică read-only reprezintă doar o imagine a datelor din sursă aşa cum se prezentau ele la un anumit moment (un instantaneu), se foloseşte adesea termenul **snapshot.**

Este interesantă istoria acestui termen. Iniţial, el a fost utilizat în unele baze de date relaţionale centralizate (Oracle), pentru a "îngheţa" anumite seturi de date.

Un **snapshot** se defineşte exact ca un view (deci prin intermediul unei selecţii) dar, spre deosebire de acesta, este o tabela stocată - nu virtuală, ca în cazul unui view. Un snapshot este accesibil doar pentru citire şi este "reîmprospătat" la anumite intervale de timp. Rolul unui snapshot într-o bază centralizată era (şi încă este) de a permite efectuarea unor operaţii de analiză bazate pe interogări complexe, fără ca acestea să interfereze (prin blocări) cu procesările "pe viu", rezultând astfel performanţe superioare atât pentru procesările tranzacţionale (OLTP) cât şi pentru cele de analiză (OLAP).

De exemplu când a păşit în lumea procesării distribuite, Oracle s-a folosit de acest mecanism, furnizând astfel "prima generaţie" de instrumente de replicare. Termenul s-a impus, aşa că atunci când Oracle a introdus o tehnologie de replicare bidirecţională ("simetrică") a păstrat termenul, introducând noţiunea de "updatable snapshot", deşi este o flagrantă contradicţie de termeni.

Atenţie deci la terminologie: Oracle foloseşte termenul snapshot ca sinonim pentru replică, în timp ce restul lumii înţelege prin snapshot doar "replică read-only" (mai mult, Sybase numeşte snapshot o replică read-only non-tranzacţională).

**Replicarea bidirecţională** implică faptul că replica este la rândul ei actualizabilă şi că modificările din replică vor fi reflectate în sursă. Observaţi că în acest caz noţiunile de "sursă" şi "replică" îşi cam pierd sensul: o sursă poate fi considerată replică şi invers, motiv pentru care se foloseşte adesea termenul de "replicare simetrică" (introdus tot de Oracle). De altfel, trebuie spus că atunci când vom lua în considerare diferitele topologii posibile pentru replicare, nici termenul "bidirecţional" nu este foarte corect (de fapt, procesul este multidirecţional) şi nici "simetric" (câteodată una dintre locaţii este "mai simetrică" decât celelalte).

**Configuratii Combinate**

În lumea reală situaţiile nu sunt atât de "pure" cum apar prezentate în secţiunea precedentă. În cele mai multe cazuri, avem de-a face cu configuraţii combinate sau cu configuraţii paralele în cadrul aceleiaşi aplicaţii. În scenariul (a) nomenclatorul central aparţine locaţiei centrale, dar datele despre clienţi pot aparţine filialelor (replicate apoi la celelalte filiale, eventual prin intermediul unui nod de referinţă situat la nivel central), în timp ce procesarea comenzilor se poate face în regim de workflow ownership. De pildă, preluarea comenzilor se face la nivelul magazinelor, livrarea se face la nivelul filialei, facturarea se efectuează la nivelul central etc.

În plus, mecanismul relativ simplu de tip publish and subscribe pe care se bazează administrarea replicării se complică şi mai mult când luăm în considerare faptul că nu întotdeauna replicarea implică doar tabele întregi (aşa cum, oarecum implicit, am considerat până acum), cu atât mai puţin baze de date întregi. În multe cazuri, datele care trebuie replicate constituie o submulţime a liniilor şi/sau a coloanelor unei tabele, dar pot fi şi date compuse printr-o definiţie de tipul celor folosite pentru crearea unui view, implicând mai multe tabele legate prin asocieri complexe. Uneori trebuie replicat un grup de obiecte ale bazei de date.

Dacă la aceasta mai adăugăm faptul că există şansa ca sistemul de baze de date să nu fie omogen, implicând diverse SGBD-uri (de pildă, utilizatorii mobili nu vor putea purta într-un notebook un server SQL complet), uneori de la producători diferiţi, obţinem un tablou destul de bogat al nivelului de complexitate la care se poate ajunge.

Cu toate acestea, replicarea poate fi stăpânită. Există astăzi numeroase produse dedicate configurării, administrării şi monitorizării procesului de administrare. Pentru a putea evalua avantajele şi dezavantajele acestora în funcţie de problema pe care o aveţi de rezolvat, este necesar să cunoaşteţi la modul general etapele procesului de replicare şi câteva dintre variantele tehnologice ale fiecărei etape. În cele ce urmează voi încerca să schiţez principalele elementele ale acestei problematici.

Înainte de a începe, iată etapele procesului:

* Stabilirea surselor şi a ţintelor;
* Capturarea actualizărilor şi evaluarea acestora în vederea propagării;
* Propagarea actualizărilor de la sursă la ţintă;
* Aplicarea actualizărilor la baza de date ţintă (cu detectarea şi rezolvarea eventualelor conflicte).

**Configuraţii, topologii, proprietate**

Diverse produse utilizează diverse terminologii pentru a descrie modul de configurare a replicării, oferind desigur şi posibilităţi diferite. Una dintre cele mai simple şi evocatoare terminologii este cea bazată pe metafora publish and subscribe (publicare şi abonare). Deşi utilizată ca atare doar de Microsoft, cu anumite adaptări este valabilă la majoritatea produselor.

Pe scurt, baza de date (serverul) care va servi ca sursă pentru replicare va fi numită "editor" (publisher). În cadrul acestei baze de date se definesc aşa-numite "publicaţii" (publications), care sunt de fapt seturile de date disponibile pentru replicare. Alte baze de date se pot "abona" (subscribe) la aceste publicaţii. Pentru a defini procesul de replicare se specifică editorul, publicaţia şi abonatul, precum şi nişte informaţii adiţionale, cum ar fi tipul replicării (unidirecţională sau bidirecţională), intervalul de timp între sincronizări, modul de rezolvare a conflictelor etc. În principiu, orice server poate să fie editor al anumitor publicaţii şi totodată abonat la publicaţii editate de alte servere. Mai mult, se poate ajunge la situaţii în care aceeaşi tabelă poată să conţină date sursă şi date replicate (prin partiţionare).

O problemă ceva mai delicată apare atunci când replicarea este selectivă, în sensul că "publicaţia" nu este o tabelă întreagă. În acest caz se recurge la un procedeu numit data subsetting. Replicarea selectivă este importantă din cel puţin două motive. În primul rând, astfel se poate minimiza traficul în reţea (să ne imaginăm că fiecare produs din nomenclator este însoţit de scheme, desene sau chiar videoclipuri de prezentare - replicarea acestora poate fi extrem de costisitoare). În al doilea rând, este foarte posibil ca drepturile de acces la anumite date să fie rezervate proprietarului, caz în care replicarea acestora afectează securitatea sau confidenţialitatea informaţiilor.

De regulă, indiferent de modul de specificare a sursei, se ajunge la o instrucţiune SELECT prin care se defineşte subsetul dorit. Restricţiile impuse asupra acestei selecţii depind de posibilităţile fiecărui produs în parte, dar câteva sunt în general valabile. Cel mai important aspect este că fiecare linie selectată trebuie să corespundă unei linii din tabela sursă. Aceasta înseamnă, la modul minimal, că subsetul de coloane selectat trebuie să cuprindă toate coloanele care formează cheia primară a tabelei sursă. În majoritatea produselor de replicare, în selecţia datelor pentru replicare nu se pot folosi funcţii de agregare sau distinct, nu se admit clauze GROUP BY sau CONNECT BY, asocieri (join), operaţii cu seturi (de pildă UNION) şi anumite forme de interogări înglobate.

Pentru exemplificare, mă voi referi la Oracle8 Replication, unde definirea "publicaţiei" se face printr-o instrucţiune de tipul:

CREATE SNAPSHOT nume

AS selecţie.

Oracle se referă la două niveluri de complexitate pentru replicare: nivelul de bază (lucrează doar unidirecţional) şi nivelul avansat (poate lucra şi bidirecţional). Replicile pot fi la rândul lor simple sau complexe. Ambele categorii pot fi folosite în replicarea de bază, însă numai replicile simple pot fi actualizabile.

O replică simplă se bazează pe o singură tabelă sursă iar selecţia pe care se bazează se supune tuturor restricţiilor amintite mai sus. Dacă vom considera că în scenariul (a) avem în baza de date centrală o tabelă Produse având coloanele cod, den, preţ, comandat şi disponibil, atunci o replică se poate crea astfel:

CREATE SNAPSHOT produse AS

SELECT cod, den, pret FROM produse@centru.ro

În acest caz, replica preia doar o submulţime a coloanelor tabelei sursă. Actualizările care afectează în sursă doar coloanele excluse din replică nu vor fi propagate. Prin folosirea unei clauze WHERE se pot selecta pentru replicare doar anumite linii:

CREATE SNAPSHOT produse AS

SELECT \* FROM produse@centru.ro p

WHERE p.disponibil > 100

O caracteristică interesantă în tehnologia de replicare de la Oracle este că selecţia liniilor se poate baza şi pe parcurgerea referinţelor many-to-one în tabele asociate. În exemplul care urmează, locaţia centrală va prelua de la locaţia locală Mureş doar comenzile care corespund unor clienţi din judeţul Mureş:

CREATE SNAPSHOT comenzi AS

SELECT \* FROM comenzi@mures.ro a

WHERE EXISTS

(SELECT cod

FROM clienti@mures.ro b

WHERE a.cod\_client = b.cod

AND b.jud = 'MS');

Replicile complexe nu au limitările replicilor simple, în schimb nu pot fi actualizabile iar sincronizarea nu se poate face incremental.

Este utilă o analogie între definirea unei replici şi definirea unui view. Restricţiile privind actualizarea unui view sunt de obicei aceleaşi în cazul unei replici. Este interesant de remarcat că instrumentele de replicare de la IBM permit (e drept, printr-o configurare specială) replicarea unui view. O altă remarcă este că Oracle face o importantă distincţie între replicile actualizabile (updatable snapshots) şi replicarea simetrică între servere (multimaster replication). În acest ultim caz nu se admite replicarea selectivă.

În fine, trebuie spus că la configurarea replicării, sistemele generează diverse obiecte specifice în bazele de date, obiecte care vor fi apoi utilizate pentru replicare. De exemplu, Oracle generează automat pentru fiecare snapshot un index bazat pe cheia primară şi un view. Pentru fiecare sursă de replicare, Oracle construieşte un jurnal (snapshot log), materializat într-o tabelă care cuprinde, pentru fiecare linie actualizată cheia primară, marca de timp şi, eventual identificatorul unic de linie (ROWID). Informix foloseşte un catalog global al replicării (cu meta-date despre fiecare surse, participanţi, definiţii, opţiuni etc.), care este la rândul său replicat în toate serverele implicate în replicare. Alte sisteme folosesc tabele adiţionale pentru modificări, tabele "umbră" (shadow), jurnale specializate etc.

**Capturarea modificărilor**

Odată ce sursele şi ţintele au fost stabilite, următoarea problemă o reprezentă mecanismele prin care programele (sau procesele, în cazul implementărilor în SGBD) de replicare să sesizeze comiterea unei modificări în sursă şi să poată identifica elementele acestei actualizări.

Există două mari categorii de mecanisme de replicare: bazate pe "declanşatoare" (triggers) şi bazate pe jurnal (log).

Capturarea actualizărilor prin "declanşatoare" este cea mai veche. Un trigger este o procedură stocată ceva mai specială, deoarece nu este apelată explicit, ci se execută automat ca răspuns la tentativa de a actualiza (prin INSERT, DELETE sau UPDATE) tabela căreia îi este ataşat trigger-ul. Unele sisteme permit specificarea unor anumite coloane din tabelă a căror modificare declanşează trigger-ul (desigur, pentru UPDATE).

Un trigger se defineşte într-un limbaj procedural, care variază de la server la server (în ultima vreme mai mulţi producători au introdus în produsele lor posibilitatea de a scrie proceduri stocate în Java). Iată un exemplu, în care voi folosi un limbaj procedural fictiv dar sugestiv (inspirat din Oracle şi Postgres):

CREATE TRIGGER produse\_up

BEFORE UPDATE ON prod

FOR EACH ROW

BEGIN

IF OLD.cod != NEW.cod THEN

DELETE prod

WHERE cod = ODL.cod;

INSERT INTO prod VALUES

(NEW.cod, NEW.den, NEW.pret);

END IF;

END;

Exemplul este sugestiv pentru posibilităţile unui trigger. În primul rând, declanşatorul este legat de o tabelă (în cazul nostru prod) şi de o anume actualizare (aici UPDATE, dar se pot menţiona mai multe, conectate prin OR). Se remarcă faptul că acest trigger se declanşează înainte de efectuarea actualizării (opţiunea AFTER ar lansa trigger-ul după ce s-a realizat actualizarea). În fine, constatăm că trigger-ul are acces atât la valorile dinainte de actualizare (OLD) cât şi la cele de după actualizare (NEW). Codul în sine nu este grozav: în cazul în care se încearcă modificarea cheii primare, comanda UPDATE este înlocuită de o ştergere urmată de o înserare. Este posibil ca inserarea să fie şi ea "păzită" de un trigger, ilustrând astfel declanşarea în cascadă. Nu voi intra în detalii legate de valoarea de retur, de parametrizare etc.

Acest mecanism a fost introdus de producătorii de baze de date mai ales pentru a permite controlul procedural al integrităţii datelor, dar a fost utilizat şi pentru a implementa o primă generaţie de aplicaţii de replicare. Este uşor de imaginat un set de trigger-e care adaugă într-o coadă mesaje privind actualizările efectuate într-o tabelă.

Dezavantajul major al acestei abordări este că trigger-ele se execută în baza de date, fiind procese destul de consumatoare de resurse, fapt care afectează performanţele. Pe de altă parte, trigger-ele utilizator nu au acces la informaţii despre tranzacţia în cadrul cărei se execută, deci nu permit decât replicarea la nivel de linie. În fine, apar şi problemele de interferenţă cu trigger-e destinate unor alte scopuri (de pildă validări complexe de consistenţă).

Capturarea modificărilor din jurnal este o metodă oarecum mai simplă. Orice server de baze de date serios jurnalizează într-un fel sau altul tranzacţiile, în scopul de a oferi o posibilitate de refacere a bazei de date în cazul unui incident. Sistemele de jurnalizare sunt diverse, dar în general se consemnează un identificator al tranzacţiei, utilizatorul care a comis tranzacţia, actualizările realizate, valorile intermediare (dinainte şi de după fiecare actualizare) şi momentul comiterii. Numărul mare de informaţii jurnalizate se explică prin faptul că anularea unei tranzacţii (rollback) se face tot pe baza acestor informaţii.

Mecanismul de capturare poate fi integrat în mecanismul de jurnalizare, caz în care identificarea tranzacţiilor care trebuie replicate se poate face dinamic, sau poate fi exterior, caz în care jurnalul este examinat periodic, într-o ordine strictă. Deoarece unele sisteme scriu în jurnal într-o manieră circulară (cele mai noi intrări le suprascriu pe cele mai vechi intrări), procesele de jurnalizare şi capturare trebuie corelate astfel încât capturarea să nu rămână în urmă şi să rateze anumite tranzacţii (în astfel de situaţii, toate tranzacţiile în curs de desfăşurare sunt blocate până când programul de capturare avansează suficient - situaţia poate apare la o defectuoasă configurare a spaţiului rezervat fişierelor log). Deşi unele sisteme marchează chiar în jurnal tranzacţiile care trebuie replicate, majoritatea recurg la structuri de stocare specializate (de genul unui "jurnal consolidat" sau a unor tabele care consemnează schimbările).

Avantajul capturării bazate pe jurnal constă în faptul că se pot identifica tranzacţiile şi că nu se produc interferenţe cu procesările curente (exceptând situaţii de genul celei descrise mai sus). Pe de altă parte, programul sau procesul care realizează capturarea implică procesări destul de complexe şi, mai ales, multe operaţii de transfer de date, ceea ce conduce la o încărcare mai mare a serverului şi/sau a reţelei.

Ambele tehnici sunt folosite şi fiecare producător insistă pe avantajele soluţiei pe care a ales-o. Oracle merge pe soluţia trigger, dar într-o variantă optimizată. Trigger-ele pentru replicare sunt standardizate şi, mai ales, internalizate. Aceasta înseamnă că sunt compilate şi incluse chiar în motorul bazei de date, ceea ce conduce la performanţe foarte bune, plus posibilitatea de a avea acces la toate informaţii relative la tranzacţiile utilizatorilor care realizează actualizări. Sybase se bazează pe un software specializat (Replication Server) care se ocupă numai de replicare. O componentă specializată a acestuia (Log Transfer Manager) rulează alături de fiecare server sursă şi alimentează serverul de replicare cu actualizările capturate din jurnal. Informix şi IBM folosesc de asemenea mecanisme bazate pe jurnal.

**Evaluarea datelor pentru replicare**

Toate sistemele de replicare încearcă să minimizeze volumul informaţiilor care trebuie propagate. În acest sens, se recurge adesea la o evaluare a informaţiilor capturate despre modificări. Această etapă se desfăşoară de regulă la momentul compunerii mesajelor care vor fi depuse în coadă. În cazul sistemelor bazate pe trigger-e, este posibil ca o parte din aceste procesări să fie realizate chiar de trigger-ul de capturare. Pe de altă parte, este posibil ca trigger-ele să nu scrie direct în coada de mesaje ci într-un jurnal propriu, de unde informaţiile pot fi evaluate în vederea replicării (aşa procesează Oracle cu replicile actualizabile).

Pentru exemplificare, mă voi referi la Informix Enterprise Replication. O primă evaluare se referă la momentul comiterii tranzacţiei şi este necesară pentru a plasa în coadă doar tranzacţii deja comise. Importantă este însă evaluarea imaginii liniilor care trebuie replicate. Această evaluare are ca scop determinarea efectului net al unui serii de actualizări care se realizează asupra unei linii în cadrul unei singure tranzacţii. În felul acesta se evită propagarea fiecărei operaţii şi se trimite doar o singură actualizare (sau poate nici atât), ceea ce minimizează atât traficul cât şi volumul actualizărilor la nodul ţintă.

Deoarece în cadrul unei tranzacţii operaţiile sunt strict secvenţiale, este suficient să analizăm modul cum se evaluează grupuri de două actualizări (actualizarea rezultantă este apoi evaluată împreună cu următoarea şi aşa mai departe).

Cazul cel mai simplu este cel în care inserarea unei linii este urmată de ştergerea liniei, caz în care efectul este nul. Dacă însă inserarea este urmată de o modificare (UPDATE) atunci se verifică dacă imaginea finală a liniei corespunde criteriului de selecţie în vederea replicării. Dacă nu, efectul este iarăşi nul. Dacă da, atunci efectul net este inserarea unei linii cu imaginea finală rezultată.

Dacă o modificare a unei linii este urmată de ştergerea linei, atunci trebuie aplicat criteriul de selecţie asupra rezultatului modificării. Dacă linia rezultată nu trebuie replicată atunci rezultatul net este nul. Dacă însă linia modificată corespunde criteriului, atunci rezultatul este ştergerea imaginii iniţiale a liniei.

De exemplu, dacă replicarea se referă doar la clienţii din judeţul Mureş, atunci tranzacţia următoare:

BEGIN WORK;

INSERT INTO pers

VALUES (1325, 'Ion', 'MS');

UPDATE pers SET nume = 'Vasile'

WHERE marca = 1325;

COMMIT WORK;

va avea ca efect net trimiterea către ţintă a operaţiei:

INSERT INTO pers

VALUES (1325, 'Vasile', 'MS');

Dacă mai apoi se execută următoarea tranzacţie:

BEGIN WORK;

UPDATE pers SET jud = 'CJ'

WHERE marca = 1325;

DELETE FROM pers

WHERE nume = 'Vasile';

COMMIT WORK;

efectul net va fi replicarea operaţiei:

DELETE FROM pers

WHERE marca = 1325

AND nume = 'Vasile'

AND jud = 'MS';

Observaţie: Desigur, ar fi suficient să se specifice cheia primară pentru ştergere. Condiţiile suplimentare sunt însă utile pentru detectarea unor eventuale conflicte.

Dacă o linie care nu corespundea criteriului va fi actualizată astfel încât să corespundă criteriului, rezultatul net este inserarea în replică a unei linii cu imaginea finală. O situaţie inversă conduce la o ştergere. Este, cred suficient de clar, modul în care se tratează alte combinaţii.

Un caz particular îl reprezintă grupurile de modificări care schimbă cheia primară a unei linii. În acest caz, dacă atât imaginea iniţială cât şi imaginea finală corespund criteriului de selecţie pentru replicare, se procedează la ştergerea liniei iniţiale şi inserarea liniei finale.

O precizare: câmpurile de tip blob sau text sunt tratate oarecum diferit.

**Mecanisme de propagare**

Propagarea modificărilor se realizează printr-un mecanism bazat pe cozi de mesaje (message queuing). Această tehnologie este suficient de complexă pentru a merita o tratare separată dar, în acest context, este suficient dacă menţionez că aceste mecanisme (fie integrate în baza de date, fie implementate ca middleware) garantează stocarea sigură a mesajelor atât la sursă cât şi la destinaţie, garantează livrarea fiecărui mesaj şi permit urmărirea fiecărui mesaj în parte. Deocamdată, o analogie cu sistemele de poştă electronică este suficientă (unele sisteme folosesc chiar MAPI), deşi produsele de tip MOM (Message Oriented Middleware) sunt adesea mult mai complexe.

În principiu, actualizările capturate se consemnează în nişte mesaje stocate într-o "coadă stabilă" (care nu poate fi afectată de incidente soft sau hard) pe sistemul sursă. Mesajele sunt preluate de mecanismul de transport asincron şi sunt expediate către ţinte, unde sunt stocate tot în cozi stabile. Prin acelaşi mecanism de mesagerie, ţintele răspund cu mesaje de confirmare pentru fiecare mesaj primit. Odată de recepţia a fost confirmată de toţi destinatarii, mesajul poate fi şters din coada de plecare. Protocolul poate fi destul de complex, implicând retransmiterea mesajelor, mecanisme de verificare etc. Este important ca mesajele să poată fi serializate înainte de aplicare, să nu se piardă şi să nu fie dublate.

În general, replicarea se face incremental: fie se trimit operaţiile care au fost efectuate asupra sursei de la ultima sincronizare (eventual evaluate ca efect net al tranzacţiilor care le-au realizat), fie se trimit aşa-numite "fişiere delta", care cuprind diferenţele intervenite în imaginile liniilor între două sincronizări succesive (în cazul replicării non-tranzacţionale). Există însă situaţii în care se recurge la replicarea totală a unei surse. Un caz tipic este cel în care un server ţintă este indisponibil o perioadă mai lungă de timp. În această situaţie, mesajele din coadă care-i sunt adresate nu vor putea fi şterse iar spaţiul ocupat va deveni foarte mare. Soluţia este îndepărtarea ţintei respective din configuraţia de replicare. De îndată ce serverul este din nou disponibil, este reintrodus în configuraţie, ceea ce va conduce la o replicare totală a sursei (echivalentă cu ştergerea tuturor liniilor şi inserarea tuturor liniilor din sursă).

**Aplicarea actualizărilor**

După ce mesajele cuprinzând actualizările sosesc la serverul ţintă, ele trebuie aplicate (de preferinţă cât mai repede) asupra bazei de date ţintă. Procesul în sine nu este interesant. Eventual se poate menţiona faptul că există posibilitatea ca regulile de integritate din baza de date ţintă să fie diferite de cele din sursă şi că ele vor superviza actualizările. Aceasta înseamnă că deşi la sursă actualizările au fost acceptate, ele pot fi respinse la baza de date ţintă. Situaţiile de acest gen nu sunt tipice, ci fiind mai degrabă rezultatul unor erori de proiectare sau configurare.

Ca regulă generală, în cadrul configuraţiilor de tip master/slave regulile de integritate din baza de date slave sunt mai relaxate decât cele din master, iar în cadrul configuraţiilor peer-to-peer regulile de integritate sunt echivalente în toate locaţiile. În aceste condiţii se poate presupune că nu vor fi replicate decât actualizări realizate de tranzacţii valide la sursă.

Chiar şi în aceste condiţii, caracterul asincron al replicării poate conduce la situaţii de conflict. Ajungem astfel la subiectul cel mai interesant din cadrul acestei secţiuni: detectarea şi rezolvarea conflictelor.

În primul rând, nu orice anomalie este un conflict. Considerând scenariul (a), o actualizare a preţului unui produs P la sediul central efectuată la momentul T1 va fi replicată la filiala Ms la momentul T2. Între momentele T1 şi T2, filiala Ms va vinde produsul P cu un preţ incorect. E o anomalie, dar nu e un conflict. O astfel de anomalie nu poate fi detectată de mecanismul de replicare, deci rămâne pe seama aplicaţiei. O posibilă variantă ar fi ca (în locaţia replicată) un trigger de actualizare să caute în tabela de vânzări toate înregistrările introduse după momentul actualizării linei respective în locaţia primară şi să le listeze (urmând ca rezolvarea situaţiei să fie făcută manual).

Conflictele sunt caracteristice configuraţiilor cu replici actualizabile şi apar la momentul în care aplicarea actualizărilor în baza de date replicată nu este posibilă datorită unor actualizării contradictorii realizate de diverse locaţii. Cele mai comune conflicte (numite uneori coliziuni) sunt cele datorate unor erori în ordinea replicării. Iată câteva exemple:

* Replicarea unei operaţii DELETE nu găseşte linia care trebuie ştearsă (conflict de ştergere). Explicaţia poate fi faptul că în timpul propagării, o altă operaţie a actualizat linia.
* O operaţie UPDATE replicată nu găseşte linia care trebuie actualizată (conflict de modificare). Explicaţia este aceeaşi ca în cazul ştergerii.
* O operaţie INSERT replicată găseşte deja în replică o linie având aceeaşi cheie primară (conflict de unicitate). Explicaţia poate fi faptul că în timpul propagării, a intervenit o operaţie de inserare sau o operaţie de modificare asupra cheii primare a unei linii existente.

Multe sisteme permit aplicarea unor reguli prestabilite pentru rezolvarea automată a unor astfel de conflicte. Cele mai comune sunt regulile bazate pe mărci de timp (timestamps). Există şi metode de rezolvare a conflictelor bazate pe un sistem de priorităţi acordate locaţiilor şi/sau utilizatorilor. O altă variantă de rezolvare o constituie construirea unor proceduri stocate care să fie executate atunci când este detectat un conflict. Acestea au acces la toate informaţiile referitoare la actualizările conflictuale, prin urmare pot combina metode bazate pe mărci de timp, priorităţi sau alte reguli specifice aplicaţiei.

Informix, de exemplu, permite rezolvarea automată a conflictelor prin mărci de timp (ultima actualizare învinge), prin proceduri stocate (logica acesteia determină actualizarea care învinge) sau prin... ignorarea conflictului. Dacă pentru o anumită replică se alege regula mărcilor de timp, se poate adăuga ca regulă secundară lansarea unei proceduri stocate (pentru cazul în care actualizările în conflict au aceeaşi marcă de timp). Un detaliu: pentru ca regulile bazate pe mărci de timp să poată funcţiona corect, sistemele participante trebuie să aibă ceasurile sincronizate şi să ţină seama de eventualele diferenţe de fus orar.

Să considerăm un exemplu, în versiune Informix. Fie trei locaţii (A, B şi C) în configuraţie peer-to-peer şi o tabelă Pers în care a fost inserată o linie prin instrucţiunea:

INSERT INTO pers

VALUES (1325, 'Vasile', 'MS');

Regula de rezolvare a conflictelor este cea bazată pe mărci de timp. Considerăm ca la momentul T0 (să zicem, ora 12:00), linia există în toate replicile.

La momentul T1 (12:05), la locaţia B se execută instrucţiunea:

UPDATE pers SET jud = 'CJ'

WHERE marca = 1325;

La momentul T2 (12:10), la locaţia C se execută instrucţiunea:

DELETE FROM pers

WHERE marca = 1325

AND nume = 'Vasile'

AND jud = 'MS'

La momentul T3 (13:00), actualizările de la locaţia C sosesc la locaţia A şi sunt aplicate în tabela Pers. Linia cu cheia primară 1325 este ştearsă din tabelă Pers. Sistemul păstrează însă linia într-o "umbră" (shadow) a tabelei.

La momentul T4 (13:15), sosesc la locaţia A actualizările de la B. Se încearcă aplicarea modificării, care însă nu găseşte linia. Sistemul detectează astfel apariţia unui conflict.

Deoarece linia cu cheia primară căutată lipseşte din tabela Pers, se scanează "umbra". Este găsită linia în cauză şi se compară marca de timp aplicată de ultima actualizare (în cazul nostru T2, adică 12:10) cu marca de timp a noii actualizări (T1, adică 12:05). Este considerată învingătoare actualizarea cu marca de timp mai mare (în cazul nostru, ştergerea), deci modificarea nu se mai aplică şi linia rămâne ştearsă.

Faptul că actualizarea de la C au sosit mai repede decât cele de la B este întâmplător. Ce s-ar fi întâmplat dacă ordinea ar fi fost inversată? S-ar fi aplicat întâi modificarea (jud ar fi devenit CJ) după care ştergerea n-ar fi găsit linia (observaţi importanţa condiţiilor suplimentare pentru ştergere). S-ar fi căutat linia cu cheia primară 1325 şi s-ar fi comparat mărcile de timp. Ştergerea ar fi fost declarată învingătoare, deci s-ar fi anulat modificarea şi s-ar fi şters linia.

Este de remarcat faptul că în ambele cazuri s-ar fi ajuns la acelaşi rezultat, care ar fi fost apoi replicat către celelalte locaţii. Să observăm că dacă instrucţiunea UPDATE ar fi modificat cheia primară, putem presupune că ştergerea n-ar mai fi putut găsi linia dinainte de modificare. Nu este aşa, deoarece la evaluarea lui UPDATE, aceasta ar fi fost înlocuită cu o ştergere urmată de o înserare, deci linia ar fi fost găsită în tabela "umbră".

S-ar putea însă ca această rezolvare să nu fie ceea ce ne-am dorit. Poate am fi dorit ca prima actualizare să învingă. Unele sisteme permit stabilirea unei astfel de reguli, dar în Informix va trebui să scriem o procedură stocată în acest scop. (Dacă am fi ales varianta "ignore" am fi obţinut rezultate diferite în funcţie de momentul replicării.)

O ultimă precizare: Informix permite şi stabilirea aplicabilităţii regulii de rezolvare la nivel de linie sau la nivel de tranzacţie. În primul caz, este posibil ca dintre actualizările realizate în cadrul unei tranzacţii unele să fie aplicate şi altele nu. Dacă rezolvarea se face la nivel de tranzacţie, atunci fie întreaga tranzacţie este aplicată, fie este anulată (rollback). Această ultimă variantă este menită să păstreze integritatea referenţială.

**Prevenirea conflictelor**

Conflictele reprezintă partea cea mai sensibilă a sistemelor distribuite pe bază de replicare. O regulă de bun simţ spune că este preferabil ca acestea să fie prevenite decât să se mizeze pe rezolvarea lor automată. În scopul prevenirii conflictelor, varianta cea mai bună este stabilirea strictă a apartenenţei datelor şi evitarea pe cât posibil a configuraţiilor peer-to-peer. În exemplul precedent, datele despre personalul unei companii se pretează cel mai bine la partajare, astfel încât drepturile de actualizare să fie deţinute de fiecare filială doar pentru angajaţii proprii. În aceste condiţii, transferul unui angajat de la filiala MS la filiala Cj (operaţie care a fost simulată în exemplu) s-ar petrece puţin mai complicat (persoana ar ştearsă din baza de date la MS şi ar fi inserată din nou la Cj) dar de bună seamă mai sigur.

Tot o metodă de prevenire a conflictelor poate fi considerată tehnologia promovată de IBM, care propune înlocuirea configuraţiilor peer-to-peer cu configuraţii ierarhice cu un nod de referinţă. În această configuraţie, toate actualizările sunt replicate întâi spre nodul de referinţă (rădăcina ierarhiei), de unde sunt apoi replicate către celelalte noduri. Eventualele conflicte sunt rezolvate la nodul de referinţă, printr-o metodă bazată pe aşa-numitele "tranzacţii de compensare". Atunci când două tranzacţii încearcă să realizeze actualizări contradictorii, una dintre ele este aleasă ca "victimă". Aceasta va fi anulată la nivelul nodului de referinţă, după care sistemul va genera o "tranzacţie de compensare", care va fi propagată în scopul de a anula efectele tranzacţiei "victimă" la nivele inferioare ale ierarhiei. Este interesant de notat că această metodă funcţionează şi în cazul conflictelor mai complexe, implicând date din mai multe tabele.

Să considerăm, ca exemplu, o configuraţie peer-to-peer cu trei noduri (A şi B), în care avem definite tabelele F (furnizori) şi P (produse), asociate printr-o cheie străină (f\_id) în tabela P (orice produs trebuie să aparţină unui furnizor). Regula pentru ştergere în restricţia referenţială pentru cheia străină este ON DELETE RESTRICT (un furnizor nu poate fi şters decât dacă nu are nici un produs care să-i fie asociat).

Să considerăm că la locaţia A s-a făcut inserarea:

INSERT INTO F VALUES (12, ...);

Să mai presupunem că nu există nici un produs de la furnizorul 12 şi că această inserare a fost replicată cu succes, astfel încât la momentul T0 (ora 12:00) starea este consistentă.

La momentul T1 (12:05), la locaţia A se execută instrucţiunea:

DELETE FROM F WHERE id = 12;

La momentul T2 (12:07), la locaţia B se execută instrucţiunea:

INSERT INTO P

VALUES (1023, 12, 'Ciocan');

La momentul T3 (12:10), la locaţia A soseşte mesajul cu actualizarea produsă la momentul T2 la B. Inserarea nu se poate face deoarece nu există furnizorul referit (id = 12).

La momentul T4 (12:15), la locaţia B este replicată ştergerea efectuată la momentul T1 la locaţia A. Furnizorul cu codul 12 nu poate fi şters deoarece există un produs care-l referă.

Observăm că s-a ajuns la o stare inconsistentă: la nodul A nu mai există nici furnizorul 12 nici produsul 1023, în timp ce la nodul B ambele există.

În varianta propusă de IBM, vom considera că un nod (C) ca fiind nod de referinţă. Considerând momentele aceleaşi operaţii în nodurile A şi B, la aceleaşi momente (T1 şi T2), scenariul poate fi continuat astfel:

La momentul T3 (12:10), inserarea de la momentul T2 din nodul B este replicată în nodul C. Este aplicată cu succes, după care este trimisă spre replicare nodului A.

La momentul T4 (12:15), ştergerea de la momentul T1 din nodul A este replicată în nodul C. Deoarece furnizorul 12 are un produs care-l referă, nu poate fi şters. Se aplică o metodă de rezolvare a conflictului, de pildă last timpstamp wins. Conform acesteia, tranzacţia care a comis inserarea învinge, iar cea care a comis ştergerea este considerată "victimă". Prin urmare, ştergerea nu este replicată către B şi se generează o tranzacţie de compensare care este trimisă către A. Aceasta ar putea fi de forma:

BEGIN WORK;

DELETE FROM P

WHERE cod = 1023

AND f\_id = 12

AND den = 'Ciocan';

INSERT INTO F

VALUES (12, ...);

INSERT INTO P

VALUES (1023, 12, 'Ciocan');

COMMIT WORK;

Chiar dacă nu s-au evitat astfel conflictele, prin centralizare numărul lor s-a redus iar posibilităţile de rezolvare au sporit. Cu toate că nodurile sunt consistente, problema nu poate fi considerată complet rezolvată. Considerând că tranzacţia de compensare a fost comisă în nodul A la momentul T5 (să zicem 12:17), orice tranzacţie care a citit tabela F de la nodul A, a avut ocazia să ia o decizie greşită bazată pe faptul că furnizorul cu codul 12 a lipsit din tabelă.

Concluzia finală este că nici una dintre tehnicile de rezolvare a conflictelor nu este infailibilă şi că vor exista mereu situaţii în care se impun intervenţii manuale. Este un preţ care trebuie plătit în schimbul avantajelor pe care replicarea asincronă le oferă. Corolarul acestei concluzii este că pentru anumite probleme replicarea asincronă este aplicabilă şi pentru altele nu.

Nu am detaliat în această prezentare subiecte extrem de interesante cum ar replicarea actualizărilor în schema bazelor de date sau replicarea prin apeluri asincrone de proceduri la distanţă (asynchronous RPC). Nu am vorbit despre mecanismele de replicare la nivel de câmp şi de rezolvările conflictelor prin versiuni multiple din Lotus Notes.

Nu am vorbit despre replicarea procedurală practicată de Oracle şi nici despre modelele de propagare pull şi push utilizate de IBM şi de alţi producători. Am amintit doar de problemele speciale pe care le ridică replicarea pentru lucrătorii mobili.

O problematică largă este replicarea în medii eterogene, cuprinzând baze de date de la producători diferiţi. În fine, nu am amintit despre produsele independente de la terţi producători, cum ar fi PeerDirect sau ThinkNet. Şi ar mai fi multe altele.

**Tipuri de Date**

Toate valorile datelor ale unei coloane trebuie sa fie de acelasi tip de date. (Singura exceptie specifica valorile tipului de data SQL\_VARIANT). Transact-SQL utilizeaza diferite tipuri de date, care pot fi clasificate dupa cum urmeaza:

* tipuri de date numerice
* tipuri de date caracter
* tipuri de date temporale (data si/sau timp)
* diverse tipuri de date

Urmatoarele sectiuni descriu toate aceste categorii.

**Tipuri de Date Numerice**

Tipurile de date numerice sunt folosite pentru a reprezenta numere.Urmatorul tabel arata lista de toate tipurile de date numerice:

INTEGER - Reprezinta valorile intregi ce pot fi stocate pe 4 biti.

SMALLINT - Reprezinta valorile intregi ce pot fi stocate pe 2 biti.

TINYINT - Reprezinta valorile intregi ce pot fi stocate pe 1 bit.

DECIMAL(p,s) - Descriu valori de punct-fix. „p” reprezita precizia, „s” reprezinta scara.

NUMERIC - Sinonim pentru Decimal.

REAL - Utilizat pentru valori de punct-variabil.

FLOAT(p) - Reprezinta valori de punct-variabil, exemplu Real. „p” reprezinta precizia.

MONEY - Utilizat pentru a reprezenta valori monetare pe 8 biti.

SMALLMONEY - Corespunde pentru valori monetare reprezentate pe 4 biti.

**Tipuri de Date Caracter**

Există doua forme generale de tipuri de date caracter. Ele pot fi siruri de caractere de un singur octet sau siruri de caractere Unicode. (Unicode foloseste cativa biti pentru a specifica un caracter). Mai mult, sirurile de caractere pot avea o lungime fixa sau variabila. Urmatoarele tipuri de date caracter sunt utilizate:

CHAR (n) - Reprezinta un sir de caractere de lungime fixa, n este numarul de caractere din interiorul sirului de caractere.

VARCHAR (n) - Descrie un sir de caractere de lungime variabila.

NCHAR (n) - Stocheaza siruri de caractere de lungime fixa de caractere Unicode.

NVARCHAR (n) - Stocheaza siruri de caractere de lungime variabila de caractere Unicode.

**NOTA**

Tipul de date VARCHAR este identic cu tipul de date CHAR, cu exceptia ca: în cazul în

care continutul unui sir de caractere CHAR (n) este mai scurta decat n, restul de sir este

completat cu spaţii. (O valoare a datelor VARCHAR tip este intotdeauna stocat in

lungime de real).

**Tipuri de Date Temporale**

Transact-SQL suporta urmatoarele tipuri de date temporale:

- DATETIME

- SMALLDATETIME

- DATE

- TIME

- DATETIME2

-DATETIMEOFFSET

Tipurile de date DATETIME si SMALLDATETIME specifica data si ora, fiecare valoare fiind stocata ca un intreg, cu valoare de 4 octeti sau 2 octeti, respectiv. Valorile DATETIME si SMALLDATETIME sunt stocate intern separat ca doua valori numerice.Valoarea de data DATETIME este stocata in intervalul 01/01/1753 -12/31/9999. Valoarea analogica pentru SMALLDATETIME este stocata în gama de  
01/01/1900 la 06/06/2079. Componenta de timp este stocata in al doilea 4-octet  
(sau 2-octet pentru SMALLDATETIME), domeniu ca numar de trei sutimi de secunda  
(DATETIME) sau minute (SMALLDATETIME), care au trecut de miezul noptii.

Utilizarea de DATETIME si SMALLDATETIME este destul de incomoda, daca  
doriţi să stocaţi numai o parte sau o parte de timp. Din acest motiv, SQL Server 2008  
introduce noi tipuri de date DATE şi TIME, care păstrează doar data şi ora a unei DATETIME. Tipul de date DATA este stocat în 3 octeţi si are gama de la 01/01/0001 la 12/31/9999. Tipul de date TIME este stocat în 3-5 octeţi şi are o precizie de 100 de nanosecunde (ns).

Tipul de date DATETIME2 este, de asemenea, un tip nou de date care stochează de înaltă precizie data şi timpul.Tipul de date poate fi definit pentru lungimi variabile în funcţie de  
cerinţă. (Capacitatea de stocare este de 6-8 octeţi).Precizie de o parte de timp este de 100 ns. Acest tip de date nu este conştient de orei de vară.

Toate tipurile de date descrise până în prezent, nu au suport pentru fusul orar. Noul tip de date numit DATETIMEOFFSET are parte de fus orar. Din acest motiv, el este stocat în 6-8 octeţi. (Toate celelalte proprietăţi ale acestui tip de date sunt similare pentru proprietăţile corespunzătoare ale DATETIME2.) Valoare de dată în Transact-SQL este implicit specificata intr-un sir de forma "mmm dd aaaa" (de exemplu, "10 ianuarie 1993). (Reţineţi că ordinea relativă de luni, zi şi an pot fi controlate de declaraţia SET DateFormat. În plus, sistemul recunoaşte numeric valorile de lună cu delimitatori de / sau -). În mod similar, valoarea de timp este specificată în formatul "HH: mm" şi motorul de baze de date foloseste 24 de ore (23:24, de exemplu).

**Tipurile de Date BIT si BINARY**

BINARY şi VARBINARY sunt cele două tipuri de date binare. Ele descriu obiecte de date fiind reprezentate în formatul intern al sistemului. Ele sunt folosite pentru a stoca biţi  
siruri de caractere. Din acest motiv, valorile sunt introduse folosind numere hexazecimale. Valorile de tip bit de date sunt stocate într-un singur bit. Prin urmare, până la 8 biţi coloanele sunt stocate într-un octet. Tabelul următor rezumă proprietăţile acestor  
tipuri de date:

BINARY (n) - Specifică un şir de biţi de lungime fixă, cu exact n octeţi.

VARBINARY (n) - Specifică un şir de biţi de lungime variabilă cu până la n octeţi.

BIT - Folosit pentru specificarea tipului de date boolean cu trei valori posibile: FALSE, TRUE, şi NULL.

**Tipuri de Date Large Object**

Obiecte de mari dimensiuni (LOBs) sunt obiecte de date, cu lungimea maximă de 2GB. Aceste obiecte sunt, în general, folosite pentru a stoca tipuri de date mari de text şi să încarce modulele si fisiere audio / video. Transact-SQL suportă două moduri diferite de a specifica şi de a accesa LOBs:

* Utilizeaza tipurile de date VARCHAR (MAX), NVARCHAR (MAX), şi VARBINARY (MAX).
* Utilizeaza asa-numitul text/imagine tip de data.

Urmatoarele paragrafe descriu două forme de LOBs.

**Specificatorul MAX**

Incepand cu SQL Server 2005, puteţi utiliza acelaşi mod de programare pentru a accesa valorile de tipuri standard de date şi LOBs. Cu alte cuvinte, aveţi posibilitatea să utilizaţi  
funcţii convenabile de sistem şi operatorii de coarde pentru a lucra cu LOBs. Motorul de baza de date utilizează specificatorul MAX cu tipurile de date Varchar, Nvarchar, şi pentru a defini VARBINARY de lungime variabilă coloane. Când utilizaţi MAX implicit (în loc de o valoare explicită), sistemul analizează lungimea unui anumit şir de caractere şi decide dacă pentru a stoca şirul ca o valoare convenabila sau ca un lob.Specificatorul MAX indică faptul că dimensiunea de valori de coloane se poate ajunge la dimensiunea maximă a sistemului actual. (Într-o versiune viitoare a SQL Server, este posibil ca MAX sa aiba o valoare maximă mai mare de siruri de caractere.)

Deşi sistemul de baze de date decide modul în care un LOB va fi stocat, puteţi suprascrie această specificaţie implicit prin procedura sistemului de sp\_tableoption cu optiunea LARGE\_VALUE\_TYPES\_OUT\_OF\_ROW. Dacă valoarea opţiunii este setata la 1, datele din coloane declarate folosind specificatorul MAX vor fi depozitate separat de toate celelalte date. Dacă această opţiune este setată la 0, motorul bazei de date stochează toate valorile pentru dimensiunea rândului mai mic decat 8060 octeti ca date regulate de rând. În SQL Server 2008, aveţi posibilitatea să aplicaţi atributul FILESTREAM nou la un VARBINARY (MAX), coloana de a stoca date binare mari, direct într-un fişier NTFS  
sistem.Principalul avantaj al acestui atribut este că mărimea corespunzătoare unui LOB  
este limitată doar de dimensiunea volumului sistemul de fişiere.

**Datele de tip TEXT, NTEXT şi IMAGE**

Tipurile de date tipuri de TEXT, NTEXT şi IMAGE constituie aşa-numitele text / imagine tipuri de date. Obiectele de date de tip imagine pot conţine orice tip de date (module de sarcină, audio / video), în timp ce obiectele de date de tip TEXT si NTEXT pot conţine orice alt tip de date text (care este, datele imprimabile).  
Tipurile de text / imagine de date sunt stocate în mod implicit separat de toate celelalte valori ale bazei de date folosind o structură de arbore B, care indică fragmentele de date. (A, B-arbore este o structură de date neted, în care toate nodurile de jos au acelaşi număr de niveluri de la rădăcină de copac) pentru coloanele unui text / imagine de tip de date, baza de date.

Motorul stochează un pointer de 16 octeti în rând de date care specifică în cazul în care datele pot fi găsite. În cazul în care cantitatea de text / imagine de date este mai mică de 32KB, indicatorul de puncte de la nodul radacina din structura de B-arbore, care este de 84 bytes. Punctele de nodul rădăcină la blocuri fizice ale datelor. În cazul în care cantitatea de date este mai mare de 32KB, motorul bazei de date construieşte noduri intermediare între blocuri de date şi nodul rădăcină. Pentru fiecare tabel care conţine mai mult de o coloană cu astfel de date, toate valorile de coloanele sunt stocate împreună. Cu toate acestea, o pagină fizică poate deţine doar text / imagine de tipuri de date dintr-un singur tabel.

Deşi tipurile text / imagine de date sunt stocate separat de toate celelalte date, aveţi posibilitatea să modificaţi acest lucru, folosind procedura sistemului de sp\_tableoption cu opţiunea TEXT\_IN\_ROW. Folosind această opţiune, puteţi specifica numărul maxim de octeti, care sunt stocati împreună cu datele regulate. Tipurile text / imagine de date discutate pana acum au mai multe limitari. Nu puteţi le utiliza ca variabile locale (în proceduri stocate sau în grupuri de Transact-SQL declaraţii). De asemenea, ele nu pot fi o parte a unui indice sau nu pot fi utilizate în următoarele clauze de declaraţia SELECT: WHERE, ORDER BY, şi GROUP BY. Cea mai mare problema cu privire la toate tipurile text / imagine de date este că va trebui să utilizaţi operatori speciali  
(READTEXT, WRITETEXT, şi UPDATETEXT) pentru a lucra cu astfel de date.

**NOTA**

Tipurile de date text / imagine sunt marcate ca fiind o caracteristică depăşita şi vor fi eliminate într-o versiune viitoare a motorului bazei de date. Utilizaţi VARCHAR (MAX), NVARCHAR (MAX) şi VARBINARY (MAX) în loc.

**Tipuri de Date UNIQUEIDENTIFIER**

Aşa cum sugerează şi numele, o valoare de tipul de date UNIQUEIDENTIFIER este un număr unic de identificare stocat ca un şir binar de 16 biţi. Acest tip de date este strâns legat de identificatorul unic global (GUID), care garantează unicitatea la nivel global. Prin urmare, folosind acest tip de date, puteţi identifica in mod unic date şi obiecte în sisteme distribuite.

Iniţializarea unei coloane sau unei variabile de tip UNIQUEIDENTIFIER poate fi furnizata folosind funcţiile NEWID şi NEWSEQUENTIALID, precum şi cu un sir constant scris într-o formă specială folosind cifre hexazecimale şi cratime. (Funcţiile NEWID şi NEWSEQUENTIALID sunt descrise în secţiunea "Functii de Sistem"mai târziu în acest capitol.)  
O coloană de tipul de date UNIQUEIDENTIFIER poate fi consultata utilizând cuvantul cheie ROWGUIDCOL într-o interogare pentru a specifica faptul că coloana conţine valori de identitate. (Acest cuvânt cheie nu generează nici o valoare.) Un tabel poate avea mai multe coloane de tipul UNIQUEIDENTIFIER, dar numai una dintre ele poate avea drept cuvant ROWGUIDCOL.

**Tipul de Date SQL\_VARIANT**

Tipul de date SQL\_VARIANT poate fi folosit pentru a stoca valori de diferite tipuri de la date de acelaşi tip, cum ar fi valori numerice, siruri de caractere, si valori data. (Singurele tipuri de valori care nu pot fi stocate sunt valorile timestamp). Fiecare valoare a unei coloane SQL\_VARIANT are două părţi: valoarea de date şi informaţii care descriu acea valoarea. (Aceste informaţii conţin toate proprietăţile valorii tipului de date, cum ar fi lungimea, scara sau precizia).  
Transact-SQL suportă funcţia SQL\_VARIANT\_PROPERTY, care afişează informaţii ataşate pentru fiecare valoare a unei coloane SQL\_VARIANT. Pentru utilizarea tipului de date SQL\_VARIANT, vedeti exemplul 5.5 în Capitolul 5.

**NOTA**

Declarati o coloana a unui tabel utilizând SQL\_VARIANT numai dacă este cu adevărat necesar. O coloană ar trebui să aiba acest tip de date în cazul în care valorile sale pot fi de diferite tipuri sau dacă nu este posibil pentru a determina tipul de coloană în timpul procesului de proiectare a bazei de date.

**Tipul de Date HIERARCHYID**

Tipul de date HIERARCHYID este utilizat pentru a stoca o întreaga ierarhie. Acesta este implementat ca un Common Language Runtime (CLR) definit de utilizator care cuprinde mai multe funcţii de sistem pentru crearea şi operarea pe noduri de ierarhie. Următoarele  
funcţii, printre altele, fac parte din metodele de acest tip de date: GetAncestor (),  
GetDescendant (), Read (), şi Write (). (Descriere detaliată a acestui tip de date este  
în afara domeniului de aplicare al acestei cărţi.)

**Tipul de Date TIMESTAMP**

Tipul de date TIMESTAMP specifică o coloană fiind definit ca un VARBINARY (8) sau  
BINARY (8), în funcţie de „nullability” coloanei. Sistemul menţine o valoare curenta (nu o dată sau de timp), pentru fiecare bază de date, pe care le incrementeaza ori de câte ori orice rand cu o coloana TIMESTAMP este introdus sau actualizat si seteaza TIMESTAMP coloanei cu valoarea respectiva. Astfel, coloanele TIMESTAMP pot fi folosite pentru a determina un timp relativ atunci când rândurile au fost modificate ultima data. (ROWVERSION este un sinonim pentru TIMESTAMP)

**NOTA**

Valoarea stocata într-o coloană TIMESTAMP nu este importanta în sine. Această coloană este de obicei folosita pentru a detecta dacă unanumit rând a fost schimbat de la ultima dată când a fost accesat.

**DECIMAL (Zecimal) cu formatul de stocare VARDECIMAL**

Tipul de date DECIMAL este de obicei stocat pe disc ca lungime fixă ​​de date. Odata cu  
SQL Server 2005 SP2, acest tip de date poate fi stocat ca o coloană de lungime variabilă folosind formatul de stocare nou, denumit VARDECIMAL. Folosind VARDECIMAL, putem reduce în mod semnificativ spaţiul de stocare pentru o coloană DECIMAL, în care valorile au diferenţă semnificativă în lungimile lor.

**NOTA**

VARDECIMAL este un format de stocare, nu si un tip de date.

Formatul de stocare VARDECIMAL este util atunci când trebuie să specificaţi cea mai mare valoare posibilă pentru un domeniu în care valorile stocate, de obicei, sunt mult mai mici. Tabelul 4-1 arată acest lucru.

**NOTA**

Formatul de stocare VARDECIMAL funcţionează în acelaşi mod pentru tipul de date DECIMAL ca si cum tipul de date VARCHAR lucreaza pentru datele alfanumerice.

**Functiile TRANSACT-SQL**

Functiile TRANSACT-SQL pot fi functii agregate sau functii scalare. Urmatoarele sectiuni descriu exact aceste tipuri de functii.

**Functii Agregate**

Funcţiile agregate se aplică la un grup de valori de date dintr-o coloană. Functiile agregate returneaza întotdeauna o valoare unică. Transact-SQL suporta mai multe grupuri de funcţii agregate:

* functii agregate convenabile;
* functii agregate statistice;
* functii agregate definite de utilizator;
* functii agregate analitice;

Agregatele statistice şi analitice sunt discutate în Capitolul 24. Cele definite de utilizator  
sunt dincolo de sfera de aplicare a acestei cărţi. Functiile agregat convenabile sunt descrise in următoarele:

**AVG** - Calculează media aritmetică a valorilor de date conţinute într-o coloană. Coloana trebuie să conţină valori numerice.

**MAX si MIN** - Calculeaza valoarea maximă şi minimă de date din coloana

respectiva. Coloana poate conţine numere, şiruri, data/ora.

**SUM -** Calculeaza totalul valorilor din coloana. Coloana trebuie sa contina valori numerice.

**COUNT -** Calculează numărul de valori de date dintr-o coloană.

**COUNT\_BIG** - Analog COUNT, singura diferenta este returneaza o valoare de tipul BIGINT.

Utilizarea de funcţii agregate convenabile cu declaraţia SELECT sunt descrise în detaliu în capitolul 6.

**Functii Scalare**

În plus faţă de funcţiile agregate, Transact-SQL oferă mai multe funcţii scalare care  
sunt utilizate în construcţia de expresii scalare. (O funcţie scalară opereaza pe o singura  
valoare sau o listă de valori, spre deosebire de funcţiile agregate, care operează asupra datelor din mai multe rânduri), funcţiile de scalare pot fi clasificate după cum urmează:

* functii numerice;
* functii data;
* functii string;
* functiile sistemului;
* functii metadata;

Următoarele secţiuni descriu aceste tipuri de funcţii.

**Functii Numerice**

Funcţiile numerice din cadrul Transact-SQL sunt funcţii matematice pentru modificarea  
valorilor numerice. Următoarele funcţii numerice sunt disponibile:

ABS(n) - Returneaza valoarea absoluta a expresiei numerice n.

ACOS(n) - Calculeaza arccos de n. Valoarea rezultata este de tip FLOAT.

ASIN(n) - Calculeaza arcsin de n. Valoarea rezultata este de tip FLOAT.

ATAN(n) - Calculeaza arctg de n. Valoarea rezultata este de tip FLOAT.

ATN2(n,m) - Calculeaza arctg de n/m. Rezultatul este de tip FLOAT.

CEILING(n) - Returneaza valoarea intreaga cea mai mica egala sau mai mare fata

de paramentru n.

COS(n) - Calculeaza cosinus de n. Rezultatul este de tip FLOAT.

COT(n) - Calculeaza cotangenta de n. Rezultatul este de tip FLOAT.

DEGREES(n) - Converteste radiani in grade.

EXP(n) - Calculeaza valoarea numarului e la puterea n.

FLOOR(n) - Calculeaza valoarea intreaga cea mai mare egala sau mai mica fata

de parametru n.

LOG(n) - Calculeaza logaritm natural de n.

LOG10(n) - Calculeaza logaritm in baza 10 de n.

PI() - Returneaza valoarea numarului PI.

POWER(x,y) - Calculeaza valoarea lui x la puterea y.

RADIANS(n) - Converteste gradele in radiani.

RAND - Returneaza un numar de tip FLOAT, aleatoriu din intervalul 0,1

ROUND(n,p,t) - Rotunjeşte valoarea numărului n cu ajutorul preciziei p. Utilizaţi valori pozitive ale lui p pentru a rotunji la partea dreapta a punctului zecimal şi de a folosi valori negative, pentru a completa pe partea stângă. Un opţional parametru t determină n pentru a fi trunchiat.

ROWCOUNT\_BIG - Returnează numărul de rânduri care au fost afectate de ultimul Transact-SQL executat de sistem.Valoarea întoarsă de această funcţie are tipul de date BIGINT.

SIGN(n) - Returneaza semnul parametrului n.

SIN(n) - Calculeaza sinus de n. Rezultatul este de tip FLOAT.

SQRT(n) - Calculeaza radical din n.

SQUARE(n) - Returneaza patratul parametrului n.

TAN(n) - Calculeaza tangenta lui n.

**Functii DATA**

Funcţii Data calculeaza data respectivă sau timpul unei expresii care returneaza valoarea dintr-un interval de timp. Transact-SQL acceptă următoarele funcţii data:

GETDATE() - Returneaza data si ora sistemului actual.

DATEPART(item,date) - Returneaza partea specificata „item” dintr-un tip de data „date” ca un intreg.

DATENAME(item,date) - Returneaza partea specificata „item” dintr-un tip de data „date” ca un string.

DATEDIFF(item,dat1,dat2) - Calculeaza diferenta dintre dat1 si dat2 si returneaza rezultatul ca un intreg.

DATEADD(i,n,d) - Aduna numarul n la unitatile specificate de valoarea lui i pentru d.

**Functii STRING**

Funcţiile STRING sunt utilizate pentru a manipula valorile de date într-o coloană, de obicei, de tipul de date caracter. Transact-SQL acceptă următoarele funcţii string:

ASCII(character) - Converteste caracterul specificat intr-un intreg asociat ASCII.

CHAR(integer) - Converteste codul ASCII in echivalentul caracter.

CHARINDEX(z1,z2) - Returneaza pozitia de start unde stringul z1 apare in z2 pentru prima data.

DIFFERENCE(z1,z2) - Returneaza un intreg de la 0 la 4 care reprezinta diferenta de valori SOUNDEX dintre z1 si z2.

LEFT(z,length) - Returneaza primele caractere de la valoarea lui length din sirul z.

LEN(z) - Returneaza numarul de caractere pentru z.

LOWER(z1) - Converteste toate caracterele mari in caractere mici.

LTRIM(z) - Sterge casutele goale din sirul z.

NCHAR(i) - Returneaza caracterul UNICODE pentru parametru.

QUOTENAME(char\_string) - Returnează un şir Unicode cu delimitatori adăugati pentru a face un şir de intrare valabil delimitat identificator.

REPLICATE(z,i) - Repeta stringul z de i ori.

REVERSE(z) - Afiseaza stringul z invers.

RIGHT(z,length) - Returneaza caracterele de lungime length din stringul z.

RTRIM(z) - Indeparteaza spatiile din sirul z.

SOUNDEX(a) - Returnează un cod de patru caractere SOUNDEX pentru a determina similitudinea dintre două şiruri.

SPACE(length) - Returnează un şir cu spaţii de lungime specificată.

STR(f,len,d) - Converteşte specificat expresia f într-un şir.

STUFF(z1,a,length,z2) - Înlocuieşte sirul partial Z1 cu sirul partial Z2 pornind de la poziţia A, înlocuind caractere lungime de Z1.

SUBSTRING(z,a,length) - Creează un şir parţial de la Z începând de la poziţia A, cu o lungime data.

UNICODE - Returneaza valoarea intreaga pentru primele caractere din stringul de input.

UPPER(z) - Converteste toate caracterele mici in caractere mari.

**Functii SISTEM**

Funcţiile sistemului Transact-SQL oferă informaţii detaliate despre baza de date. Cele mai multe funcţii de sistem utilizeaza un identificator intern numeric (ID), care este  
alocat pentru fiecare obiect din bază de date a sistemului la crearea acesteia. Folosind acest identificator, sistemul poate identifica fiecare obiect unic din baza de date. Funcţiile sistemului oferă informaţii despre sistemul de baze de date.Următorul tabel descrie sistemul cu mai multe funcţii. (Pentru lista completă a tuturor funcţiilor sistemului, vă rugăm să consultaţi cărţi online.)

CAST(a AS type [(length)] - Converteşte o expresie a în tipul specificat de date (dacă este posibil). a, ar putea fi orice exprimare valabila.

COALESCE(a1,a2,…) - Intoarce pentru o anumită listă de expresii a1, a2, ... valoarea care nu este NULL.

COL\_LENGTH(obj,col) - Returnează lungimea coloanei aparţinând obiectului de bază de date (tabel sau vedere)

CONVERT(type[(length)],a) - Echivalent cu CAST, dar argumentele sunt specificate diferit. CONVERT poate fi utilizat cu orice tip de date.

CURRENT\_TIMESTAMP - Returnează data şi ora curenta.

CURRENT\_USER - Returnează numele utilizatorului curent.

DATALENGTH(z) - Calculează lungimea (în octeţi) a rezultatului Z.

GETANSINULL('dbname') - Returnează 1 dacă utilizarea de valori NULL din baza de date dbname este în conformitate cu standardul ANSI SQL.

ISNULL(expr, value) - Returnează valoarea de expresie în cazul în care valoarea nu este nul, în caz contrar, se returnează valoarea sa.

ISNUMERIC(expression) - Determină dacă o expresie este de un tip valid numeric.

NEWID() - Creează un număr de identificare unic, care constă intr-un şir binar de 16 octeti destinati pentru a stoca valorile de tip de date UNIQUEIDENTIFIER.

NEWSEQUENTIALID() - Creează un GUID care este mai mare decât orice GUID generate anterior de această funcţie pe un calculator specificat. (Această funcţie poate fi folosită doar ca valoare implicită pentru o coloană.)

NULLIF(expr1,expr2) - Returnează valoarea NULL în cazul în care expresiile expr1 si expr2 sunt egale.

SERVERPROPERTY(propertyname) - Returnează informaţii despre proprietatea serverul de baze de date.

SYSTEM\_USER - Returnează ID-ul de conectare a utilizatorului curent.

USER\_ID([user\_name]) - Returnează identificatorul utilizatorului USER\_NAME. Dacă nu este specificat numele, identificatorul de utilizatorul curent este preluat.

USER\_NAME([id]) - Returnează numele de utilizator cu ID-ul de identificare. Dacă nu este specificat numele, numele de utilizatorul curent este preluat.

Toate funcţiile şir pot fi imbricate, în orice ordine, de exemplu, REVERS(CURRENT\_USER).

**Functii METADA**

În general, funcţiile de metadate restituie informaţii privind baza de date specificată  
şi obiectele din baze de date.Următorul tabel descrie mai multe funcţii metadate. (Pentru  
lista completă a tuturor funcţiilor de metadate, vă rugăm să consultaţi cărţi online.)

COL\_NAME(tab\_id, col\_id) - Întoarce numele unei coloane aparţinând tabelului cu ID-ul şi tab\_id ID-ul coloanei col\_id.

COLUMNPROPERTY(id, col, property) - Returnează informaţii despre coloana specificată.

DATABASEPROPERTY(database, property) - Returnează valoarea proprietăţii numita de baza de date pentru baza de date specificată şi proprietate.

DB\_ID([db\_name]) - Returnează identificatorul DB\_Name al bazei de date. Dacă nu este specificat numele, identificatorul bazei de date curentă este returnat.

DB\_NAME([db\_id]) - Returnează numele bazei de date cu db\_id de identificare. Dacă identificatorul nu estespecificat, numele bazei de date curente este afişat.

INDEX\_COL(table, i, no) - Returnează numele coloanei indexate în tabelul table, definit de indicele i şi poziţia coloanei în index.

INDEXPROPERTY(obj\_id, index\_name,property) - Returnează indicele de nume sau valoarea statistica a proprietatii unui tabel specificat in numărul de identificare, numele sau indicele statisticii şi numele proprietăţii.

OBJECT\_NAME(obj\_id) - Returnează numele obiectului bazei de date print indentificatorul obj\_id.

OBJECT\_ID(obj\_name) - Returnează identificatorul obiectului bazei de date obj\_name.

OBJECTPROPERTY(obj\_id,property) - Returnează informaţii despre obiectele din baza de date curentă.

**OPERATORII SCALARI**

Operatorii scalari sunt utilizati pentru operaţiuni cu valori scalare. Transact-SQL suporta  
operatorii numerice şi booleani, precum şi cei de concatenare. Există operatori unari şi aritmetici binari. Operatorii unari sunt + şi - . Operatorii aritmetici binari sunt +, -, \*, / şi %. Operatorii booleeni au două coduri diferite, în funcţie de faptul ca dacă acestia sunt  
aplicati la şiruri de biţi sau la alte tipuri de date. Operatorii nu, si, si sau sunt  
aplicati la toate tipurile de date (cu excepţia BIT). Operatorii la nivel de bit pentru manipularea sirurilor de biti sunt listati aici şi Exemplul 4.8 arată modul în care acestia sunt utilizate:

~ Complement (de exemplu, NU)  
& Conjunctia si de şiruri de biţi (de exemplu, SI)  
| Disjuncţie de şiruri de biţi (de exemplu, OR)  
^ Disjuctia Exclusiv (de exemplu, XOR sau exclusive sau)

**Variabile Globale**

Variabilele globale sunt variabile speciale de sistem care pot fi utilizate în cazul în care acestea au fost scalari constanti. Transact-SQL suporta multe variabile globale, care trebuie să fie precedate de prefixul @ @.Următorul tabel descrie mai multe variabile globale.

@@CONNECTIONS - Returnează numărul de încercări de conectare de la pornirea sistemului.

@@CPU\_BUSY - Returnează timpul total de CPU (în milisecunde) utilizat de la pornirea sistemului.

@@ERROR - Returnează informaţii despre valoarea de întoarcere de la ultima executata de Transact-SQL.

@@IDENTITY - Returnează ultima valoare introdusa pentru coloana cu proprietatea de identitate.

@@LANGID - Returnează identificatorul de limbă care este utilizat în prezentul sistemu de baze de date.

@@LANGUAGE - Returnează numele limbii care este utilizata în prezentul sistem de baze de date.

@@MAX\_CONNECTIONS - Returnează numărul maxim de conexiuni reale ale sistemului.

@@PROCID - Returnează identificatorul de procedură stocat în curs de execuţie.

@@ROWCOUNT - Returnează numărul de rânduri care au fost afectate de ultima clauza Transact-SQL executata de către sistem.

@@SERVERNAME - Preia informaţiile referitoare la serverul de baze de date locale. Această informaţie conţine, printre altele, numele serverului şi numele de instanţă.

@@SPID - Returnează identificatorul de procesul server.

@@VERSION - Returnează versiunea curentă de software a sistemului de baze de date.

**Valori NULL**

O valoare NULL este o valoare specială care poate fi atribuita la o coloană. Această valoare este utilizata în mod normal atunci când informaţiile dintr-o coloană este necunoscuta sau nu se aplică. De exemplu, în cazul unui număr necunoscut de telefon pentru un angajat al companiei, se recomandă ca valoarea NULL sa se atribuie coloanei home\_telephone.

Orice expresie aritmetica rezulta in NULL dacă oricare operand al expresiei este el însuşi  
nul. Prin urmare, în expresii aritmetice unare (în cazul în care A este o expresie cu o  
valoarea NULL), ambele + A şi-A se intoarce NULL. În expresii binare, în cazul în care unul (sau ambele) din operanzii A sau B, are o valoare NULL, A + B, A - B, A \* B, A / B, A şi B şi, de asemenea,% duce la o valoare NULL. (Operanzii A şi B trebuie să fie expresii numerice).

Dacă o expresie conţine o operaţie relaţionala şi unul (sau ambele) din operanzi are  
(au) valoarea NULL, rezultatul acestei operaţii va fi NULL. Prin urmare, fiecare dintre  
expresiile A = B, A <> B, A <B, si A> B, de asemenea returnează NULL.  
În Boolean AND, OR, şi NOT, comportamentul valorilor NULL este specificat  
cu următoarele tabele de adevăr, T pentru adevărat, pentru necunoscut (NULL), şi F  
pentru fals. În aceste tabele, urmaţi rândul şi coloana reprezentata de valorile  
expresiilor booleene pe care operatorul lucrează şi valoarea unde se intersecteaza  
reprezentand valoarea rezultată.

AND T U F OR T U F NOT

T T U F T T T T T F

U U U F U T U U U U

F F F F F T U F F T

Orice valoare NULL în argumentul de funcţii agregate AVG, SUM, MAX, MIN,  
şi COUNT este eliminat înainte ca funcţia respectivă sa fie calculata (cu excepţia  
Funcţia COUNT (\*)). Dacă o coloană conţine doar valori nule, funcţia returnează  
NULL. Functia agregata COUNT(\*) se ocupă de toate valorile NULL la fel şi de valorile  
non NULL. În cazul în care coloana conţine doar valori nule, rezultatul funcţiei  
COUNT (column\_name distincte) este 0.

O valoare NULL trebuie să fie diferita de toate celelalte valori. Pentru tipurile de date numerice, există o distincţie între zero şi valoarea NULL.Acelaşi lucru este valabil şi pentru un şir de caractere gol şi NULL pentru tipurile de date.  
O coloană a unui tabel permite valori NULL dacă definiţia conţine în mod explicit NULL. Pe de altă parte, valorile NULL nu sunt permise în cazul în care definiţia unei coloane conţine în mod explicit NOT NULL. Dacă utilizatorul nu specifică NULL sau NOT NULL pentru o coloană cu un tip de date (cu excepţia TIMESTAMP), următoarele valori sunt atribuite:

* NULL dacă opţiunea ANSI\_NULL\_DFLT\_ON pentru declaratia SET este setat la ON
* NOT NULL dacă opţiunea ANSI\_NULL\_DFLT\_OFF pentru declaraţia SET  
  este setat la ON

În cazul în care declaraţia SET nu este activata, o coloană va conţine valoarea NOT NULL în mod implicit. (Coloanele de tip de date TIMESTAMP poate fi declarat ca NOT  
NULL). Există, de asemenea, o altă opţiune pentru declaraţia SET: CONCAT\_NULL\_YIELDS\_NULL. Această opţiune influenţează funcţionarea concatenarii, cu o valoare NULL, astfel încât orice se concateaneaza cu NULL va afisa din nou NULL. De exemplu: "San Francisco" + NULL = NULL.

**CONCLUZIE**

Caracteristicile de bază ale Transact-SQL sunt compuse din tipuri de date, predicate, şi funcţii. Tipurile de date în conformitate cu tipurile de date din standardul ANSI SQL92. Transact-SQL suportă o varietate de funcţii de sistem utile. Următorul capitol vă introduce declaratii Transact-SQL în legătură cu SQL Data Definition Language. Această parte a Transact-SQL cuprinde toate situaţiile necesare pentru crearea, modificarea, eliminarea şi obiectele bazei de date.