Curs 4
Fragmentarea datelor, replicare și consistență în sisteme de baze de date distribuite și mobile (partea I)

4.1 Tehnici de fragmentare

Fragmentarea datelor (denumită și *partiționare*) este procesul de divizare a unei baze de date sau a unui tabel în părți mai mici, numite **fragmente**, care pot fi stocate pe noduri diferite ale sistemului distribuit. Scopul fragmentării este de a localiza datele aproape de punctele de utilizare, de a reduce traficul în rețea și de a permite scalarea orizontală (adăugarea de noduri pentru a mări capacitatea sistemului). Un **SGBD distribuit** (SGBDD) ar trebui să mascheze fragmentarea față de utilizator – adică să ofere *transparență la fragmentare*, astfel încât interogările să poată fi scrise ca și cum datele ar fi într-o singură bază de date, sistemul ocupându-se de reunirea fragmentelor după nevoie[1].

Tipuri de fragmentare a datelor

Există trei metode principale de fragmentare a datelor într-un sistem distribuit[2]:

1) **Fragmentare orizontală** – împarte relațiile (tabelele) în fragmente conținând subseturi de **rânduri** (tupluri). Fiecare fragment orizontal conține un subset de înregistrări, de obicei definite printr-o condiție de selecție. De exemplu, o tabelă *Angajați* poate fi împărțită în două fragmente: unul cu angajații din departamentul *Vânzări* și altul cu angajații din *IT*. Formal, fragmentarea orizontală corespunde aplicării unei operații de selecție (σ) pe tabelă, de forma Fragment1 = σ_condiție(Tabel)[3][4]. Fragmentele orizontale păstrează toate atributele originale, dar conțin rânduri diferite și mutual exclusive (un rând apare într-un singur fragment). Dacă reunim (UNION) toate fragmentele orizontale, reconstruim tabela inițială: Tabel = Frag1 ∪ Frag2 ∪ ... ∪ FragN[5][6].
2) **Fragmentare verticală** – împarte o relație după **coloane** (atribute). Fiecare fragment vertical conține un subset de coloane ale tabelului, astfel încât fiecare fragment reprezintă o proiecție a tabelului inițial. Pentru a reconstrui tabela originală din fragmente verticale este necesar un *JOIN natural* pe cheile comune[7]. De aceea, la fragmentarea verticală, cheia primară a tabelului trebuie inclusă (ca atribut redundant) în fiecare fragment, pentru a asigura coeziunea datelor[8]. Exemplu: un tabel *Angajați(ID, Nume, Departament, Salariu)* ar putea fi fragmentat vertical în două: unul cu coloanele (ID, Nume, Departament) și altul cu (ID, Salariu). Cheia ID apare în ambele fragmente, permițând reîntregirea datelor prin operația de join pe ID. Fragmentarea verticală reduce volumul de date transferat la interogări care necesită doar anumite coloane (ex: o aplicație de HR care accesează frecvent numele și departamentele, dar rar salariile, ar beneficia ca aceste atribute să fie într-un fragment separat, evitând citirea salariilor inutil).
3) **Fragmentare mixtă (hibridă)** – combină fragmentarea orizontală cu cea verticală. Aceasta se poate realiza în două moduri: (1) mai întâi se aplică o fragmentare verticală a unei relații, apoi fragmentele rezultate sunt fragmentate orizontal; sau (2) invers, se fragmentează orizontal relația și apoi fiecare fragment orizontal este împărțit vertical. Fragmentarea hibridă este utilă în scenarii complexe, unde datele trebuie distribuite atât pe sub-seturi de coloane, cât și pe sub-seturi de rânduri. Reconstrucția unei relații

fragmentate hibrid implică atât operații de reuniune cât și de join, în ordinea inversă fragmentării (de exemplu, dacă s-a fragmentat mai întâi vertical apoi orizontal, se vor aplica întâi operații de join pentru a recompune fragmentele verticale, apoi reuniuni pentru a reface relația completă, sau viceversa)[9][10].

Exemplu : Să presupunem o bază de date a unei companii cu tabela *Angajat(ID, Nume, Departament, Salariu, ...)*. O posibilă strategie de fragmentare mixtă ar fi: fragmentare orizontală în funcție de departament (toți angajații din Departament=IT într-un fragment, cei din Departament=Vânzări în alt fragment, etc.), apoi, pentru fragmentele foarte mari (ex. fragmentul IT cu mii de angajați), aplicăm o fragmentare verticală separând informațiile personale de cele salariale. Astfel, un nod al bazei de date ar putea stoca datele personale ale angajaților IT, alt nod datele salariale ale angajaților IT, alt nod datele angajaților de vânzări ș.a.m.d. Interogările care accesează doar numele angajaților IT pot fi satisfăcute local (pe fragmentul corespunzător), pe când o interogare complexă care cere și nume și salariu va necesita unire de date între fragmente (cost suplimentar de comunicare).

Corectitudinea fragmentării – reguli și considerații

Pentru ca o fragmentare să fie corectă (adică să nu piardă sau să dubleze date neintenționat), trebuie respectate trei reguli fundamentale[11][12]:

1) **Completitudine (fără pierdere de date):** Reuniunea (sau join-urile adecvate) tuturor fragmentelor trebuie să refacă întreaga mulțime de date originală. Niciun tuplu al relației inițiale nu trebuie omis – orice dată din tabela originală trebuie să se regăsească în cel puțin un fragment. Cu alte cuvinte, fragmentarea este o **decompozare fără pierdere** a datelor.
2) **Reconstructibilitate:** Există o operație sau un set de operații prin care se poate **reconstrui exact relația inițială** din fragmentele sale. În cazul fragmentării orizontale, operația de reuniune (UNION) a fragmentelor redă relația originală; în cazul fragmentării verticale, operația de join natural pe cheia primară reface relația originală; pentru fragmentări mixte, o combinație de join-uri și reuniuni în ordinea potrivită va reface datele. Practic, sistemul trebuie să poată răspunde corect la interogări ca și cum baza ar fi unitară, combinând datele din fragmente după necesitate.
3) **Disjuncție (fragmente ne-suprapuse):** Fragmentele nu trebuie să aibă date redundante comune. Un tuplu al relației inițiale trebuie să apară *doar într-unul* din fragmentele orizontale (fragmentele orizontale sunt disjuncte între ele)[13]. În fragmentarea verticală, același atribut (coloană) nu trebuie să apară în două fragmente diferite (cu **excepția** atributelor de cheie folosite pentru reconstrucție, care inevitabil apar în toate fragmentele verticale ca legătură)[12]. Asigurând disjuncția, evităm inconsistențe cauzate de existența aceleiași date stocate de două ori în fragmente diferite.

Distribuția datelor și transparența: Într-o bază de date distribuită corect fragmentată, utilizatorii nu știu (și nici nu trebuie să știe) pe ce nod se află un anumit fragment – această *transparență de localizare* face parte din cerințele unui SGBD distribuit robust. Sistemul răspunde interogărilor distribuind automat sub-interogări către nodurile relevante și combinând rezultatele. Desigur, fragmentarea are impact asupra optimizării interogărilor: dacă o interogare necesită date din fragmente multiple (de pe noduri diferite), costul de comunicare crește. De aceea, un principiu de proiectare este să **fragmentezi datele astfel încât majoritatea acceselor**

să fie locale (pe un singur nod), minimizând operațiile distribuite (join-uri sau reuniuni între noduri)[14][15].

Considerații practice la fragmentare: În proiectarea fragmentării, se urmăresc următoarele obiective:

1) **Localitate:** Plasarea fiecărui fragment pe nodul (sau nodurile) care folosesc cel mai des acele date. Astfel, se maximizează procentul de interogări locale (care nu traversează rețeaua). Exemplu: datele despre angajații din biroul din Cluj pot fi stocate pe serverul din Cluj, iar cele despre angajații din București pe serverul din București, dacă aplicațiile locale accesează preponderent angajații proprii.
2) **Echilibrarea încărcării:** Fragmentarea ar trebui să conducă la o distribuție cât mai uniformă a datelor și sarcinilor de prelucrare între noduri. Niciun nod nu trebuie să devină *hotspot* (supraîncărcat) în timp ce altele stau aproape idle. De exemplu, dacă un fragment ajunge de 10 ori mai mare decât altul, avantajele distribuției scad – în acest caz se pot aplica fragmentări suplimentare (sub-fragmentare) sau realocarea datelor.
3) **Minimizarea comunicației între noduri:** Prin fragmentare adecvată, se încearcă reducerea interogărilor care implică date din situri multiple. Dacă o interogare necesită combinarea fragmentelor de pe mai multe noduri, atunci costul de rețea și latența cresc. În unele cazuri, se preferă chiar **replicarea unor fragmente** (vom discuta replicarea la secțiunea 4.2) pentru a evita accesul remote, cu prețul menținerii consistenței acelor copii în plus[16][17]. Însă acest beneficiu vine cu *dezavantajul major al costurilor de actualizare și complexității de consistență* a datelor replicate[17].
4) **Securitate și confidențialitate:** Uneori fragmentarea se face și pentru a izola date sensibile. De exemplu, se pot păstra date personale într-un fragment separat cu acces restricționat, iar datele generice în alt fragment, asigurând că doar anumite noduri (și utilizatori) pot vedea informațiile critice.

Exemple moderne de fragmentare (sharding) în industrie

În practică, conceptul de fragmentare este implementat sub denumirea de **sharding** în multe sisteme de baze de date distribuite moderne, în special în zonele NoSQL și NewSQL :

a) **MongoDB (bază de date documentară NoSQL):** MongoDB suportă *sharding* nativ. O colecție (echivalentă cu o tabelă) poate fi împărțită în **shard-uri** pe baza unui *cheie de shard* (un câmp ales ca determinant al fragmentării). Fragmentarea este orizontală – fiecare shard conține un subset de documente. MongoDB folosește fie *fragmentare pe intervale (range sharding)* – ex: toți clienții cu ID între 1 și 10000 pe shard-ul 1, 10001–20000 pe shard-ul 2 etc., fie *fragmentare hash* – cheile sunt trecute printr-o funcție hash pentru a distribui uniform datele. Un cluster Mongo tipic are un server de configurare care ține evidența fragmentelor și rutează interogările către nodurile corespunzătoare. Astfel, dacă o aplicație caută un document după cheia de shard, driverul Mongo știe direct pe ce nod (fragment) se află. Fragmentarea în MongoDB permite scalarea la volume mari de date și trafic, multe aplicații web folosind această abordare.

b) **Apache Cassandra (sistem NoSQL tip key-value/coloane):** Cassandra implementează o arhitectură complet distribuită, fără master, bazată pe **fragmentare prin hashing consistent**. Datele sunt organizate în rânduri identificate de o cheie primară; un *partitioner* hash asociază fiecărei chei un token numeric pe un inel (ring) abstract, iar **nodurile** Cassandra preiau zone de valori hash. În acest mod, rândurile sunt automat

distribuite uniform pe toate nodurile clusterului[18][19]. De exemplu, Netflix (un caz real de scară mare) folosește Cassandra pentru stocarea jurnalelor de vizionare și a altor date de utilizator. Cheia de fragmentare este aleasă cu grijă – de obicei un identificator unic (user ID, de pildă) posibil combinat cu un element temporal – pentru a asigura *distribuția uniformă* a datelor și pentru a evita creșterea necontrolată a oricărui fragment[20]. Fragmentarea astfel realizată (shard-uirea datelor de utilizator) permite Netflix să stocheze petabytes de date pe sute de noduri, asigurând că nicio mașină nu devine punct unic de încărcare.

c) **Google Spanner (bază de date distribuĠtă NewSQL):** Spanner (folosit intern de Google și expus și ca serviciu Cloud Spanner) împarte automat datele în **partiții** pe criterii de interval de chei primare. Aceste partiții (denumite *splits*) sunt unitatea de distribuție și replicare. Spanner mută dinamic granițele fragmentelor pe măsură ce volumul de date și traficul evoluează, pentru a menține echilibrul încărcării. În plus, Spanner replică sincron aceste fragmente (vom detalia la replicare) menținând o vedere consistentă global. Un studiu de caz celebru este aplicația Google Photos, care stochează miliarde de fotografii în Spanner, partiționate după un ID și replicate pe continente diferite, permițând acces rapid și coerent de oriunde.

d) **Baze de date relaționale cu partajare (sharding) la nivel de aplicație:** Înainte de apariția soluțiilor NoSQL, multe companii au implementat *manual* fragmentarea pentru baze de date SQL. Un exemplu istoric este Twitter, care și-a fragmentat tabela de *tweets* pe mai multe instanțe MySQL (ex: tweet-urile utilizatorilor cu ID mod 10 = 0 pe serverul 0, mod 10 = 1 pe serverul 1 etc.), pentru a putea scala volumul uriaș de mesaje. Aplicația era conștientă de această împărțire – practic transparența de fragmentare nu era totală, fiind gestionată la nivel de logică de business. Astăzi, există și soluții middleware (ex. Vitess pentru MySQL) sau baze de date NewSQL (CockroachDB, Yugabyte) care fac automat sharding-ul datelor relaționale, combinând consistența tranzacțională cu fragmentarea și replicarea.

Beneficii și provocări: Fragmentarea permite *scalare orizontală* (adăugarea de noduri crește capacitatea sistemului aproape liniar), izolarea defecțiunilor (dacă un nod cade, restul fragmentelor pot fi încă disponibile, cel puțin parțial) și flexibilitate în amplasarea datelor (ex: datele europene pe servere din UE, cele americane în SUA pentru conformitate la reglementări). Pe de altă parte, fragmentarea necesită un efort de proiectare (alegerea corectă a schemei de fragmentare este critică) și un suport robust din partea SGBD pentru a optimiza interogările distribuie. De asemenea, intervine problema menținerii **integrității referențiale** în medii fragmentate: dacă, de exemplu, tabela *Comenzi* este pe un nod iar tabela *Clienți* pe alt nod, constrângerea foreign key devine mai dificil de aplicat. Adesea, sistemele distribuite sacrifică constrângerile între fragmente sau le validează prin mecanisme speciale. Vom vedea în secțiunile următoare că fragmentarea este adesea folosită împreună cu replicarea – fragmentele pot fi replicate pentru a crește disponibilitatea.

4.2 Replicarea datelor: sincronică vs. asincronă

Replicarea datelor constă în menținerea unor **copii identice** ale datelor pe mai multe noduri (servere) din sistem. Scopurile principale ale replicării sunt: creșterea **fiabilității** și **disponibilității** (dacă un nod cade, există o copie a datelor pe alt nod), îmbunătățirea performanței la citire (multiple copii pot deservi cereri de citire în paralel) și distribuirea geografică a datelor (copii mai aproape de utilizatori din diferite regiuni, pentru latență redusă).
Într-o bază de date distribuită, replicarea poate fi aplicată la nivel de întreagă bază de date, la nivel de tabel sau chiar la nivel de fragment de tabel.

Astfel se disting adesea:

- **replicare totală** (întreaga bază de date este duplicată pe fiecare nod – rar folosită pentru că introduce multă redundanță și cost la actualizare),
- **replicare parțială** (doar anumite fragmente/tabele sunt replicate pe anumite noduri) sau combinații (ex: unele fragmente replicatate, altele nu).

4.2.1 Mecanisme de replicare

Există mai multe modalități tehnice de a realiza replicarea, însă din punct de vedere al **temporizării și consistenței** update-urilor, replicarea se încadrează în două categorii de bază:

a) **Replicare sincronică:** atunci când un nod sursă (de obicei numit *master* sau *primar*) execută o tranzacție ce modifică date, modificările sunt propagate *imediat* la replici (nodurile secundare) **în cadrul aceleiași tranzacții**.

Cu alte cuvinte, operația de scriere nu se consideră completă până când toate (sau cel puțin un anumit set de) replicile au confirmat aplicarea modificării. Astfel, în replicarea sincronă, toate copiile datelor rămân **aliniate în timp real** – imediat după commit, orice citire de pe orice replică va vedea noile valori.

Acest mod asigură o consistență puternică între replici, însă are cost de latență: tranzacțiile trebuie să aștepte comunicația cu replicile, ceea ce poate introduce întârzieri.

Replicarea sincronă este **ideală pentru medii care necesită acuratețe strictă a datelor**, unde nu ne permitem ca vreo copie să rămână în urmă[21]. De exemplu, în sisteme bancare de contabilitate centralizată, soldurile conturilor ar trebui replicate sincron pe un site de backup; în caz contrar, un failover ar putea folosi o copie depășită, cu consecințe grave.

Exemplu:

Galera Cluster (tehnologie de replicare multi-master pentru MySQL/MariaDB) folosește o formă de replicare sincronică (mai exact *certified commit*, unde tranzacțiile sunt certificate pe toate nodurile înainte de a fi angajate). Astfel, într-un cluster Galera de 3 noduri, o inserție pe oricare nod este trimisă tuturor; dacă cel puțin o replică nu confirmă, tranzacția e anulată. Rezultatul: toate nodurile au același conținut (consistență puternică), dar cu un cost de latență suplimentar.

Un alt exemplu este **Google Spanner**, care implementează replicare sincronă pe mai multe centre de date folosind un protocol bazat pe **Paxos** și sincronizare de ceas (TrueTime) – Spanner garantează că datele sunt replicate atomic și vizibile global în același moment (consistență **linearizabilă**), permițând distribuția globală fără a sacrifica consistența strictă.

b) **Replicare asincronă:** în acest caz, replicațiile nu sunt parte din tranzacția inițială – un nod primar realizează modificările local și confirmă tranzacția către client fără a aștepta ca alte copii să fie la zi. Actualizările sunt apoi propagate către replici **cu o întârziere** (de la secunde la minute, în funcție de sistem). Astfel, replicile pot rămâne temporar *în urmă* față de sursă[21].

Replicarea asincronă este utilizată atunci când sincronizarea strictă nu este critică, prioritară fiind viteza și disponibilitatea.

Avantajul major este că aplicațiile care scriu pe baza de date nu mai așteaptă rețeaua, deci latența de scriere e scăzută și sistemul e mai disponibil – chiar dacă unele replici sunt căzute, primarul continuă să proceseze tranzacții, acestea urmând să fie recuperate ulterior de replici (conceptul de **catch-up** sau **replay** al jurnalului de tranzacții).

Dezavantajul evident este că în ferestrele de timp dintre scrierea pe primar și actualizarea replicilor, datele nu sunt consistente pe toate nodurile: un client care citește de pe o replică secundară ar putea vedea **date vechi** (*stale*) față de cele de pe primar.

Replicarea asincronă se potrivește multor scenarii de web și aplicații distribuite unde se poate tolera consistență eventuală (de exemplu, într-un site de e-commerce poate nu este critic ca inventarul afișat să fie actualizat la milisecundă pe toate serverele global, atâta timp cât sistemul finalizează convergența datelor într-un timp rezonabil).

Exemplu:

Cel mai cunoscut exemplu este replicarea standard **master-slave** în MySQL: baza de date primară (master) scrie toate modificările într-un binlog, iar replicile secundare (slaves) preiau asincron acest jurnal și aplică modificările. Dacă master-ul cade, un slave poate prelua rolul (failover), însă orice tranzacții care nu apucaseră să fie replicate se pot pierde. Mulți utilizatori MySQL aleg replicarea asincronă pentru scalarea la citire (slaves multiple deservesc interogări SELECT), acceptând un mic lag.

Un alt exemplu, în zona NoSQL, este **MongoDB Replica Set**: Mongo folosește un model primar-secundar, unde secundarele aplică asincron fluxul de operații (oplog) de la primar. Implicit aplicațiile citesc de pe primar (date consistente), dar dacă configurezi citiri de pe secundare poți observa întârzieri – dacă imediat după un update încerci să citești de pe o replică secundară, s-ar putea să nu vezi modificarea instantaneu, evidențiind natura asincronă.

4.2.2 Sincron vs. asincron – rezumat comparativ:

		Descriere
1.	**Consistența datelor**	replicarea sincronă asigură **consistență imediată** între toate copiile (toți clienții văd aceleași date în orice moment după commit), pe când replicarea asincronă permite **inconsistențe temporare** – copiile pot conține versiuni anterioare ale datelor până ce update-urile ajung la ele[22]. Sistemele asincrone oferă de obicei doar **consistență eventuală** – se garantează că dacă nu mai intervin noi actualizări, în timp toate replicile vor converge la același conținut.
2.	**Performanță**	replicarea sincronă adaugă latență fiecărei tranzacții de update (trebuie să aștepte confirmări de la replici), deci **scrierile sunt mai lente**, și dacă o replică nu răspunde, sistemul poate bloca (dacă se așteaptă confirmarea

		ei). Replicarea asincronă are **throughput mai mare** la scriere și reziliență mai bună la noduri căzute (primarul nu așteaptă replicile, deci poate continua să proceseze tranzacții), îmbunătățind disponibilitatea percepută. În schimb, citirile pot fi servite de oricare replică; în sisteme asincrone, dacă permitem citiri de pe replici pot fi foarte rapide (load balancing), dar pot returna date stale, pe când în sisteme sincrone orice replică are date up-to-date dar citirile sunt eventual mai lente din cauza sincronizării (uneori replicarea sincronă implică replici în aceeași locație sau mecanisme optimizate, deci impactul la citire e minim).
3.	**Complexitate**	replicarea asincronă este mai **simplu de implementat** (de exemplu prin jurnalizare și replay ulterior) și decuplată, pe când replicarea sincronă necesită protocoale distribuite de commit (ex: două faze – 2PC, sau consens distribuit) care sunt mai complexe și pot avea probleme de blocaj (de exemplu, 2PC blochează în caz de eșec al coordonatorului, de aceea sisteme moderne preferă protocoale de consens Paxos/Raft). Totodată, replicarea asincronă necesită proceduri de **recuperare** a datelor pierdute la failover (dacă un primar cade înainte ca replicile să primească ultimele update-uri, acele update-uri pot fi pierdute definitiv sau trebuie recuperate din jurnal/restaurare).

Bibliografie:

[1] [2] [3] [5] [35] Fragmentation in Distributed DBMS - GeeksforGeeks
https://www.geeksforgeeks.org/dbms/fragmentation-in-distributed-dbms/

[4] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] Fragmentarea Bazelor de Date Distribuite | PDF
https://ro.scribd.com/doc/269106699/Fragmentarea-Bazelor-de-Date-Distribuite

[18] [19] [20] [23] [24] [27] [28] [29] [30] [32] [33] Case Study: Navigating the CAP Theorem — Netflix's Balance of Consistency, Availability, and Partition Tolerance | by Disant Upadhyay | Medium
https://disant.medium.com/case-study-navigating-the-cap-theorem-netflixs-balance-of-consistency-availability-and-4f8794c7aac7

[21] [22] Ce este replicarea bazelor de date? Și cum funcționează - OPSWAT
https://romanian.opswat.com/blog/database-replication

[25] [26] [31] [34] TSM - Bazele de date NoSQL - o analiză comparativă
http://www.todaysoftmag.ro/article/304/bazele-de-date-nosql-o-analiza-comparativa