Curs 5
Fragmentarea datelor, replicare și consistență în sisteme de baze de date distribuite și mobile (partea II)

5.1 Strategii de replicare – topologii

Indiferent de sincron/asynchron, replicarea poate fi organizată în diverse topologii:

A. **Primar-secundar (Master-Slave):** Un nod primar preia toate operațiile de scriere, iar unul sau mai multe noduri secundare primesc copii ale datelor (de obicei doar în regim read-only). Acesta e modelul clasic folosit de multe SGBD relaționale (MySQL, PostgreSQL cu streaming replication etc.) și de unele NoSQL (MongoDB). Avantajul e simplitatea – conflictele de scriere sunt evitate deoarece doar un nod face scrieri. Dezavantajul e că primarul poate deveni bottleneck la scrieri și este un punct unic de eșec (deși se poate face failover la un secundar dacă primarul pică). Replicarea poate fi configurată sincronă (unele sisteme suportă semi-sincron – primarul așteaptă confirmare de la cel puțin un secundar) sau asincronă (cel mai des, pentru performanță).

B. **Multi-master (Primar-Primar sau replicare activ-activ):** Mai multe noduri acceptă tranzacții de actualizare simultan, replicând modificările între ele. Acest mod asigură disponibilitate ridicată (oricare nod poate prelua scrieri, deci dacă unul cade, altul e disponibil) și e util pentru geo-distribuție (utilizatorii din Europa scriu pe nodul UE, cei din Asia pe nodul local, etc., cu replicare între centre). Provocarea majoră este **rezolvarea conflictelor**: dacă doi masteri diferiți modifică aceeași înregistrare aproape concomitent, sistemul trebuie să decidă o ordine sau o fuziune a acestor actualizări. Replicarea multi-master poate fi implementată *sincron* (ex: sisteme care folosesc algoritmi de consens – toți masterii ajung la acord pentru fiecare tranzacție, practic transformând scrierea într-un commit distribuit; de exemplu, CockroachDB, YugabyteDB folosesc Raft pentru a replica în mod sincron pe mai multe noduri fiecare range de date) sau *asincron* (ex: sistemele DNS sau unele directoare LDAP unde masterii se sincronizează periodic și eventual au mecanisme de *last write wins* sau vectori de versiune pentru conflicte). Un exemplu notabil de replicare multi-master asincron a fost Microsoft SQL Server Merge Replication sau Oracle Multi-Master, care necesitau definirea de proceduri de rezolvare a conflictelor. În zona NoSQL, **Cassandra** poate fi văzut ca un sistem multi-master asincron: oricare nod poate primi scrieri, pe care le propagă la replicile sale; Cassandra evită conflictele având o politică deterministă – fiecare scriere are un timestamp și *cel mai recent timestamp câștigă* în cazul conflictelor, asigurând convergența[23]. Totuși, aplicațiile trebuie proiectate având în vedere acest comportament (de exemplu, două incrementări paralele pe același cont bancar ar fi problematice, deoarece ultima scriere ar suprascrie-o pe cealaltă dacă nu se iau măsuri speciale).

C. **Replicare pe bază de instantanee (snapshot) vs. replicare continuă:** Unele sisteme folosesc replicare prin **instantanee periodice** – adică la intervale regulate se ia o copie a întregii baze sau a modificărilor și se aplică pe replici. Aceasta este utilă când datele nu se schimbă foarte des sau consistența strictă nu e necesară; de exemplu, pentru baze de date de raportare (reporting) care se actualizează o dată pe zi din baza operațională. Cealaltă

abordare este replicarea **tranzacțională continuă**, unde fiecare tranzacție (sau fiecare operație de DML) este capturată și trimisă imediat către replici (fie sincronic, fie asincronic). Majoritatea arhitecturilor de replicare modernă (inclusiv cele discutate mai sus) sunt tranzacționale continue.

Exemple industriale:

a) *MySQL:* replicare primar-secundar prevalent asincronă. Recent, plugin-ul *Group Replication* și *Galera* permit replicare multi-master sinc, dar cu anumite limitări (de exemplu, tablele trebuie să aibă cheie primară, tranzacții optimiste etc.). Folosirea replicării MySQL a stat la baza multor arhitecturi web 2.0, de exemplu Wikipedia a folosit replicare master-slave pentru a separa încărcarea de citire (mulți slaves pentru citire) de scrieri (un master).

b) *PostgreSQL:* are replicare streaming (asincronă de obicei) primar-secundar și opțiune de *synchronous_standby* (poți configura ca master-ul să aștepte confirmarea de la cel puțin un secundar – replicare semi-sincronă). De asemenea, există proiecte ca Postgres-BDR (bi-directional replication) pentru multi-master asincron.

c) *Cassandra:* replică datele pe baza unui *factor de replicare* configurat (ex: RF=3 înseamnă că fiecare fragment (partiție de date) este stocat pe 3 noduri diferite). Replicarea este *de tip eventual consistent*, iar Cassandra oferă **consistență ajustabilă (tunable consistency)**: clientul alege la fiecare operație câte replici trebuie să confirme. De exemplu, la scriere se poate specifica CL=ONE (confirmare doar de la un nod – restul actualizărilor se fac asincron), CL=QUORUM (majoritatea replicilor trebuie să confirme – asigură un echilibru, cum vom vedea în CAP), sau ALL (toate replicile – practic devine echivalent cu o replicare sincronă strictă). Netflix, menționat anterior, folosește **consistență local-quorum** pentru scrieri critice: când un user finalizează vizionarea unui episod, scrierea în Cassandra e considerată reușită după confirmarea de la un *quorum* (de obicei 2 din 3) de noduri din centrul local[24]. Astfel, datele sunt redundante suficient, iar dacă un nod nu primește imediat scrierea din cauza unei defecțiuni de rețea, sistemul nu se oprește – acea scriere va fi transmisă asincron când nodul revine (mecanismul *hinted handoff* păstrează „indicații" pentru a trimite ulterior update-uri nodurilor indisponibile[23]). Cassandra asigură astfel *disponibilitate ridicată* (nu blochează scrierea) cu *consistență eventuală* (replicile rămase în urmă se repară în timp, eventual chiar la citire prin *read repair*).

d) *MongoDB:* într-un replicaset Mongo, replicarea este asincronă pe un singur master. Totuși, aplicația poate alege nivelul de confirmare al scrierilor (*write concern*). De exemplu, w=1 înseamnă confirmare după ce primarul a scris local (asincron complet pentru replici), w=majority înseamnă așteaptă ca majoritatea nodurilor din replicaset să confirme scrierea înainte de a răspunde clientului (asemănător cu un quorum, oferind o consistență mai puternică). Un replicaset tipic are 3 noduri: 1 primar, 2 secundare; dacă primarul cade, secundarele negociază și promovează una nouă ca primar (failover automat). Acest mecanism oferă disponibilitate foarte bună și toleranță la defecțiuni, însă trebuie acceptat că între momentul căderii primarului și promovarea noului primar, unele tranzacții recente pot să nu fi ajuns la secundar – potențial pierdute. Din acest motiv, aplicațiile critice folosesc journaled writes și uneori replicare sincronă (cu write concern ridicat) pe două centre, de exemplu.

e) *Sisteme de tip masterless (Dynamo-like):* Unele baze de date NoSQL (Riak, Amazon DynamoDB, chiar Cassandra într-un fel) nu au un master static; orice nod poate primi trafic

și replicarea se face între noduri peer-to-peer. **Amazon Dynamo** (progenitorul multor NoSQL de astăzi) folosea replicare asincronă cu un model *optimist*: datele nou scrise pe un nod sunt propagate celorlalte replici în background. Dacă apar *conflicte* (două scrieri asupra aceluiași element pe noduri diferite), sistemul păstrează multiple versiuni și rezolvă conflictul la citire, de obicei folosind *vectori de versiune* pentru a determina dacă una din versiuni este „ancestral" față de cealaltă sau sunt paralele. În caz de conflict real, se putea aplica o politică de reconciliere (ultima scriere câștigă, fuziune de valori, etc.)[25][26]. Această abordare, deși complexă, permite scalarea și disponibilitatea maxime (niciun nod nu e critic), în schimb programatorii trebuie să se ocupe de eventualele inconsistențe temporare.

Nota: În sisteme distribuite mobile (ex: baze de date replicate pe dispozitive mobile ce se sincronizează cu un server central), replicarea este aproape întotdeauna asincronă.
Dispozitivul mobil lucrează offline pe o copie locală a datelor și periodic face *sync* cu serverul – trimițând modificările locale și primind pe cele de la alții.
Aceasta este de fapt o replicare multi-master asincronă: atât serverul cât și clientul pot face schimbări independent, iar la sincronizare trebuie rezolvate conflictele.

Un exemplu este sistemul de sincronizare al **CouchDB/Couchbase Mobile**, care folosește versiuni și politică de merge (sau marcarea conflictelor pentru rezolvare manuală). În mod similar, aplicații ca Evernote sau agenda de contacte de pe telefon folosesc replicare asincronă (prin internet) și eventual consistență, garantând că la final toate dispozitivele vor avea aceleași date, chiar dacă temporar pot exista discrepanțe.

5.2 Consistența și coerența datelor în sistemele distribuite

Conceptul de **consistență a datelor** într-un sistem distribuit are mai multe fațete. În sens clasic (ACID), *Consistența* se referă la asigurarea integrității: o tranzacție pornește de la o bază de date într-o stare validă și o lasă într-o stare validă, menținând toate constrângerile de integritate. În sensul arhitecturilor distribuite și al teoremei CAP însă, **Consistența (C)** se referă la faptul că toți clienții sistemului văd aceleași date la scurt timp după orice actualizare – mai formal, *oricărei citiri i se garantează că vede fie rezultatul complet al ultimei scrieri, fie o eroare*[27]. Cu alte cuvinte, datele replicate pe multiple noduri trebuie să se comporte ca și cum ar fi o singură copie: aceasta este numită și **consistență puternică** sau *consistență strictă*.
Prin contrast, noțiunea de **coerență a datelor** este deseori folosită pentru a descrie *uniformitatea și absența conflictelor* între copiile datelor. Un sistem coerent asigură că datele nu devin contradictorii – de exemplu nu vom avea două versiuni diferite „adevărate" ale aceleiași entități la noduri diferite; eventual, pe termen lung, toți vor vedea aceeași valoare. Termenul de *coerență* este

folosit frecvent în contextul *cache-urilor distribuite* sau al memoriilor distribuite (cache coherence), referindu-se la principiul că două memorii cache nu ar trebui să aibă valori diferite pentru aceeași locație de memorie. Prin extensie, în baze de date replicate, coerența datelor implică faptul că **actualizările multiple ajung în toate replicile într-un mod compatibil temporal** (respectând anumite proprietăți, ca ordine cauzală sau monoticitate).

În general, putem considera că **consistența** în sisteme distribuite înseamnă *cât de strict sincronizate sunt datele între noduri*, iar **coerența** se referă la *calitatea generală a datelor replicate* – lipsa conflictelor, integritatea lor globală și faptul că toți nodurile vor converge spre aceeași stare. Adesea termenii sunt folosiți interschimbabil, dar îi vom detalia sub aspectul **modelelor de consistență**.

Modele de consistență

Într-un sistem ne-distribuit (centralizat), există practic un singur mod de consistență: orice citire după o scriere vede efectul acelei scrieri (serializare strictă a tranzacțiilor). Într-un sistem distribuit, mai ales în prezența replicării asincrone, pot fi definite mai multe **modele de consistență (consistency models)**, care sunt garanții formale despre ce valori pot fi returnate de operațiile de citire în diverse situații. Iată câteva modele importante, de la cel mai puternic la cel mai slab:

1) **Consistență strictă (linearizabilitate / consistență puternică):** Echivalentul distribuit al unui sistem atomic centralizat. Orice operație de citire va vedea fie starea inițială, fie rezultatul complet al celei mai recente operații de scriere (oricât de distribuită ar fi aceasta). Practic, există o ordine globală a tuturor operațiilor, respectată de toate nodurile. Dacă un proces X a scris valoarea v la ora 10:00:00, orice citire făcută după acest moment (în timp absolut, folosind un ceas global) va returna fie v, fie o valoare mai nouă dacă au existat scrieri ulterioare. Acest model e ideal dar foarte dificil de implementat în medii distribuite reale (necesită sincronizare de ceas sau protocoale greoaie). **Google Spanner** este un exemplu rar care pretinde aproape linearizabilitate globală (prin TrueTime). Majoritatea SGBD-urilor distribuite nu oferă linearizabilitate peste replici în mod implicit.
2) **Consistență secvențială:** Este un model ușor mai slab, în care toți nodurile văd **aceeași ordine a actualizărilor**, dar acea ordine nu e neapărat legată de timpul real. E ca și cum toate operațiile ar fi executate într-o secvență (posibil diferită de ordinea reală de inițiere, atâta timp cât este una valabilă) și toată lumea o vede așa. Consistența secvențială menține iluzia unui sistem unic, dar nu garantează că dacă un client face o scriere și apoi citirea imediat, va vedea neapărat propria scriere (dacă scrierea lui stă la coadă în secvența globală). E un model teoretic; în practică, sistemele cu replicare asincronă nu ating nici consistența secvențială decât dacă au un arbitru global de ordine.
3) **Consistență causală:** Garantează că dacă o operație B este *cauzal dependentă* de operația A (de exemplu, A este un write, B este un read făcut de același client după ce a văzut A, sau B este un write care teoretic se bazează pe cunoașterea rezultatului lui A), atunci **toți nodurile** vor vedea A înainte de B. În schimb, dacă două operații sunt independente (nelegate cauzal), pot fi văzute în ordine diferită pe noduri diferite. Consistența cauzală este mai slabă decât secvențiala, dar mai puternică decât eventuală simplă, asigurând un nivel de intuiție: de exemplu, dacă un utilizator postează un comentariu (operația A) și apoi postează o editare la acel comentariu (operația B), ceilalți utilizatori nu vor vedea *editarea* înaintea *comentariului inițial* – ordinea cauzală se păstrează. Însă două comentarii fără legătură, de la autori diferiți, ar putea apărea în ordine diferită la două replici temporar. Implementarea practică a consistenței cauzale folosește de obicei vectori de timp (vector clocks) pentru a urmări dependențele. Unele sisteme de mesagerie și stocare partajată

implementau consistență cauzală (ex: sistemul Coda, sau unele DHT-uri avansate). Modele de consistență *de sesiune* sau *PRAM consistency* (Pipeline RAM) sunt variante slabite ale consistenței cauzale.

4) **Consistență eventuală (eventual consistency):** Este modelul folosit de multe sisteme distribuite scalabile (ex: Dynamo, Cassandra în modul default, serviciile DNS, sistemele de fișiere replicate asincron etc.). Consistența eventuală afirmă că, *în absența unor actualizări noi*, toate replicile vor converge **în cele din urmă** la aceeași stare[25]. Nu se specifică cât de repede, dar presupunem că mecanismele de replicare/anti-entropie fac ca toate update-urile să circule și sistemul să se stabilizeze. Până la convergență, datele pot fi inconsistente (necoerente) între noduri – adică poți citi o valoare veche de pe o replică și o valoare nouă de pe alta în același timp. Important e că *„eventually"* (cândva) tot sistemul va fi la zi. Consistența eventuală permite disponibilitate maximă și toleranță la partiții, conform teoremei CAP (discutăm imediat), dar vine cu sarcina suplimentară a **gestionării conflictelor** de actualizare. Dacă două actualizări concurente au loc în perioada în care replicile sunt izolate (ex: fiecare centru de date își actualizează aceeași înregistrare diferit), la reconectare sistemul trebuie fie să aplice o regulă de fuziune (de exemplu ultima scriere câștigă după un timestamp global – cum face Cassandra[28], sau se mențin ambele versiuni și se cere rezolvare manuală sau automată la nivel aplicație – cum se poate întâmpla în Dynamo/Riak cu vector clocks[26]). Astfel de situații sunt rare în anumite aplicații și dese în altele, deci alegerea modelului eventual consistent trebuie făcută ținând cont de toleranța la anomalii a aplicației.
5) **Consistență de sesiune / lectură-proprie (read-your-writes consistency):** Este un model pragmatic mai slab decât puternic, dar care îmbunătățește experiența individuală a utilizatorilor: garantează că un client (sesiune) își vede propriile actualizări. Adică după ce clientul X scrie ceva, orice citire ulterioară (de către X) va reflecta acea schimbare (fie forțând citirea de pe replica pe care s-a scris, fie propagând update-ul rapid). Însă alt client Y ar putea încă vedea vechea valoare până la convergența globală. Acest model este des întâlnit în sisteme de caching distribuit (ex: cache local per utilizator care invalidează intrările modificate de utilizatorul respectiv) sau în servicii ca Amazon S3 în modul „read-after-write consistency" pentru același client.
6) **Consistențe mai slab definite:** Unele sisteme pur asincrone vorbesc de *consistență slabă* (weak consistency) – practic orice poate fi returnat la citire, atâta timp cât eventual se converg – sau definiri speciale (ex: *monotonic read consistency* – odată ce ai citit o versiune nouă, nu vei mai vedea versiuni mai vechi la lecturile viitoare; *monotonic write consistency* – sistemul asigură că scrierile de la același client se aplică în ordine).

În practica ingineriei software, modelele de consistență sunt adesea rezumate la două extreme: **consistență puternică** vs. **consistență eventuală**.

7) Un sistem cu consistență puternică (strictă) se comportă ca și cum ar exista o singură copie a datelor: este mai simplu de gândit, însă pentru a realiza asta într-un mediu distribuit adesea trebuie sacrificate *disponibilitatea* sau *performanța*.
8) Un sistem cu consistență eventuală acceptă perioade de incoerență în schimbul *disponibilității continue* și al *performanței mai bune*, promițând doar că la final datele se vor alinia.

Teorema CAP și compromisurile consistenței

Eric Brewer a formulat așa-numita **Teoremă CAP**, care afirmă că într-un sistem de stocare distribuit, nu poți garanta simultan toate cele trei proprietăți: **Consistență** (C), **Disponibilitate**

(Availability – fiecare cerere primește răspuns, chiar dacă nu cel mai actualizat) și **Toleranță la partiții** (P – sistemul continuă să funcționeze chiar dacă rețeaua pierde/delay-ează unele mesaje între noduri)[27]. În prezența inevitabilă a partițiilor de rețea (P este nerenunțabil dacă vrei distribuție, căci rețelele pot eșua), un sistem trebuie să aleagă între a fi **consistent** sau **disponibil** atunci când apar astfel de defecțiuni de rețea[29].

Astfel se vorbește de sisteme tip **CP** (Consistent+Partition-tolerant) care sacrifică disponibilitatea în caz de probleme de rețea – de exemplu, preferă să refuze sau să întârzie răspunsul la unele cereri decât să returneze date potențial inconsistente – versus sisteme **AP** (Available+Partition-tolerant) care sacrifică consistența strictă – vor răspunde întotdeauna, chiar dacă asta înseamnă că în timpul partiției diferiți clienți pot vedea informații diferite (inconsistență temporară)[30]. Sistemele de tip **CA** (Consistent+Available fără partiții) sunt cele centralizate sau distribuite pe rețele perfecte – practic, în realitate, dacă apare o partiție majoră, ele nu pot fi simultan consistente și disponibile. Exemple de **sisteme CP**: baze de date NewSQL precum Spanner, CockroachDB (preferă consistența, iar în caz de partiție pot bloca scrieri sau chiar opri operațiuni până la reconectare), sau scenarii precum servicii de plăți/banking – mai bine opresc serviciul câteva secunde decât să permită o actualizare divergentă a balanțelor. Exemple de **sisteme AP**: multe baze NoSQL orientate pe *high availability*, precum Cassandra, Riak, Dynamo – ele au fost proiectate să nu oprească niciodată serviciul (Availability), acceptând modelul eventual consistent. De exemplu, **Riak** (o bază de date key-value) consideră toate nodurile egale, replică datele în 3 locuri și se laudă cu *reziliență extremă*, asigurând disponibilitate și toleranță la partiții; în schimb, oferă doar consistență eventuală, rezolvând conflictele prin vectori de versiune și merge de date[25][31]. **MongoDB** într-un cluster multi-site configurat cu replicare asincronă ar fi mai aproape de AP (poate servi date potențial stale dintr-un datacenter izolat, decât să refuze lectura).

Merită menționat că teorema CAP simplifică puțin lucrurile – există și concepte ca **consistența parțială** sau **toleranță la partiții în anumite limite**, dar ca principiu de bază ea ne ghidează în proiectarea sistemelor: **alegerea modelului de consistență este un compromis direct cu disponibilitatea sub condiții de rețea dificilă.**

Consistență vs. performanță: Chiar și fără partiții, consistența puternică costă în termeni de latență (datorită replicării sincrone, commit distribuit etc.). Astfel, un alt compromis este *consistență vs. latență*. S-a observat în practică că pentru **aplicații interactive la scară globală**, utilizatorii preferă viteză și disponibilitate în locul unei actualizări imediate a tuturor nodurilor. De pildă, la **Netflix**, serviciul de streaming global, arhitectura aleasă (bazată pe Cassandra) prioritizează disponibilitatea și viteză: datele sunt replicate cross-regional, însă în caz de partiție, serviciul funcționează mai departe servind eventual date ușor vechi, iar sincronizarea se face ulterior (consistență eventuală)[30]. Astfel, un utilizator ar putea să nu vadă *imediat* progresul vizionării sincronizat pe toate dispozitivele, dar sistemul nu-l blochează – iar în câteva secunde/minute, totul se aliniază.

Asigurarea coerenței și consistenței – mecanisme moderne

Cum menținem datele **coerente** și **fără conflicte** în medii distribuite complexe? Am văzut câteva mecanisme: confirmarea de la mai multe replici (quorum) pentru a decide ce operații sunt vizibile, marcarea fiecărei scrieri cu vectori de timp sau timestamp-uri și alegerea uneia ca dominantă, *anti-entropy* (procese de sincronizare periodică între replici pentru a corecta diferențe). Vom enumera câteva tehnici și studii de caz:

9) **Quorum pentru consistență parțială:** Mecanismul de quorum (folosit în Cassandra, Dynamo, RavenDB etc.) implică definirea a două numere: R = câte replici trebuie să confirme o **lectură** și W = câte replici trebuie să confirme o **scriere**. Pentru a avea

consistență puternică, se alege R + W > N (N = număr total de replici). De exemplu, în Cassandra cu N=3 replici, dacă setăm W=2 și R=2 (quorum = 2), atunci orice citire și scriere se suprapun într-o replică comună cel puțin, garantând că citirea quorum vede cea mai recentă scriere quorum. Netflix, în exemplul anterior, folosește local quorum pentru scrieri, iar la citire în mod obișnuit CL=1 sau QUORUM în funcție de nevoie[24][23]. Acest model este un compromis: e mai consistent decât eventual (pentru că cel puțin o majoritate se pune de acord), dar nu la fel de strict ca ALL. De asemenea, dacă noduri sunt căzute, atâta timp cât există un quorum disponibil, sistemul merge mai departe.

10) **Rezolvarea conflictelor la consistență eventuală:** În sisteme AP, când două noduri au actualizat diferit aceeași entitate, *coerența* se restabilește prin reconcilieri. Un exemplu: **Cassandra** folosește *timestamps* asociate fiecărei scrieri; când replicile compară date, se păstrează varianta cu timestamp-ul cel mai mare (presupunând ceasuri sincronizate aproximativ). Acest mecanism *„ultima scriere câștigă”* este simplu dar poate duce la pierderea unei actualizări (dacă două scrieri au suprascris valori diferite, una va fi ignorată). Alte sisteme, precum **Riak** sau **CouchDB**, păstrează ambele versiuni ale unui document când detectează un conflict (ramificând istoricul în doi *părinți*). Apoi, oferă fie aplicației, fie unui proces automat oportunitatea să *fuzioneze* versiunile (de exemplu, dacă erau adăugări într-o listă, se pot uni listele; dacă erau editări de același câmp, poate se ia decizia business de a păstra una). Pentru a ști că două versiuni au un strămoș comun sau nu, se folosesc **vector clocks** (vectori de versiune) – o structură ce ține pentru fiecare replică un contor al update-urilor, permițând detectarea relațiilor de cauzalitate. Dacă două versiuni au vectori incomparabili (adică fiecare are ceva ce celălalt nu are), înseamnă că au apărut concurrent – deci conflict real. Amazon Dynamo descria acest mecanism în celebrul lor whitepaper, fiind un pionier al gestionării conflictelor la scară mare.
11) **Cohorentența datelor în caches și sisteme de memorie distribuită:** Un subiect adiacent este cum păstrezi datele coerente între baza de date și eventual un cache distribuit (ex: **memcached** sau **Redis**). Practic, cache-ul e ca o replică asincronă a datelor. Strategii comune includ: *invalidation* (șterge sau marchezi ca invalid entry-ul din cache pe un update în baza principală, forțând următoarea citire să ia valoarea nouă) sau *write-through* (actualizezi și baza și cache-ul sincron). Dacă aceste metode nu se aplică și se citesc date stale din cache după ce baza a fost actualizată, apare fenomenul de **incoerență** cache-database. Din nou, decizia ține de cât de critice sunt datele vs. câștigul de performanță. Exemple: la timeline-ul rețelelor sociale, e acceptabil să vezi o postare nouă cu câteva secunde întârziere dacă a fost servită din cache; la starea stocului unui produs, poate e nevoie de strictețe mai mare (evitând supravânzarea unui produs stoc 0, de exemplu).
12) **Sisteme cu consistență configurabilă:** Merită menționat **Azure Cosmos DB** – un serviciu cloud multi-model (documente, grafe, etc) distribuit global, care oferă dezvoltatorului 5 opțiuni de consistență la alegere: **Strong**, **Bounded-Staleness** (consistență puternică cu o întârziere sau fereastră de versiuni permisă – ex: toate replicile pot fi cu max 5 secunde sau 100 updates în urmă), **Session** (consistență de sesiune – fiecare client își vede scrierile), **Consistent Prefix** (fiecare replică vede modificările în aceeași ordine, dar poate să fie întârziată – fără a sari peste vreuna; practic nicio replică nu va vedea o scriere nouă fără să le fi văzut pe toate anterioare, doar că poate fi în urmă cu un prefix), și **Eventual**. Acest mod de a expune consistența ca parametru oferă flexibilitate – de exemplu, pentru un container de date unde integritatea e critică se poate alege Strong, iar pentru altul (ex: telemetrie, loguri) se alege Eventual, obținând latențe mai mici.

13) **Toleranța la scrieri conflictuale prin CRDT-uri:** Un avans recent în menținerea coerenței datelor distribuite sunt **CRDT (Conflict-free Replicated Data Types)** – structuri de date replicabile care garantează convergența automată fără coordonare, indiferent de ordinea aplicării operațiilor. Exemple: seturi care doar adaugă elemente (G-Set) sau care pot adăuga și elimina cu tracking (2P-Set, OR-Set), contoare care numai cresc (PN-Counter) sau cresc/scad (PN-Counter cu parți), registre ultimul-câștigă etc. Folosirea CRDT-urilor în aplicații (ex: colaborare în timp real gen Google Docs, sau stocare de preferințe user distribuită) permite obținerea consistenței eventuale fără ca două modificări paralele să se anuleze incorect – CRDT-urile asigură proprietatea de *merge fără conflict*. De exemplu, două persoane care editează același document offline pot avea fluxuri de editare care la reunire sunt combinate coerent (prin structuri ca trellis, lamport timestamps per caracter etc.). Deși aceste concepte sunt avansate, ele devin tot mai practice (Redis și Riak au suport pentru unele CRDT-uri, iar librării de sincronizare ca Automerge, Yjs etc. le folosesc pentru editor colaborativ).

Rezumat: Consistența și coerența datelor în sisteme distribuite reprezintă de fapt *alegeri de design* – nu există o soluție universal mai bună, ci depinde de cerințele aplicației. Dacă aplicația este critică pentru finanțe, sănătate, procesarea de tranzacții care nu se pot pierde – se va opta pentru un model puternic (fie centralizare, fie replicare sincronă și eventual întreruperea serviciului la probleme). Dacă aplicația este un serviciu global de social media sau streaming unde uptime-ul continuu e vital și utilizatorii pot tolera mici inconsecvențe (ex: numărul de like-uri afișat poate fi vechi de câteva secunde) – atunci se alege un model eventual consistent, cu replicare asincronă, caching agresiv etc., știind că datele vor fi coerente pe termen lung și sistemul va fi mereu disponibil.

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