Curs 7
Controlul Concurenței în Baze de Date Distribuite și Mobile

6.1 Protocolul de blocare în două faze (2PL)

Definiție: Protocolul de blocare în două faze (Two-Phase Locking – 2PL) este o metodă *pesimistă* de control al concurenței bazată pe blocări (lock-uri), care garantează *serializabilitatea* execuției tranzacțiilor concurente[1][2]. Numele provine de la faptul că execuția fiecărei tranzacții este împărțită în două faze distincte în ceea ce privește manipularea blocărilor:

1) **Faza de creștere (faza de** expansiune**):** tranzacția poate *dobândi* noi blocări după nevoie, dar **nu are voie să elibereze** niciun lock în această fază[3]. Numărul de lock-uri deținute de tranzacție **doar crește** în această perioadă.
2) **Faza de descreștere (faza de** contracție**):** odată ce tranzacția începe să *elibereze* un lock, intră în faza a doua, în care **nu mai are voie să obțină noi blocări**[3]. Toate lock-urile deținute sunt eliberate treptat (sau simultan la final), numărul lor **doar scade**.

Regula 2PL: Nicio tranzacție nu poate solicita noi blocări *după* ce a eliberat vreun lock. Cu alte cuvinte, toate operațiile de LOCK trebuie să precede orice operație de UNLOCK în cadrul tranzacției[4]. Această regulă asigură că intercalarea tranzacțiilor produce un schedule conflict-serializabil (echivalent cu o execuție serială)[1][2]. Practic, dacă două tranzacții intră în conflict (accesează aceleași date, cu cel puțin una scriind), 2PL garantează că ordinea lor de acces respectă o ordine serială legală: dacă tranzacția **A** intră în conflict cu tranzacția **B**, atunci B fie a terminat complet înainte ca A să înceapă (a eliberat toate lock-urile), fie B va aștepta ca A să se termine, evitând intercalări nepermise[5].

Tipuri de blocări:

Protocolul 2PL folosește de obicei două tipuri de lock-uri pe obiectele de date: *blocare partajată* (read-lock sau S) pentru operații de citire și *blocare exclusivă* (write-lock sau X) pentru operații de scriere[6].

Reguli de compatibilitate: mai multe tranzacții pot deține simultan lock-uri de citire pe același element (acces concurent la citire este permis), dar un lock de scriere este exclusiv – el blochează orice alt acces concurent (nici citire, nici altă scriere nu sunt permise pe obiectul respectiv)[6][7].

O tranzacție trebuie să obțină un lock S pe un obiect înainte de a-l citi și un lock X înainte de a-l modifica. Dacă un alt tranzacție deține deja un lock incompatibil, tranzacția curentă va **aștepta** (este blocată) până când acel lock este eliberat[8][9].

Acest mecanism previne *interferența* (de ex. împiedică două tranzacții să scrie simultan același obiect sau ca o tranzacție să citească un obiect aflat în curs de modificare nefinalizată de altcineva).

Varianta Strictă (Strict 2PL): O problemă potențială a protocolului 2PL de bază este apariția *lecturilor murdare* sau a abortărilor în cascadă – situații în care o tranzacție B citește date modificate dar necomise de tranzacția A, iar dacă A face rollback, B trebuie și ea anulată. Pentru a preveni acest scenariu, în practică majoritatea SGBD-urilor folosesc *Protocolul 2PL Strict*, în care **toate lock-urile exclusive (X)** sunt păstrate până la finalul tranzacției (până la commit sau rollback), nelăsând alte tranzacții să vadă date necomise[10][11]. Strict 2PL implică eliberarea tuturor lock-urilor *doar după commit/abort*, garantând astfel **recoverability** (nu apar citiri de la

tranzacții necomise) și izolarea de nivel serializabil strict (echivalent cu linializabilitate în multe cazuri)[12][13]. O variantă și mai restrictivă este *Strong Strict 2PL (SS2PL)*, unde și lock-urile de citire sunt reținute tot până la commit, însă acest lucru poate reduce și mai mult paralelismul. **2PL conservativ (C2PL)** este o altă variantă în care tranzacția obține *anticipat* toate blocările de care va avea nevoie (de obicei declarându-și setul de date citite/scrise la început); astfel se evită complet deadlock-urile, însă această abordare este dificil de aplicat când tranzacțiile nu își cunosc din start toate accesările și poate duce la blocarea inutilă a multor date[14][15]. În practică C2PL este rar utilizat.

Deadlock (interblocaj): Fiind un protocol *pessimist*, 2PL presupune uneori ca tranzacțiile să se blocheze așteptând lock-uri și, în consecință, pot apărea situații de *deadlock* – două sau mai multe tranzacții se așteaptă circular una pe cealaltă (ex: T1 a blocat obiectul X și cere Y, T2 a blocat Y și cere X). Protocolul 2PL nu previne deadlock-urile în forma de bază[6][16], așa că sistemele de baze de date distribuie implementate includ mecanisme separate de **gestionare a deadlock-urilor**.
Două strategii comune:

(a) *detectarea și recuperarea* – sistemul construiește un graf de așteptare al tranzacțiilor și detectează cicli de așteptare, apoi *înlătură* deadlock-ul prin abortarea uneia dintre tranzacții;

(b) *evitarea deadlock-ului* – folosirea unor discipline de așteptare precum **wound-wait** sau **wait-die**, unde tranzacțiile mai tinere sau mai vechi primesc prioritate și în caz de posibil deadlock una abandonează imediat, prevenind ciclul. De exemplu, algoritmul *wound-wait* (rănire-așteptare) permite tranzacțiilor mai vechi (cu timestamp mai mic) să preia lock-uri de la tranzacții mai tinere forțându-le abortul ("wound"), evitând astfel ca o tânără să blocheze una veche; reciproca *wait-die* face tranzacția tânără să moară (abort) dacă ar trebui să aștepte o mai veche. În sisteme distribuite, deadlock-urile pot fi globale (implicând resurse de pe noduri diferite), necesitând coordonare între managerii de blocări de pe fiecare nod sau un **manager de blocări distribuit** care să colecteze graful global[17][18].

Implementare și exemple:

Protocolul 2PL este folosit în mod tradițional de majoritatea SGBD-urilor relaționale pentru a implementa nivelul de izolare *Serializable*. De exemplu, PostgreSQL și MySQL/InnoDB folosesc intern o formă de 2PL strict combinată cu alte optimizări (InnoDB implementează *next-key locking* pentru a preveni *phantom reads*, combinând blocări de interval cu MVCC)[19][20]. Multe sisteme tranzacționale moderne preferă totuși tehnici multiversiune (discutate la secțiunea 6.3) pentru a spori paralelismul la citire, dar și acestea pot folosi 2PL pentru sincronizarea scrierilor. Un *studiu de caz important* este Google Spanner – o bază de date distribuită la scară globală (tip NewSQL): Spanner implementează controlul concurenței folosind **blocare în două faze strictă** pentru tranzacțiile cu scrieri, combinată cu un protocol de replicare (Paxos) și cu mecanismul **TrueTime** (ceasuri atomice sincronizate global) pentru a atribui fiecărei tranzacții un timestamp și a asigura *consistența externă* (serializabilitate strictă în ordine cronologică reală)[21][22]. Practic, într-o tranzacție Spanner, clientul obține întâi toate lock-urile de citire, citește datele, apoi obține lock-urile de scriere și face commit (acesta implicând un 2PC între replici) – până la commit toate lock-urile sunt deținute (Strict 2PL)[23]. TrueTime garantează că, după commit, sistemul așteaptă suficient timp înainte de a elibera timestamp-ul, astfel încât nicio altă tranzacție viitoare să nu poată avea un timestamp mai mic – eliminând anomaliile de ordine temporală[22]. Rezultatul este că Spanner oferă tranzacții distribuite **serializabile și liniarizabile**

(strict serializable) la nivel global. Alte sisteme *NewSQL* (CockroachDB, Yugabyte, etc.) s-au inspirat din Spanner, combinând 2PL sau variații de locking cu timestamp-uri hibride fizice/logice pentru ordine globală.

Avantaje și dezavantaje:

Metoda 2PL (în special varianta strictă) asigură nivelul maxim de izolare și consistență (serializabilitate), fiind relativ simplu de înțeles și de verificat formal. Dezavantajele țin de natura sa *pesimistă*: tranzacțiile pot fi blocate mult timp așteptând lock-uri, ceea ce duce la latențe crescute și utilizare ineficientă a resurselor dacă conflictul real este rar. De asemenea, deadlock-urile necesită mecanisme suplimentare de rezolvare. În medii cu **contention ridicat** (multe accesări concurente ale acelorași date), 2PL poate de fapt să fie eficient deoarece previne activitatea inutilă (nu lasă tranzacțiile să lucreze optimist doar pentru a fi apoi invalidate). Însă în medii cu **contention scăzut** (conflicte rare), strategia 2PL poate introduce *întârzieri inutile*, limitând gradul de paralelism mai mult decât ar fi necesar.

Exemplu practic 2PL:

Să considerăm o bază de date bancară distribuită unde două tranzacții actualizează soldurile aceluiași cont. Tranzacția T_1 (a lui Alice) citește soldul contului, apoi adaugă o sumă, iar tranzacția T_2 (a lui Bob) retrage o sumă. Sub 2PL strict, dacă Alice începe prima și obține un lock X pe acel cont, Bob va trebui să aștepte eliberarea acelui lock pentru a-și putea efectua operația de modificare. **Managerul de blocări** va bloca operația lui Bob până când Alice face commit și eliberează lock-urile. Astfel se evită o *actualizare pierdută* (lost update) – nu pot exista două scrieri concurente pe același cont. De asemenea, dacă Bob ar încerca să citească în timp ce Alice are un lock X (scriere necomisă), citirea lui ar fi fie blocată (dacă izolarea e serializabilă) sau ar vedea versiunea veche (dacă se folosește MVCC pentru citire, vezi secțiunea 6.3). Acest comportament este ilustrat și în figura de mai jos.

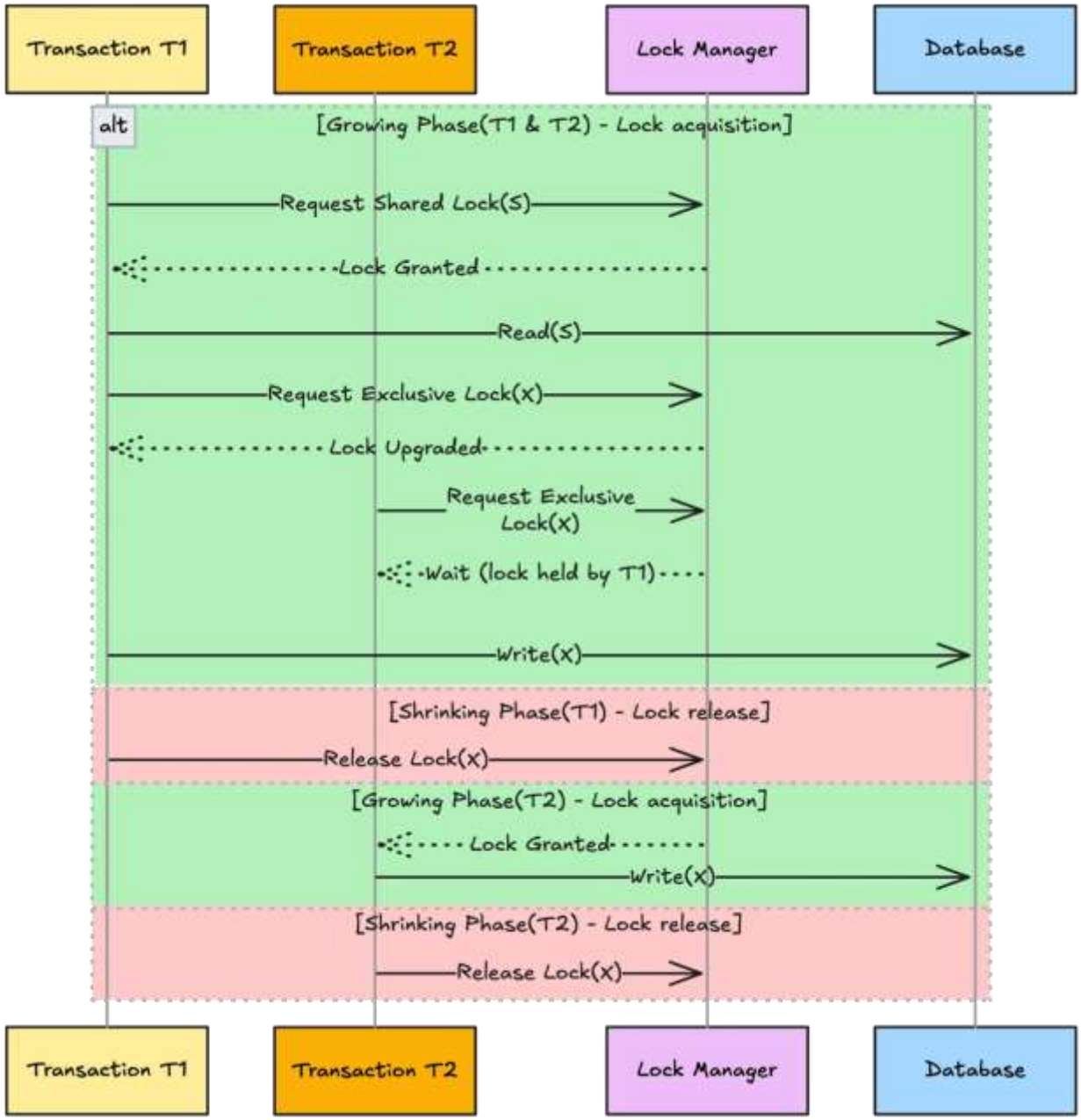

Figura 6.1: Diagramă conceptuală a protocolului 2PL într-un sistem de gestiune a blocărilor. Fiecare tranzacție parcurge o fază de growing (acumulare de lock-uri S sau X necesare) urmată de o fază de shrinking (eliberare a lock-urilor numai după ce a obținut toate resursele necesare). Sunt evidențiate două tranzacții concurente (T_1 și T_2) care accesează același obiect: T_1 obține un lock de citire (S) pe obiect, permițând citiri concurente, apoi îl upgradează la lock de scriere (X) pentru actualizare, blocând accesul altor tranzacții. T_2, care solicită un lock X pe același obiect, este pusă în așteptare de managerul de blocări până când T_1 își eliberează lock-urile la commit. Acest mecanism garantează serializabilitatea execuției tranzacțiilor.[24][25].

Studii de caz moderne (blocare distribuită):

În afara exemplului Spanner menționat, merită menționat că unele sisteme NoSQL adoptă *parțial* mecanisme de blocare în caz de conflict intens.

Apache **Cassandra**, de pildă, este un SGBD NoSQL distribuit care în mod obișnuit nu folosește tranzacții ACID multi-rând și nu aplică blocări în mod clasic (folosește consistență eventuală cu rezolvarea conflictelor pe baza ultimului timestamp al scrierii). Totuși, Cassandra oferă așa-numitele *Lightweight Transactions* (LWT) – o formă de tranzacții serializabile per rând, implementate printr-un protocol bazat pe Paxos (asimilat unui **optimistic compare-and-set**). În plus, pentru scrierile concurente obișnuite, Cassandra aplică un fel de control optimist: scrierile simultane sunt acceptate optimist și reconciliate pe baza timestamp-urilor (ultimul câștigă), iar în memorie modificările la același rând sunt aplicate atomic. Dacă însă se detectează **contention ridicat** pe un rând (multe scrieri concurente care se invalidează reciproc), Cassandra **schimbă strategia la una pesimistă**: folosește *blocări pe rând* pentru a serializa accesul și a evita numeroase abortări[26].

Astfel, Cassandra îmbină abordarea optimistă cu blocarea pesimistă la nevoie, fiind un exemplu de sistem hibrid modern. Multe alte sisteme NoSQL renunță la tranzacțiile complexe tocmai pentru a evita costurile blocărilor distribuite – ele aleg eventual *să nu ofere* izolare serializabilă la nivel multi-obiect, lăsând dezvoltatorului responsabilitatea de a trata anomaliile la nivel aplicativ sau oferind doar tranzacții restrânse (ex: tranzacții intr-o singură partiție).

6.2 Ordonarea pe bază de marcaj temporal (Timestamp Ordering – TO)

O alternativă la utilizarea blocajelor pentru asigurarea serializabilității este **ordonarea pe bază de timp** a tranzacțiilor. Algoritmul de *Timestamp Ordering (TO)* atribuie fiecărei tranzacții un **timestamp unic** (un identificator monoton crescător, ce poate fi generat pe baza unui ceas fizic, a unui ceas logic Lamport sau a unui counter global) și impune ca execuția tranzacțiilor să respecte ordinea acestor timestamp-uri[27]. Cu alte cuvinte, dacă tranzacția T_1 are timestamp-ul mai mic (mai vechi) decât tranzacția T_2, atunci sistemul de gestiune a tranzacțiilor trebuie să se asigure că efectul final este echivalent cu executarea lui T_1 înainte de T_2 (într-un schedule serial tranzacțiile apar în ordinea crescătoare a timestamp-urilor lor)[27].

Ideea de bază: În loc să prevină conflictele prin blocare, protocolul de ordonare temporală le **lasă să se întâmple** liber, dar *detectează și rezolvă* orice încălcare a ordinii de timp. Astfel, *TO este o metodă optimistă prin natură*, deși adesea este discutată separat de OCC. Fiecare element de date din baza de date poartă meta-informație cu privire la **timpul ultimei citiri** și **timpul ultimei scrieri** care s-au efectuat asupra sa (de exemplu, două marcaje: RTS(X) – *Read Timestamp* pentru obiectul X și WTS(X) – *Write Timestamp* pentru X)[28][29]. Când o tranzacție *T* (cu timestamp asociat *TS(T)*) accesează un obiect, regulile protocolului determină dacă accesul este permis sau nu, comparând timestamp-ul tranzacției cu cele ale obiectului:

a) **Reguli pentru operația de Citire** a unui obiect *X* de către tranzacția *T* cu timestamp = TS(T):
b) Dacă TS(T) < WTS(X), înseamnă că *X* a fost deja scris de o tranzacție mai *nouă* (cu timestamp mai mare) înainte ca *T* să apuce să citească. Accesul ar însemna ca *T* (mai veche) să vadă un *rezultat din viitor*, ceea ce ar încălca serializabilitatea. Prin urmare, **citirea nu**

este permisă: tranzacția *T* va fi *abortată* și eventual replanificată cu un nou timestamp[28][30].

c) Dacă TS(T) ≥ WTS(X), atunci tranzacția *T* este cea mai nouă (sau contemporană) entitate care accesează *X*. Se permite citirea, iar meta-datul RTS(X) (timestamp-ul ultimei citiri) se actualizează la max(RTS(X), TS(T))[30]. Practic, *T* vede versiunea curentă a lui *X* (ultimul WTS compatibil cu timpul său). **Notă:** Pentru a preveni lecturi murdare, de obicei dacă X este în curs de a fi modificat de o tranzacție necomisă, *T* fie va aștepta commit-ul acelei tranzacții, fie (în implementări simple) se comportă ca și cum ar fi abortată – dar în protocolul teoretic TO se presupune că scrierile devin vizibile doar la commit.

d) **Reguli pentru operația de Scriere** (update) pe un obiect *X* de către tranzacția *T*:

e) Dacă TS(T) < RTS(X), înseamnă că *T* (care este mai veche) încearcă să scrie un obiect *X* ce a fost **citit** deja de o tranzacție mai nouă *după* momentul în care *T* a început. Acceptarea acestei scrieri ar viola repeatable-read pentru tranzacția mai nouă (aceasta a citit o valoare care n-ar fi trebuit să existe dacă *T* scrie acum ceva mai vechi). Deci condiția impune **abortarea** tranzacției *T* dacă TS(T) este mai mic decât *oricare* citire ulterioară a lui *X*[31].

f) Dacă TS(T) < WTS(X), înseamnă că *X* a fost deja actualizat de o tranzacție mai nouă decât *T*. În cazul de bază, acest conflict se tratează tot prin **abortarea** lui *T* – tranzacția este prea veche și încearcă să suprascrie o valoare "viitoare" față de sine[31]. (Există și o variație a protocolului, *Thomas's Write Rule*, care spune că dacă noua valoare oricum nu ar modifica nimic – de exemplu, *T* scrie același lucru care a fost deja scris de altcineva – atunci scrierea poate fi **ignorată** fără abort, marcând totuși *T* ca realizată. Această optimizare evită aborturi inutile atunci când ordinea scrierilor nu contează pentru rezultat[32]).

g) Dacă TS(T) > WTS(X) **și** TS(T) ≥ RTS(X), atunci scrierea este sigură: tranzacția *T* este cea mai nouă atât în raport cu ultima scriere, cât și cu ultimele citiri ale lui X. *T* poate scrie obiectul, iar WTS(X) se actualizează la TS(T)[31]. Practic, *T* devine cea mai recentă tranzacție care a modificat *X*.

Dacă oricare dintre condițiile de abort se întrunește, tranzacția este anulată și repornită cu un nou timestamp (de obicei actualizat la momentul reînceperii). În acest mod, protocolul de ordonare temporală *refuză* orice operație care ar conduce la un conflict de ordine. De exemplu, dacă o tranzacție mai veche încearcă să citească un element deja modificat de una mai nouă, i se refuză citirea (pentru a nu vedea date "din viitor" în ordinea serială)[30]; similar, o tranzacție mai veche care vrea să suprascrie un element deja citit de alta mai nouă va fi oprită, deoarece în serializarea după timp, tranzacția nouă trebuia să vină după cea veche.

Proprietăți: Protocolul TO *evită deadlock-urile*, deoarece tranzacțiile nu se blochează niciodată așteptând resurse – ele fie își fac operațiile, fie sunt abortate imediat dacă ar viola ordinea de timp. Practic, tranzacțiile mai "vechi" pot fi prevenite să influențeze tranzacțiile "tinere" în mod nepermis: o tranzacție care întârzie prea mult riscă să fie abortată de altele mai noi. Astfel, **starvation** (foametea) este posibilă: o tranzacție cu timestamp foarte mic (veche) ar putea tot fi abortată de tranzacții noi care apar mereu și o prind din urmă, împiedicând-o să termine – mai ales dacă e lungă, șansele să fie lovită de altele cresc[33][34]. O altă problemă potențială este sincronizarea ceasurilor în sistem distribuit: dacă un nod generează un timestamp mult mai mare decât al celorlalți (de ex. ceasul său e înainte), tranzacțiile sale vor fi considerate "din viitor" și pot forța abortul multor alte tranzacții pe celelalte noduri[35]. Implementarea TO necesită deci fie ceasuri logice (Lamport) sau un mecanism de clock sync pentru corectitudine în medii distribuite.

Implementări practice și variații:

Protocolul de ordonare simplu, așa cum e descris mai sus, suferă de problema numeroaselor abortări, în special pentru tranzacții lungi (care acumulează timestamp-uri mici și pot fi invalidate de oricine mai nou)[33]. Ca atare, abordarea pură TO *nu este folosită frecvent ca atare* în SGBD comerciale. În schimb, a dat naștere la tehnici **multiversiune**: *Multiversion Timestamp Ordering (MVTO)* stochează versiuni multiple ale obiectelor astfel încât tranzacțiile mai vechi să poată citi **versiunea istorică** corespunzătoare timpului lor, în loc să fie abortate[36][34]. De exemplu, dacă tranzacția T_1 (TS=100) vrea să citească un obiect pe care tranzacția T_2 (TS=120) l-a modificat deja, sub MVTO s-ar păstra versiunea veche a obiectului (de dinainte de TS=120) astfel încât T_1 să poată continua fără abort, citind versiunea istorică. Scrierile creează versiuni noi cu timestamp-ul tranzacției care scrie.

Această abordare este baza mecanismului **MVCC (Multi-Version Concurrency Control)** folosit pe scară largă astăzi: cititorii și scriitorii nu se blochează reciproc deoarece cititorii pot vedea snapshot-ul consistent al datelor de la momentul începerii lor, iar scriitorii lucrează pe versiuni noi. MVCC este folosit de exemplu în Oracle, PostgreSQL, Microsoft SQL Server (pentru Snapshot Isolation), etc., deoarece *citirile nu blochează scrieri și scrierile nu blochează citiri*, îmbunătățind mult performanța în medii cu multe citiri[37]. Totuși, MVCC necesită și el un criteriu de validare pentru scrieri: de exemplu, în **Snapshot Isolation** (o variație populară a MVCC), se permite ca două tranzacții să scrie concurent obiecte diferite, dar dacă încearcă să scrie același obiect, a doua care face commit va eșua (similar cu abort în TO) – se asigură astfel o formă de serializabilitate parțială.

Există și algoritmi precum **Serializable Snapshot Isolation (SSI)** care detectează anomalii de tip *write skew* sub MVCC și fac rollback la nevoie, combinând beneficiile MVCC cu serializabilitatea completă.

Exemplu practic:

Să luăm un scenariu simplu cu două tranzacții distribuite, T_1 și T_2, care accesează aceleași resurse. T_1 are TS=10, T_2 are TS=20 (T_1 e considerată mai veche). Dacă T_1 citește un element X pe care apoi T_2 îl modifică, atunci la momentul scrierii lui X de către T_2 se va avea $TS(T_2)=20 >$ RTS(X) (RTS era 10 după citirea lui T_1) și > WTS(X), deci scrierea T_2 se poate realiza. Ulterior, dacă T_1 vrea să scrie același element X, va constata $TS(T_1)=10 <$ WTS(X)=20 (pentru că T_2 a scris deja) și atunci **T_1 va fi abortată** de protocol (nu i se permite să suprascrie modificarea "viitoare" făcută de T_2)[31]. Astfel se păstrează ordinea: tranzacția cu TS=20 (T_2) a efectuat scrierea finală pe X, deci în orice serializare echivalentă, T_1 (TS=10) trebuie să fie înainte, ceea ce nu mai e posibil deoarece ea ar fi trebuit să scrie și ea X. Abortând T_1, se menține consistența (practic, în execuția efectivă doar T_2 își duce efectele la capăt, echivalent cu ordinea T_1 înainte, dar T_1 neavând efect util – a fost ca și cum nu ar fi existat sau ar fi eșuat înainte de a produce schimbări).

Avantaje și dezavantaje (TO):

Avantajul major al timestamp ordering este *simplitatea conceptuală* și lipsa blocărilor: tranzacțiile nu se așteaptă unele pe altele, deci nu apare deadlock. În plus, citirile pot fi efectuate fără întârziere (dacă există versiuni, chiar complet concurent cu scrierile) – în implementările multiversiune, cititorii sunt neblocați complet[37].

Dezavantajul principal al versiunii cu o singură versiune (basic TO) este *rata mare de aborturi* când există conflicte, ceea ce poate duce la irosirea muncii tranzacțiilor lungi. Tranzacțiile lungi sau care pornesc cu timestamp mic sunt vulnerabile la *starvation*, fiind posibil să tot fie reluate dacă sistemul este ocupat de multe tranzacții mai noi. Pentru a ameliora acest lucru,

sistemele pot implementa politici de **prioritate** la reloare (de ex., dacă o tranzacție a tot fost abortată de k ori, la repornire i se poate acorda un timestamp mai mare artificial sau se poate pune un *delay* în startul altora, pentru a-i da șansa să termine).

Un alt neajuns în medii distribuite este necesitatea unui acord global asupra timestamp-urilor – utilizarea unui *clock global* (fizic sau logic). Sistemul **Google Spanner** combină abordarea TO cu 2PL: folosește ceasul global TrueTime. Practic, Spanner atribuie fiecărei tranzacții un timestamp de commit pe baza ceasurilor atomice, dar **controlul concurenței efectiv** tot cu 2PL îl face; timestamp-ul global este folosit abia la commit ca să decidă ordinea finală a tranzacțiilor și să introducă eventual un *wait* (așteptare) până ce incertitudinea ceasului dispare, garantând că nu există o altă tranzacție cu timestamp mai mic încă neaplicată[38][39].

Alte sisteme (ex. **DynamoDB** sau stocarea din **Cassandra**) folosesc timestamp-uri logice pentru *rezolvarea* scrierilor în conflict: de exemplu Cassandra, în modul eventual consistent, atașează fiecărei scrieri un timestamp și aplică politica "**last write wins**" – ultima scriere (cu timestamp maxim) pe un item prevalează[26]. Aceasta însă *nu garantează serializabilitatea* (deoarece nu există rollback, iar scrierile suprascrise sunt pierdute logic), dar este o folosire a timestamp-urilor pentru concilierea replicărilor concurrente, preferată în medii unde consistența strictă nu e obligatorie.

În rezumat, ordonarea după marcaje temporale este elegantă teoretic și stă la baza multor optimizări (precum MVCC), însă rareori se aplică ca unic mecanism în formă simplă, din cauza costului ridicat al aborturilor în scenarii reale cu conflict.

6.3 Controlul concurenței optimist (Optimistic Concurrency Control – OCC)

Concept:

Controlul concurenței *optimist* pornește de la premisa că **ciocnirile între tranzacții sunt rare**, deci e avantajos să lăsăm tranzacțiile să se execute *concomitent fără blocaje*, urmând ca abia la final să verificăm dacă au intrat în conflict. Spre deosebire de metodele pesimiste (care folosesc blocări înainte de acces pentru a preveni conflicte) și de metoda timestamp *basic* (care verifică la fiecare operație cu abort imediat), OCC permite tranzacțiilor să ruleze complet în paralel, *logând* local modificările, și efectuează o **validare** înainte de commit. Dacă validarea arată că tranzacția nu a încălcat izolația, atunci schimbările sale sunt aplicate; dacă a intrat în conflict cu altă tranzacție, atunci este *abortată* (și eventual reluată).

Faze OCC:

În mod clasic (conform algoritmului propus de Kung & Robinson, 1981), execuția unei tranzacții în OCC este împărțită în **trei faze** distincte[40][41]:

1. **Faza de citire (și execuție locală)** – *Read Phase*. Tranzacția își începe execuția, **citește** liber datele necesare din baza de date (fără a bloca elementele) și stochează copiile acestor date local (de ex., în spațiul de lucru al tranzacției). Toate modificările (operațiile de write/update) sunt efectuate *doar asupra copiilor locale*, **fără a scrie nimic încă în baza de date globală**[42][43]. Practic, tranzacția lucrează "în sandbox" optimist, acumulând un *Write Set* (setul de modificări pe care intenționează să le facă) și un *Read Set* (setul obiectelor citite). Această fază corespunde logic cu executarea sub izolare maximă, dar fără sincronizare efectivă – tranzacțiile pot citi date care eventual vor fi modificate de altele, însă nu-și dau seama în această etapă.

2. **Faza de validare** – *Validation Phase*. Când tranzacția termină procesarea locală și dorește să facă **commit**, trece în faza de validare. În acest moment, sistemul verifică dacă tranzacția se poate **serializa** față de celelalte tranzacții care au coexistat cu ea. Există mai multe metode de validare; una clasică este următoarea: se atribuie tranzacției un timestamp (sau un număr de ordine) TS_validare la începutul fazei de validare (care va juca rol de timestamp al tranzacției în istoria serializată). Apoi, pentru *fiecare tranzacție activă concurentă* se verifică dacă mulțimea de citiri și scrieri se suprapune conflictual. Dacă o altă tranzacție *T_j* a scris un element pe care *T_i* (tranzacția noastră) l-a citit **și** T_j a terminat (a intrat în commit) înainte ca *T_i* să înceapă commit-ul, atunci există un conflict: *T_i* a citit date care au fost modificate deja de *T_j* înainte de commit-ul lui *T_i* (deci citirea lui *T_i* nu mai reflectă starea finală – anomalia *non-repeatable read*). Similar, dacă *T_i* și *T_j* au ambele scrieri pe același obiect, ordinea trebuie să respecte timestamp-urile lor. Formal, o condiție suficientă de validare (unul din criteriile din literatură) este: pentru două tranzacții *T_i* și *T_j* cu TS(T_i) < TS(T_j) (să zicem *T_i* mai veche ca timp de validare):
    a. *T_i* nu trebuie să scrie peste ceva citit de *T_j* înainte ca *T_j* să termine. (Dacă *T_j* a citit un obiect X pe care *T_i* îl scrie, și *T_i* încă nu era comisă când *T_j* a citit, atunci la commit-ul lui *T_i* se strică izolarea lui *T_j* – conflict read-write.)
    b. *T_j* nu trebuie să scrie peste ceva citit de *T_i* înainte ca *T_i* să termine. (Asta ar însemna *T_i* a citit o versiune veche, iar *T_j* a scris între timp ceva nou pe acel obiect – deci *T_i* nu a văzut modificarea, conflict write-read din perspectiva lui *T_i*.)
    c. Dacă ambele scriu același obiect, atunci ordinea scrierilor trebuie să urmeze ordinea timestamp-urilor, altfel e conflict write-write.

    Dacă oricare conflict este detectat, **validarea eșuează** și tranzacția *T_i* (sau *T_j*, depinde de politică) este abortată[44][45]. În multe implementări se alege să se *aborteze tranzacția care intră în faza de validare mai târziu* (de obicei cea cu timestamp mai mare), considerând că cea aflată deja în commit își poate vedea de treabă, iar cealaltă se reia.
3. **Faza de scriere (commit)** – *Write Phase*. Dacă tranzacția trece de validare (adică nu s-au găsit conflicte care să încalce serializabilitatea), atunci modificările sale locale sunt **aplicate global**: se realizează efectiv operațiile de write în baza de date, persistând noile valori, după care tranzacția se consideră *commit*[41][46]. În caz contrar (validare eșuată), tranzacția este *rollback*-ată (toate modificările locale sunt aruncate) și, de obicei, se reia de la zero (opțional după un scurt timp aleator pentru a evita repetarea imediată a conflictului).

Un aspect important: în multe sisteme, *momentul exact al trecerii în faza de validare* definește și ordinea de serializare a tranzacțiilor OCC. Adesea, timestamp-ul atribuit tranzacției este timpul de început al validării (nu începutul execuției). Astfel se reduce șansa de conflict pentru tranzacții lungi: chiar dacă *T_i* a rulat mult timp, abia la commit i se stabilește poziția finală în ordine. Dacă între timp unele tranzacții mai mici au terminat, ele pot fi considerate "înainte" de *T_i*, ceea ce e acceptabil[47]. Această decizie (TS = moment validare) minimizează abortul tranzacțiilor lungi din cauza apariției multor tranzacții scurte după startul lor, crescând șansa ca *T_i* să valideze cu succes.

Proprietăți OCC:

Pentru că nu folosesc blocări în faza de lucru, tranzacțiile OCC nu se blochează reciproc – deci **fără deadlock** (nu există așteptare circulară, eventual două tranzacții pot ajunge ambele la commit și atunci una va fi abortată, dar nu rămân blocate)[48][49]. De asemenea, neaplicând modificări până la commit, OCC evită *cascade aborts* – nicio altă tranzacție nu poate citi modificări necomise (pentru că acestea nici nu sunt vizibile altora), deci dacă o tranzacție e abortată nu afectează consistența lecturilor altora[50][51]. Pe de altă parte, **riscul OCC** este că munca făcută de o tranzacție în faza de citire/execuție poate fi irosită dacă la final conflictul este detectat. Dacă încărcarea de sistem este mare și conflictul devine *frecvent*, OCC poate duce la **rulare în gol**: multe tranzacții care tot se invalidează reciproc, în special cele care accesează obiecte comune intens. În astfel de cazuri, protocoalele pesimiste (blocările) pot de fapt obține throughput mai bun, pentru că *una* dintre tranzacții ar fi așteptat, în loc ca ambele să ruleze și apoi una să fie aruncată.

Politici de validare:

Diferite implementări OCC folosesc criterii de validare diferite (mai laxe sau mai stricte) pentru a decide când două tranzacții sunt în conflict. Un criteriu comun (numit *forward validation*) pentru două tranzacții *T_i* și *T_j* este: - dacă intervalele lor de execuție se suprapun (T_i și T_j au fost parțial concomitente), atunci fie ordinea operațiilor de citire/scriere trebuie să fi fost compatibilă. Mai concret: dacă *T_i* a *citit* un element pe care *T_j* l-a *scris*, iar *T_j* a terminat commit înainte ca *T_i* să fi terminat citirea, atunci *T_i* este invalidă. Invers, dacă *T_i* a scris ceva ce *T_j* a citit înainte de commit-ul lui *T_i*, atunci *T_j* este invalidă. - Dacă ambele scriu același element, conflictul e direct – una va fi invalidată. De regulă se invalidează tranzacția cu timestamp mai mare (mai nouă) și se acordă prioritate celei ajunse prima la commit.

Aceste reguli pot fi implementate eficient ținând evidența pentru fiecare tranzacție a *ReadSet* și *WriteSet*, plus a timestamp-urilor de finalizare (sau început validare). Sistemul parcurge tranzacțiile care s-au comis între momentul începerii lui *T_i* și momentul validării sale și verifică dacă vreuna a modificat un element pe care *T_i* l-a citit – dacă da, *T_i* eșuează validarea[52][53]. De asemenea, se verifică tranzacțiile încă active. Există și variantă de *validare în spate (backward validation)* unde o tranzacție ce vrea să se comită este verificată împotriva tuturor tranzacțiilor *validate deja* (comise înainte) cu care s-a suprapus, pentru a vedea dacă ar fi trebuit să fie după ele – diferite politici schimbă performanța și gradul de paralelism acceptat.

Exemple de sisteme și utilizări OCC:

Multe sisteme moderne, în special cele optimizate pentru procesare in-memory sau distribuții de date, adoptă controlul concurenței optimist.

Spre exemplu, **Oracle Berkeley DB** oferă atât mod pesimist cât și optimist. **Microsoft Hekaton** (motorul in-memory din SQL Server) folosește în principal OCC cu versiuni: tranzacțiile lucrează pe versiuni locale și la commit verifică dacă nu au apărut conflicte. **Firebase Cloud Firestore** (bază de date NoSQL pentru aplicații mobile) folosește OCC la nivel de document: când două actualizări concurente ajung, una poate eșua dacă versiunea documentului a fost modificată între timp – dezvoltatorii trebuie să reîncerce. Un exemplu notabil este protocolul **Calvin** (un sistem de procesare determinată a tranzacțiilor): acesta pre-planifică tranzacțiile într-o ordine și astfel evită blocările, efectuând în esență o validare determinată (Calvin nu este pur OCC dar împarte idealul de a nu bloca tranzacțiile la runtime).

În zona bazelor de date distribuite, **Cassandra** (discutată anterior) implementează scrierile obișnuite în mod optimist: la nivel de partiție, scrierile concurente încearcă să se realizeze și doar

dacă se calcă pe picioare în mod repetat, sistemul va serializa (prin lock) pentru a evita abort-uri continue[26]. Alte sisteme NoSQL cu replicare multi-master (ex: CouchDB) folosesc de asemenea un stil optimist: fiecare nod sau client face modificări locale și la replicare se *detectează conflicte* (de exemplu, două editări offline ale aceluiași document) – rezolvarea conflictelor se face prin versiuni (ex. CouchDB atribuie fiecărei revizii un identificator de versiune și dacă la sincronizare există două capete diferite, le păstrează pe ambele și lasă aplicația să aleagă). Acesta e tot un fel de OCC la nivel aplicație: nu s-au folosit lock-uri când s-au făcut modificările, s-a **acceptat conflictul** și s-a lăsat pe seama unei faze ulterioare reconcilierea.

Avantaje și dezavantaje OCC:

Principalul avantaj este *maximizarea paralelismului* în medii unde conflictele sunt într-adevăr rare: tranzacțiile nu așteaptă niciodată după altele, toată lumea rulează în paralel cât de mult posibil, ceea ce în scenarii cu workload dispersat duce la throughput superior metodei cu blocări[54][55]. De asemenea, lipsa blocărilor elimină complet problemele de deadlock și reduce supraîncărcarea de context switching asociată cu fire blocate. În plus, integritatea este mai ușor de asigurat: fiindcă nimic nu e scris până la commit, nu există riscul ca o tranzacție abortată să fi lăsat date murdare vizibile – izolarea este implicită până la commit. Pe de altă parte, dacă **conflictul este frecvent**, OCC poate duce la scăderea dramatică a performanței: multe tranzacții vor fi refăcute de mai multe ori până reușesc, irosind CPU și lățime de bandă I/O. În cazuri extreme (ex: update simultan pe același obiect foarte des), OCC degradează spre un throughput aproape zero, deoarece tranzacțiile se invalidează reciproc repetat – în astfel de situații un lock simplu ar fi permis măcar uneia să treacă imediat, restul așteptând puțin. De asemenea, OCC nu oferă un mecanism în sine de prioritate: tranzacții scurte pot repetat să-l "bată" pe un tranzacție lungă care tot e restartată la final din cauza lor, ducând la **starvation** pentru tranzacțiile lungi[56][57]. În practică se implementează soluții (contorizarea numărului de restarturi și eventual ridicarea priorității sau abortarea preferențială a tranzacțiilor care au rulat mai puțin etc.).

Studiu de caz modern (OCC): Un exemplu clar al eficienței OCC este în sisteme de tip **low-contention** cum ar fi gestiunea coșului de cumpărături într-un magazin online: majoritatea tranzacțiilor utilizatorilor afectează obiecte diferite (produsele din coșuri diferite), deci pot rula în paralel fără probleme. OCC ar permite tuturor adăugărilor în coș să aibă loc fără vreun lock global; doar dacă doi utilizatori modifică exact același coș în același timp s-ar face o validare care poate duce la restart (ceea ce e rar). Pe de altă parte, într-un sistem de **trading bursier** cu volum mare pe aceleași acțiuni, OCC ar suferi – mii de tranzacții ar încerca să modifice prețul aceleiași acțiuni concomitent, știind una de alta abia la commit, cauzând numeroase retry. De aceea, multe sisteme folosesc adaptiv: mod optimist pentru majoritatea operațiilor, dar când se identifică *hot-spot*-uri de conflict, se poate trece la măsuri pesimiste (blocaje locale) – exact cum face Cassandra cu lock-urile pe rând sub conflict intens[58].

Un alt exemplu întâlnit este *versiunea web* a controlului optimist: pentru aplicațiile web, menținerea unui lock de editare e impracticabilă (utilizatorul poate deschide un formular și abandona, ținând lock-ul inutil). În schimb, se folosește OCC: de exemplu, la editarea unei pagini Wikipedia, sistemul nu blochează pagina, dar când două persoane salvează modificări concurente, al doilea care salvează primește un mesaj de **edit conflict** – echivalentul unei tranzacții care a picat la validare, cerând utilizatorului să îmbine manual modificările. Acest model seamănă cu OCC, unde commit-ul este condiționat de lipsa intervenției altcuiva în interval[59][60].

Bibliografie

[1] [2] [3] [6] [7] [14] [15] [16] Two-phase locking - Wikipedia
https://en.wikipedia.org/wiki/Two-phase_locking
[4] [PDF] 13.Realizarea serializabilitatii prin blocari
http://inf.ucv.ro/documents/danciulescu/BD-fizica-informatica/BD13.pdf
[5] [27] [28] [29] [30] [31] [33] [37] [40] [41] [44] [45] [46] [47] [52] [53] Learn about Database Concurrency Control Methods | Medium
https://alibaba-cloud.medium.com/a-deep-dive-into-database-concurrency-control-bc90d19a587a
[8] [9] [10] [11] [17] [18] [54] [55] [61] [62] [65] [66] Concurrency Control Mechanisms in Distributed Systems | by Roopa Kushtagi | Medium
https://medium.com/@roopa.kushtagi/concurrency-control-mechanisms-in-distributed-systems-4c7e510b2427
[12] [13] [25] [63] How does the 2PL (Two-Phase Locking) algorithm work - Vlad Mihalcea
https://vladmihalcea.com/2pl-two-phase-locking/
[19] [20] [24] Database Transactions Deep Dive
https://thuva4.com/blog/transactions/
[21] [22] Title
https://www.eecg.utoronto.ca/~ashvin/courses/ece1724/2024f/lectures/5-spanner.pdf
[23] [38] [39] Xusheng Chen's Homepage | On the Usage of Atomic Clocks in Spanner
https://michaelxschen.github.io/blog/2022/spanner/
[26] [58] Cassandra - Database of Databases
https://dbdb.io/db/cassandra
[32] Thomas Write Rule in DBMS - GeeksforGeeks
https://www.geeksforgeeks.org/dbms/thomas-write-rule-in-dbms/
[34] [35] [36] Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems
http://muratbuffalo.blogspot.com/2022/11/timestamp-based-algorithms-for.html
[42] [43] Validation Based Protocol in DBMS - GeeksforGeeks
https://www.geeksforgeeks.org/dbms/validation-based-protocol-in-dbms/
[48] [49] [59] [60] Optimistic concurrency control - Wikipedia
https://en.wikipedia.org/wiki/Optimistic_concurrency_control
[50] [51] [56] [57] Validation Based Protocol in DBMS. | by Laxmi priya Asam | Apr, 2025 | Medium
https://medium.com/@asamlaxmipriya/validation-based-protocol-in-dbms-e156d7abbbcd
[64] Principles of Distributed Database System 2020 | PDF | Databases | Computer Architecture
https://www.scribd.com/document/753190192/principles-of-distributed-database-system-2020