
Przetwarzanie transakcyjne

Trasakcja, to podstawowa "jednostka pracy" w systmie zarządzania bazą danych. Rolą SZBD jest efektywne, poprawne i bezpieczne wykonanie transakcji. Traksakcja może składać się z wielu operacji elementarnych, przy czym interesują nas jedynie operacje odwołujące się do bazy danych.

ACID

Podstawowe cechy transakcji, które powinien zapewnić SZBD zostały ujęte w popularny skrót ACID, którego poszczególne litery rozkodowuje się następująco:

atomowość (A – atomicity) — transakcja musi zostać wykonana w całości lub nie wykonana wcale;

spójność (C – consistency) — transakcja uruchomiona na spójnym stanie bazy danych powinna przekształcać bazę w stan spójny;

niezależność (I – independence) — transakcja, pomimo że wykonywana na bazie danych wraz z innymi transakcjami, powinna mieć stworzone takie warunki, jakby była wykonywana samodzielnie;

trwałość (D – durability) — wyniki transakacji zakończonej (wypełnionej), powinny być na trwale zapisane w bazie danych.

ACID jest często traktowany, jako wyznacznik zasad przekształcania trasakcyjnego. W różnych SZBD cechy ACID są zaimplementowane w różnym stopniu. Pominięcie niektórych z nich może wynikać z:

- trudności w implementacji (proste SZBD)
- celowej rezygnacji np. na rzecz szybkości działania (SZBD dedykowane np. do obsługi gier czy portali społecznościowych).

Za zapewnienie ACID w większości odpowiada SZBD, choć częściowo może to być zadaniem użytkownika (aplikacji). Za poszczególne cechy ACID odpowiadają różne elementy SZBD, np. za atomowość i trwałość odpowiada system odtwarzania i zapisywania danych. Za spójność odpowiada użytkownik (definiując odpowiednie więzy integralności) oraz system gwarantując zachowanie tych więzów. Za niezależność zasadniczo odpowiada SZBD. Techniki kontrolowania i zapewniania **niezależności** są tematem tej notatki.

Przykłady transakcji

Potrzebę przetwarzania transakcyjnego ilustrują poniższe przykłady.

Przykład 1. Rozważmy konta bankowe zapisane w bazie danych. Przelew wykonany przez klienta polega na pobraniu pieniędzy z jednego konta i wpłaceniu ich na inne konto. Gdyby ten ciąg operacji został przerwany, to pieniędze zniknęłyby z banku. Przelew musi więc być transakcją, które zawsze jest wykonywana w całości (lub wcale).

Przykład 2. Klient A biura podróży ogląda ofertę wycieczek. Widzi, że jest jeszcze jedno miejsce na atrakcyjny wyjazd do Kenii. Rezerwuje to miejsce i przelewa pieniądze za wycieczkę. Zanim A zarezerwował wycieczkę, zobaczył ją także klient B. Jemu także się spodobała, więc on także rezerwuje ją i płaci. Jest to sytuacja niepoprawna, która wyniknęła stąd, że każdy z klientów "pobrał"dane z bazy i oglądał swoją kopię. Każdy z nich podjął decyzję i przesłał ją do bazy, ale jeden z nich podejmował decyzję na podstawie danych, które nie były aktualne.

Przykład 3. Program Dziekanat ma opcję wyliczania średniej ocen i wyznaczania 10% najlepszych studentów, którym należy się stypendium. Program został uruchomiony w chwili t_0. Gdy już przeczytał połowę bazy, powiedzmy w chwili $t_1 > t_0$, ktoś zmienił już przeczytaną przez program ocenę A (powiedzmy z a_0 na a_1) i jeszcze nie przeczytaną ocenę B (z b_0 na b_1). Program wyliczający średnią kończy działanie w chwili $t_2 > t_1$. To, co obliczył, nie jest średnią z żadnego momentu czasowego — do obliczeń wzięte są wartości a_0 oraz b_1, co nie odpowiada ani stanowi sprzed t_1, ani stanowi po t_1.

1 Problem szeregowalności

1.1 Podstawowe pojęcia

Operacja elementarna — z punktu widzenia baz danych liczą się jedynie operacje odwołujące się do danych przechowywanych w bazie. Operacja elementarna może polegać na pobraniu (odczytaniu) danych, zmodyfikowaniu wartości danych (całkowity dostęp do danych) lub zapisaniu nowej wartości (zapis). W niektórych przypadkach warto wyróżnić dodatkowo rodziny operacji przemiennych, jak np. inkrementację.

Transakcja — ciąg elementarnych operacji odwołujących się do bazy danych zadanych w porządku chronologicznym. Ciąg ten musi zostać wykonany w zadanej kolejności, w całości albo wcale.

Jednostka bazy danych — dane z bazy udostępniane w całości w celu wykonania elementarnej operacji bazodanowej. Możemy wyróżniać w bazie małe jednostki (np.

rekordy) lub duże (np. tabele). Jednostki mogą być niezależne (rekordy między sobą), powiązane z punktu widzenia dostępu (jak węzły B-drzewa) lub zależne hierarchicznie (jak rekordy w stosunku do tabeli, do której należą, jeśli jednostką blokowania może być i rekord, i tabela).

Blokada — transakcja może zablokować określoną jednostkę w celu wykonania na niej operacji elementarnej i zapobieżeniu dostępu przez inne transakcje do jednostki. Blokadami zarządza zazwyczaj wydzielony moduł SZBD, do którego transakcje zwracają się o przyznanie blokady.[1] W SZBD mogą być stosowane różne rodzaje blokad, np. do zapisu, do odczytu, całkowita,...

Harmonogram — dla transakcji T_1, T_2,..., T_k złożonych z operacji elementarnych $T_i = (O_1^i, \ldots, O_{r_i}^i)$ dla $1 \leq i \leq k$, harmonogram to ciąg operacji $H = (H_1, H_2, \ldots, H_\ell)$, złożony z operacji elementarnych $\{O_j^i \mid 1 \leq i \leq k \text{ oraz } 1 \leq j \leq r_i\}$, przy czym operacje elementarne każdej transakcji T_i występują w H w takim samym porządku jak w T_i.

Harmonogram sekwencyjny — dla transakcji T_1, T_2,..., T_k to harmonogram postaci:

$$T_{\pi(1)}, T_{\pi(2)}, \ldots, T_{\pi(k)},$$

gdzie π jest dowolną permutacją zbioru $\{1, 2, \ldots, k\}$. Innymi słowy jest to wykonanie transakcji T_1, T_2,..., T_k w dowolnym porządku, przy czym każda transakcja jest wykonywana w całości, bez przeplatania z operacjami innych transakcji.
Przyjmuje się, że wykonanie sekwencyjne oznacza niezależne wykonanie transakcji współbieżnych, bo odpowiada pewnemu wykonaniu, w którym transakcje są obsługiwane pojedynczo, a nie współbieżnie.

Harmonogram szeregowalny — harmonogram transakcji T_1, T_2,..., T_k , którego działanie jest równoważne pewnemu harmonogramowi sekwencyjnemu $T_{\pi(1)}$, $T_{\pi(2)}$,..., $T_{\pi(k)}$, gdzie π jest permutacją $\{1, 2, \ldots, k\}$.
Pojęcie szeregowalności zależy istotnie od tego, co rozumiemy przez równoważność działania dwóch harmonogramów. Zasadniczo przyjmuje się, że harmonogramy działają równoważnie, jeśli wszystkie zawarte w nich transakcje widzą takie same dane i zapisują takie same wartości w bazie danych. Precyzyjniej można to opisać, gdy zdecydujemy się, jakie operacje i jednostki wyróżniamy w bazie danych (patrz kolejny rozdział).

[1] Blokady na poziomie baz danych są "wyższym" pojęciem niż blokady sprzętowe dbające np. o jednoznaczność zapisu konkretnej wartości do konkretnej komórki pamięci. Rozważając problem wielodostępu w bazach danych zazwyczaj zakładamy, że bezpośrednie kolizje wynikające z chęci jednoczesnego zapisania wartości w tej samej komórce przez różne transakcje są rozwiązywane przez system operacyjny.

Problem szeregowalności — zapewnienie przez SZBD takich warunków, w których działanie przeplatających się transakcji będzie równoważne ich pewnemu sekwencyjnemu wykonaniu, czyli wykonywany harmonogram będzie szeregowalny.

1.2 Równoważność harmonogramów

Model bazy danych określa, jakie jednostki wyróżniamy w bazie danych i z jakich operacji elementarnych składają się transakcje:

- w modelach bardzo prostych (np. tylko tabela jako jedostka bazy danych i jeden rodzaj operacji — dostęp całkowity) — kontrola szeregowalności jest stosunkowo łatwa; model jest jednak bardzo ogólny i nie pozwala na wykonanie harmonogramów, które przy uwzględnieniu specyfiki jednostek i operacji mogą okazać się szeregowalne (np. można wykonać przeplatające się operacje transakcji, które chcą działać na różnych rekordach tabeli);
- w modelach bardziej złożonych (np. jako jednostki wyróżniamy zarówno tabele, jak i rekordy bazy danych, oraz odróżniamy operację czytania od operacji zapisu) — można dokładniej przeanalizować harmonogram i rozpoznać (i zaakceptować) taki, który gwarantuje niezależną pracę transakcji; problem analizy komplikuje się jednak (czasami aż za bardzo, np. do poziomu NP-zupełnego).

Przyjrzyjmy się kilku modelom różniącym się rozróżnianymi operacjami na bazie danych:

Model 1 Rozróżniamy operacje odczytu i dostępu całkowitego (lepiej nie nazywać go zapisem, ze względu na kolizję z oznaczeniami z kolejnego modelu). Rozważając równoważność harmonogramów w tym modelu zauważamy, że:

- względna kolejność odczytów tej samej jednostki przez różne transakcje nie ma znaczenia dla efektów działania transakcji, więc może być różna w równoważnych harmonogramach;
- kolejność odczytu względem dostępu całkowitego do danej jednostki musi być zachowana w harmonogramach równoważnych;
- zachowana także musi być względna kolejność dostępów całkowitych do jednostki.

Model 2 Rozróżniamy operacje odczytu i zapisu (zwanego także "ślepym zapisem"). Rozważając równoważność harmonogramów w tym modelu zauważamy, że:

- istotna jest kolejność operacji zapisu i odczytu danej jednostki; jeśli w harmonogramie H transakcja T zapisuje wartość jednostki A ($h = T:\ \mathit{write}\ A$), a transakcja T' czyta tę wartość ($h' = T':\ \mathit{read}\ A$), to w każdym harmonogramie równoważnym H operacja h musi poprzedzać operację h';

- dodatkowo, żaden zapis jednostki A nie może zostać wykonany pomiędzy h i h', czyli każda operacja zapisu $g = S : \ write\ A$, dla $g \neq h$, z harmonogramu H w każdym harmonogramie równoważnym H musi zostać wykonana albo przed h, albo po h'.

Model 3 Rozróżniamy operacje odczytu, inkrementacji (zwiększania) i dostępu całkowitego. Rozważając równoważność harmonogramów w tym modelu zauważamy, że:

- względna kolejność odczytów tej samej jednostki nie ma znaczenia;
- względna kolejność inkrementacji tej samej jednostki nie ma znaczenia;
- kolejność odczytu względem dostępu całkowitego i inkrementacji dla tej samej jednostki musi być zachowana;
- kolejność inkrementacji względem dostępu całkowitego dla tej samej jednostki musi być zachowana.
- względna kolejność dostępów całkowitych dla tej samej jednostki musi być zachowana.

W modelach 1 i 3 problem szeregowalności harmonogramu może być rozwiązany w czasie wielomianowym — sprowadza się do zapewnienia acykliczności pewnego grafu (patrz kolejny rozdział). W modelu 2 badanie szeregowalności jest NP-trudne, gdyż sprowadza się do próby uporządkowania transakcji w sytuacji, w której dla pewnych transakcji mamy warunki typu: musi być przed albo po pewnej parze transakcji.[2]

1.3 Podejście zachowawcze i optymistyczne

Organizując wielodostęp transakcji do bazy danych możemy starać się zapewnić transakcji dostęp do wszystkich potrzebnych jej danych, tym samym gwarantując jej szansę na wykonanie się. Jest to *podejście zachowawcze oparte na blokadach*, które zazwyczaj powoduje długotrwałe trzymanie zasobów i zmusza inne transakcje do oczekiwania. Inne podejście jest stosowane *w metodach optymistycznych*, w których transakcje na bieżąco wykonują swoje operacje — sprawdzenie, czy operacje danej transakcji są właściwie uszeregowane w czasie względem innych transakcji, jest przeprowadzane wówczas, gdy transakcja odwołuje się do bazy danych lub wręcz na końcu transakcji. Jeśli wynik sprawdzenia jest negatywny, transakcja musi być wycofana i, ewentualnie, uruchomiona ponownie.

[2] W takim przypadku także można zbudować graf pierwszeństwa, analogiczny jak dla modeli 1 i 3. Trzeba jednak do niego potem dodać pary krawędzi "alternatywnych" — z każdej takiej pary do grafu musimy wybrać dokładnie jedną. Pytamy, czy graf można tak skonstruować, by był acykliczny. Intuicja podpowiada, że taki wybór prowadzi do czasu wykładniczego. Jest też dowód na to, że problem jest NP-trudny.

2 Metody blokad dla jednostek niezależnych

Rozważmy Model 1 z poprzedniego rozdziału, czyli bazę danych złożoną z niezależnych rekordów z operacjami odczytu i dostępu całkowitego. Przyjmijmy, że w celu uzyskania dostępu całkowitego do jednostki A transakcja T musi dostać (od SZBD) blokadę wyłączną (nazywaną `T: LOCK A` lub `T: EXCLUSIVE LOCK A`). System może przydzielić transakcji blokadę wyłączną, jeśli nie przydzielił żadnej innej transakcji żadnej blokady na tę jednostkę. W celu odczytu jednostki A transakcja T musi dostać (od SZBD) blokadę dzieloną (nazywaną `T: RLOCK A` lub `T: SHARED LOCK A`). System może przydzielić transakcji blokadę dzieloną, jeśli nie przydzielił żadnej innej transakcji blokady wyłącznej na tę jednostkę. Po wykonaniu operacji transakcja powinna zwolnić w pewnym momencie blokadę jednostki (`T: UNLOCK A`) — najpóźniej w momencie zakończenia T (wypełnienia: `COMMIT` lub wycofania: `ROLLBACK`). Widzimy, że w powyższej sytuacji blokady tej samej jednostki wykluczają się ($-$) zgodnie z następującą tabelą:

	`LOCK (EXCLUSIVE LOCK)`	`RLOCK (SHARED LOCK)`
`LOCK (EXCLUSIVE LOCK)`	$-$	$-$
`RLOCK (SHARED LOCK)`	$-$	$+$

2.1 Szeregowalność w modelu z operacjami odczytu i dostępu całkowitego

Załóżmy, że w rozważanym modelu mamy dany pewien harmonogram H transakcji $T_1, T_2, \ldots, T_k$ i zamierzamy sprawdzić, czy jest on równoważny jakiemukolwiek harmonogramowi sekwencyjnemu tych transakcji. W tym celu możemy zbudowac graf, w którym zaznaczymy krawędziami skierowanymi (łukami) kolejność transakcjami wynikającą z ich działania na jednostkach w H.

Grafem pierwszeństwa dla harmonogramu H transakcji $T_1, T_2, \ldots, T_k$ nazywamy graf skierowany $G = (V, E)$, gdzie $V = \{T_1, T_2, \ldots, T_k\}$. Krawędź $(T_i, T_j) \in E$, jeśli istnieje jednostka A taka, że:

- operacja T_i : `LOCK A` poprzedza w H operację T_j : `LOCK A` i pomiędzy tymi operacjami nie występuje T_k : `LOCK A`, dla innej transakcji T_k;
- operacja T_i : `RLOCK A` poprzedza w H operację T_j : `LOCK A` i pomiędzy tymi operacjami nie występuje T_k : `LOCK A`, dla innej transakcji T_k;
- operacja T_i : `LOCK A` poprzedza w H operację T_j : `RLOCK A` i pomiędzy tymi operacjami nie występuje T_k : `LOCK A`, dla innej transakcji T_k.

Twierdzenie 1 Harmonogram w rozważanym modelu jest szeregowalny wtedy i tylko wtedy, gdy jego graf pierwszeństwa jest acykliczny. Równorzędny takiemu harmonogramowi harmonogram sekwencyjny, to rozszerzenie liniowe porządku częściowego transakcji zadanego przez graf. □

Uwagi:

1. Dowód twierdzenia jest prosty.
2. Algorytm sprawdzania szeregowalności wynikający z twierdzenia jest łatwy i szybki (liniowy względem liczby krawędzi).
3. Algorytm sprawdzania szeregowalności wynikający z twierdzenia jest niepraktyczny, bo zazwyczaj nie przetwarzamy harmonogramu wsadowo, więc nie mamy kompletnego zbioru operacji do wykonania, który możemy przeanalizować. W SZBD transakcje pojawiają się dynamicznie, nowe pojawiają się, gdy inne jeszcze trwają, stare chcielibyśmy zakończyć nie czekając na dokończenie tych najnowszych.

2.2 Protokół dwufazowy dla modelu z operacjami odczytu i dostępu całkowitego

Praktyczną metodę zapewnienia szeregowalności harmonogramu transakcji uzyskamy definiując *protokół* (zbiór zasad, których powinna przestrzegać każda transakcja) gwarantujący szeregowalność harmonogramu.

Protokół 2-fazowy (*2 phase locking, 2PL*) — transakcji nie wolno blokować jednostek, jeśli wykonała już chociaż jedną operację odblokowania; to oznacza, że transakcja przestrzegająca protokołu ma dwie kolejne fazy: pierwszą fazę: *wzrostu*, gdy może blokować nowe jednostki, i drugą fazę: *zmniejszania*, gdy zwalnia zablokowane wcześniej jednostki.

Twierdzenie 2: Harmonogram transakcji przestrzegających protokołu dwufazowego jest szeregowalny.
Dowód: Dowód jest prosty. Polega na przeanalizowaniu grafu pierwszeństwa dla transakcji przestrzegających protokołu 2-fazowego i wykazaniu, że nie może w nim istnieć cykl. □

2.3 Brudne dane i anulowanie transakcji

Protokół dwufazowy nie rozwiązuje wszystkich problemów związanych z szeregowalnością. Zauważmy, że zawsze może dojść do sytuacji, gdy jakaś transakcja nie może zostać dokończona. Powody tego mogą być rozmaite:

- wewnętrzne działanie transakcji: przerwanie przez użytkownika, błąd obliczeniowy, brak dostatecznych uprawnień do wykonania operacji;
- interakcje pomiędzy transakcjami: kolizja z innymi transakcjami przy dostępie do zasobów, czyli zakleszczenie (T zablokowała A i czeka na B; T' zablokowała B i czeka na A) lub zagłodzenie (T nie może doczekać się na jednostkę A, bo cały czas system przydziela tę jednostkę innym transakcjom);
- przyczyny zewnętrzne: awaria pamięci, systemu, dysku, sieci...

Atomowość wymaga, by taką transakcję wycofać i usunąć z bazy danych wszystkie ślady jej działania. *Brudne dane* (ładniej: *dane niepewne*) to dane zapisane do bazy i odblokowane przez transakcję, która jeszcze nie jest wypełniona. Dane takie:

- stają się pewne, gdy transakcja kończy się wypełnieniem (`COMMIT`);
- muszą zostać wycofane z bazy, a jednostkom trzeba przywrócić poprzednie wartości, gdy transakcja kończy się wycofaniem (`ROLLBACK`).

Jeśli udostępnimy pewnej transakcji T brudne dane utworzone przez transakcję S, to w razie konieczności wycofania S trzeba także wycofać T, a być może także inne transakcje, które przeczytały dane utworzone przez T (te transakcje nawet nie czytały danych niepewnych!). Sytuację taką nazywamy *anulowaniem kaskadowym*. Jest ona bardzo nieprzyjemna, bo może dotyczyć nawet dawno wypełnionych transakcji i stwarza problemy z zapewnieniem trwałości transakcji.

Protokół ściśle dwufazowy. Można zabezpieczyć się przed anulowaniem kaskadowym. W tym celu modyfikujemy protokół dwufazowy wydłużając czas blokowania danych zmienionych przez transakcję do momentu wypełnienia tej transakcji. Wówczas transakcja zwalnia dane dopiero, gdy są pewne, więc nie naraża innych transakcji na anulowanie kaskadowe, nawet jeśli sama musi zostać anulowana.

3 Inne modele danych

Można zaobserwować, że oprócz modelu bazy danych złożonej ze zbioru niezależnych rekordów typowe są również następujące modele:

- baza tworząca *strukturę drzewiastą* — jednostki bazy są niezależne, jeśli chodzi o zawieranie, ale tworzą drzewo ze względu na powiązania nawigacyjne — w bazie przechodzi się od rekordu do jego dzieci i/lub ojca; w bazie przykładem danych tej postaci jest indeks o strukturze B-drzewa;
- baza tworząca *hierarchię* — jednostki bazy są różnej wielkości i jednostka nadrzędna składa się z jednostek podrzędnych; z taką sytuacją mamy do czynienia,

gdy transakcje mogą traktować jako jednostkę zarówno całą tabelę (jednostka nadrzędna), jak i pojedynczy rekord (jednostka podrzędna); ewentualnie, gdy baza opisuje strukturę złożonej konstrukcji i transakcja może zablokować zarówno dużą część, jak i pojedynczy element.

Dla harmonogramu transakcji działających na bazach opisanych powyższymi modelami zawsze można zastosować blokady i protokół dwufazowy (ewentualnie ściśle dwufazowy) gwarantując szeregowalność harmonogramu. Nie jest to jednak rozwiązanie efektywne:

- stosując protokół dwufazowy dla indeksu w postaci B-drzewa, zmuszamy transakcję, która korzysta z indeksu, do trzymania blokady na korzeniu indeksu przez całą fazę wzrostu; to może generować długie kolejki innych transakcji czekających na dostęp do indeksu; warto poszukać innego rozwiązania, które pozwala wcześniej zwalniać korzeń indeksu, gdy transakcja jest pewna, że do niego nie wróci;
- aby zachować poprawność blokowania w modelu hierarchicznym, zmuszamy transakcję chcącą zablokować jednostkę większą (np. tabelę) do zablokowania wszystkich zawartych w niej jednostek mniejszych (np. rekordów tabeli); powoduje to duży nakład pracy na blokowanie i zwiększa prawdopodobieństwo zakleszczeń i zagłodzeń; warto zastanowić się nad sposobem blokowania, który zagwarantuje legalność w modelu hierarchicznym i będzie sprawniejszy.

3.1 Protokół drzewa

Protokół drzewa pozwoli nam odejść od protokołu dwufazowego, czyli pozwolić transakcjom na wcześniejsze zwalnianie blokad, przy jednoczesnym zachowaniu szeregowalności harmonogramu. Rozważmy działanie protokołu dwufazowego dla transakcji korzystających z indeksu w postaci B-drzewa.

- W czasie **odczytu danych** każda transakcja rozpoczyna od czytania korzenia drzewa a następnie przechodzi w dół (od ojca do dziecka) w poszukiwaniu właściwego rekordu.
- W czasie operacji modyfikacji indeksu (np. wstawiania/usuwania krotek do tabeli) transakcja również rozpoczyna dostęp od korzenia drzewa. W razie konieczności (przepełnienia indeksu) modyfikuje wierzchołek i przechodzi do jednego z jego dzieci. Dalej postępuje analogicznie idąc w dół drzewa i ewentualnie modyfikując jego węzły.[3]

[3]Warto w tym miejscu przypomnieć, że mamy tradycyjne B-drzewa, które możemy nazwać dwuprzebiegowymi, oraz ulepszoną wersję, którą możemy nazwać jednoprzebiegową. W drzewach dwuprzebiegowych wykonując operację modyfikacji (wstawienia/usunięcia rekordu) najpierw schodzimy do liścia i wstawiamy tam dane. Jeśli wstawienie spowodowało przepełnienie w liściu, to rozszczepiamy go na dwa i wracamy do ojca, by dodać do niego klucz i wskaźnik dla nowego węzła. To może także

W obu przypadkach transakcja przestrzegająca protokołu dwufazowego może zwolnić korzeń indeksu (i udostępnić go innym transakcjom) dopiero po dojściu do liścia. Jest to spore opóźnienie nawet, jeśli typowy indeks ma wysokość ok. 3-4. Warto więc rozejrzeć się za rozwiązaniem, które usprawni korzystanie z danych o strukturze B-drzewa zapewniając jednocześnie szeregowalność.

Poniżej założymy, że dane mają strukturę drzewa a każda transakcja może (chcieć) zablokować dowolną jednostkę. Przyjmijmy dla uproszczenia, że mamy tylko jeden rodzaj blokady — całkowity. Możemy wówczas zastosować następujący protokół.

Protokół drzewa:

- transakcja zakłada pierwszą blokadę na dowolny węzeł (oczywiście niezablokowany przez inną transakcję);
- transakcja może zablokować kolejny węzeł drzewa tylko wtedy, gdy blokuje jego ojca;
- transakcja może zwolnić dowolną blokadę w dowolnym momencie (czyli nie jest dwufazowa);
- transakcja nie może zablokować żadnej jednostki dwukrotnie.

Twierdzenie 3: Harmonogram transakcji przestrzegających protokołu drzewa jest szeregowalny.
Dowód: W dowodzie, który nie jest trywialny, można skonstruować graf pierwszeństwa i zaobserwować, że protokół drzewa nie pozwala na powstawanie w nim cykli. □

Uwaga: Można stosować na rozłącznych częściach bazy danych różne rozwiązania gwarantujące szeregowalność, np. protokół drzewa na indeksie i protokół ściśle dwufazowy na rekordach z danymi. Pozwala to zachować szeregowalność na całej bazie.

3.2 Protokół z ostrzeżeniami

Rozważmy bazę danych, w której jednostki są różnego rozmiaru i mogą się w sobie zawierać. Przykładem takiej sytuacji jest zwykła relacyjna baza danych, w której można zablokować tylko pole rekordu, sam rekord, całą tabelę lub cały schemat.... To oznacza, że blokowane jednostki tworzą hierarchię ze względu na relację zawierania.

spowodować przepełnienie, które czasami przenosi się aż do korzenia. W takiej sytuacji transakcja wstawiająca element musi trzymać blokadę korzenia indeksu do końca, bo nie jest pewna, czy nie będzie musiała go zmodyfikować. W drzewach jednoprzebiegowych, zanim wejdziemy do poddrzewa, sprawdzamy, czy jego korzeń nie jest na granicy zapełnienia. Jeśli tak, to od razu go modyfikujemy (rozszczepiamy, jeśli jest pełny, lub łączymy, gdy jest w połowie pusty), co daje nam gwarancje, że wszystkie dalsze operacje wykonamy w poddrzewie bez konieczności powrotu do korzenia i jego modyfikacji. Stąd w jednoprzebiegowych B-drzewach mamy szansę na szybsze zwalnianie węzłów B-drzewa.

Zauważmy, że zwykłe zablokowanie większej jednostki, np. tabeli, wymaga wówczas zablokowania wszystkich jej jednostek podrzędnych, czyli rekordów i ich pól. Taka operacja blokowania może być bardzo czasochłonna i nieefektywna. Warto więc poszukać innego rozwiązania, które pozwoli efektywnie i poprawnie blokować jednostki wraz z jednostkami podrzędnymi.

Przyjmijmy, że każda transakcja w bazie może blokować każdą jednostkę całkowicie lub do odczytu. Mamy więc blokady, które oznaczymy odpowiednio `LOCK` i `RLOCK`. Dodatkowo wprowadzimy blokady pomocnicze, tzw. ostrzeżenia: `WARN` i `RWARN`. Zakładamy także, że jednostki w bazie tworzą hierarchię **w postaci drzewa**, tak by do każdej jednostki można było dojść od korzenia hierarchii tylko jedną drogą. Poniższy protokół pozwala na sprawne i legalne blokowanie jednostek w tym modelu.

Protokół z ostrzeżeniami:

- każda transakcja przestrzega protokołu dwufazowego (ewentualnie ściśle dwufazowego);
- traksakcja w celu zablokowania jednostki A blokadą `LOCK` (odpowiednio `RLOCK`) przechodzi ścieżkę od korzenia hierarchii do jednostki A zakładając po drodze ostrzeżenia `WARN` (odpowiednio `RWARN`) na wszystkie węzły na ścieżce i blokadę `LOCK` (odpowiednio `RLOCK`) na A.
- przy próbie założenia blokady na każdy węzeł obowiązują następujące zasady wykluczania się blokad:

	WARN	RWARN	LOCK	RLOCK
WARN	+	+	−	−
RWARN	+	+	−	+
LOCK	−	−	−	−
RLOCK	−	+	−	+

- każda transakcja zdejmuje blokady i ostrzeżenia w kolejności odwrotnej do kolejności zakładania — idąc od jednostki do korzenia hierarchii.

Powyższy protokół gwarantuje, że wszystkie zakładane blokady są **legalne**, czyli nie ma niedopuszczalnej kolizji na żadnej blokowanej przez transakcje jednostce. Liczba zakładanych fizycznie blokad może być przy tym znacznie mniejsza niż w przypadku stosowania prostej zasady zablokowania dużej jednostki przez zablokowanie wszystkich jej jednostek składowych.

4 Metoda znaczników czasowych

Model Pierwszą metodę optymistyczną rozważmy w modelu danych, w kórym baza składa się z niewielkich, niezależnych jednostek (np. rekordów), a operacje dostępu do danych to:

read — odczyt jednostki,

write — zapis wartości jednostki bez odczytu (czyli "ślepy" zapis).

Znaczniki czasowe (timestamps). Każda transakcja ma przypisany znacznik czasowy *time(T)*, który system nadaje jej w momencie uruchomienia. Znaczniki transakcji są unikalne i można je interpretować jako dokładny czas rozpoczęcia transakcji. Każda jednostka danych w bazie posiada dwa znaczniki czasowe: *rtime(A)* — znacznik transakcji, która ją czytała jako ostatnia (czyli "czas" ostatniego odczytu A), oraz *wtime(A)* — znacznik transakcji, która zapisała A jako ostatnia (czyli "czas" ostatniego zapisu A).

W metodzie znaczników czasowych pozwalamy działać transakcjom dość swobodnie, ale na działanie na jednostce bazy danych zezwalamy tylko wtedy, gdy kolejność operacji odpowiada znacznikom czasowym transakcji. To gwarantuje, że wykonywany harmonogram jest szeregowalny — równoważny harmonogramowi sekwencyjnemu zadanemu kolejnością znaczników czasowych. Z tego założenia wynikają następujące procedury kontrolne:

- gdy transakcja T chce przeczytać jednostkę A, to sprawdzamy, czy jednostka A nie została już zapisana przez transakcję późniejszą niż T ($wtime(A) > time(T)$) — jeśli tak się stało, to trzeba wycofać transakcję T;
- gdy transakcja T chce zapisać jednostkę A, to po pierwsze trzeba sprawdzić, czy jakaś transakacja późniejsza od T nie przeczytała już A ($rtime(A) > time(T)$)— w takim przypadku trzeba wycofać T; gdy nie ma przeszkód z tego powodu, to trzeba sprawdzić, czy jakaś późniejsza transakcja T' nie zapisała już swojej wartości w jednostce A ($wtime(A) > time(T)$) — jeśli tak się stało, to zapis T trzeba pominąć, ale transakcję T można kontynuować (gdyby T zapisała A o czasie, to jej wartość i tak zostałaby zamazana przez zapis T' i nikt by jej w międzyczasie nie przeczytał).

Podane zasady pozwalają napisać następujący algorytm obsługi transakcji w metodzie znaczników czasowych:

```
T: read(A)
----------
if time(T) < wtime(A) then {
  rollback(T);
  exit(rollback T and restart T);
};
rtime(A) := max { rtime(A), time(T)};
execute read A by T;
```

```
T: write(A)
-----------
if time(T) < rtime(A) then {
  rollback(T);
 exit(rollback T and restart T);
}
elseif time(T) < wtime(A) then {
  skip(T: write A);
  continue T;
}
else {
wtime(A) := time(T);
execute write A by T;
}
```

Uwagi:

1. Metoda znaczników czasowych gwarantuje szeregowaność harmonogramu zgodną z porządkiem znaczników czasowych transakcji.
2. Metoda ta działa sprawnie, gdy kolizje pomiędzy transakcjami występują sporadycznie.
3. W metodzie tej powstaje wiele brudnych danych.
4. W razie częstych kolizji może łatwo dochodzić do wycofywania transakcji i (być może) anulowania kaskadowego innych transakcji, które przeczytały zapisane przez nią jednostki. Anulowanie kaskadowe może dotyczyć nawet transakcji zakończonych, co jest wyjątkowo kłopotliwe.

4.1 Bit wypełnienia

Metodę znaczników czasowych można uzupełnić o kontrolę brudnych danych. W tym celu każdej jednostce danych przypisujemy dodatkowy bit $C(A)$, który ma wartość `true`, jeśli transakcja o znaczniku *wtime(A)*, czyli transakcja która zapisała A jako ostatnia, jest wypełniona. W przeciwnym razie A jest brudną daną i bit $C(A)$ ma wartość `false`.

Po wprowadzeniu bitu C trzeba zmodyfikować i uzupełnić podane poprzednio procedury:

`T: read(A)` — jeśli transakcja T może przeczytać A ze względu na znacznik czasowy, to dodatkowo sprawdza $C(A)$; jeśli bit ma wartość `true`, to czyta A, w przeciwnym razie jest dołączana do kolejki transakcji oczekujących na odczyt A;

`T: write(A)` —— transakcja T nie może zapisać A, gdy jest to dana niepewna, nawet, jeśli pozwala na to relacja pomiędzy jej znacznikiem czasowym i znacznikami czasowymi A; możliwe jest bowiem, że w kolejce związanej z A czekają transakcje, które chcą A odczytać; T musi więc czekać z zapisem A aż sytuacja się wyjaśni i bit $C(A)$ zostanie ustawiony na `true`.

`T: commit` —— transakcja, która się wypełniła, może zatwierdzić wszystkie zapisane przez siebie wartości (ustawić ich bit C na `true`) i wpisać im swój znacznik czasowy jako *wtime*; procedura ta powinna dodatkow "obudzić" transakcje oczekujące na dostęp do tej jednostki (zarówno odczyt, jak i zapis) —— system wznawia ich przetwarzanie w kolejności, w jakiej są w kolejce (lub w kolejności znaczników czasowych);

`T: rollback` —— wycofując transakcję system musi "odwiedzić" wszystkie zapisane przez nią jednostki, odtworzyć ich wartości sprzed zapisu przez wycofywaną transakcję i ustawić bit C na `true`; następnie system może wznowić wykonywanie wszystkich transakcji oczekujących na dostęp do jednostki.

4.2 Wiele wersji jednostek

Metoda znaczników czasowych może być stosowana w bazach, w których pamiętanych jest kilka ostatnich, ewentualnie wszystkie, wartości, jakie przyjmowała jednostka. Z każdą taką wersją przechowywany jest czas jej utworzenia. W tak uzupełnionej bazie można wykonać więcej odczytów — w szczególności transakcja spóźniona może czytać wartości historyczne. Należy jednak wówczas zrezygnować z pomijania wycofywania spóźnionych transakcji zapisujących, które poprzednio po prostu pomijały swój zapis — teraz taka transalcja zmieniłaby historię jednostki, która jest dostępna dla innych transakcji. Baza z wartościami historycznymi może służyć do kilku celów:

- wartości historyczne mogą być wykorzystywane tylko do usprawnienia przetwarzania transakcyjnego i wtedy trzeba przechowywać tylko wartości datujące się na czas rozpoczęcia najstarszej aktywnej transakcji;
- wartości historyczne mogą być traktowane jako "logi" i wykorzystywane do odtwarzania stanu bazy danych w przypadku awarii;
- wartości historyczne mogą stać się integralną częścią bazy danych, w której możemy zadawać pytania o stan bazy z dowolnego momentu czasowego lub o wartość jednostki na przestrzeni pewnego okresu czasu — mamy wówczas do czynienia z *temporalną bazą danych.*

5 Metoda walidacji

W metodzie znaczników czasowych system kontroluje, czy transakcja działa "zgodnie z harmonogramem" przy każdej operacji odczytu/zapisu. W metodzie walidacji takie

sprawdzenie jest wykonywane dla każdej transakcji tylko raz –– w chwili, gdy zakończy ona obliczenia i jest gotowa do zapisu wyników w bazie. Jeśli sprawdzenie (walidacja) przebiegnie pomyślnie, to transakcja jest kończona i zapisuje swoje wyniki w bazie. W przeciwnym razie transakcja jest wycofywana i uruchamiana ponownie.

Cykl życia transakcji W momencie uruchomienia transakcja otrzymuje znacznik *START(T)* (jak w przypadku *timestamp*, często jest to moment uruchomienia transakcji w systemie). Ma także przydzielony prywatny obszar roboczy — w trakcie obliczeń wczytuje do niego wszystkie potrzebne jej dane i tam je przetwarza. Zbiór jednostek bazy danych odczytanych przez transakcję oznaczmy RS(T). Po zakończeniu obliczeń transakcja zgłasza gotowość do zakończenia i zapisania wyników w bazie –– zbiór jednostek, które transakcja zamierza zapisać oznaczmy WS(T). System nadaje transakcji kolejny znacznik czasowy: czas walidacji *VAL(T)*. W trakcie walidacji system kontroluje, czy można zapisać wyniki transakcji nie naruszając porządku zadanego wartościami *VAL* poszczególnych transakcji. Jeśli walidacja wypadnie pomyślnie, to wyniki transakcji są zapisywane w bazie i system nadaje transakcji znacznik *FIN(T)* — czas zakończenia T.

Transakcje w systemie Z powyższego wynika, że w każdym momencie t mamy w systemie kilka grup transakcji:

zbiór START — transakcje T już uruchomione, ale jeszcze niezgłoszone do walidacji; zachodzi da nich $START(T) < t$, są w fazie czytania danych z bazy i wykonywania obliczeń w swoim obszarze roboczym; nie mają określonych wartości $VAL(T)$ i $FIN(T)$.

zbiór VAL –– transakcje po walidacji (mają nadane $VAL(T) < t$) będące w trakcie zapisywania wyników do bazy (czyli nie mające jeszcze nadanego *FIN(T)*).

zbiór FIN –– transakcje po walidacji, które zakończyły także zapisywanie wyników i wypełniły się. Mają nadaną wartość $FIN(T) < t$.

Walidacja W procesie walidacji nie dopuszczamy do wykonania tych transakcji, dla których istnieje ryzyko naruszenia porządku działań zadanego wartościami *VAL(T)*. W tym celu sprawdzamy oddzielnie, czy transakcja T mogła przeczytać poprawne wartości jednostek RS(T) oraz czy może zapisać poprawnie w bazie swoje wyniki — zbiór WS(T). Przeszkodą w każdym z tych wypadków jest:

czytanie –– transakcja T nie może ryzykować przeczytania danych, jeśli transakcja od niej wcześniejsza może je jeszcze zmienić; taka sytuacja może wystąpić, jeśli istnieje już zwalidowana transakcja S ($VAL(S) < VAL(T)$, której okres zapisu nakłada się na okres czytania T ($START(T) < FIN(S)$) i obie transakcje odwoływały się do tej samej jednostki: $RS(T) \cap WS(S) \neq \emptyset$. W takiej sytuacji istnieje możliwość, że T przeczytała pewną jednostkę X zanim S zdążyła ją zapisać. To

oznacza, że trzeba wycofać T, bo przecież powinna wykonać się po S i czytać wartości utworzone przez S.

pisanie — transakcja T nie powinna zapisywać danej, jeśli istnieje transakcja S, wcześniejszą od T ($VAL(S) < VAL(T)$), która także tę daną chce zapisać i okresy zapisu S i T nakładają się ($VAL(T) < FIN(S)$, czyli w momencie walidacji T transakacja S jeszcze nie zakończyła zapisywania); rodzi to bowiem ryzyko, że T zapisze przed S, a ostatecznym wynikiem powinna być wartość zapisana przez T.

Powyższe zasady można zapisać w postaci następującej procedury walidacji transakcji T, wykonywanej w momencie $VAL(T)$:

```
if ∃S : (VAL(S) < VAL(T)) ∧
        (RS(T) ∩ WS(S) ≠ ∅) ∧
        ⟨VAL(S), FIN(S)⟩ ∩ ⟨START(T), VAL(T)⟩ ≠ ∅
then rollback(T);
```

```
if ∃S : (VAL(S) < VAL(T) ∧
        WS(T) ∩ WS(S) ≠ ∅ ∧
        ⟨VAL(S), FIN(S)⟩ ∩ ⟨VAL(T), FIN(T)⟩ ≠ ∅
then rollback(T);
```

6 Poziomy niezależności w SQL

W standardzie SQL zaproponowano cztery poziomy niezależności dla transakcji. Użytkownik może ustalić wybrany poziom poleceniem:

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
```

Wymagania w stosunku do każdego z poziomów są następujące:

READ UNCOMMITTED — transakcja może czytać brudne dane, gdyż czytanie przez nią nie wymaga żadnego blokowania jednostek; transakcji na tym poziomie niezależności nie wolno nic zapisywać, gdyż jest on dopuszczalny jedynie dla transakcji typu `READ ONLY`;

READ COMMITTED — transakcja może jedynie czytać rekordy utworzone przez transakcje zatwierdzone; nie grozi jej anulowanie kaskadowe z powodu czytania brudnych danych; działanie transakcji na tym poziomie niezależności musi być równoważne działaniu transakcji blokującej czytane dane na czas operacji, a dane zapisywane — do momentu wypełnienia;

REPEATABLE READ — transakcja na tym poziomie niezależności czyta tylko dane pewne i dodatkowo wymaga, by wartości jednostek przez nią czytanych nie zostały zmienione (przez inne transakcje) do końca działania danej transakcji (na wypadek, gdyby chciała powtórnie przeczytać taką jednostkę — powinna wówczas zobaczyć tę sama wartość jednostki, co za pierwszym razem). Działanie transakcji na tym poziomie niezależności odpowiada działaniu transakcji utrzymujących do końca blokady jednostek zarówno czytanych, jak i zapisywanych.

SERIALIZABLE — jeśli transakcja wykonuje skanowanie całej tabeli, to w czasie działania transakcji zawartość tabeli nie może ulec zmianie, w szczególności nie mogą zostać ani do niej dopisane, ani z niej usunięte żadne rekordy (rekordy dopisane i ńiezauważone"przez taką transakcję nazywmy *fanotomami*); gwarantuje to szeregowalność transakcji zarówno, gdy rozważamy jako jednostki bazy danych rekordy, jak i tabele. Działanie transakcji na tym poziomie niezależności jest równoważne działaniu transakcji trzymających do końca blokady pojedyńczych rekordów czytanych i zapisywanych przez transakcję oraz dodatkowo blokujących całe tabele, jeśli transakcja wykonuje operację skanowania czy wyboru z całej tabeli.

W poniższej tabeli podsumowane są efekty działania transakcji na kazdym z poziomów niezależności:

Poziom niezależności	Blokada czytanego rekordu	Blokada zapisywanego rekordu	Blokada skanowanej tabeli
READ UNCOMMITTED	brak	brak	brak
READ COMMITTED	zdejmowana po przeczytaniu	trzymana do końca	brak
REPEATABLE READ	trzymana do końca	trzymana do końca	brak
SERIALIZABLE	trzymana do końca	trzymana do końca	trzymana do końca

Jak widać, zaproponowane metody odpowiadają blokowaniu ściśle dwufazowemu na poziomie rekordów lub tabel. Oczywiście w konkretnym SZBD nie musi to być implementowane poprzez protokół ściśle dwufazowy.

7 Bazy i transakcje rozproszone

Rozproszona baza danych to logicznie spójna baza przechowywana w wielu węzłach fizycznych połączonych siecią. Przyjmuje się, że komunikacja pomiędzy węzłami trwa istotnie dłużej niż operacje bazodanowe wewnątrz węzła, więc liczba takich komunikatów jest podstawą do wyznaczania złożoności operacji na bazie danych.

7.1 Dane rozproszone

Zawartość bazy danych może być podzielona pomiędzy węzły:

podział pionowy — różne tabele znajdują się w różnych węzłach; najbardziej kłopotliwą operacją jest w takiej sytuacji złączenie naturalne tabel z różnych węzłów;

podział poziomy — rekordy jednej tabeli są przechowywane w różnych węzłach, np. dane studentów są na ich wydziałach albo dane studentów dużej uczelni są losowo (funkcją haszującą) rozrzucone pomiędzy kilka serwerów przechowujących bazę danych;

powielenie — rekordy (ewentualnie całe tabele) mogą być przechowywane w bazie w kilku kopiach w celu ułatwienia dostępu; w ten sposób dość łatwo można przyspieszyć odczyt danych, ale dużym problemem staje się zachowanie spójności wszystkich kopii.

7.2 Transakcje rozproszone

W rozproszonej bazie danych transakcja może wykonywać operacje w różnych węzłach. Z racji rozposzenia w sieci i rozciągnięcia w czasie ciągi operacji wykonywane w różnych węzłach wyróżniamy jako podtransakcje. Niezależnie od tego chcielibyśmy zachować atomowość transakcji, czyli zapewnić, że wszystkie podtransakcje podejmą taką samą decyzję w sprawie wypełnienia lub anulowania. Spełnienie tego warunku może nie być łatwe z racji rozproszenia, możliwości awarii i opóźnień w różnych węzłach. Istnieje kilka protokołów, które mają zapewnić atomowosc transakcji. Różnią się one efektywnością (liczbą koniecznych komunikatów) i odpornością na błędy w sieci. Jednym z takich protokołów jest wypełnienie dwufazowe.

Wypełnienie dwufazowe. W wypełnieniu dwufazowym biorą udział podtransakcje, które nazywamy *uczestnikami*. Jeden z uczestników jest wyróżniony — nazywamy go *koordynatorem*. W celu zakończenia transakcji rozproszonej (wypełnieniem lub anulowaniem) koordynator i uczestnicy postępują zgodnie ze swoimi protokołami podanymi poniżej.

1. Koordynator po czasie, który uznaje za wystarczający do wykonania podtransakcji, rozsyła uczestnikom komunikat: "głosujemy".

2. Jeżeli uczestnik wykonał się i jest gotów do wypełnienia, to odpowiada koordynatorowi: "głosuję za wypełnieniem". Jeżeli uczestnik nie zdążył się wykonać lub musiał się anulować, odpowiada: "jestem za anulowaniem" i anuluje się.

3. Jeśli koordynator otrzymał od wszystkich uczestników głos za wypełnieniem, to podejmuje decyzję o wypełnieniu i rozsyła uczestnikom komunikat: "wypełnienie". Jeżeli jakiś uczestnik zagłosował za anulowaniem lub nie odpowiedział na czas, to koordynator rozsyła do uczestników komunikat: "anulowanie".

4. Uczestnicy wykonują COMMIT albo ROLLBACK, w zależności od tego, czy komunikat rozesłany przez koordynatora w kroku 3. brzmiał "wypełnienie" czy "anulowanie".

Powyższy protokół jest odporny na błędy (awarie węzłów i awarie sieci) po stronie uczestników. Wymaga jedynie, by uczestnik, który zagłosował za wypełnieniem i uległ awarii, skontaktował się z koordynatorem po usunięciu awarii i dowiedział się od niego, jaka była ostateczna decyzja. Jeśli było to wypełnienie, to uczestnik wypełnia się, a w przypadku anulowania — anuluje się w swoim węźle..

Protokół 3-fazowy i wybór lidera. Niestety, protokół dwufazowy nie jest odporny na awarie koordynatora — uczestnicy nie otrzymują wówczas komunikatów, ale nie mogą samodzielnie podjąć decyzji o anulowaniu, bo może to nie być decyzja spójna. Aby oczekiwanie na decyzję koordynatora nie doprowadziło do zablokowania węzłów, trzeba zaplanować procedurę wychodzenia z blokady lub "udoskonalić" (jednocześnie skomplikować) protokół wypełnienia. Istnieje np. protokół 3-fazowy, w którym koordynator w dodatkowej rundzie uprzedza, że ma wszelkie dane do podjęcia decyzji. Wówczas:

- milczenie przed otrzymaniem takiej informacji od koordynatora pozwala uczestnikom protokołu podjąć na nowo procedurę uzgadniania wypełnienia (wiedzą, że nikt nie dostał decyzji od koordynatora);
- milczenie koordynatora po uprzedzeniu pozwala uczestnikom przejść do etapu poszukiwania, czy ktoś otrzymał końcową decyzję od koordynatora.

Oddzielnym zadaniem jest także wybór nowego koordynatora, jeśli poprzedni nie podejmuje działania. Jest sporo protokołów pozwalających rozwiązać ten problem, bo jest to typowe zadanie w systemach rozproszonych (nie tylko bazodanowych) nazywane *leader election*. Jeden z prostszych, to wysłanie pozostałym uczestnikom swojego unikalnego numeru i przyjęcie, że nowym koordynatorem będzie uczestnik o najmniejszym numerze.

7.3 Blokowanie jednostek powielonych

Przyjmując, że niektóre jednostki rozproszonej bazy danych występują w wielu kopiach, trzeba zastanowić się nad poprawnym blokowaniem jednostek tak, by nie dochodziło nigdy do faktycznego zablokowania tej samej jednostki logicznej przez różne transakcje. Istnieje kilka metod, w których, aby zablokowac jednostkę, trzeba zablokować pewną jej kopię. Metody różnią się wyborem tej kopii.

Metoda węzła centralnego. Możemy dla rozproszonej bazy danych wybrać węzeł odpowiedzialny za blokowanie. W węźle tym znajduje się tablica blokad wszystkich

jednostek zapisanych w różnych węzłach w wielu kopiach. By zablokować jednostkę, transakcja musi zwrócić się do węzła centralnego i uzyskać od niego blokadę jednostki. Metoda węzła centralnego jest bardzo uzależniona od sprawnego działania węzła zarządzającego blokadami i powoduje duże natężenie komunikacji pomiędzy nim i innymi węzłami.

Metoda kopii pierwotnej. Dla każdej jednostki z rozproszonej bazy danych możemy wskazać, która kopia tej jednostki odpowiada za blokady. Wtedy transakcja chcąca zablokować jednostkę, musi zablokować tę kopię w jej węźle. W tej metodzie działanie całej bazy nie zależy już od jednego węzła i nie ma dużego obciążenia jednego węzła komunikacją. Wszystkie transakcje muszą jednak wiedzieć, gdzie szukać kopii pierwotnej każdej jednostki.

Metoda żetonu. W tej metodzie za blokowanie jednostki jest odpowiedzialna kopia, która posiada żeton tej jednostki (można rozróżniać żeton zapisu i odczytu lub rozważać jeden żeton — blokady całkowitej). Jeśli transakcja chce zablokować jednostkę, musi uzyskać odpowiedni żeton tej jednostki. W tym celu zwraca się do wszystkich węzłów bazy — jeśli jakiś ma żeton i go nie potrzebuje (nie bokuje jednostki), to go przekazuje. Jeśli nikt nie ma żetonu, to potrzebująca transakcja może go u siebie wygenerować. W metodzie żetonu rozproszona baza danych dostosowuje swoją konfigurację dynamicznie do potrzeb. Wymagany jest jednak dodatkowo protokół przekazywania żetonu.

7.4 Blokowanie do odczytu i do zapisu.

W opisanych w poprzednim rozdziale metodach za blokowanie jednostki odpowiadała jedna kopia, która była wyznaczana w różny sposób. Inne metody opierają poprawność blokowania jednostki na zablokowaniu odpowiednio dużej liczby kopii tej jednostki.

Blokowanie większością. Przyjmijmy, że jednostka ma n kopii. Aby zablokować jednostkę blokadą wyłączną, możemy wymagać założenia blokady LOCK (lub WLOCK) na więcej niż $\frac{n}{2}$ kopii. Jednocześnie, aby zablokować jednostkę blokadą dzieloną (RLOCK), wymagamy zablokowania co najmniej $\frac{n}{2}$ kopii. W pojedynczym węźle blokada LOCK (WLOCK) wyklucza się z jakąkolwiek inna blokadą, natomiast blokada do odczytu (RLOCK) może być założona jednocześnie przez wiele transakcji. Łatwo sprawdzić, że podane warunki gwarantują, że przy podanych warunkach blokady zakładane przez transakcje są poprawne.

"Write locks all". Aby zablokować jednostkę blokadą wyłączną transakcja musi zablokować blokadą LOCK (lub WLOCK) wszystkie jej kopie. Aby zablokować jednostkę do odczytu, musi założyć blokadę RLOCK na jedną kopię jednostki. Przestrzeganie tych zasad powoduje, że blokady uzyskane przez transakcje są poprawne.

Metody pośrednie. Można łatwo zauważyć, że w pierwszej z podanych metod założenie blokady wyłącznej jest maksymalnie ułatwione i zablokowanie do odczytu jest prawie tak samo trudne (z dokładnością do tego, że blokadę do odczytu łatwiej uzyskać w dowolnym węźle). W drugiej z przedstawionych metod maksymalnie ułatwiony jest odczyt — wymaga założenia tylko jednej blokady. Metody pośrednie pozwalają ustalić próg $t > \frac{n}{2}$ taki, by blokada wyłączna wymagała założenia co najmniej t blokad LOCK, a zablokowanie do odczytu założenia co najmniej $n - t + 1$ blokad RLOCK.

Stosowanie opisanych wyżej metod powoduje, że istotnym problemem staje się zakleszczenie transakcji. Gdy w środowisku rozproszonym transakcje usiłują zablokować wiele kopii i każdej udaje się zablokować tylko cześć z tego, co potrzebuje, nie uzyskuje ona blokady, ale skutecznie uniemożliwia innym transakcjom uzyskanie blokady.