Uniwersytet Mikołaja Kopernika

Wydział Matematyki i Informatyki

Danuta Posiadała
Nr albumu:158137

Praca magisterska
na kierunku Informatyka

Mechanizm transakcji w relacyjnych i obiektowych bazach danych

Praca wykonana pod kierunkiem
dr. Piotra Wiśniewskiego
Katedra Algebry i Geometrii

Toruń 2006

Rodzicom

Spis treści

Wstęp

Jedną z najbardziej znanych i użytecznych dziedzin informatyki są bazy danych. Dzisiaj informacja jest czymś „na wagę złota" i to nie tylko w świecie biznesu, ale i w wielu innych dziedzinach życia. Stąd też często określa się współczesne pokolenie „społeczeństwem informacyjnym". Konieczność w życiu codziennym gromadzenia, przetwarzania i wymiany danych przyczyniło się do powstania systemów komputerowych, które te właśnie potrzeby zaspakajają.

Początkowo bazy danych były proste, scentralizowane i jednodostępowe. Służyły tylko do przechowywania danych w strukturach zapewniających im należyty porządek oraz do w miarę czytelnej i wygodnej ich prezentacji. Jednak „złośliwość rzeczy martwych" oraz zasada, że „jeśli coś może się zepsuć to na pewno się zepsuje", szybko uświadomiła użytkownikom i twórcom baz danych konieczność wprowadzenia mechanizmów, które by zapewniło bezpieczeństwo danych w sytuacji awarii. Nie chodzi tu o uszkodzenia mechaniczne komputera. Nad zapewnieniem fizycznego bezpieczeństwa systemu komputerowego czuwają administratorzy i producenci poszczególnych urządzeń. Dzięki ich staraniom takie awarie dzisiaj zdarzają się rzadko Dużo częstszy problem stanowią awarie systemowe. Mogą być one spowodowane błędnym działaniem niedopracowanych operacji samej bazy danych, procesów innych aplikacji działających na tym samym komputerze, czy wreszcie niedoskonałością samego systemu operacyjnego.

Najbardziej newralgicznymi punktami działania bazy danych są momenty, w których dokonywane są operacje na danych, czyli ich modyfikacje, wstawianie czy usuwanie. Jeśli właśnie w takiej chwili zdarzy się awaria, wówczas użytkownik nie wie, czy operacja zdążyła się wykonać do końca. Taka częściowo wykonana operacja może powodować zaburzenie spójności bazy danych, tworzyć konflikty natury logicznej ze strukturami czy z założonymi wymaganiami i ograniczeniami bazy danych. Użytkownik, który natknąłby się na tego rodzaju sytuację musiałby wykonać dodatkową pracę: sprawdzić jakie zmiany dokonała ostatnia operacja i dokonać korekty danych tak, by były poprawne. Niestety, nie zawsze jest to proste i nie zawsze użytkownik posiada wystarczającą wiedzę i uprawnienia by to uczynić. Dlatego powstała konieczność dodania do systemów zarządzania bazami danych takich mechanizmów, które gwarantowałyby, że każda zlecona operacja (lub zestaw operacji) wykona się w całości albo nie wykona się wcale. Dzięki tej gwarancji wiadomo, że stan bazy

danych będzie zawsze poprawny.

Mechanizm, który rozwiązuje wyżej opisany problem, to *transakcje* oraz *menadżer transakcji*, który nimi zarządza. Przeciętny użytkownik bazy danych nie ma pojęcia o ich istnieniu, natomiast żaden administrator bazy danych nie wyobraża sobie, by ich mogło nie być.

Kolejnym aspektem, który niejako automatycznie nasuwa się przy tworzeniu bazy danych, jest konieczność stworzenia takiego systemu, który pozwalałby na dostęp do tych samych danych jednocześnie wielu użytkownikom. W każdej, nawet nie wielkiej firmie, instytucji, czy organizacji, często się zdarza, że kilka osób musi pracować w oparciu o te same dane i to w tym samym czasie. Rozwiązanie, w którym każda osoba posiadałaby swoją kopię danych jest bardzo nie praktyczne, gdyż każda najdrobniejsza modyfikacja danych musiałaby zostać dokonana na każdej z kopii. Przesyłanie aktualnej wersji danych do każdego stanowiska byłoby bardzo uciążliwe i kosztowne. Rozwój sieci umożliwił powstanie wielodostępowej, scentralizowanej bazy danych. W strukturze takiej dane przechowywane są w jednym miejscu,a użytkownicy za pomocą sieci komunikują się z nimi i je modyfikują. Dzięki temu każdy ma dostęp do aktualnej bazy danych. Należy się jednak zastanowić, czy zezwolenie na jednoczesny dostęp wszystkich do tych samych danych jest w pełni bezpieczne. Sytuacja, w której kilka osób jednocześnie chce czytać i/lub modyfikować te same dane może być kłopotliwa.

Tym problemem również zajmują się transakcje. Pozwalają one rozstrzygnąć, kiedy jednoczesny dostęp jest w ogóle możliwy i w jaki sposób i w jakiej kolejności zezwalać na odczyt i zapis informacji.

Niniejsza praca opisuje mechanizmy i algorytmy jakie powinny być zaimplementowane w bazie danych, by zapewnić poprawne działanie transakcji i by mogły one w pełni spełniać swoją funkcję. Przedstawione zostaną różne problemy z jakimi programista transakcji może się natknąć oraz różne techniki ich rozwiązania. Celem pracy jest stworzenie własnego systemu transakcyjnego do powstającego zespołowego projektu „SBQLLite” - obiektowej bazy danych – pod kierunkiem pana dr Piotra Wiśniewskiego.

W pierwszym rozdziale wyjaśnię czym właściwie jest transakcja i menadżer transakcji oraz w jaki sposób przebiegają różne operacje w bazie danych. Zostanie poruszony problem bezpieczeństwa systemu i automatycznego zachowania się bazy po awarii. Przedstawię różne sposoby rozwiązania tego problemu oraz techniki usprawniające działanie procesów odtwarzających stan bazy po awarii.

Drugi rozdział poświęcony zostanie *współbieżności* w bazach danych, czyli zarządzaniu operacjami jednoczesnymi w systemach wielodostępowych. Opiszę mechanizmy i algorytmy, które pozwalają na uporządkowanie wszelkich działań dokonywanych na bazie danych, rozstrzygające które operacje, kiedy i gdzie mogą się wykonywać. Ponadto przedstawię rozwiązania problemów, które mogą powstawać w trakcie działania transakcji w systemie współbieżnym.

Trzeci rozdział będzie dotyczyć części praktycznej pracy, czyli napisanego przeze mnie sytemu transakcyjnego. Przybliżę podstawy teoretyczne całego projektu „SBQLLite" oraz opiszę konstrukcję menedżera transakcji. Uzasadnię wybór zastosowanych przeze mnie technik transakcyjnych oraz w jaki sposób rozwiązałam problemy implementacyjne.

1.Transakcje i mechanizmy odtwarzania

1.1.Transakcje

Pojęcie *transakcji* jest jednym z podstawowych pojęć dotyczących baz danych. Przeciętnemu człowiekowi transakcja kojarzy się z operacją finansową. Najprostszym jej przykładem jest przelew pewnej kwoty pieniężnej z jednego konta na drugie. W wyniku takiej transakcji dokonywane są następujące operacje: odjęcie od stanu konta pierwszego danej kwoty, przesłanie informacji o przelewie do drugiego konta i dodanie danej kwoty do stanu drugiego konta. Ten bardzo prosty przykład znajduje swoje odwzorowanie w środowisku baz danych. Transakcja baz danych to nic innego jak ciąg operacji mających się wykonywać na danych z bazy. Operacjami może tu być zapis i odczyt danych oraz wykonywanie na nich wszelakich obliczeń. Istotą transakcji jest grupowanie operacji i łączenie ich w pewną całość. Często się zdarza, że dwie lub kilka operacji musi się wykonać w określonej kolejności i niedopuszczalne jest, żeby choć jedna z operacji składowych się nie wykonała (tak jak w transakcji finansowej nie może się zdarzyć, żeby kwota z konta została odjęta a później nie została dodana do konta drugiego). Dlatego zamyka się je w całość i dzięki temu gwarantuje, że albo wszystkie operacje się wykonają albo żadna. Oto najczęściej spotykane definicje transakcji:

> *Transakcja – operacja lub ciąg operacji wykonany przez jednego użytkownika lub program aplikacji odwołujący się (czytający lub modyfikujący) do zawartości bazy danych.* [2]
>
> *Transakcja to niepodzielny logicznie blok instrukcji.*[3]

Transakcje mogą być *niejawne* lub *jawne*. Transakcje niejawne to takie, które odbywają się bez wiedzy użytkownika. W systemach, w których ustawione jest „autozatwierdzanie” (AUTOCOMMIT), każde zapytanie stanowi oddzielną transakcję niejawną. Transakcja jawna to taka, którą użytkownik sam definiuje, czyli określa blok instrukcji, jaki ma się wykonać w ramach jednej transakcji. W większości baz danych transakcja rozpoczyna się słówkiem BEGIN, a kończy się wykonaniem jawnej instrukcji COMMIT lub ROLLBACK. Jeśli w systemie nie zostało ustawione „autozatwierdzanie”, wówczas nie trzeba określać początku, gdyż zatwierdzenie lub wycofanie transakcji jest jednocześnie początkiem następnej.

Na razie jest wszystko proste. Jednak świat, w którym „żyją” transakcje jest dość

skomplikowany i niestety nie jest wolny od błędów. W systemach baz danych jednocześnie wykonuje się kilka lub kilkadziesiąt transakcji, kilka z nich może chcieć jednocześnie operować na tych samych danych. Dużym problemem jest również sytuacja, w której w trakcie trwania transakcji następuje awaria systemu. Aby w wyniku działania transakcji w bazie danych nie powstał „bałagan" oraz by wszystkie mogły się wykonać poprawnie, system musi zapewnić by transakcje spełniały tzw. własności **ACID** [1]:

- *niepodzielność (atomicity)* – transakcja wykona się albo w całości albo wcale,
- *spójność (consistency)* – żadna transakcja nie naruszy spójności bazy, czyli pewnych związków między danymi (więzy integralności) określonymi przez klucze czy ograniczenia takie jak: wartość minimalna/maksymalna, unikalność wartości itp.
- *izolacja (isolation)* – każda transakcja musi przebiegać tak, jakby żadna inna transakcja w tym czasie się nie wykonywała,
- *trwałość (durability)* – wynik działania zakończonej transakcji nie zostanie utracony.

Za spełnienie powyższych własności oraz zapewnienie porządku w zachowaniu transakcji jest odpowiedzialna specjalna jednostka zwana *menedżerem transakcji*.

Transakcja służy nie tylko do uporządkowania wykonywania się operacji na bazie danych, stanowi również podstawową *jednostkę odtwarzania* systemu bazy danych.

Jak wspomniałam we wstępie, system komputerowy jest narażony na awarię, a co za tym idzie, zagrożone są bazy danych znajdujące się na nim.

Awarie, mogą być różnego typu:[2]

- awarie systemu – wynikające z wad sprzętu lub błędów oprogramowania, wpływają na pamięć operacyjną,
- awarie nośników – powodują utratę części pamięci operacyjnej,
- błąd oprogramowania aplikacji – np. logiczne błędy w programie odwołującym się do bazy danych, powodujące awarię jednej lub kilku transakcji,
- naturalne katastrofy – przypadkowe wyłączenie komputera, pożar,
- nieostrożność,
- sabotaż.

Jeśli awaria spowodowała tylko utratę pamięci operacyjnej i jest możliwość odczytu danych z dysku, to dzięki mechanizmom transakcyjnych jest możliwość odtworzenia spójnego stanu bazy danych sprzed awarii, a nawet przywrócenia danych utraconych z pamięci operacyjnej. Aby lepiej to zrozumieć prześledźmy działanie transakcji.

1.2.Działanie transakcji

Przy działaniu wszelkich operacji na bazie mamy do czynienia z danymi znajdującymi się na dysku oraz danymi znajdującymi się w przestrzeni wirtualnej.

Jeśli chcemy wykonać transakcję, która będzie czytała element bazy danych, modyfikowała, a następnie zapisywała na dysk, wówczas muszą zostać wykonane następujące czynności:

- skopiowanie do bufora elementu bazy danych,
- skopiowanie elementu z bufora do zmiennej lokalnej w przestrzeni adresowej transakcji,
- wykonanie operacji na zmiennej lokalnej,
- skopiowanie wartości elementu ze zmiennej lokalnej do bufora,
- zrzucenie zawartości bufora na dysk.

Żeby móc dalej szczegółowo rozważać przebieg transakcji i zrozumieć proces odtwarzania zdefiniujmy operacje pierwotne odpowiedzialne za wykonywanie wyżej wymienionych czynności:

- INPUT(X) – kopiowanie bloku dysku, zawierającego element bazy danych X, do bufora pamięci,
- OUTPUT(X) – skopiowanie bufora zawierającego X na dysk,
- READ(X,t) – kopiowanie elementu bazy danych X z bufora do lokalnej zmiennej transakcji t,
- WRITE(X,t) – kopiowanie wartości lokalnej zmiennej t do elementu bazy danych X w buforze pamięci.

Polecenia READ i WRITE pochodzą z menedżera transakcji, natomiast INPUT i OUTPUT wydaje ta struktura bazy danych (często zwana menadżerem buforów), która odpowiada za kopiowanie danych z dysku do bufora i z bufora na dysk. Działanie tych operacji obrazuje następujący przykład:

Przykład 1.1.

Niech transakcja T wykonuje dwie następujące operacje na elementach bazy danych A i B:

$A:=A*2;$

$B:=B*2;$

i niech jedynym warunkiem spójności bazy jest równość $A=B$.

Wykonanie tej transakcji obejmuje przeczytanie zawartości elementów danych *A* i *B* z dysku i skopiowanie ich do bufora. Następnie menedżer transakcji wczytuje te wartości do zmiennych lokalnych, wykonuje na nich obliczenia i wyniki przesyła z powrotem do bufora. Dalej elementy te są kopiowane na dysk. Ciąg tych czynności i ich efekty można przedstawić następująco:

Kroki	***Czynności***	***t***	***Pamięć A***	***Pamięć B***	***Dysk A***	***Dysk B***
1)	INPUT(*A*)		8		8	8
2)	INPUT(*B*)		8	8	8	8
3)	READ(*A*,*t*)	8	8	8	8	8
4)	*t*:=*t**2	16	8	8	8	8
5)	WRITE(*A*,*t*)	16	16	8	8	8
6)	READ(*B*,*t*)	8	16	8	8	8
7)	*t*:=*t**2	16	16	8	8	8
8)	WRITE(*B*,*t*)	16	16	16	8	8
9)	OUTPUT(*A*)	16	16	16	16	8
10)	OUTPUT(*B*)	16	16	16	16	16

Przeniesienie zawartości bufora z powrotem na dysk, czyli wykonanie instrukcji OUTPUT nie zawsze dzieje się natychmiastowo. Względy optymalizacyjne często decydują o tym, że moment ten jest odraczany w czasie. Należy jednak pamiętać, że zbyt długie przetrzymywanie danych w buforze zwiększa prawdopodobieństwo ich utraty, ze względu na możliwość wystąpienia awarii.

1.3.Menadżer transakcji

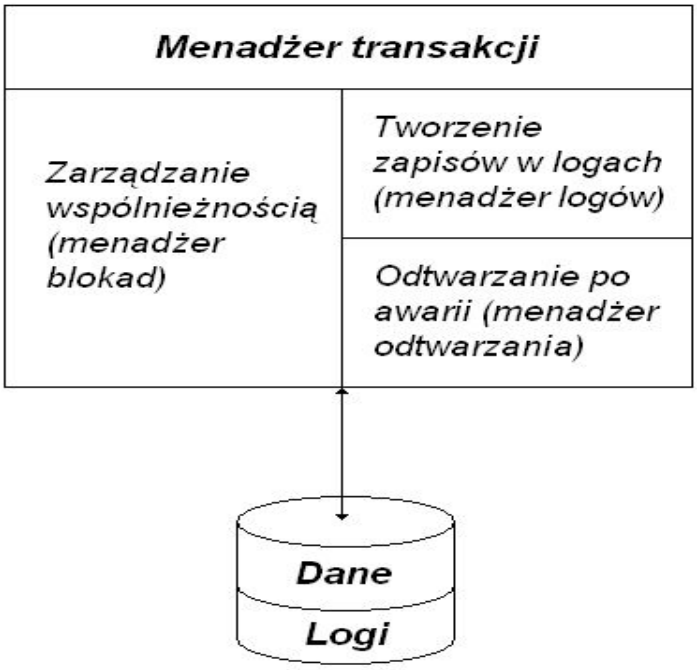
Menadżer transakcji jest to podsystem, który zajmuję się zapewnieniem poprawności wykonania transakcji. *Zasada poprawności* mówi, że jeśli transakcja wykonuje się osobno przy spójnym stanie bazy i nie występują żadne błędy systemowe to po zakończeniu transakcji stan bazy jest również spójny[1]. Ta zasada wymusza od menedżera transakcji spełnienia własności ACID transakcji z wyjątkiem trzeciej - *izolacji*.

Izolacja musi być spełniona przez menedżera w przypadku systemów wielodostępowych, z

możliwością jednoczesnego wykonywania się kilku transakcji. Problemem jest tutaj możliwa chęć dostępu do tych samych danych w jednym czasie. Menadżer transakcji zarządza przydziałem danych transakcjom, ustala które z nich należy wstrzymać dłużej, a którym przydzielić dane w pierwszej kolejności. Menadżer musi więc sprawnie zarządzać współbieżnością transakcji i, ze względu na to, że najbardziej popularnym rozwiązaniem stosowanym do rozwiązania tego problemu jest mechanizm blokad, tą funkcję menedżera nazywa się *menadżerem blokad*.

Kolejną, wspomnianą już funkcją jest zapewnienie odporności na awarie systemu. W przykładzie 1.1, gdyby awaria nastąpiła między 9) a 10) krokiem, wówczas zawartość bufora zostałaby utracona i wartości elementów *A* i *B* w bazie byłyby różne, czyli stan bazy byłby niespójny. By uniknąć takich sytuacji część menadżera, zwana często *menedżerem logów*, tworzy na dysku tzw. *logi*, w których zapisuje informacje dotyczące aktywności każdej transakcji od momentu rozpoczęcia aż do zakończenia. Dzięki tym zapisom tzw. *menedżer odtwarzania* jest w stanie w przypadku awarii systemu odtworzyć stan systemu sprzed awarii i zdecydować co należy zrobić z przerwanymi transakcjami, czy należy je unieważnić i usunąć z bazy skutki ich częściowego działania, czy należy je dokończyć. Decyzja ta zależy głównie od tego, w jaki sposób został zaimplementowany log. W niniejszym rozdziale przedstawię różne rodzaje logów i systemów odtwarzania.

Poniższy rysunek przedstawia ogólny schemat menedżera transakcji:



Rys. 1.1. Schemat menedżera transakcji, źródło własne

1.4.Rodzaje logów i zachowanie się menedżera odtwarzania.

Za zarządzanie logiem i dokonywaniem w nim zapisów odpowiedzialny jest menedżer logów. Na podstawie tych zapisów, w sytuacji awarii, menedżer odtwarzania jest w stanie sprawdzić na jakim etapie prac transakcje zostały przerwane i zdecydować co należy zrobić ze zmodyfikowanymi przez nie danymi. Ze względu tak ważną rolę, jaką odgrywa log w procesie odtwarzania, wskazane jest by pliki te były zapisywane w co najmniej dwóch różnych miejscach. Z drugiej strony, w logach przechowywane jest bardzo wiele informacji i rozmiar pliku może przewyższać rozmiar samej bazy, stąd czasami zachodzi konieczność usuwania starych logów bądź archiwizowania ich w pamięci zewnętrznej.

Do podstawowych zapisów w logu należą:

- <START *T*> - transakcja T rozpoczęła się,
- <COMMIT *T*> - transakcja T zakończyła się pomyślnie,
- <ABORT *T*> - transakcja T nie powiodła się.

Oprócz tych zapisów istnieje jeszcze tzw. *zapis aktualizacj*i i *punkty kontrolne*, ale ich postać zależy od rodzaju logu i będzie omówiona szczegółowo przy opisie kolejnych rodzajów logów. Często w logu umieszcza się dodatkowo czas rozpoczęcia i zakończenia transakcji.

Log może być również wykorzystywany do innych celów niż odtwarzanie. Jednym z nich jest celowe wycofywanie całej (lub częściowej) transakcji oraz jej powtarzanie przez użytkownika. Jest bardzo cenna funkcja, gdyż dzięki niej możliwe jest szybki naprawienie błędu użytkownika, np. przywrócenie przypadkowo usuniętego wiersza. Czasem wykorzystuje się log do monitorowania systemu, wówczas zapisuje się tam dodatkowe informacje dotyczące np. logowania użytkowników.

1.4.1.Log z unieważnianiem

Konstrukcja logów z unieważnianiem oraz mechanizmów odtwarzania opiera się na założeniu, że jeśli nie ma pewności, że transakcja wykonała się w całości i jej wyniki zostały w całości zapisane na dysku, to wszystkie zmiany, które mogły za jej sprawą zajść w bazie danych, są unieważniane i jest przywracany stan bazy danych sprzed wykonania się transakcji.

Zapis aktualizacji wygląda następująco:

- *<T,X,v>* - zapis ten oznacza, że transakcja *T* zmienia wartość elementu bazy danych *X*, a jego poprzednia wartość wynosiła *v*

To właśnie na podstawie tego zapisu system jest w stanie odtworzyć stan bazy danych sprzed wykonania transakcji, gdyż stara wartość jest przechowywana w tym zapisie w logu. Należy również pamiętać o tym, że zapis ten pojawia się w logu w wyniku wykonania przez transakcję operacji WRITE, a nie OUTPUT. Oznacza, to że pojawi się on w logu zanim zmiana bazy danych zostanie dokonana na dysku. W związku z tym, w sytuacji awarii, menedżer nie sprawdza czy transakcja zapisała już zmiany na dysku, tylko przypisuje starą wartość *v* elementu *X*.

Aby stosować ochronę awaryjną za pomocą logu z unieważnianiem transakcje muszą stosować się do następujących reguł:

U1: *Jeśli element bazy danych X został zmieniony w wyniku transakcji T, to zapis w logu postaci <T,X,v> musi pojawić się na dysku, zanim nowa wartość X zostanie zapisana na dysk.*

U2*: Jeśli transakcja kończy się pomyślnie, to w logu zapis COMMIT musi pojawić się na dysku, ale tylko po zapisaniu na dysk zmian wykonanych w bazie przez transakcję, tak szybko, jak jest to możliwe.*[1]

Podsumowując, transakcja powinna dokonywać zmian na dysku w następującej kolejności:

a) zapisy aktualizujące w logu,

b) zmienione elementy bazy danych,

c) zapis COMMIT.

Aby lepiej zrozumieć działanie logów z unieważnianiem prześledzimy kolejny przykład.

Przykład 1.2.

Transakcja T będzie wykonywać te same obliczenia na elementach bazy danych co transakcja z przykładu 1.1. Ponieważ przy logach z unieważnianiem istotna jest kolejność zapisywania zmian z bufora na dysk, wprowadzona zostanie dodatkowa operacja FLUSH LOG. Jest to instrukcja nakazująca, by menedżer buforów skopiował na dysk wszystkie bloki z logami, które jeszcze nie były kopiowane albo od czasu ostatniego kopiowania były zmienione.

Krok	*Czynność*	*t*	*Pamięć A*	*Pamięć B*	*Dysk A*	*Dysk B*	*Log*
1)					8	8	<START *T*>
2)	INPUT(*A*)		8		8	8	
3)	INPUT(*B*)		8	8	8	8	
4)	READ(*A,t*)	8	8	8	8	8	
5)	*t*:=*t**2	16	8	8	8	8	
6)	WRITE(*A,t*)	16	16	8	8	8	<*T,A,8*>
7)	READ(*B,t*)	8	16	8	8	8	
8)	*t*:=*t**2	16	16	8	8	8	
9)	WRITE(*B,t*)	16	16	16	8	8	<*T,B,8*>
10)	FLUSH LOG	16	16	16	8	8	
11)	OUTPUT(*A*)	16	16	16	16	8	
12)	OUTPUT(*B*)	16	16	16	16	16	
13)							<COMMIT *T*>
14)	FLUSH LOG						

Kolumna „Log” w powyższej tabeli obrazuje zawartość logu. Pierwszą instrukcją jaką wykonuje transakcja jest zapis <START *T*> w logu informujący o rozpoczęciu działania transakcji. Kolejne zapisy w logu to <*T,A,8*> i <*T,B,8*>. W momencie kiedy menedżer transakcji dokonuje zmian na elementach bazy danych, czyli wykonuje operację WRITE, wysyła sygnał do menedżera logów, aby ten w logu umieścił zapis aktualizujący (w przykładzie krok 6 i 9). Jak na razie zapisy te są umieszczone w buforze. Dopiero po wykonaniu jawnej instrukcji FLUSH LOG (krok 10) są one kopiowane na dysk. Gdy zapisy logów znajdują się już na dysku, transakcja może rozpocząć kopiowanie zmienionych elementów *A* i *B* na dysk. Zaraz po tym menedżer transakcji informuje menedżera logów o zakończeniu transakcji, ten zapisuje w logu <COMMIT *T*> i wysyła log na dysk.

Odtwarzanie:

Jeśli w trakcie działania transakcji T następuje awaria, to po ponownym wznowieniu pracy bazy danych zostaje uruchamiany proces odtwarzający. W przypadku logu z unieważnianiem proces ten polega na wycofaniu przerwanej transakcji i usunięciu ewentualnych skutków jej działania.

Menedżer odtwarzania analizuje log znajdujący się na dysku od końca. Jeśli znajduje się

tam zapis <COMMIT *T*> wówczas wie, że transakcja *T* zakończyła się pomyślnie, wobec tego nie musi nic robić.

Jeśli zapisu <COMMIT *T*> nie ma, a istnieją zapisy aktualizujące, to znaczy że awaria nastąpiła między krokiem 10) a 14). Wówczas może się zdarzyć, że pewne zmiany na elementach bazy danych zostały już dokonane na dysku. Menedżer odtwarzania nie sprawdza, czy i które operacje OUTPUT zostały wykonane. Brak zapisu <COMMIT *T*> dla menedżera odtwarzania jest informacją, że transakcja nie zakończyła się i że trzeba ją wycofać. Na podstawie zapisów aktualizacji menedżer odtwarzania przypisuje elementom *A* i *B* starą wartość $v = 8$ (nawet jeśli awaria nastąpiła tuż przed krokiem 11) i stan bazy nie został naruszony) oraz umieszcza w logu zapis <ABORT *T*> i przesyła go na dysk.

Gdy awaria następuje podczas wykonywania przez transakcję kroków od 1) do 9) wówczas log nie został jeszcze skopiowany na dysk, stąd menedżer odtwarzania nic nie musi robić. Wiadomo, że wszelkie dotychczasowe operacje miały miejsce tylko w buforze, czyli stan bazy na dysku został nienaruszony.

Co się stanie jeśli w trakcie odtwarzania znowu nastąpi awaria? Nie stanie się nic. Po kolejnej awarii znowu sprawdzany jest log. Jeśli znajdują się tam zapis <COMMIT *T*> lub <ABORT *T*> to wiadomo, że wszystko jest w porządku, a jeśli nie ma tych zapisów przystępuje do unieważniania transakcji *T*.

Punkty kontrolne

Jak już wspomniałam wcześniej w logu mogą pojawić się tzw. *punkty kontrolne*. Jak wiemy, w systemie może jednocześnie działać wiele transakcji i zapisy dotyczące ich działania są zapisywane w jednym logu. Wobec tego, w przypadku awarii, menedżer odtwarzania musi przeszukać cały log by sprawdzić czy dla każdej transakcji istnieje zapis <COMMIT> lub <ABORT>. Jeśli baza danych pracuje już dłuższy czas, log staje się dość pokaźnym plikiem i przeszukiwanie go może być zbyt czasochłonne. Najprostszy sposób rozwiązania tego problemu polega na tworzeniu punktów kontrolnych, które byłyby wstawiane co jakiś czas i które dawałyby pewność, że wszystkie zapisy znajdujące się przed ostatnim punktem kontrolnym dotyczą transakcji poprawnie zakończonych.

Wstawianie punktów kontrolnych polega na:[1]

1. wstrzymaniu uruchomień nowych transakcji,
2. zaczekaniu, aż wszystkie rozpoczęte transakcje wprowadzą do logu zapisy COMMIT lub ABORT,

3. przesłaniu logu na dysk,
4. wprowadzeniu do logu zapisu <CKPT> i ponownym przesłaniu logu na dysk ,
5. wznowieniu wstrzymanych transakcji.

Teraz, gdy menedżer odtwarzania rozpocznie sprawdzanie logu od końca i natknie się na punkt kontrolny <CKPT>, wie że wszystkie transakcje, które rozpoczęły swe działanie przed wprowadzeniem tego zapisu zostały już sprawdzone i zakończone. Stąd menedżera interesują tylko zapisy od ostatniego punktu kontrolnego do końca pliku. Chcąc oszczędzić miejsce na dysku, dane sprzed punktu kontrolnego można by usunąć.

Przykładowy log typu unieważnianie z punktami kontrolnymi może wyglądać tak:

Przykład 1.3.

1) <START T_1 >
2) < T_1 , *A, 5*>
3) <START T_2 >
4) < T_2 , *B, 10*>
5) < T_2 , *C, 15*>
6) < T_1 , *D, 20*>
7) <COMMIT T_1 >
8) <COMMIT T_2 >
9) <CKPT>
10) <START T_3 >
11) < T_3 , *E, 10*>
12) < T_3 , *F, 25*>

Dla procesu odtwarzania istotne są zapisy po <CKPT>.

Problem tej techniki polega na tym, że aby wstawić punkt kontrolny należy wstrzymać system i zaczekać na zakończenie wszystkich rozpoczętych transakcji, co może trochę potrwać. W tym czasie system dla nowych transakcji jest właściwie nieaktywny.

Dużo lepszym rozwiązaniem jest stosowanie tzw. *bezkonfliktowych punktów kontrolnych*. Jest to technika, która nie ogranicza rozpoczynania nowych transakcji.

Aby utworzyć bezkonfliktowe punkty kontrolne wykonuje się kolejno:[1]

1. wprowadzenie do logu zapisu <START CKPT(T_1 ,..., T_k)> i przesłanie logu na dysk. T_1 ,..., T_k to wszystkie transakcje aktywne
2. zaczekanie na zakończenie transakcji T_1 ,..., T_k , ale dopuszczenie rozpoczęcia się innych transakcji

3. gdy wszystkie transakcje T_1 ,..., T_k zakończą się, wprowadza się do logu zapis <END CKPT> i przesyła log na dysk.

Mając log z bezkonfliktowymi punktami kontrolnymi menedżer odtwarzania również przeszukuje plik od końca w poszukiwaniu transakcji nie zakończonych.

Jeśli najpierw znajdzie zapis <END CKPT> to znaczy, że wszystkie transakcje nie zakończone rozpoczęły swe działanie po zapisie <START CKPT(T_1 ,..., T_k)>, a transakcje T_1 ,..., T_k na pewno się zakończyły. Zatem menedżer transakcji może przeszukiwać log od ostatniego zapisu <START CKPT(T_1 ,..., T_k)>.

Jeśli natomiast najpierw natknie się na zapis <START CKPT(T_1 ,..., T_k)>, to znaczy, że jedynymi transakcjami niezakończonymi mogą być transakcje ze zbioru T_1 ,..., T_k i te które rozpoczęły swe działanie po zapisie <START CKPT(T_1 ,..., T_k)>. Dlatego menedżer odtwarzania nie musi się cofać dalej niż do momentu rozpoczęcia najwcześniejszej z tych transakcji.

Przykład 1.4.

1) < START T_1 >
2) < T_1 , *F, 10* >
3) < START T_2 >
4) < T_1 , *E ,15* >
5) < T_2 , *A, 5* >
6) < START T_3 >
7) < T_3 , *B,10* >
8) < T_1 , *D, 5* >
9) < COMMIT T_2 >
10) < START T4 >
11) < COMMIT T_1 >
12) < T_4 , *C, 5*>
13) **< START CKPT(** T_3 **,** T_4 **) >**
14) < T_4 , *G, 20* >
15) < START T_5 >
16) < COMMIT T_3 >
17) < T_5 , *H, 10* >
18) < COMMIT T_4 >
19) **< END CKPT >**
20) < T_5 , *A, 20* >

W powyższym przykładzie w momencie rozpoczęcia wstawiania punktu kontrolnego

jedynymi transakcjami aktywnymi są transakcje T_3 i T_4 . W przedziale kontrolnym (wiersze 13-19) rozpoczęła się jeszcze jedna transakcja T_5 . Gdyby po awarii w logu znajdowałyby się zapisy jak powyżej, wówczas menedżer odtwarzania w poszukiwaniu transakcji cofnąłby się do linii 13. Zainteresowałby się wówczas tylko transakcją T_5 , a zapisy dotyczące innych transakcji by pominął, gdyż transakcje T_3 i T_4 są wymienione w początku punktu kontrolnego i wiadomo, że zostały zakończone. Jeśli w logu byłoby tylko 18 linii (tzn. nie byłoby zapisu <END CKPT>) wówczas menedżer odtwarzania w przeszukiwaniu logu cofnąłby się do początku najwcześniejszej z transakcji wymienionej w punkcie kontrolnym, w naszym przykładzie do linii 6), gdzie rozpoczyna się transakcja T_3 .

1.4.2.Log z powtarzaniem

Innym sposobem prowadzenia logów są tzw. logi z powtarzaniem. Podstawowymi własnościami różniącymi je od logu z unieważnianiem jest to, że w sytuacji odtwarzania systemu po awarii ignoruje się transakcje niezatwierdzone, natomiast powtarza się transakcje zatwierdzone. Ma to sens, ponieważ przy logach z powtarzaniem zapis danych na dysku i zapis logu na dysku następuje w innej kolejności niż to miało miejsce przy logu z unieważnianiem. Stosując technikę powtarzania zapis COMMIT zostaje zapamiętany, zanim zmienione wartości zapisze się na dysku. W tej sytuacji w logu nie występują stare wartości tylko nowe, stąd inne jest znaczenie zapisu aktualizującego:

- *<T,X,v>* - transakcja *T* zapisała w elemencie bazy danych *X* nową wartość *v*

Stosując technikę logu z powtarzaniem należy trzymać się jednej zasadniczej reguły:

> ***R1:*** *Zanim zmiana elementu bazy danych X zostanie zapamiętana na dysku, wszystkie zapisy logu odnoszące się do tej zmiany X, zarówno zapisy zmiany <T,X,v>, jak i zapis <COMMIT T> muszą wystąpić na dysku.*[1]

Stosując się do powyższej reguły wszelkie zmiany na dysku będą zapisywane w następującej kolejności:[1]

1. zapis w logu wskazujący na zmieniany element,
2. zapis <COMMIT *T*>,
3. zmiany elementów w bazie.

Przebieg przykładowej transakcji (rozważanej już w poprzednio) będzie wyglądał

następująco:

Przykład 1.5.

Krok	***Czynność***	***t***	***Pamięć A***	***Pamięć B***	***Dysk A***	***Dysk B***	***Log***
1)					8	8	<START *T*>
2)	INPUT(*A*)		8		8	8	
3)	INPUT(*B*)		8	8	8	8	
4)	READ(*A*,*t*)	8	8	8	8	8	
5)	*t*:=*t**2	16	8	8	8	8	
6)	WRITE(*A*,*t*)	16	16	8	8	8	<*T,A,16*>
7)	READ(*B*,*t*)	8	16	8	8	8	
8)	*t*:=*t**2	16	16	8	8	8	
9)	WRITE(*B*,*t*)	16	16	16	8	8	<*T,B,16*>
10)		16	16	16	8	8	<COMMIT *T*>
11)	FLUSH LOG	16	16	16	8	8	
12)	OUTPUT(*A*)	16	16	16	16	8	
13)	OUTPUT(*B*)	16	16	16	16	16	

Przebieg tej transakcji różni się od tego przedstawionego w przykładzie 2 tym, że w zapisach aktualizacyjnych jest nowa wartość oraz log jest przesyłany na dysk tylko raz, zaraz po pojawieniu się zapisu COMMIT.

Odtwarzanie:

Istotą logu z powtarzaniem przy odtwarzaniu jest fakt, że jeśli w logu nie ma zapisu <COMMIT *T*>, to mamy pewność, że zmiany w bazie danych nie zostały dokonane i menedżer może uznać taką transakcję za niepomyślnie zakończoną. Natomiast jeśli zapis COMMIT występuje wtedy może się zdarzyć, że część zmian została zapisana na dysku, a część nie. Wówczas następuje powtórzenie zapisu zmian, nawet jeśli zostały one całkowicie zapisane przed awarią.

Prześledźmy możliwe sytuacje awarii w naszym przykładzie.

Jeżeli awaria nastąpiła po kroku 11) to zapis <COMMIT *T*> z pewnością znajduje się na dysku. Menedżer odtwarzania sprawdza log od początku do przodu i przy zapisach <*T,A,16*> i <*T,B,16*> ponownie zapisze wartość 16 na elementach *A* i *B*.

Jeśli awaria miałaby miejsce w kroku 11) i zapis <COMMIT *T*> znalazłby się na dysku to postępowanie jest takie samo jak powyżej. Jeśli natomiast występuje brak tego zapisu wówczas menedżer odtwarzania uznaje transakcję *T* za niezakończoną i wpisuje do logu zapis <ABORT *T*>. Gdyby awaria miała miejsce wcześniej, to również transakcja zostałaby unieważniona.

W zachowaniu menedżera odtwarzania istotnym jest, że przeszukuje log od początku do końca. Mogło się bowiem zdarzyć, że kilka transakcji zatwierdzonych pisało na elemencie *X*. Wobec tego menedżer odtwarzania musi powtórzyć wszystkie przypisania w takiej kolejności w jakiej transakcje pisały. W przypadku odtwarzania z unieważnianiem sytuacja była odwrotna, log byłby przeszukiwany od końca do początku. Dzięki temu wartość elementu *X* byłaby przywracana zgodnie ze stanem sprzed rozpoczęcia transakcji.

Punkty kontrolne

Stosowanie punktów kontrolnych nie może się odbywać tak samo jak to miało miejsce w logach z unieważnianiem. Problem jest następujący: ponieważ transakcje zatwierdzone mogą być skopiowane na dysk znacznie później niż nastąpiło ich zatwierdzenie, więc przy wstawianiu punktu kontrolnego nie można się ograniczać tylko do transakcji aktywnych. Wobec tego przy wstawianiu punktu kontrolnego należy zapamiętać na dysku wszystkie te elementy bazy danych, które zostały zmienione przez transakcje zatwierdzone, ale nie zostały wcześniej przesłane na dysk. Nakłada to dodatkowy obowiązek dla menedżera buforów śledzenia tzw. „brudnych" buforów, czyli tych, z których wartości nie zostały jeszcze zapamiętane na dysku. Natomiast przy wstawianiu zakończenia bezkonfliktowego punktu kontrolnego nie trzeba czekać na zatwierdzenie transakcji aktywnych, gdyż i tak żadne aktywne transakcje nie zostaną w tym czasie zapisane na dysku.

Podsumowując, przy wstawianiu bezkonfliktowych punktów kontrolnych w logu z powtarzaniem należy wykonać następujące czynności:[1]

1. wprowadzić do logu zapis <START CKPT(T_1 ,..., T_k)>, gdzie T_1 ,..., T_k są wszystkimi transakcjami aktywnymi i zapisać log na dysku
2. zapamiętać na dysku wszystkie elementy bazy danych, które znajdują się w buforach, ale nie zostały zapamiętane na dysku przed zapisaniem do logu START CKPT
3. zapisać w logu <END CKPT> i przesłać log na dysk

Przykład 1.6.

1) < START T_1 >
2) < T_1 , *A, 5* >
3) < START T_2 >
4) < COMMIT T_1 >
5) < T_2 , *B, 10*>
6) < START CKPT(T_2) >
7) < T_2 , *C, 15* >
8) < START T_3 >
9) < T_3 , *D, 20* >
10) < END CKPT >
11) < COMMIT T_2 >
12) < COMMIT T_3 >

W powyższym przykładzie jedyną transakcją zatwierdzoną przed wstawieniem punktu kontrolnego jest transakcja T_1 . Jeśli zmieniona wartość *A* nie została jeszcze skopiowana na dysk, należy to uczynić jeszcze przed wstawieniem zakończenia punktu kontrolnego.

Przy odtwarzaniu za pomocą logu z bezkonfliktowymi punktami kontrolnymi mogą się zdarzyć następujące sytuacje:

- Jeśli w logu znajduje się zapis <END CKPT> to wiadomo, że wszystkie zmiany transakcji zatwierdzonych przed wstawieniem początku punktu kontrolnego zostały już zapisane (w naszym przykładzie transakcja T_1 została zapisana). Wobec tego przeszukiwanie logu można ograniczyć do najwcześniejszego z zapisów <START T_i >, gdzie T_i jest transakcją ze zbioru transakcji aktywnych wymienionych w zapisie <START CKPT(T_1 ,..., T_k)>. Od tego miejsca menedżer odtwarzania postępuje zgodnie z zasadami logu z powtarzaniem.
- Jeśli natomiast w logu istnieje zapis <START CKPT(T_1 ,..., T_k)>, a po nim nie występuje zapis zamykający punkt kontrolny, wówczas nie możemy mieć pewności, czy zmiany dokonane przez transakcje zatwierdzone przed punktem kontrolnym zostały zapisane na dysku. Wobec tego należy odszukać wstecz poprzedni zapis <END CKPT> i odpowiadający mu zapis <START CKPT(S_1 ,..., S_n)> i powtórzyć zachowanie jak powyżej.

W naszym przykładzie jeśli awaria nastąpiłaby po kroku 12), to powtórzone by zostały transakcje T_2 i T_3 , jeśli między 11) a 12) to transakcja T_2 zostałaby powtórzona a

transakcja T3 unieważniona i zostałby wpisany do logu zapis <ABORT T_3 >, jeśli natomiast ostatnim zapisem jest <END CKPT> to zarówno transakcja T_2 jak i T_3 zostałyby unieważnione. Jeżeli natomiast awaria miałby miejsce przed krokiem 10), wówczas menedżer szukałby poprzedniego punktu kontrolnego. Ponieważ w naszym przykładzie go nie ma przeszukałby log od początku i powtórzył transakcję T_1 , gdyż została ona zatwierdzona, natomiast unieważniłby transakcje T_2 i T_3 .

1.4.3.Logi typu unieważnianie/powtarzanie

Kolejnym typ logu łączy niejako dwa poprzednie typy, gdyż w zapisach aktualizacyjnych przechowuje zarówno starą jak i nową wartość. Zaletą tego logu, przewyższającą dwa poprzednie, jest fakt, że kolejność zapisywania logu i danych na dysk nie jest tak istotna jak w pozostałych przypadkach.

Log typu unieważnianie/powtarzanie różni się tylko zapisem aktualizującym, mającym teraz postać:

- *<T,X,v,w>* - transakcja *T* zmieniła element bazy danych *X*, poprzednia jego wartość wynosiła *v*, natomiast nowa wartość wynosi *w*

  System, który chce pisać do takiego logu musi stosować się do następującej reguły:

  UR1: *zanim na dysku zapisze się zmianę wartości elementu X, spowodowaną działaniem pewnej transakcji T, należy najpierw na dysk wprowadzić zapis <T,X,v,w>*

  Reguła ta nie wymusza, kiedy ma zostać zapisany na dysk <COMMIT *T*>.

Przykład 1.7.

Krok	***Czynność***	***t***	***Pamięć A***	***Pamięć B***	***Dysk A***	***Dysk B***	***Log***
1)					8	8	<START *T*>
2)	INPUT(*A*)		8		8	8	
3)	INPUT(*B*)		8	8	8	8	
4)	READ(*A*,*t*)	8	8	8	8	8	
5)	*t*:=*t**2	16	8	8	8	8	
6)	WRITE(*A*,*t*)	16	16	8	8	8	*<T,A,8,16>*

Krok	*Czynność*	*t*	*Pamięć A*	*Pamięć B*	*Dysk A*	*Dysk B*	*Log*
7)	READ(*B,t*)	8	16	8	8	8	
8)	*t*:=*t**2	16	16	8	8	8	
9)	WRITE(*B,t*)	16	16	16	8	8	<*T,B,8,16*>
10)	FLUSH LOG	16	16	16	8	8	
11)	OUTPUT(*A*)	16	16	16	16	8	
12)		16	16	16	16	8	<COMMIT *T*>
13)	OUTPUT(*B*)	16	16	16	16	16	

W przypadku logu unieważniane/powtarzanie musi zapisy zmian muszą zostać przesłane na dysk przed zapisaniem zmian elementów i przed wprowadzeniem do logu zapisu COMMIT. Nie jest istotne natomiast, kiedy to zatwierdzenie nastąpi. Dlatego zapis <COMMIT *T*> mógłby równie dobrze nastąpić przed krokiem 11) albo po 13). Nie jest również określone, kiedy ma nastąpić przesłanie zapisu COMMIT na dysk. Może to powodować pewne problemy związane z tym, że jeżeli menedżer buforów przez długi czas nie przesyłałby logu na dysk i nastąpiłaby awaria, wówczas transakcje zatwierdzone musiałby zostać po pewnym czasie unieważnione, co może powodować problem z poinformowaniem użytkownika systemu o wycofaniu transakcji. Jeśli takie opóźnione zatwierdzenie jest problemem, wówczas należy przy logach z unieważnianiem/powtarzaniem stosować dodatkową regułę:

***UR2**: zapis <COMMIT T> należy przesłać na dysk, natychmiast, gdy zostanie wprowadzony do logu.*

Do naszego przykładu należałoby wówczas między krokiem 12) a 13) dodać krok z instrukcją FLUSH LOG.

Odtwarzanie

Strategia odtwarzania za pomocą logów typu unieważnianie/powtarzanie polega na:

1. powtórzeniu wszystkich transakcji zakończonych, zaczynając od najwcześniejszej z nich,
2. unieważnieniu wszystkich niezakończonych transakcji, zaczynając od najpóźniejszej.

Wykonanie tych dwóch kroków jest możliwe, gdyż w logu przechowywane są zarówno stare jak i nowe wartości elementów bazy danych. Ponadto, wykonanie obydwu tych kroków jest konieczne ze względu na dowolność momentu wpisania zapisu COMMIT do logu i kopiowania zmian bazy danych na dysk. Stąd jest możliwa sytuacja, że zatwierdzone

transakcje i zmiany bazy danych będą przechowywane tylko w buforze albo niezatwierdzone transakcje i zmiany znajdą się na dysku.

Prześledźmy możliwe sytuacje:

- Jeżeli awaria nastąpiłaby zaraz po przesłaniu <COMMIT *T*> na dysk, to menedżer odtwarzania nie wie, czy wszystkie zmiany zostały zapisane, gdyż nie wiadomo, kiedy to zatwierdzenie nastąpiło(mogło ono nastąpić zarówno przed krokiem 11) jak i po kroku 13)). Wobec tego transakcja *T* zostanie zidentyfikowana jako zatwierdzona,a następnie powtórzona.
- Jeżeli zapisu <COMMIT *T*> nie ma w logu, wówczas transakcja *T* zostanie uznana za niezatwierdzoną, mimo iż zmiany elementów mogły już zostać zapisane w bazie danych. W takiej sytuacji transakcja *T* zostanie unieważniona i elementom *A* i *B* zostanie ponownie przypisana wartość 8.

Punkty kontrolne.

Aby wstawić bezkonfliktowe punkty kontrolne w logach typu unieważnianie/powtarzanie należy:

1. wstawić do logu zapis <START CKPT(T_1 ,..., T_k)>, gdzie T_1 ,..., T_k to wszystkie transakcje aktywne i przesłać log na dysk,
2. zapisać na dysku wszystkie bufory zawierające zmienione elementy bazy danych, bez względu czy zostały zmienione przez transakcje zatwierdzone czy niezatwierdzone,
3. zapisać do logu zapis <END CKPT> i przesłać log na dysk.

Ze względu na punkt 2 należy również przyjąć założenie, że

Transakcja nie może wprowadzić nowej wartości (nawet do bufora pamięci), jeśli nie jest pewne, że nie zostanie odwołana.

Przykład 1.8.

1) < START T_1 >
2) < T_1 , *A, 4, 5* >
3) < START T_2 >
4) < COMMIT T_1 >
5) < T_2 , *B, 9,10* >
6) < START CKPT(T_2) >
7) < T_2 , *C, 14, 15* >
8) < START T_3 >

9) < T_3 , *D, 19, 20*>
10) < END CKPT >
11) < COMMIT T_2 >
12) < COMMIT T_3 >

Jeżeli awaria nastąpiłaby, a w logu znalazłby się zapis <END CKPT> wówczas menedżer odtwarzania musiałby się cofnąć tylko do początku punktu kontrolnego, czyli do linii 6). Jeśli istnieje w logu również zapis <COMMIT T_2 >, to transakcja T_2 zostanie powtórzona, ale tylko od linii 6), bo wiadomo, że wszystkie wcześniejsze zmiany zostały zapisane przy wstawianiu punktu kontrolnego. Podobnie, jeśli byłby zapis <COMMIT T_3 >, to transakcja T_3 zostałaby powtórzona. Jeśli natomiast w logu byłoby tylko 10 pierwszych linii, wówczas menedżer odtwarzania musiałby unieważnić transakcje T_2 i T_3 . W związku z tym konieczne byłoby odnalezienie początku transakcji T_2 i wycofanie zapisanych zmian.

Jeśli w logu występuje zapis rozpoczynający punkt kontrolny, a nie ma mu odpowiadającego zapisu <END CKPT>, wówczas menedżer postępuje jak przy logach z powtarzaniem, tzn. poszukuje wcześniejszego punktu kontrolnego i od niego rozpoczyna proces odtwarzania.

Przy odtwarzaniu może nastąpić jeszcze jeden problem. Wystarczy wyobrazić sobie sytuację, w której dwie transakcje piszą na jednym elemencie bazy danych, z czego jedna została zatwierdzona a druga nie. Po awarii nie wiadomo, czy należy najpierw prowadzić proces unieważniania czy powtarzania. Poza tym powstaje pytanie, czy pozwolić transakcji na czytanie elementu, który został zapisany przez transakcję niezatwierdzoną. Ten problem dotyczy zagadnienia współbieżności i izolacji transakcji w systemie i omówię go w następnym rozdziale.

1.5. Transakcje w popularnych systemach baz danych

W każdym poważnym systemie bazodanowym istnieją mechanizmy transakcyjne. Przyjrzałam się bliżej trzem znanym systemom: Oracle, MySQL i PostgreQSL, by dowiedzieć się jak do problemów związanymi z transakcjami i odtwarzaniem podeszli ich twórcy.

Oracle 9i

Domyślnie w systemie Oracle już w momencie zalogowania się do systemu poprzez konsolę użytkownika rozpoczyna się transakcja. Dopiero wydanie jawnej komendy ROLLBACK lub COMMIT kończy transakcję i powoduje automatyczne uruchomienie nowej transakcji, a wszystkie instrukcje wypisane przed tymi komendami są wykonywane zamykane w całość, czyli wykonają się wszystkie albo żadna. Wyjątkiem są instrukcje DDL (CREATE TABLE..., ALTER TABLE...), które niejawnie powodują zatwierdzenie transakcji. Oczywiście istnieje możliwość ustawienia opcji AUTOCOMMIT, która spowoduje, że każda wydana instrukcja będzie wykonywana jako oddzielna transakcja.

W Oraclu zaimplementowane jest również pełne wsparcie dla archiwizowania i odtwarzania. Jednostką odpowiedzialną za te procesy jest RMAN (*Recover MANager – zarządca odtwarzania*). Może on przeprowadzać archiwizację pełną, czyli tworzyć zrzuty całej bazy z określonego momentu w czasie, jak i archiwizację zmian w logach. Istnieje również możliwość wprowadzenia do logów punktów kontrolnych co jakiś odstęp czasu. Ten odstęp czasowy jest definiowany w pliku startowym INIT.ORA. Na podstawie tych plików archiwalnych po awarii RMAN, stosując operacje ponownego wykonywania i wycofywania transakcji (log powtarzanie/unieważnianie), przywraca bazę danych do stanu spójnego. Dzięki systemowym numerom zmian (SCN) wstawianym przy każdym zapisie w logu możliwe jest odtworzenie bazy z konkretnego momentu.

MySQL 5.1

W MySQL'u zaimplementowanych jest kilka typów tabel:

- MyISAM – ten typ tabel nie posiada wsparcia transakcyjnego, każda instrukcja jest automatycznie zatwierdzana, za to działa szybciej
- InnoDB i Berkley DB (od wersji 3.23)– te typy tabel posiadają pełne wsparcie transakcyjne, ale działają wolniej niż MyISAM

  Dzięki temu użytkownik ma wybór: szybkość albo bezpieczeństwo.

Serwer MySQL tworzy cztery rodzaje logów: *General Query Log, Binary Log, Slow Log, Error Log*. Logiem *sensu stricte* związnym z transakcjami jesr *Binary Log*. W nim tworzone są zapisy związane z modyfikacjami danych. *General Query Log* zawiera informację o

połączeniach z klientami oraz zapytania jakie od nich otrzymuje. *Slow Log* dotyczy zapytań o długim czasie trwania, natomiast *Error Log* to dziennik błędów. Logi zapisywane są w plikach, ale od wersji 5.1.6 jest możliwość przechowywania logu w postaci tabel bazy danych. Przy każdym stracie serwera tworzone są nowe pliki logów. Istnieje również możliwość jawnego zamknięcia logu i ponownego jego otwarcia lub utworzenie nowego za pomocą instrukcji FLUSH LOG.

PostgreSQL 8.1.3

W PostgreSQL transakcje spełniają wszystkie własności ACID. Domyślnie każda instrukcja jest automatycznie zatwierdzana, ale oczywiście istnieje możliwość wykonywania bloków instrukcji w ramach jednej transakcji. Wówczas blok taki należy poprzedzić komendą BEGIN WORK i zakończyć COMMIT WORK lub ROLLBACK WORK.

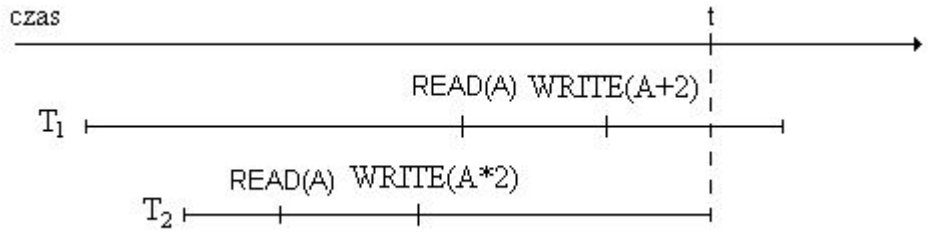
Dla każdej transakcji w log jest zapis składający się z ID transakcji oraz flagi: *in progress, commited, aborted*. Wszelkie wpisy zostają zapisane w logu zanim zmiany danych zostaną zapisane w bazie. Istnieje również możliwość wstawiania checkpoint'ów do logu.

Wyżej wymienione bazy danych dodają do obsługi transakcji dodatkową funkcję: możliwość wstawiania savepoint'ów. Polega to na zaznaczaniu miejsc w bloku instrukcji transakcji do których można by się ewentualnie później cofnąć, za pomocą komendy SAVEPOINT *nazwa.*

Takich punktów można wstawić do bloku transakcji kilka. Następnie za pomocą komendy ROLLBACK TO SAVEPOINT *nazwa* można cofnąć się do określonego miejsca w bloku transakcji, czyli wycofać te instrukcje, które zostały wykonane po wstawieniu wskazanego savepoint'a. Komenda ta nie powoduje przerwania transakcji. Należy również pamiętać, że jeżeli cofamy się do jakiegoś miejsca, to wszystkie savepoint'y, które znajdowały się dalej w bloku, zostają usunięte.

2.Sterowanie współbieżnością

W poprzednim rozdziale głównym rozważanym zagadnieniem był problem odtwarzania transakcji po awarii jako niezbędnego mechanizmu zapewniającego poprawność bazy danych. Ale nie tylko awaria może zakłócić jej spójność. Większość obecnych baz danych to systemy wielodostępowe z możliwością jednoczesnego wykonywania się transakcji. „Jednoczesne wykonywanie" oznacza, że czasy trwania transakcji się pokrywają, ale w istocie poszczególne ich operacje wykonywane są sekwencyjnie, gdyż większość komputerów to maszyny jednoprocesorowe. Współbieżność transakcji sprowadza się więc do wykonania przeplotu operacji z różnych transakcji. Taką sytuację będziemy dalej rozpatrywać. Ustalenie kolejności przeplotu operacji jest konieczne, gdyż wiele współbieżnych transakcji może chcieć pracować jednocześnie na tym samym elemencie bazy danych, a zezwolenie na nieograniczony dostęp do danych może zakłócić ich poprawność. Wyobraźmy sobie sytuację, gdy transakcja T_1 chce element (liczbowy) A zwiększyć o 2, a transakcja T_2 zwiększyć go 2 razy. W zależności od kolejności wykonania się tych transakcji końcowa wartość elementu A będzie inna. Trzeba więc się zastanowić, która operacja zapisu powinna się pierwsza: czy ta z transakcji T_1 , czy z T_2 . Jedno z najprostszych rozwiązań brzmi „kto pierwszy ten lepszy", czyli ta transakcja, która pierwsza zgłosi chęć zapisu, uzyskuje dostęp do elementu. Jednak gdyby system nie był współbieżny, to dostęp do danych dostała by ta transakcja, która pierwsza się rozpoczęła, a wówczas kolejność zapisu mogła by być inna. Sytuację tego typu ilustruje poniższy rysunek:



Rys. 2.1.źródło własne

Niezależnie od tego jak ustalimy kolejność wykonywania się transakcji, pojawia się pytanie czy transakcja powinna widzieć to, co zapisała inna transakcja. W naszym przykładzie, gdyby transakcja T_2 była wykonywana pierwsza i w momencie *t* nie zakończyła się zatwierdzeniem, tylko komendą ROLLBACK, wówczas powstałby problem, jak tą transakcję wycofać skoro inna transakcja zdążyła już przeczytać i nadpisać

zmodyfikowaną wersję elementu A. Jedna z własności ACID transakcji – *izolacja* - mówi, że transakcje powinny wykonywać się tak, jakby żadna inna transakcja się w tym samym czasie nie wykonywała, stąd transakcja T_1 nie powinna widzieć co zrobiła T_2 i na odwrót. Ale jaka powinna być ostateczna wartość elementu A, po zakończeniu obydwu transakcji? Wiemy też, że ze względów optymalizacyjnych, dane są często zrzucane na dysk później niż na to wskazuje operacja WRITE. Jeżeli transakcje zakończą się tak jak na rysunku, to efekty działania T_2 zostaną całkowicie nadpisane i element będzie miał wartość A+2. Gdyby transakcja druga trwała nieco dłużej i zakończyłaby się później niż T_1 , to ostateczna wartość elementu byłaby A*2. Ponadto obydwa te wyniki są różne od tego jaki uzyskalibyśmy w systemie jednodostępowym.

Celem sterowania współbieżnością jest zachowanie poprawności danych. *Zasada poprawności* mówi, że transakcja wykonywana w izolacji przekształca spójny stan bazy danych w inny stan spójny[1]. W systemach niewspółbieżnycm spójność nie jest zagrożona. We współbieżnych natomiast należy tak uszeregować transakcje by dały wynik taki sam, jakby transakcje wykonywały się pojedynczo. Zatem menadżer transakcji ma do wykonania dwa zadania:

1. zapewnienie wykonywania się operacji w określonej kolejności,
2. zapewnienie izolacji transakcji.

2.1.Plany transakcji

By móc wykonać pierwsze zadanie, konieczne jest określenie „planu" działania wielu transakcji.

> *„Plan (Harmonogram)* – jest ciągiem ważnych czynności jednej lub wielu transakcji, uporządkowanych w czasie". [1]

Do tych ważnych czynności należą: INPUT i OUTPUT -czyli pobieranie danych do bufora i zrzucanie na dysk, READ i WRITE – czyli czytanie bufora i przypisanie wartości do zmiennej oraz zapisywanie wartości elementu oraz operacje wykonane na zmiennej.Okazuje się jednak, że przy rozważaniu współbieżności istotne znaczenie mają tylko operacje WRITE i READ, oraz te, które modyfikują wartość elementu.

Plany możemy rozróżnić na sekwencyjne i szeregowalne. *Plan sekwencyjny* zakłada, że

najpierw wykonują się wszystkie czynności pierwszej transakcji, a następnie wszystkie czynności drugiej. Transakcje zazwyczaj wykonują się w takiej kolejności w jakiej trafiły do menedżera transakcji. W *planie szeregowalnym* jest dozwolone przeplatanie się czynności z różnych transakcji, ale w taki sposób, by wynik planu szeregowalnego był taki sam jak w przypadku planu sekwencyjnego tych samych transakcji. Oto przykłady planów :

Przykład 2.1.

Niech wartość elementów A i B wynosi 10. Transakcja T_1 dodaje 2 do elementów, a transakcja T_2 je podwaja.

Plan sekwencyjny

	T_1	T_2	***A***	***B***
1			10	10
2	READ(A,t)			
3	t:=t+2			
4	WRITE(A,t)		12	
5	READ(B,t)			
6	t:=t+2			
7	WRITE(B,t)			12
8		READ(A,s)		
9		s:=s*2		
10		WRITE(A,s)	24	
11		READ(B,s)		
12		s:=s*2		
13		WRITE(B,s)		24

Plan szeregowalny

	T_1	T_2	***A***	***B***
1			10	10
2	READ(A,t)			
3	t:=t+2			
4	WRITE(A,t)		12	
5		READ(A,s)		
6		s:=s*2		
7		WRITE(A,s)	24	
8	READ(B,t)			
9	t:=t+2			
10	WRITE(B,t)		12	
11		READ(B,s)		
12		s:=s*2		
13		WRITE(B,s)		24

Wynik obu tych planów jest taki sam, zarówno element A i B ostatecznie mają wartość 24.

Za ułożenie planu odpowiada część menedżera transakcji, zwana często planistą. Transakcja przesyła planiście żądanie odczytu lub zapisu natomiast planista wykonuje je w buforze bądź je opóźnia. Należy się więc zastanowić na jakiej podstawie planista powinien szeregować operacje. Zauważmy, że zamiana miejscami wierszy 5-7 i 8-10 w planie sekwencyjnym z przykładu nie wpłynęła na wynik jego wykonania. Natomiast gdyby operacja z wierszy 5-7 planu sekwencyjnego wykonały się na samym końcu, wówczas taki plan

szeregowalny dałby inny wynik, gdyż element B zostałby najpierw podwojony, a potem zwiększony. Gdyby obydwie transakcje T_1 i T_2 wykonywały operację dodawania, to nie byłoby problemu, ponieważ dodawanie jest przemienne. Natomiast w przypadku dodawania i mnożenia istotna jest kolejność wykonywania działań. Można by nauczyć planistę podstawowych zasad arytmetyki. Jednak nie rozwiązałoby to problemu, gdyż operacje wykonywane na elementach bazy przez transakcje mogą być bardzo skomplikowane i wcale nie być operacjami arytmetycznymi. Zauważmy również, że zamiana miejscami operacji 4 i 5 z planu szeregowalnego z przykładu zakłóciłaby jego poprawność. Istotna jest więc kolejność operacji odczytu i zapisu, jeśli dotyczą one tego samego elementu. Gdybyśmy mieli maszynę wieloprocesorową i transakcje mogłyby się wykonywać jednocześnie, to aby zachować poprawność, prawo pierwszeństwa do zapisu zmian w buforze miałaby transakcja T_1 . Nawet jeśli transakcja T_2 wcześniej zakończyłaby swe obliczenia to nie mogłaby ich zapisać dopóki nie wykonałaby tego transakcja T_1 .

Podsumowując: przy układaniu szeregowalnego planu znaczenie mają operacje odczytu i zapisu. Nie ważne jest jakie obliczenia są wykonywane na danych, ważne jest, kiedy są czytane i kiedy zapisywane są zmiany. Dlatego w dalszych rozważaniach będziemy się interesować tylko operacjami READ i WRITE.

Dla uproszczenia notacji będę się teraz posługiwała następującymi oznaczeniami:

- T_i – transakcja (ciąg czynności) o indeksie *i*,
- $r_i(X)$ – czynność odczytu elementu *X* przez transakcję T_i ,
- $w_i(X)$ - czynność zapisu elementu *X* przez transakcję T_i ,
- Plan *S* dla zbioru transakcji T – ciąg czynności transakcji $T_i \in T$, w którym dla każdej transakcji T_i jej czynności występują w planie S w takiej samej kolejności, jak w definicji transakcji T_i .

Jak już zostało wspomniane w Przykładzie 2.1, niektóre zamiany miejscami operacji READ i WRITE były obojętne dla wyniku planu, a niektóre jemu szkodziły. Jeżeli dwie czynności muszą wykonywać się w ustalonej kolejności, to wówczas mówimy, że tworzą one *konflikt*. Następujące przypadki par czynności są w konflikcie:

- $r_i(X)$ i $w_i(Y)$,

- $w_i(X)$ i $w_j(X)$,
- $r_i(X)$ i $w_j(X)$.

Czynności tworzą konflikt, jeśli dotyczą tego samego elementu, co najmniej jedna z nich jest czynnością WRITE lub pochodzą z tej samej transakcji.

Natomiast następujące sytuacje nie tworzą konfliktu:

- $r_i(X)$ i $r_j(Y)$, gdzie $i \neq j$,
- $r_i(X)$ i $w_j(Y)$, gdzie $i \neq j$ oraz $X \neq Y$,
- $w_i(X)$ i $w_j(Y)$,gdzie $i \neq j$ oraz $X \neq Y$.

Dla poprawności planu istotna jest więc kolejność operacji tworzących konflikt. Dokładniej:

„O harmonogramie *S* składającym się z *n* transakcji $T_1, T_2, \ldots, T_n$ mówimy, że jest harmonogramem pełnym, jeżeli spełnione są następujące warunki:

1. Operacje harmonogramu S są dokładnie tymi samymi operacjami, które występują w transakcjach $T_1, T_2, \ldots, T_n$,
2. Dla dowolnej pary operacji należącej do tej samej transakcji T_i kolejność ich występowania w harmonogramie S jest taka sama jak kolejność w transakcji T_i ,
3. Dla dowolnych dwóch transakcji konfliktowych jedna z nich musi występować w harmonogramie przed drugą”.[7]

Warunki te nie wymuszają określenia kolejności występowania par operacji bezkonfliktowych. Dla poprawności planu najważniejsze są czynności, które są w konflikcie, wobec tego problem budowania planów szeregowalnych można rozwiązać zapewniając nieco silniejszy warunek, a mianowicie „szeregowalności konfliktów”.[1]

„Plan nazywamy szeregowalnym ze względu na konflikt, jeśli jest on równoważny ze względu na konflikty z innym planem sekwencyjnym.

Dwa plany są równoważne ze względu na konflikty, jeśli można przekształcić jeden z nich w drugi, wykonując ciąg bezkonfliktowych zmian sąsiadujących czynności.”

Zamiany czynności bezkonfliktowych (czyli takich, które nie spowodują wystąpienia konfliktu) można dokonywać nieskończenie wiele, a każdy plan szeregowalny uzyskany przez takie przekształcenia będzie poprawny. W szczególności każdy plan szeregowalny ze względu na konflikt da się doprowadzić do planu sekwencyjnego za pomocą zamian czynności bezkonfliktowych.

Przykład 2.2

Dany jest plan szeregowalny ze względu na konflikt:

$r_1(A); w_1(A); r_2(A); \underline{w_2(A)}; \underline{r_1(B)}; w_1(B); r_2(B); w_2(B);$

Przekształcamy go do planu sekwencyjnego w następujący sposób (w kolejnych krokach dokonujemy zamian czynności podkreślonych):

$r_1(A); w_1(A); \underline{r_2(A)}; \underline{r_1(B)}; w_2(A); w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_2(A)}; \underline{w_1(B)}; r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; \underline{w_1(B)}; w_2(A); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

Kolejnym problemem jest jak sprawdzić czy dany plan niesekwencyjny jest szeregowalny. Przy prostych planach można spróbować znaleźć równoważny plan sekwencyjny za pomocą wyżej opisanych przekształceń. Jednak jest to metoda mało efektywna, a przy dużych planach wręcz niemożliwa do wykonania. Do testowania szeregowalności planów służy technika oparta na *grafie poprzedzania*. Dla planu *S* graf poprzedzania to graf skierowany *G=(N,E)*, gdzie zbiór wierzchołków *N* tworzą wszystkie transakcje z planu *S* (a właściwie ich numery indeksów *i*). Między wierzchołkiem *i* a wierzchołkiem *j* istnieje krawędź od *i* do *j*, jeśli transakcja T_i *poprzedza* transakcję T_j , tzn. istnieją takie czynności A_i w T_i oraz A_j w T_j ,że:

1. w planie *S* czynność A_i występuje przed A_j ,
2. obie czynności A_i i A_j operują na tym samym elemencie bazy danych,
3. co najmniej jedna z tych czynności jest zapisem.

Relację poprzedzania oznaczamy tak: $T_i <_S T_j$.

Testowanie szeregowalności polega na szukaniu cykli w grafie poprzedzania, Jeśli taki cykl zostanie znaleziony, oznacza to, że plan *S* nie jest szeregowalny.

Przykład 2.3.

Dany jest plan:

$S: r_1(A); w_1(A); r_2(A); w_2(A); r_2(B); w_2(B); r_1(B); w_1(B);$

Graf poprzedzania dla tego planu wygląda następująco:

Rys. 2.2. Występuje cykl, więc plan *S* nie jest szeregowalny.

Zapewnienie szeregowalności konfliktów gwarantuje wygenerowanie poprawnego szeregowalnego planu, ale spowoduje również odrzucenie planów, które nie są szeregowalne ze względu na konflikt, ale nie spowodowałby żadnych zakłóceń. Algorytm ten może okazać się zbyt restrykcyjny, przez co mało wydajny. Dlatego zdefiniowano inne, mniej wymagające rodzaje równoważności planów, które nie zapewniają pełnej szeregowalności, ale są efektywniejsze, gdyż dopuszczaja plany, które w przypadku szeregowalności konfliktów zostałyby odrzucone. Przykładem może być *równoważność perspektywiczna*. „Idea równoważności perspektywicznej polega na tym, że o ile każda operacja odczytu danej transakcji odczytuje wynik tej samej operacji zapisu w obu harmonogramach, operacje zapisu każdej z tych transakcji muszą dać te same wyniki. Stąd o operacjach odczytu mówi się, że *mają dostęp do tej samej perspektywy w obu harmonogramach...*"[7]. Można udowodnić, że każdy harmonogram szeregowalny konfliktowo jest też szeregowalny perspektywicznie, ale nie odwrotnie.

Czasem można zrezygnować i z tego rodzaju szeregowalności planów na rzecz jeszcze mniej restrykcyjnych. Taką decyzję możemy podjąć, jeżeli dokładnie znamy rodzaj operacji jakie się wykonują na bazie. Jeśli transakcje, które się wykonują, tylko dodają albo odejmują wartość elementów, to wiemy, że nie muszą spełniać szeregowalności konfliktów, ani szeregowalności perspektywicznej, gdyż ze względu na przemienność dodawania, nie spowodują zakłócenia spójności bazy danych.

2.2.Izolacja

Drugim zadaniem dla menadżera transakcji w związku z współbieżnością jest zapewnienie własności izolacji transakcji. Pełna izolacja transakcji powoduje, że wszelkie modyfikacje czynione przez jedną transakcje, nie są widoczne przez inne transakcje, dopóki transakcja

modyfikująca nie zostanie zatwierdzona. Zapewnienie tej własności jest trudne i niekiedy sprowadza się do tego, że w czasie gdy wykonuje się jedna transakcja, inne czekają, aż ta skończy swoje działanie. Tracimy wówczas możliwość współbieżnego wykonywania się transakcji, co bardzo spowalnia system. Zdarzają się również tzw. transakcje o długim czasie trwania, dla których zapewnienie pełnej izolacji jest często niemożliwe, gdyż ewentualne wycofanie tej transakcji zmuszałoby do kaskadowego wycofania szeregu innych transakcji.

Ponieważ często istnieją poważne przesłanki przemawiające za szybkością działania bazy danych i dostępu do najbardziej aktualnej wersji danych, czasami się osłabia warunek izolacji, a tym samym dopuszcza się możliwość naruszenia szeregowalności. Standard SQL określa trzy sytuacje, które mogą naruszyć szeregowalność wykonania się transakcji:[6]

- *brudny odczyt*: Załóżmy, że transakcja T_1 modyfikuje pewien wiersz. Następnie transakcja T_2 pobiera dane z tego wiersza, po czym transakcja T_1 zostaje zakończona wycofaniem. Wówczas transakcja T_2 widziała wiersz, który już nie istnieje, a w zasadzie nigdy nie istniał, ponieważ transakcja T_1 praktycznie nigdy nie zaszła,
- *niepowtarzalny odczyt*: Załóżmy, że transakcja T_1 czyta pewien wiersz. Następnie transakcja T_2 aktualizuje ten wiersz, a później transakcja T_1 ponownie czyta „ten sam" wiersz. W tej sytuacji transakcja T_1 uzyskała dwa różne zestawy tego samego wiersza,
- *fantomy*: Załóżmy, że transakcja T_1 pobrała zestaw wszystkich wierszy spełniających pewne warunki. Następnie transakcja T_2 wstawia nowy wiersz spełniający ten sam warunek. Jeżeli teraz transakcja T_1 powtórzy to wyszukiwanie, to zobaczy wiersz, którego nie było poprzednio, czyli „fantom".

Pełna izolacja oznacza nie wystąpienie, którejkolwiek z wyżej opisanych sytuacji. Ponieważ użytkownikowi może nie zależeć na pełnej izolacji, dano mu możliwość wyboru i zdefiniowano cztery poziomy izolacji:

- SERIALIZABLE – poziom pełnej izolacji, gwarantuje szeregowalność, jedyny całkowicie bezpieczny poziom,
- REPEATABLE READ – na tym poziomie dopuszczalny jest odczyt „fantomów", natomiast pozostałe dwie sytuacje naruszające szeregowalność wystąpić nie mogą,
- READ COMMITTED – nie dopuszczalną sytuacją jest jedynie brudny odczyt,

natomiast może wystąpić niepowtarzalny odczyt jak i fantomy,

- READ UNCOMMITTED – najsłabszy poziom izolacji, dopuszczalne są wszystkie trzy sytuacje zakłócające szeregowalność.

Nie zawsze w bazie danych zaimplementowane są wszystkie poziomy. Jeśli dostępne są co najmniej dwa poziomy, to wówczas użytkownik ma możliwość ustawienia dla każdej transakcji poziom izolacji, z jakim ma się dana transakcja wykonać. Najbezpieczniej jest jeśli wszystkie transakcje wykonują się z poziomem SERIALIZABLE.

2.3.Mechanizmy sterowania współbieżnością

Jak już wcześniej wspomniałam za zarządzanie i wykonanie planu odpowiada planista. Jego zadanie polega na tym, żeby nie dopuścić do powstania i wykonania planu, który nie byłby szeregowalny. On również odpowiada za to, żeby transakcje wykonywały się z odpowiednim poziomem izolacji. Sposobów zapewniających te warunki jest kilka. Do najbardziej popularnych mechanizmów należą:

- blokady
- znaczniki czasowe
- walidacja

Mechanizmy te można podzielić na *pesymistyczne* i *optymistyczne*. Podejście pesymistyczne zakłada, że konflikty między transakcjami będą się zdarzały często. Wobec tego należy tak zaimplementować techniki sterowania współbieżnością, by zapobiegały powstawaniu konfliktów. Takim mechanizmem jest stosowanie blokad. Dzięki nim zapewnia się odpowiedni poziom izolacji dla transakcji i właściwe ich uszeregowanie. Skutkiem blokowania elementów jest konieczność opóźniania niektórych transakcji, czyli wstrzymywanie ich na pewien czas, aż sytuacja konfliktowa minie. Minusem takiego postępowania jest możliwość wystąpienia zakleszczenia, czyli wstrzymania transakcji na nieskończony czas (problem ten zostanie bardziej szczegółowo omówiony w następnym podrozdziale). Techniki optymistyczne zakładają, że sytuacje konfliktowe zdarzają się rzadko, więc opłaca się zaryzykować wykonanie transakcji bez opóźnień, koniecznych do zapewnienia szeregowalności. Dopiero gdy dochodzi do sytuacji, w której np. zostaje zakłócona spójność, transakcje są przerywane, wycofywane i uruchamiane ponownie. W

systemach optymistycznych nie występują zakleszczenia i są dużo efektywniejsze , gdy większość transakcji wykonuje tylko czytanie. Natomiast jeśli problemy zdarzają się często, to koszty związane z wycofywaniem transakcji mogą być zbyt wysokie i wówczas lepiej jest stosować techniki pesymistyczne. Do mechanizmów optymistycznych należą znaczniki czasowe i walidacja.

2.4.Blokady

Mechanizm blokowania jest jednym znajczęsciej stosowanych rozwiązań. Jeżeli jakaś transakcja chce pracować na elemencie bazy danych, to planista, zanim go udostępni transakcji, nakłada na niego blokadę. Dzięki temu żadna inna transakcja nie będzie miała dostępu do tego elementu, aż do momentu, w którym transakcja pierwsza zakończy się i planista zwolni blokadę. Żeby blokada została zwolniona transakcja musi powiadomić planistę o tym, że już skończyła swe działania. Podsumowując, transakcja oprócz żądań zapisu i odczytu wysyła do planisty żądania nakładania i zwalniania blokad. Dla zaznaczenia tych czynności w planie będę używać następującej notacji:

$l_i(X)$ - transakcja T_i żąda zablokowania elementu *X*.

$u_i(X)$ - transakcja T_i żąda zwolnienia elementu *X*.

Blokady przechowywane są w *tablicy blokad*. Może być ona zaimplementowana jako tablica z haszowaniem, gdzie kluczem jest adres elementu blokowanego. Ponadto musi zawierać informację, która transakcja nałożyła blokadę, gdyż tylko ona może ją potem zwolnić. Szczegółową strukturę tablicy blokad omówię nieco później. Gdy planista otrzymuje żądanie dostępu do elementu od transakcji, najpierw sprawdza w tablicy blokad czy na tym elemencie nie została nałożona blokada i albo umożliwia działanie transakcji, albo ją opóźnia.

Powyżej sformułowane zasady stosowania składają się na dwa założenia, które gwarantują poprawność działania mechanizmu blokad. Założenia te, to:

- *Spójność transakcji* – czynności i blokady muszą być od siebie zależne w następujący sposób:
  1. Transakcja może odczytać lub zapisać element , jeśli uprzednio wystąpiła z żądaniem blokady i jeszcze nie wykonała zwolnienia ,
  2. Jeśli transakcja spowodowała blokadę elementu, to musi go potem zwolnić

- *Legalność planu* – żadne dwie transakcje nie mogą zablokować tego samego elementu, o ile nie został on zwolniony.

Przykład 2.4.

Oto plan szeregowalny z Przykładu 1. wzbogacony o czynności zakładania i zdejmowania blokad:

	T1	***T2***	***A***	***B***
1			10	10
2	$l_1(A); r_1(A)$			
3	t:=t+2			
4	$w_1(A); u_1(A)$		12	
5		$l_2(A); r_2(A)$		
6		s:=s*2		
7		$w_2(A); u_2(A)$	24	
8	$l_1(B); r_1(B)$			
9	t:=t+2			
10	$w_1(B); u_1(B)$		12	
11		$l_2(B); r_2(B)$		
12		s:=s*2		
13		$w_2(B); u_2(B)$		24

Powyższy plan jest poprawny, spełnia założenia spójności transakcji i legalności planu.

2.4.1. Dwufazowe blokowanie

Z dotychczasowych rozważań wynika, że planista powinien zorganizowań taki plan, żeby był legalny, spełniał własność spójności oraz był szeregowalny. Okazuje się, że istnieje taki warunek, który gwarantuje, że plan legalny jest też szeregowalny, a przez to i spełniona jest spójność transakcji. Warunek ten nazywa się *dwufazowym blokowaniem* lub 2PL (*two-phase locking*) i jest powszechnie stosowany w systemach bazodanowych. Brzmi on następująco:

W każdej transakcji wszystkie żądania blokowania występują przed jakimkolwiek żądaniem zwolnienia.

Jak sama nazwa wskazuje postępowanie blokowania spełniającego ten warunek składa się z dwóch faz:

- *faza pierwsz*a, w trakcie której wykonywane są wszystkie blokowania wymagane do realizacji transakcji,
- *faza druga*, w trakcie której zwalniane są wszystkie blokady nałożone w fazie pierwszej.

Plan z Przykładu 4. nie spełnia warunku 2PL, gdyż np. transakcja T_1 zwalnia blokadę z elementu A zanim nałoży blokadę na element B.

Wiemy, że plan *S* z transakcjami 2PL, jest szeregowalny. Wiemy też, że każdy plan szeregowalny jest równoważny ze względu na konflikt z planem sekwencyjnym i w związku z tym można jeden przekształcić w drugi. Okazuje się, że transakcje w równoważnym planie sekwencyjnym występują w tej samej kolejności jak pierwsze ich zwolnienia w planie szeregowalnym z 2PL.

Wiemy już jak można skonstruować plan szeregowalny i poprawny. Jednak to nie koniec problemów z transakcjami. Niestety dwufazowe blokowanie może spowodować *zakleszczenie* transakcji. „Jest to taka sytuacja, gdy dwie transakcje (lub więcej) czekają na zwolnienie blokady jednostek, które nawzajem blokują”[2].

Przykład 2.5.

$T_1 : l_1(A); r_1(A); w_1(A); l_1(B); u_1(A); r_1(B); w_1(B); u_1(B);$
$T_2 : l_2(B); r_2(B); w_2(B); l_2(A); u_2(B); r_2(A); w_2(A); u_2(A);$

Jeżeli czynności tych transakcji przeplecie się w następujący sposób:

$S : l_1(A); r_1(A); l_2(B); r_2(B); w_1(A); w_2(B); l_1(B); l_2(A);$

to dojdzie do zakleszczenia. Transakcja T_1 chce zablokować element B, ale nie może, gdyż został on zablokowany przez transakcję T_2 i nie został jeszcze zwolniony. Zostaje więc wstrzymana i czeka na zwolnienie blokady. Transakcja T_2 natomiast, chce dostępu do elementu A zablokowanego wcześniej przez transakcję T_1 i równie czeka na jego zwolnienie. Transakcje te czekają tak bez końca.

Problem zakleszczenia można rozwiązać na dwa sposoby:

1. poprzez protokoły zapobiegania zakleszczeniom,
2. poprzez protokoły wykrywania zakleszczeń i limity czasowe.

Jeden z protokołów pierwszego rodzaju wymaga, aby każda transakcja „z góry blokowała wszystkie wymagane elementy”. Jest to warunek często nie możliwy do spełnienia i mocno

ograniczający współbieżność.

Inne rozwiązanie opiera się na pojęciu *znacznika czasowego transakcji*, czyli unikalnego identyfikatora transakcji. Znacznik ten porządkuje transakcje według kolejności ich powstania, czyli transakcje starsze maja mniejszą wartość znacznika. Załóżmy, że transakcja *T* chce zablokować element *X*. Jeden ze schematów „czekaj-kończ" zakłada, że jeśli *X* posiada już blokadę założoną przez transakcję młodszą, wówczas *T* jest wstrzymywana, w przeciwnym przypadku transakcja jest anulowana i uruchamiana ponownie z nowym znacznikiem czasowym. Drugi schemat „zakończ-czekaj" działa w ten sposób, że jeśli element *X* został zablokowany przez transakcję młodszą to zostaje ona anulowana, by ustąpić miejsca transakcji starszej. Metoda ta daje pierwszeństwo transakcjom starszym i nigdy one nie oczekują. Może jednak sprawiać „przykre niespodzianki" użytkownikom transakcji młodszych.

Kolejne metody zapobiegania zakleszczeniom to tzw. algorytmy *nie oczekiwania* lub *oczekiwania ostrożnego*. Pierwszy algorytm zakłada, że żadna transakcja nie oczekuje, a w sytuacji, gdy nie może uzyskać blokady, zostaje anulowana i za jakiś czas uruchamiana ponownie. W tej metodzie może dojść do niepotrzebnego anulowania wielu transakcji, gdyż nie sprawdzane jest czy w ogóle zagłodzenie mogło wystąpić. W tym względzie lepszy jest algorytm oczekiwania ostrożnego. Jeśli transakcja *T* chce uzyskać blokadę będącą w konflikcie z blokadą już nałożoną na elemencie *X*, najpierw sprawdzane jest, czy transakcja, która zablokowała *X*, jest wstrzymywana. Jeśli tak, to transakcja *T* zostaje anulowana, w przeciwnym przypadku trafia na listę transakcji oczekujących.

Protokoły drugiej grupy polegają na wykryciu zakleszczenia i jego wyeliminowaniu. Najbardziej znany mechanizm oparty jest o tzw. *graf oczekiwania*. Wierzchołki grafu stanowią transakcje. Krawędź skierowana między wierzchołkami T_i i T_j ($T_i \rightarrow T_j$) jest tworzona wtedy, gdy transakcja T_i czeka na zablokowanie elementu, który jest już blokowany przez T_j . Jeśli w tak skonstruowanym grafie występują cykle, oznacza to, że w systemie występuje zakleszczenie. W celu eliminacji zakleszczenia, należy wybrać „ofiarę", czyli ustalić, która transakcji tworzących cykl ma zostać wycofana. Menedżer powinien wybrać tą transakcję, która ma mniejszy koszt wycofania operacji. Problemem tego podejścia jest kwestia, kiedy i jak często system powinien uruchamiać algorytm budowania grafu i jego przeszukiwania.

Najprostszym sposobem radzenia sobie z zakleszczeniami są *limity czasowe*. Polega ona

na tym, że transakcja, która czeka zbyt długo (więcej niż to określa limit czasowy), jest unieważniana. Minusem tej metody jest fakt, że unieważniane są wszystkie transakcje, które wystarczająco długo czekają, a nie tylko te tworzące zakleszczenie.

2.4.2. Typy blokad

Do tej pory w rozważaniach stosowane był jeden typ blokad. Transakcja blokująca zamykała na pewien czas dostęp do elementu innym transakcjom. Nie zawsze jednak takie zachowanie jest konieczne. Jeżeli dwie transakcje wykonują tylko zapytanie typu SELECT, czyli czytają dane i ich nie modyfikują, to nie jest konieczne by jedna z transakcji czekała, aż druga zakończy czytanie. Taka czynność może się wykonać jednocześnie. Niewskazane jest natomiast by w czasie, kiedy jedne transakcje czytają element inna chciała je modyfikować. Również elementy modyfikowane nie powinny być widoczne dla transakcji czytających, aż do momentu zakończenia modyfikacji. Stąd właśnie wziął się najbardziej podstawowy podział na dwa typy blokad na blokadę *wspólną* (dzieloną) i *wyłączną*.

Blokada wspólna jest tzw. „blokadą do odczytu". Jeśli transakcja chce tylko czytać nakłada na element blokadę wspólną. Na jeden element może być nałożonych przez różne transakcje wiele takich blokad jednocześnie. Należy pamiętać, że muszą to być tylko i wyłącznie transakcje czytające. Blokada wspólna nie pozwala modyfikować elementu, czyli nałożyć przez inną transakcję blokady do zapisuj. Na jednym elemencie bazy może być nałożona w jednej chwili tylko jedna blokada wyłączna. Nałożenie tej blokady oznacza, że ani inne transakcje piszące, ani czytające nie mają dostępu do elementu.

Przy kolejnych analizach transakcji blokady te będę oznaczać w następujący sposób:

$sl_i(X)$ - oznacza nałożenie blokady wspólnej na elemencie X przez transakcję T_i .

$xl_i(X)$ - oznacza nałożenie blokady wyłącznej na elemencie X przez transakcję T_i .

Natomiast zdejmowanie blokad będzie dla obu typów oznaczone tak jak poprzednio, czyli $u_i(X)$.

W tym nowym kontekście warunki spójności, dwufazowego blokowania i legalności planu brzmią następująco:

1. *Spójność transakcji*: Nie można pisać beż nałożenia blokady wyłącznej i nie można czytać bez blokady wspólnej. Po każdej blokadzie musi nastąpić zwolnienie tego samego

elementu,

2. *Dwufazowe blokowanie*: Blokowanie musi występować przed zwolnieniem,
3. *Legalność planów*: Element może mieć blokadę wyłączną nałożoną przez jedną transakcję lub wiele blokad wspólnych, ale nie może mieć jednocześnie blokady wspólnej i wyłącznej.

Zauważmy też, że jedna transakcja może blokować ten sam element zarówno wspólnie jak i wyłącznie, o ile nie kłóci się to z innymi transakcjami. Ponadto dla tych dwóch typów blokad aktualna jest własność, że legalne plany dla spójnych transakcji 2PL są szeregowalne ze względu na konflikt.

Jeżeli w sterowaniu współbieżnością stosuje się kilka typów blokad, to w tablicy blokad przy każdej blokadzie powinna się znaleźć informacja o tym jakiego ona jest typu.

W takiej sytuacji, planista musi wiedzieć, kiedy może daną blokadę nałożyć i według jakiej strategii. Należy więc planistę nauczyć wszystkich zależności pomiędzy nimi. Do tego celu służy *macierz kompatybilności*. Wiersze w macierzy kompatybilności odpowiadają blokadom nałożonym na element *X*, natomiast kolumny - żądane typy blokowań. Macierz wypełniona jest wartościami „Tak” lub „Nie”. Jeżeli transakcja żąda nałożenia blokady danego typu, to planista sprawdza w tablicy blokad czy element nie został już zablokowany. Jeśli tak to odczytuje jakiego typu jest ta blokada, a następnie sprawdza w macierzy kompatybilności, co znajduje się na przecięciu wiersza z typem posiadanej blokady i kolumny z typem blokady żądanej. Znaleziona wartość jest odpowiedzią na pytanie o zezwolenie nałożenia blokady.

Macierz kompatybilności dla blokowania wspólnego i wyłącznego wygląda następująco:

		Blokady żądane	
		bl. wspólna	bl. wyłączna
Posiadane blokady	bl. wspólna	Tak	Nie
	bl. wyłączna	Nie	Nie

Stosując powyższy schemat blokowania transakcja , która chce najpierw czytać element bazy danych *X*, a potem na nim pisać, od razu zakłada blokadę wyłączną, co blokuje dostęp innym transakcjom. Może się zdarzyć, że etap czytania jest dość długi i właściwie w tym czasie nie ma przeciwwskazań, żeby inna transakcja mogła też przeczytać ten element. W celu

usprawnienia systemu stosuje się następującą modyfikację: transakcja, która chce czytać i pisać najpierw zakłada blokadę wspólną na element *X*, a dopiero gdy przystępuje do pisania zakłada blokadę wyłączną, z tym że nie ma zwolnienia poprzedniej blokady tylko jej zamiana. W czasie czytania inne transakcje również mogą nakładać blokady wspólne. W momencie gdy transakcja chce rozpocząć pisanie musi poczekać, aż wszystkie inne transakcje, które czytały razem z naszą transakcją zwolnią swoje blokady wspólne. Modyfikacja ta ma jeden duży minus. Może się bowiem zdarzyć, że kilka transakcji zablokuje element wspólnie z zamiarem późniejszego pisania. Gdy nadejdzie moment zapisu wszystkie te transakcje będą czekały na wzajemne zwolnienie blokad wspólnych. Dochodzi więc do zakleszczenia i system się zawiesza.

Żeby rozwiązać ten problem można wprowadzić dodatkowy tryb blokowania – tzw. *blokadę aktualizującą*. Jest ona blokadą tylko do czytania, ale jest ona również informacją dla innych transakcji, że wkrótce stanie się blokadą do pisania. Blokadę aktualizującą można nakładać na element, który jest już blokowany wspólnie. Jeśli natomiast element ma blokadę aktualizującą, to inne transakcję nie mogą już na niego nałożyć żadnej innej blokady. Tylko transakcja, która nałożyła blokadę aktualizującą może zablokować element wyłącznie.

Dla trzech trybów blokowania: wspólnego, wyłącznego i aktualizującego macierz kompatybilności wygląda tak:

		Blokady żądane		
		bl. wspólna	bl. wyłączna	bl. aktualizująca
Posiadane blokady	bl. wspólna	Tak	Nie	Tak
	bl. wyłączna	Nie	Nie	Nie
	bl. aktualizująca	Nie	Nie	Nie

Istnieje jeszcze jeden rodzaj blokad - *blokady przyrostowe*. Są to blokady przeznaczone tylko do określonego typu transakcji, które zwiększają albo zmniejszają wartość elementu. Wobec tego blokadę przyrostową można stosować tylko do transakcji, które modyfikują element liczbowy poprzez wykonanie podstawowych operacji arytmetycznych dodawania i odejmowania. Do operacji zwiększania i zmniejszania elementu, można by oczywiście stosować wyżej wymienione blokady: wspólne, wyłączne i aktualizujące. Wystarczą one do

zachowania poprawności systemu. Blokady te wyróżniono po to, by usprawnić systemy, w których najczęściej wykonywanymi transakcjami są właśnie transakcje przyrostowe np. Systemy bankowe, które głównie zajmują się odejmowaniem lub dodawaniem określonych wartości do konta. Jak już wcześniej wspomniałam operacje te są przemienne, więc kolejność ich wykonania na tym samym elemencie nie jest istotna. Stąd spada obowiązek z planisty pilnowania, która transakcja powinna wykonywać się pierwsza. Należy jednak pamiętać, że operacje przyrostu nie są przemienne z operacjami czytania i pisania.

W dalszych rozważaniach zapis $inc(X,c)$ będzie oznaczał operację przyrostu elementu X o wartość c; składa się ona z następującego ciągu czynności: READ(X,t); t:=t+c; WRITE (X,t). Zapis $il_i(X)$ oznacza, że transakcje T_i żąda nałożenia blokady przyrostowej na elemencie X. W celu zachowania spójności transakcji i legalności planów powinny być zachowane następujące zasady:

1. Transakcją spójna może wykonywać operację przyrostu na elemencie X wtedy, kiedy X jest zablokowany przyrostowo,
2. W planach legalnych w danej chwili wiele transakcji może blokować jeden element X przyrostowo,
3. Czynność $inc_i(X)$ jest w konflikcie zarówno z $r_j(x)$, jak i $w_j(X)$, dla $i \neq j$, ale nie jest w konflikcie z $inc_j(X)$.

Stąd łatwo zdefiniować macierz kompatybilności dla blokad wspólnych, wyłącznych i przyrostowych:

		Blokady żądane		
		bl. wspólna	bl. wyłączna	bl. przyrostowa
Posiadane blokady	bl. wspólna	Tak	Nie	Nie
	bl. wyłączna	Nie	Nie	Nie
	bl. przyrostowa	Nie	Nie	Tak

Przykład 2.6

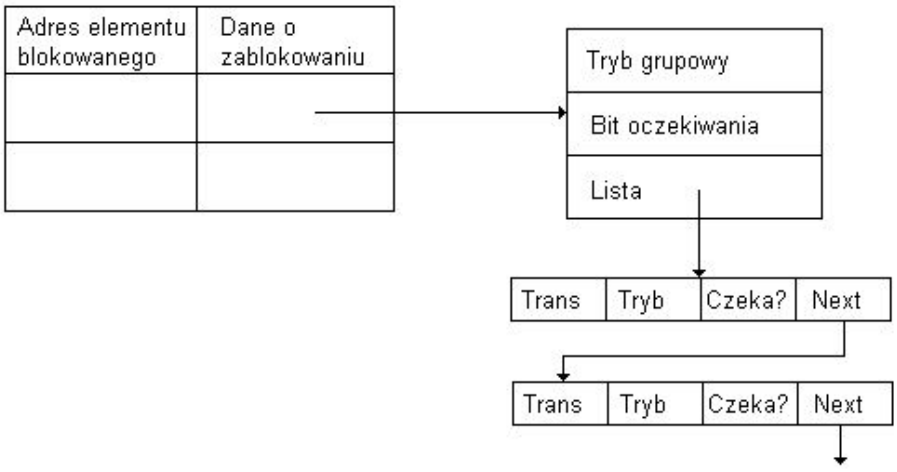
Rozważmy następujące transakcje i przeplatanie się ich czynności.

	T_1	T_2
1	$sl_1(A); r_1(A);$	
2		$sl_2(A); r_2(A)$
3		$il_2(B); inc_2(B)$
4	$il_1(B); inc_1(B)$	
5		$u_2(A); u_2(B);$
6	$u_1(A); u_1(B);$	

Zauważmy, że planista nie musi opóźniać żadnej z transakcji, a kolejność operacji w wierszu 3 i 4 nie ma znaczenia.

2.4.3.Tablica blokad

Jak już wcześniej wspomniałam blokady przechowywane są w tablicy blokad. Zazwyczaj do tego celu wydzielony jest oddzielny obszar w pamięci operacyjnej. W tablicy przechowywane tylko elementy blokowane więc wielkość jej nie zależy od wielkości bazy ale zmienia się. Może być zaimplementowana jako tablica z haszowaniem, gdzie jak kluczy używa się adresów elementów. Poniższy obrazek przedstawia strukturę tablicy blokad:



Rys. 2.3. Schemat tablicy blokad; źródło [1]

W pierwszej kolumnie przechowywane są adresy elementów. W drugiej znajduje się struktura, składająca się z następujących pól:

1. *Tryb grupowy* - jest informacją w jakim trybie został zablokowany element przez transakcję lub grupę transakcji. Jeśli stosowany jest system blokowania wspólnego(S)-wyłącznego(X)-aktualizującego(U), to wartość tego pola będzie wynosić S, w przypadku gdy na element będzie nałożona co najmniej jedna transakcja wspólna, X-oznacza jedną blokadę wyłączną, U-gdy element został zablokowany jedną blokadą aktualizującą lub blokadami wspólnymi i blokadą aktualizującą,
2. *Bit oczekiwania* – informuje o tym, czy jakieś inne transakcje czekają na zablokowanie danego elementu:1-tak, 0 – nie,
3. *Lista* – jest to lista transakcji, które czekają na zablokowanie lub transakcji, które blokują właśnie element wspólnie/aktualizująco. Każdy element tej listy zawiera informację o tym:
   a) jaka transakcja blokuje lub oczekuje na blokadę, czyli identyfikator transakcji,
   b) w jakim trybie blokuje chce zablokować (S,X lub U),
   c) czy jest transakcją blokującą czy oczekującą ,
   d) wskaźnik na następny element listy.

Prześledźmy teraz proces zakładania i zwalniania blokad. Załóżmy, że transakcja *T* chce zablokować element *X* . Najpierw planista sprawdza, czy element *X* ma swój wpis w tablicy, przeszukiwana jest więc pierwsza kolumna tablicy. Jeśli takiego wpisu nie ma, do tablicy dodawana jest nowa pozycja zawierająca adres elementu *X*, tryb grupowy jest ustawiany na tryb żądanej blokady, bit oczekiwania wynosi 0, a lista zawiera jeden element z numerem transakcji, trybem i bitem oczekiwania 0. Jeśli natomiast element *X* jest już blokowany to planista odczytuje tryb grupowy elementu oraz tryb żądanej blokady i sprawdza w macierzy kompatybilności czy blokady te mogą być nałożone jednocześnie. Jeśli tak to, w zależności od rodzajów blokad, tryb grupowy jest modyfikowany albo pozostaje bez zmian. Do listy natomiast zostaje dodany element z numerem nowej transakcji i bitem czekania równym 0. Jeśli natomiast planista decyduje się opóźnić transakcję, to wówczas tryb grupowy pozostaje bez zmian, grupowy bit oczekiwania zostaje zmieniony na 1, a do listy dodana zostaje transakcja z bitem oczekiwania 1.

W przypadku zwalniania blokady najpierw usuwa się z *listy* transakcji blokująco-czekających informację o transakcji, która zakończyła już swe działanie. Jeśli bit oczekiwania wynosi 0, to tryb grupowy wówczas nie zmienia się: w przypadku, gdy

zdejmowana jest blokada wspólna, to tryb grupowy może być S albo U i po usunięciu blokady taki zostanie. W przypadku blokady wspólnej należy sprawdzić, czy na liście znajdują się inne transakcje. Jeśli nie to usuwany jest cały wpis o elemencie blokowanym z tablicy. Tak samo jest w przypadku zdejmowania blokady wyłącznej i aktualizującej, gdyż z ich definicji wiadomo, że w momencie zdejmowania tych blokad inne transakcje nie blokują elementu. Jeśli natomiast bit oczekiwania (grupowy) jest ustawiony na 1 to po zdjęciu blokady X lub U, lub ostatniej blokady z grupy typu S, należy przydzielić blokadę, jednej z oczekujących transakcji. Przydzielanie blokad transakcjom oczekującym może się odbywać według następujących zasad:

1. pierwszy przyszedł, pierwszy obsłużony,
2. priorytet blokad wspólnych,
3. priorytet blokad wyłącznych.

Pierwsza z wymienionych zasad odzwierciedla system kolejkowy i wydaje się być najbardziej sprawiedliwa, gdyż nie powoduje zagłodzenia żadnej transakcji. Ale w zależności od potrzeb i założeń systemu bazy danych, można stosować pozostałe zasady.

W przypadku stosowania pierwszej zasady, bierzemy pierwszą transakcję z listy i przydzielamy jej blokadę ustawiając tryb grupowy taki jak typ blokady. Jeśli blokada ta jest typu U lub X to sprawdzamy czy za nią na liście znajdują się jakieś wpisy. Jeśli tak to bit oczekiwania należy ustawić na 1. Jeśli pierwszą blokadą w kolejce była blokada typu S, należy rozważyć następujące sytuacje:

- w kolejce znajdują się jeszcze inne transakcje czekające tylko na blokadę wspólną. Wówczas wszystkim transakcjom przyznana zostaje blokada, a ich bity oczekiwania zmieniają się na 0, bit grupowy również ustawiany jest na 0,
- jeśli w kolejce oprócz wyżej wymienionych czekają jeszcze transakcje aktualizujące, to pierwszej z nich przydziela się blokadę, tryb grupowy zostaje zmieniony na U i w zależności czy transakcji aktualizujących jest więcej niż jedna ustawia się grupowy bit oczekiwania. W przypadku, gdy transakcje na liście są ustawione w kolejności: wspólne, aktualizujące, wspólne, to należy się zastanowić, czy przydzielamy blokady wszystkim transakcjom wspólnym i jednej aktualizującej, czy tylko tym wspólnym , które znajdują się przed pierwszą blokada aktualizującą. Pierwszy sposób promuje blokady wspólne i może powodować nieco sprawniejsze działanie transakcji ale i zakłócenie spójności bazy danych. Drugi sposób jest bardziej poprawny, gdyż transakcje, które przyszły do systemu później niż transakcja aktualizująca powinny

przeczytać wersję danych po ich modyfikacji,

- podobnie jest gdy, na liście znajduje się choć jedna transakcja wyłączna. Wówczas stosujemy postulat przydzielania blokad tylko transakcjom wspólnym występującym przed pierwszą transakcją wyłączną, pozostałe transakcje nadal są wstrzymywane. W tym przypadku tryb grupowy nie zmienia się.

2.4.4. Blokowanie hierarchicznych elementów bazy danych

Wiemy już jak zarządzać blokadami pojedynczych elementów, kiedy zezwalać na ich przydzielanie, a kiedy nie. Należy się teraz zastanowić jak zarządzać blokadami, kiedy mamy do czynienia z hierarchicznymi elementami bazy danych, czyli kiedy baza danych ma strukturę drzewiastą. W przypadku obiektowej bazy danych układem hierarchicznym elementów jest obiekt i jego podobiekty. Podobnie w relacyjnych bazach danych elementem nadrzędnym jest cała relacja, czyli tabela, a podrzędnym np. wiersz lub krotka. Okazuje się, że omówione do tej pory mechanizmy blokowania nie wystarczą do zachowania poprawności bazy danych.

Przykład 2.7

Niech A będzie elementem bazy danych, a B, C i D jego podelementami. Załóżmy, że transakcja T_1 chce modyfikować element B. Wobec tego żąda nałożenia na ten element blokady wyłącznej. Następnie do systemu przychodzi transakcja T_2, która chce pracować nad elementem A, obojętnie czy chce pisać czy czytać. Na elemencie A nie ma nałożonej blokady, wobec tego planista zezwala na jej działanie. W tym momencie dochodzimy do błędu, gdyż transakcja T_2, jeśli chce przeczytać element A, to musi przeczytać wszystkie jego podelementy, czyli również element B, który jest właśnie modyfikowany. Jeśli transakcja T_2 chciałaby pisać na elemencie A, to wówczas dwie transakcje miały by prawo w jednym czasie modyfikować element B. A taki plan nie jest już legalny.

Jednym z rozwiązań wyżej opisanego problemu jest nakładanie blokady na obiekt nadrzędny, jeśli przynajmniej jeden jego podobiekt jest blokowany. Czyli transakcja T_1, chcąc modyfikować element B, powinna uprzednio zablokować element A. Jednak, jeśli w

tym czasie jakaś inna transakcja T_3 chciałaby pracować na elemencie C, to transakcja ta zostałaby wstrzymana, gdyż wówczas musiałaby zablokować element A. Z logicznego punktu widzenia nie ma żadnych przeciwwskazań, by obie transakcje mogły się wykonywać jednocześnie, gdyż de facto dotyczą one różnych elementów. Poza tym gdyby transakcja T_1 blokowała element B tylko do czytania, to również można by zezwolić innym transakcjom na czytanie nadrzędnego elementu A. To rozwiązanie ogranicza mocno współbieżność wykonywania się transakcji.

Rozważmy odwrotny przypadek, kiedy pierwsza transakcja blokuje najpierw element nadrzędny A. Czy planista powinien udostępnić innym transakcjom jego podelementy? Zależy to od tego w jakim trybie został zablokowany element A i w jakim trybie chcą kolejne transakcje blokować jego podelementy. Jeśli na A została nałożona blokada wyłączna, to żadna inna transakcja nie powinna mieć dostępu do elementów B, C i D. Jeśli natomiast blokada byłaby wspólna, to można by zezwolić innym transakcjom na czytanie elementów podrzędnych.

Najlepszym rozwiązaniem tego problemu jest wprowadzenie takiego mechanizmu blokowania, który w przypadku blokowania elementu podrzędnego w hierarchii najpierw sprawdza, czy elementy nadrzędne są blokowane i w jakim trybie. A gdy tą blokadę nakłada, to elementy nadrzędne dostają informację, że któryś z ich podelementów jest blokowany. Mechanizm ten nazywany jest *blokowaniem ostrzegawczym*. Pojawiają się tu dwa nowe typy blokad:

- *IS* – jest blokadą ostrzegawczą informującą, że co najmniej jeden podelement tak blokowanego elementu jest blokowany wspólnie,
- *IX* – informuje, że co najmniej jeden podelement tak blokowanego elementu jest blokowany wyłącznie.

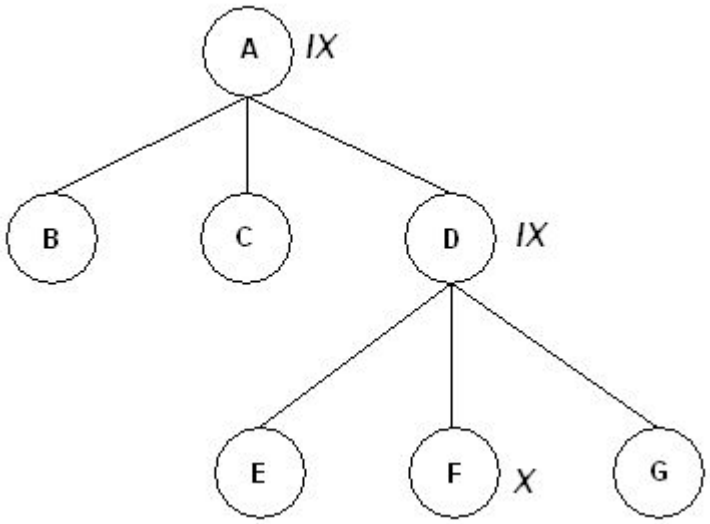
Ponieważ hierarchiczne struktury bazy danych można przedstawić w postaci drzewa, to mechanizm z wykorzystaniem blokad ostrzegawczych stosujmy następująco:

1. Aby uzyskać blokadę typu *S* lub *X* na jakimś elemencie rozpoczynamy przeszukiwania drzewa od korzenia.
2. Jeśli jest to element, który chcemy blokować, to po sprawdzeniu w tablicy blokad, czy nie został on wcześniej zablokowany, nakładamy blokadę i na tym koniec.
3. Jeśli natomiast element, który chcemy zablokować znajduje się niżej w hierarchii to przy korzeniu umieszczamy blokadę *IS* lub *IX*, w zależności od tego w jakim trybie będziemy

blokować. Następnie przechodzimy do poddrzewa, w którym znajduje się szukany element i wracamy do kroku 2

Przykład 2.8

Oto przykład hierarchicznego układu elementów i blokady nałożone przy stosowaniu blokowania ostrzegawczego, gdy chcemy nałożyć blokadę na element F leżący najniżej w hierarchii.



Rys.2.4.Herarchia obiektów; źródło [1]

Blokady ostrzegawcze *IX* nie pozwalają na blokowanie *S* i *X* przez inne transakcje elementów A i D, natomiast zezwalają na dostęp do elementów pozostałych. Wobec tego jeśli chcielibyśmy zablokować element E tylko do czytania, to nie byłoby żadnego problemu, a na elementy A i D zostałyby nałożone dodatkowo blokady ostrzegawcze *IS*. Możliwość jednoczesnego nakładania różnych blokad określa jak zwykle macierz kompatybilności.

		Blokady żądane			
		bl. wspólna (S)	bl. wyłączna (X)	bl. ostrzegawcza wspólna (IS)	bl. ostrzegawcza wyłączna (IX)
Posiadane blokady	bl. wspólna(S)	Tak	Nie	Tak	Nie
	bl. wyłączna(X)	Nie	Nie	Nie	Nie
	bl. ostrzegawcza wspólna(IS)	Tak	Nie	Tak	Tak
	bl. ostrzegawcza wyłączna(IX)	Nie	Nie	Tak	Tak

2.5.Znaczniki czasowe

Metoda znaczników czasowych różni się istotnie od mechanizmu blokowania. Nie ma w niej oczekiwania, a transakcje, które wejdą w kolizję, są wycofywane i uruchamiane ponownie.

Istotą tej metody jest przypisanie każdej transakcji unikalnego znacznika czasowego *TS(T)* – *timestamp*. Znacznik ten jest przypisywany w momencie rozpoczęcia się transakcji i odpowiada wartości czasu zegara systemowego w momencie rozpoczęcia się transakcji *T*, lub wartości nadanej przez menedżera transakcji na podstawie utrzymywanego w systemie licznika. Planista musi przechowywać listę transakcji aktywnych i ich znaczników czasowych. Oprócz znacznika czasowego transakcji menedżer nakłady dwa inne znaczniki na elementy bazy danych::

- *RT(X) – read timestamp* – znacznik czasowy ostatniej transakcji, która odczytywała element *X,*
- *WT(X) – write timestamp* – znacznik czasowy ostatniej transakcji, która zapisywała element *X.*

Dzięki znacznikom czasowym, w sytuacji konfliktu, pierwszeństwo mają transakcje starsze, czyli te, które rozpoczęły się wcześniej. Metoda ta gwarantuje uszeregowanie transakcji w takiej kolejności, jak w planie sekwencyjnym. Jeśli jakaś transakcja chce pracować z elementem bazy danych *X*, sprawdza najpierw, kiedy nastąpiła jej ostatnia modyfikacja, czyli znacznik *WT(X)*. Jeśli znacznik ten jest starszy od znacznika bieżącej transakcji *TS(T)*, to nie ma żadnych przeszkód do kontynuowania transakcji. W przeciwnym przypadku transakcja jest wycofywana i uruchamiana jeszcze raz z nowym znacznikiem czasowym.

Reguły działania mechanizmu znaczników czasowych można przedstawić następująco:

1. Transakcja *T* chce wykonać instrukcję READ(*X*):
   a) jeśli *TS(T)<WT(X)*, to oznacza, że doszło do *zbyt późnego czytania*, czyli transakcja *T* chce przeczytać to, co zapisała młodsza transakcja. W takiej sytuacji transakcja *T* zostaje odrzucona i uruchomiona ponownie z nowym znacznikiem czasowym,
   b) jeśli *TS(T)>WT(X*), to operacja odczytu może zostać wykonana, a znacznik zapisu zostaje zmieniony *WT(X):= TS(T),*
2. Transakcja T chce wykonać instrukcję WRITE(*X*):

a) jeśli *TS(T)<RT(X)*, to znaczy, że doszło do *zbyt późnego pisania*, czyli transakcja *T* chce zapisać nową wartość elementu *X*, który został już wcześniej przeczytany przez transakcję młodszą. Transakcja *T* zostaje więc unieważniona i uruchamiana jeszcze raz z nowym znacznikiem czasowym,

b) jeśli *TS(T)<WT(X)*, tzn. że transakcja *T* chce modyfikować element, który już został zmodyfikowany przez transakcję młodszą, czyli chce go zmodyfikować niepotrzebnie. Transakcja *T* zostaje unieważniona i uruchomiona ponownie z nowym znacznikiem czasowym,

c) w przeciwnym przypadku zapis może zostać wykonany, a *WT(X):=TS(T)*.

Wyżej opisane reguły tworzą *podstawową metodę znaczników czasowych*. Plan powstały dzięki tej metodzie jest szeregowalny. Czasem do tej metody wprowadza się tzw. *zasadę zapisu Thomasa*, która usprawnia dodatkowo mechanizm znaczników czasowych, ale za to osłabia nieco szeregowalność planów. Zasada ta powoduje, że w sytuacji opisanej w punkcie 2b) transakcja, która spóźniła się z zapisem nie zostaje unieważniona, a zapis jej zostaje uznany za „przestarzały" i pominięty.

Metoda znaczników czasowym nie jest jednak wolna od błędów. Problem stanowią sytuacje, kiedy transakcja jest unieważniana, ale zmodyfikowane przez nią dane zostały już przeczytane przez inne transakcję, czyli tzw. czytanie brudnopisów . Kolejny problem wynika z zastosowania zasady zapisu Thomasa. Jeśli wystąpi sytuacja jak w punkcie 2b) i zapis transakcji starszej zostanie pominięty, a następnie z jakiś powodów transakcja młodsza zostanie unieważniona, to należałoby modyfikowanemu elementowi nadać wartość z zapisu z transakcji starszej. Jednak jest to nie możliwe, gdyż przez zasadę zapisu Thomasa, ta wartość nie została nigdzie zapamiętana.

Jeden ze sposobów radzenia z takimi problemami polega na dodaniu trzeciego znacznika na element bazy danych, tzw. *bitu zatwierdzenia:*

- *C(X)* - bit zatwierdzenia transakcji, przyjmuje wartość logiczną: prawda lub fałsz; wartość prawda jest przypisywana wtedy i tylko wtedy, gdy ostatnia transakcja , która modyfikowała element X, zakończyła się

W tej „wzbogaconej" wersji metody znaczników czasowych stosuje się reguły jak w metodzie podstawowej, ale dodatkowo musi zostać sprawdzony znacznik bitu zatwierdzenia. Jeśli *C(X)=false,* oznacza to, że ostatnia transakcja, która pracowała z tym elementem nie została jeszcze zatwierdzona, a więc istnieje ryzyko jej wycofania. W takiej sytuacji transakcja zostaje wstrzymana i oczekuje, aż do momentu, w którym *C(X)* przyjmie wartość

prawda. Kiedy transakcja może być wykonana, znaczniki czasowe są aktualizowane, bit zatwierdzenia przyjmuje wartość fałsz, natomiast planista kopiuje poprzednią wartość elementu i jego poprzednie znaczniki czasowe. Dopiero w momencie zatwierdzenia transakcji planista odszukuje elementy, z którymi transakcja pracowała, i nadaje wartość prawda bitowi zatwierdzenia: *C(X):=true.* Gdyby transakcja zostałaby unieważniona, planista byłby wstanie przywrócić jej poprzednie wartości i poprzednie znaczniki czasowe, a transakcje oczekujące mogłyby zostać wznowione.

Wprowadzenie bitu zatwierdzenia do mechanizmu znaczników czasowych rozwiązuje problem „brudnopisu", ale powoduje wstrzymywanie transakcji, czyli spowalnia system. Metoda podstawowa jest najszybsza, ale za to ewentualne wycofywanie transakcji staje się bardzo kosztowne i problematyczne. Dlatego mechanizm znaczników czasowych stosuje się w systemach, w których występują głównie transakcje czytające, a wycofywanie zdarza się bardzo rzadko. W innym przypadku mechanizm blokad okazuje się być lepszy.

2.5.1.Znaczniki czasowe z wieloma wersjami

Metoda znaczników czasowych z wieloma wersjami jest kolejną modyfikacją mechanizmu podstawowego. Zakłada ona możliwość dostępu transakcji nie tylko do jednej – ostatniej wersji elementu bazy danych, ale także do jego wcześniejszych wersji. Wobec tego transakcja pisząca nie nadpisuje wartości elementu, ani jego znaczników czasowych, tylko tworzy jego kolejną wersję. Dzięki temu transakcja, która byłaby odrzucona lub wstrzymana w podstawowej metodzie znaczników czasowych, mogłaby zostać wykonana. Zwiększa to wielodostęp i sprawność systemu, ale jest za to bardziej pamięciochłonne. Reguły dla planisty posługującego się znacznikami czasowymi w wielu wersjach wyglądają następująco:

1. Transakcja T chce wykonać instrukcję READ(X) – planista wyszukuje taką wersję elementu X, np. X' ,której znacznik czasowy spełnia zależność: $WT(X') \leq TS(T)$ oraz nie ma innej wersji elementu X np. X'' takiej, że $WT(X') < WT(X'') \leq TS(T)$. Następnie dla tej wersji elementu zostaje ustalony nowy znacznik czasowy $RT(X') := TS(T)$,
2. Transakcja T chce wykonać instrukcję WRITE(X) – wówczas tworzona jest nowa wersja elementu X z nowymi znacznikami czasowymi, ale zanim zostanie to dokonane planista

musi sprawdzić, czy inna transakcja – młodsza od T - nie czytała tego elementu, tzn. czy nie zachodzą następujące zależności:

Dla wersji elementu X'*, takiego, że* $WT(X') \leq TS(T)$ *oraz dla którego nie istanieje* X'' *takie, że* $WT(X'') < WT(X') \leq TS(T)$*, zachodzi* $RT(X') \geq TS(T)$

Gdyby te warunki zostały spełnione, transakcja musiałby zostać odrzucona i uruchomiona jeszcze raz z nowym znacznikiem czasowym, gdyż inaczej doszłoby do zbyt późnego pisania.

Zauważmy, że w tak zorganizowanym mechanizmie znaczników czasowych, jedynymi transakcjami, które mogą zostać przerwane są transakcje piszące, natomiast transakcje czytające wykonają się zawsze.

Przechowywanie wielu wersji elementów bazy danych wymaga dodatkowego miejsca w pamięci. Jednak nie jest to aż tak duży problem, gdyż stare wersje elementów, takie, które nie będą już potrzebne można usunąć. Aby stwierdzić, które wersje będą potrzebne, a które nie, należy znaleźć najstarszą działającą transakcję w systemie. Następnie porównujemy znaczniki czasowe wersji ze znacznikiem czasowym tej transakcji. Jeśli dla elementu X, istnieją co najmniej dwie wersje elementu takie, że $WT(X_1) < WT(X_2) < .... < WT(X_k) \leq TS(T)$, gdzie $k \leq 2$,a T jest najstarszą transakcją w systemie , to wówczas możemy usunąć wszystkie wersje starsze od X_k , czyli $X_{1,} X_{2,} ..., X_{k-1}$.

2.6.Walidacja

Kolejnym optymistycznym mechanizmem sterowania współbieżnego jest tzw. *walidacja*. Tak jak w przypadku znaczników czasowych, walidację stosuje się w sytuacji, gdy jest pewność, że wycofywanie transakcji będzie miało miejsce rzadko. W metodzie tej nie przechowuje się znaczników czasowych dla wszystkich elementów. Nie jest to potrzebne, gdyż transakcje nie pracują na nich bezpośrednio na danych, lecz tylko na kopiach. Znaczniki czasowe przypisywane są tylko i wyłącznie transakcjom.

Mechanizm sterowania współbieżnego z walidacją charakteryzuje się występowaniem trzech faz:

1. Faza odczytu - W trakcie tej fazy, transakcja odczytuje wszystkie potrzebne elementy bazy danych i umieszcza ich kopie w zmiennych lokalnych. Wszystkie obliczenia i modyfikacje

są dokonywane na tych kopiach,

2. Faza walidacji – Jest to tzw. faza kontroli poprawności,, w trakcie której sprawdzane jest, czy zatwierdzenie transakcji nie naruszyłoby szeregowalności. Jeśli planista stwierdzi, że dana transakcja nie zaburza szeregowalności i spójności danch, to ją *uprawomocnia*, czyli zezwala na jej dalsze wykonanie. W przeciwnym wypadku transakcja jest wycofywana,
3. Faza zapisu – Faza ta następuje, jeśli transakcja pomyślnie przeszłą fazę walidacji. Tutaj transakcja jest faktycznie wykonywana, czyli wszelkie modyfikacje są przenoszone na elementy w pamięci trwałej bazy danych (nie dotyczy transakcji czytających).

W trakcie przebiegu transakcji przez kolejne fazy przypisywane są niej następujące znaczniki czasowe:

- START(*T*) – czas rozpoczęcia się transakcji,
- VAL(*T*) – czas zakończenia fazy walidacji, (czas uprawomocnienia transakcji),
- FIN(*T*) – czas zakończenia się transakcji (zakończenia fazy trzeciej).

Ponadto transakcja przechowuje dwa zbiory, w których przechowuje kopie elementów. Są nimi:

- *zbiór odczytów – RS(T)*, w którym przechowywane są wszystkie elementy czytane przez transakcje,
- *zbiór zapisów – WS(T*), w którym znajdują się elementy, które transakcja zapisuje.

Zbiory te są wypełniane w pierwszej fazie działania transakcji, czyli w fazie odczytu. Jeśli użytkownik zechce wycofać transakcję i wyda instrukcję ROLLBACK, wówczas zwalniana jest pamięć zajmowana przez transakcję i zbiory odczytów i zapisów zostają usunięte. Jeśli natomiast transakcja kończy się zatwierdzeniem, to przechodzi do fazy drugiej.

Walidacja polega na sprawdzeniu, czy transakcje nie powodują kolizji, czyli czy nie pracują na tych samych danych i czy wszelkie zapisy dokonują w odpowiedniej kolejności. Aby transakcja *T* mogła zostać uprawomocniona, jej zbiory odczytów i zapisów są porównywane ze wszystkimi transakcjami zatwierdzonymi lub tymi, które już przeszły fazę walidacji. Porównywanie z transakcjami, które rozpoczęły się, ale nie zostały uprawomocnione, nie jest potrzebne, gdyż takie transakcje pracują tylko w pamięci i nie podjęły jeszcze istotnych operacji na danych. Wobec tego nie mają jeszcze znaczenia dla problemu szeregowalności. W związku z tymi porównaniami planista musi przechowywać informacje o wszystkich transakcjach (ich znaczniki czasowe i zbiory odczytów i zapisów), które pomyślnie przeszły przez fazę walidacji. Regułę walidacji można sformułować następująco:

Niech S będzie zbiorem, w którym będą umieszczane wszystkie transakcji zatwierdzone lub uprawomocnione. W trakcie fazy walidacji transakcji T sprawdzane są następujące warunki:

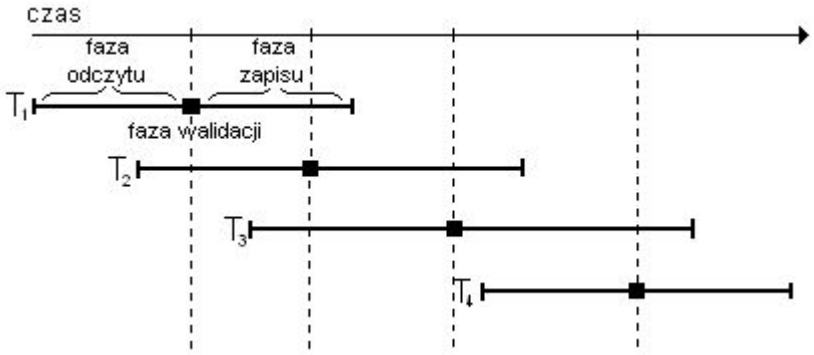
1. czy dla $T_i \in S$ zachodzi FIN(T_i)<START(*T*),
2. czy dla $T_i \in S$ takich, że START(*T*)<FIN(T_i) zachodzi $RS(T) \cap WS(T_i) = \emptyset$,
3. czy dla $T_i \in S$ takich, że <VAL(*T*)<FIN(T_i) zachodzi $WS(T) \cap WS(T_i) = \emptyset$

Warunek pierwszy oznacza, że transakcje, które zakończyły się przed rozpoczęciem transakcji *T* nie tworzą z nią kolizji. Drugi i trzeci warunek dotyczy transakcji, które zakończyły się po rozpoczęciu transakcji *T*. Jeżeli jakaś transakcja zakończyła się przed rozpoczęciem fazy walidacji transakcji *T*, to wymagane jest sprawdzenie, czy zbiór zapisów transakcji zakończonej i zbiór odczytów transakcji *T* nie ma elementów wspólnych (warunek drugi). Dzięki temu wiemy, czy elementy pobrane przez transakcję *T* w fazie odczytu są nadal aktualne. Jeśli natomiast jakaś transakcja nie zakończyła się przed rozpoczęciem fazy walidacji transakcji *T*, tylko została uprawomocniona, to konieczne jest sprawdzenie czy jej zbiór zapisów nie posiada elementów wspólnych ze zbiorem zapisów i zbiorem odczytów transakcji *T* (warunek drugi i trzeci). W tym wypadku musi zostać sprawdzona nie tylko aktualność przeczytanych elementów, ale również czy pisanie przez transakcję *T* nie zakłóca szeregowalności, tzn. czy inna transakcja nie powinna dokonać zapisu przed transakcją *T*. Jeśli dla którejkolwiek transakcji $T_i \in S$ odpowiedź na wszystkie powyższe trzy pytanie brzmi „nie", to transakcja *T* nie może zostać uprawomocniona. Musi więc zostać wycofana i po pewnym czasie uruchomiona ponownie. Takie wycofanie nie jest kosztowne, gdyż aby anulować transakcję wystarczy zwolnić zajmowaną przez nią pamięć.

W momencie, w którym transakcja uzyskuje uprawomocnienie, przechodzi do fazy trzeciej. Wszelkie modyfikacje zostają faktycznie zapisane w bazie danych i tylko awaria może ten proces przerwać.

Przykład 2.9

Na poniższym rysunku faza walidacji jest oznaczona jako punkt, gdyż w systemach jednoprocesorowych w czasie jej wykonywania faktycznie nie działają żadne inne operacje. W systemach wieloprocesorowych powstaje problem, gdyż wiele transakcji może przechodzić przez walidację współbieżnie.



Rys. 2.5. źródło własne

Rozważmy przedstawiony system. W momencie walidacji transakcji T_1 , planista na swojej liście nie ma jeszcze żadnej transakcji uprawomocnionej, więc T_1 nie wchodzi w kolizję z żadną inną transakcją. Pomyślnie przechodzi walidację, trafia na listę planisty i przechodzi do fazy zapisu. Gdy transakcja T_2 przechodzi fazę walidacji, sprawdzane jest, czy nie następuje kolizja z T_1 . Zbiór zapisów $WS(T_2)$ jest porównywany ze zbiorami zapisów i odczytów transakcji T_2 , gdyż spełnione są założenia warunków 2 i 3 z reguły walidacji. Jeśli iloczyny zbiorów są puste, transakcja T_2 może przejść do fazy zapisu. W fazie walidacji T_3 , dla transakcji T_1 wykonywany jest warunek drugi, a dla transakcji T_2 warunek drugi i trzeci. W momencie walidacji T_4 dla T_1 wykonywany jest warunek pierwszy, dla T_2 i T_3 warunek drugi. Oczywiście transakcje T_1 , T_2 i T_3 są uwzględniane, jeśli uprzednio pomyślnie przeszły przez swoje fazy walidacji. W przeciwnym przypadku nie zostają one umieszczone na liście planisty, wobec tego nie zostają uwzględnione przy walidacji kolejnych transakcji.

Wszystkie trzy omówione wyżej techniki sterowania: blokady, znaczniki czasowe i walidacja mają swoje wady i zalety. Wybór odpowiedniego mechanizmu zależy od kilku czynników, tj. poziom współbieżności systemu, rodzaj transakcji wykonywanych na danych (czy są to głównie transakcje czytające czy piszące), wielkość dostępnej pamięci operacyjnej, wymagana szybkość wykonywania transakcji, problemy związane z wycofywaniem się transakcji itp. Jeśli współbieżność systemu jest duża to preferowany jest mechanizm blokowania. Wówczas transakcje działają nieco wolniej, ale za to unika się cofania transakcji, co zwłaszcza przy transakcjach o długim czasie trwania jest bardzo kosztowne. Natomiast przy małej współbieżności lub gdy występują głównie zapytania czytające wydajniejsze są

walidacja i znaczniki czasowe. Z koleji te techniki optymistyczne są bardziej pamięciochłonne. W przypadku znaczników czsowych trzeba przechowywać znaczniki dla każdego elementu bazy danych, dla walidacji konieczne jest zapamiętanie zbioru zapisów i odczytów wielu transakcji, nawet tych zakończonych. Natomiast w przypadku blokad zajęta pamięć jest proporcjonalna do ilości blokowanych elementów.

Decyzja należy do projektanta bazy.

2.7.Mechanizmy sterowania współbieżnością w popularnych bazach danych

Zdecydowana większość baz danych to systemy wielodostępowe, a więc współbieżne. Czas więc przyjrzeć się temu, jak twórcy popularnych systemów bazodanowych poradzili sobie z problemem współbieżności.

Oracle 9i

W systemie Oracle transakcje mogą działać na dwóch poziomach izlolacji: READ COMMITTED (domyślny tryb) i SERIALIZABLE. Określenie trybu izolacji odbywa się za pomocą SQL-owej instrukcji SET TRANSACTION ISOLATION LEVEL *poziom izolacji*. Jest również możliwość zdefiniowania typu transakcji, czyli czy jest to transakcja czytająca czy pisząca (instrukcja SET TRANSACTION READ/WRITE). Domyślny typ to transakcja READ-WRITE, czyli czytająco-pisząca.

Zastosowanym mechanizmem sterowania współbieżnością jest mechanizm blokad. Oracle rozróżnia następujące typy blokad:

- blokady DDL – stosowane przy instrukcjach DDL, czyli dotyczących struktury tabel, perspektyw, schematów
- blokady DML - do blokowania danych: *exclusive,share*(wyłączne i wspólne zakładane na tabele), *exlusive-row, share-row*(wyłączne i wspólne zakładane na poszczególne wiersze), *share-row-exclusive*(blokada aktualizująca zakładana na wiersz w wyniku zapytań SELECT FOR UPDATE), *share-update* (do modyfikowania wierszy). Blodady *exlusive-row, share-row-exclusive, share-update* są nakładane na wiersze przez transakcje

modyfikujące. Różnica między nimi jest taka, że tryb *exlusive-row* nie pozwala na nałożenie jednoczesne żadnej innej blokady. W przypadku *share-row-exclusive* inne transakcje mogą nakładać blokady *share-row* lub *share-update*, ale z modyfikacją muszą poczekać aż *share-row-exlusive* zostanie zdjęta. Natomiast *share-update* nie jest kompatybilna tylko z blokadami *exlusive*. Jeśli jakaś transakcja nałoży blokadę *share* na wiersz zablokowany już w trybie *share-update*, to transakcja, która założyła *share-update* nie może modyfikować danych dopóki blokady dzielone nie zostają zdjęte. Blokady typu *share* mogą zostać przekształcona na *exlusive* przez tą samą transakcję pod warunkiem, że jest to jedyna transakcja blokująca aktualnie dany element.

Blokady zakładane są bez wiedzy użytkownika i zdejmowane dopiero w momencie wypełnienia się transakcji. Istnieje jednak instrukcja SQL: LOCK TABLE *nazwa tabel*i IN *typ blokady* MODE, która daje możliwość użytkownikowi jawnego założenia blokady na tabelę w sytuacji, gdy obawia się, że dany poziom izolacji nie wystarczy do pełnego zapewnienia spójności. Informacja o blokadach nie jest przechowywana w globalnej tablicy blokad, ale w bloku danych, w którym znajduje sie dany wiersz.

W systemie Oracle istnieje również automatyczny system wykrywania zakleszczeń. Usuwanie zakleszczenia odbywa się poprzez wycofanie jednego z poleceń (nie całej transakcji) uwikłanego w zakleszczenie. Transakcja otrzymuje odpowiedni komunikat o wycofaniu i zazwyczaj musi ona już zostać wycofana jawnie.

MySQL 5.1

W bazie danych MySQL również problem współbieżności jest rowiązywany za pomocą blokad. Dotyczy on, tak jak w przypadku odtwarzania, tylko tabel typu InnoDB. Stosowane są następujące tryby blokowania: wspólne (*S*) i wyłączna (*X*). Blokady te są nakładane na poziomie wierszy. Ponadto używane są blokady ostrzegawcze: *IS, IX*. Domyślnie w przypadku wykonania zapytania typu SELECT... nie jest nakładana blokada dzielona. Ma to na celu zwiększyć współbieżność bazy. Blokada S jest nakładana dopiero w wyniku jawnego określenia w zapytaniu: SELECT ... IN SHARE MODE. Nałożenie blokady na wiersz może powodować tzw. *next-key-locking,* czyli zablokowanie przestrzeni między wierszami. Powoduje to, że inne transakcje nie mogą wstawiać nowych wierszy i dzięki temu rozwiązuje

się problem z fantomami. Rozszerzenie blokady o *next-key-locking* zależy od przyjętego poziomu izolacji.

W MySQL-u dostępne są wszystkie cztery poziomy izolacji. Domyślnym jest REPEATABLE READ. W tym trybie zapytania SELECT ... FOR UPDATE, SELECT ... IN SHARE MODE oraz UPDATE, DELETE, które są używają do wyszukania elementu unikalnego indexu, nie powodują blokowania przestrzeni między rekordami, natomiast pozostałe zapytania już tą funkcję posiadają. Tryb SERIALIZABLE blokuje tak jak REPEATABLE READ oraz dodatkowo nakłada blokady dzielone w wyniku zwykłych zapytań SELECT ... . W trybie READ COMMITTED funkcja *next-key-locking* jest stosowana tak jak w REPEATABLE READ. Różnica między tymi dwoma trybami jest taka, że w REPEATABLE READ każde zapytanie typu SELECT ... w ramach jednej transakcji zwraca zawsze tą samą wersję bazy danych utworzoną przy pierwszym takim zapytaniu. Natomiast transakcja wykonująca się w trybie READ COMMITTED widzi zawsze świeżą wersję danych, czyli modyfikacje innych transakcji zatwierdzonych jak i swoje. Na poziomie izolacji READ UNCOMMITTED zapytania SELECT ... odbywają się bez żadnego blokowania. W pozostałych przypadkach działa jak READ COMMITED.

MySQL posiada również mechanizm do wykrywania zakleszczeń. W takiej sytuacji jedna transakcja jest przerywana i wycofywana. Na „ofiarę” wybierana jest transakcji mniejsza, czyli taka, która zmodyfikowała mniejszą ilość wierszy. Jednak mechanizm ten nie wykryje zakleszczenia, jeśli blokada została założona jawnie na całą tabelę za pomocą instrukcji LOCK TABLE ... W takich sytuacjach zalecane jest ustawienie odpowiedniej wartości dla zmiennej systemowej *innodb_lock_wait_timeout.* Spowoduje to przerwanie transakcji po tak określonym czasie oczekiwania na przydzielenie blokady.

PostgreSQL 8.1.3

Tak jak w poprzednich bazach danych PostgreSQL stosuje mechanizm blokowania do sterowania współbieżnością. Planista posługuje się następującymi typami blokad: wspólne, wyłączne i aktualizujące. Zakleszczenia są wykrywalne i powodują przerwanie jednej z transakcji, umożliwiając kontynuację drugiej. Dostępne poziomy izolacji to READ COMMITTED i SERIALIZABLE. Ten pierwszy jest trybem domyślnym. Jeśli dwie transakcje chcą modyfikować ten sam element bazy danych to na poziomie READ

COMMITTED transakcja późniejsza jest wstrzymywana i oczekuje na przydzielenie blokady. Natomiast w trybie SERIALIZABLE transakcja późniejsza zostaje wycofana. Menedżer transakcji dysponuje więc znacznikami czasowymi każdej transakcji.

3.Transakcje w obiektowej bazie danych „SBQLLite”[1]

W poprzednich rozdziałach omówione zostały zagadnienia dotyczące transakcji w bazach danych. Większość literatury temat ten przedstawia w odniesieniu do obiektowo-relacyjnych baz danych, gdyż taki model jest najczęściej spotykany. Jednak problem transakcyjności dotyczy jednakowo relacyjnych, obiektowych, hierarchicznych jak i mieszanych modeli baz danych. Przedstawione wcześniej rozwiązania są uniwersalne i dają się zastosować do każdego modelu, różnice występować będą głównie w sposobie implementacji.

Moim celem przy pisaniu niniejszej pracy było zapoznanie się z szeroko pojętym tematem transakcji w bazach danych i zaimplementowanie mechanizmu transakcyjnego do powstającego aktualnie projektu akademickiego - obiektowej bazy danych „SBQLLite”

3.1.O projekcie „SBQLLite”

„SBQLLite” jest projektem eksperymentalnym mającym na celu powstanie systemu zarządzania obiektową bazą danych. Projekt jest tworzony przez grupę studentów informatyki czwrtego i piątego roku, a głównym jego koordynatorem jest dr Piotr Wiśniewski.

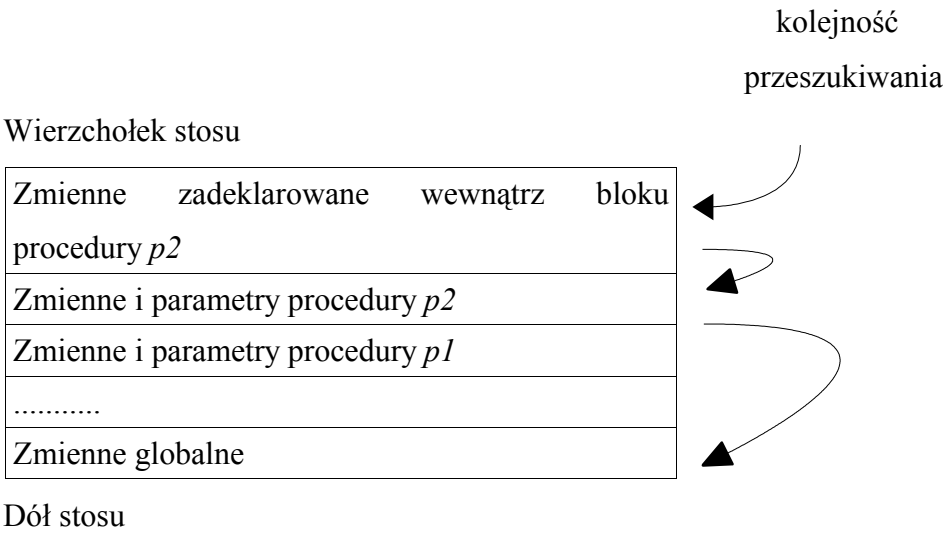
Coraz większa popularność i rozwój obiektowości w technologiach informatycznych zmusiły informatyków do zastosowania tej koncepcji w dziedzinie baz danych. Podstawowe zalety obiektowości jak: obiekt, klasa, interfejs, metoda, dziedziczenie, itp. spowodowały powstanie zupełnie nowych jakości w dziedzinie baz danych. Jednak brak standardów, precyzyjnych podstaw teorii oraz luźne założenia dotyczące obiektowości sprawiają, że powstałe obiektowe i relacyjno-obiektowe bazy danych charakteryzuje koncepcyjny brak spójności, a stosowane w nich języki zapytań różnią się znacznie i trudno je nazwać obiektowymi językami programowania. Przyczyną tego chaosu jest fakt, że do tej pory twórcy obiektowych baz danych próbowali przenieść rozwiązania stosowane w modelu relacyjnym na grunt obiektowy. Dominują więc tu modyfikacje i rozszerzenia teorii relacyjnego modelu bazy danych, próby tworzenia algebry obiektów w miejsce algebry relacji oraz przypadkowe rozwiązania praktyczne. Celem projektu „SBQLLite” jest więc stworzenie takiego systemu zarządzania obiektową bazą danych, który byłby spójny koncepcyjnie zarówno w zakresie

1 Nazwa „SBQLLite” jest nazwą roboczą dla projektu prototypowego

konstrukcji składu bazy danych, jak i semantyki języka zapytań. Podstawą projektu jest obiektowy język zapytań SBQL (*Stack-Based Query Language*), którego pełny opis składni, cechy i zastosowania zostały opisane w książce Kazimierza Subiety „Teoria i konstrukcja obiektowych języków zapytań".

Główną ideą konstrukcji tego języka jest „podejście stosowe", gdyż na zasadzie stosu środowisk skonstruowana jest większość języków programowania. Mechanizm ten pozwala na określanie zakresu nazw, wiązanie nazw, wywoływanie procedur i realizację cech obiektowości (tj. definiowanie klas, dziedziczenie itp). Podejście stosowe polega na tym, że dla każdego programu tworzone jest środowisko, podzielone na kilka podśrodowisk (tzw. sekcje). W ramach danego sekcji znajdują się zmienne, stałe, procedury, klasy i inne byty programistyczne wykorzystywane w trakcie działania programu. Podśrodowiska tworzą układ hierarchiczny w postaci stosu, czyli sekcje znajdujące się na wierzchu stosu mają najwyższy priorytet. W czasie działania programu tworzone są nowe podśrodowiska i zawsze dokładane do stosu na jego wierzch. Usunięcie się środowiska odbywa się przez zdjęcie go ze stosu, z tym że zdejmować można tylko z wierzchu. Działanie stosu środowisk wyjaśnię na przykładzie.

Przykład 3.1.



rys.3.1.źródło.[8] str 177

Załóżmy, że mamy program składający się z procedury *p1*, która to wywołuję procedurę *p2*. W momencie startu programu tworzone jest pierwsze środowisko na stosie zawierające byty globalne. W chwili wywołania procedury *p1* na stos dokładana jest sekcja zawierająca środowisko lokalne procedury *p1*. Gdy wywołana zostanie procedura *p2*, na wierzch stosu

dołożona zostanie kolejna sekcja będąca podśrodowiskiem procedury *p2*. Następnie dla każdego bloku procedury *p2*, na wierzchu stosu tworzona będzie kolejna sekcja. W rezultacie powstanie stos środowisk jak na rysunku 3.1.

Przyjmijmy, że program będzie w momencie wykonywania wewnętrznego bloku procedury *p2* i będzie konieczne odwołanie się do zmiennej np. *x*, czyli związanie nazwy *x* z odpowiednim bytem programistycznym. Poszukiwanie tej zmiennej będzie się odbywało od wierzchołka stosu. Jeżeli wewnątrz bloku nie ma zmiennej *x*, to zostanie przeszukane środowisko znajdujące się poniżej na stosie, czyli środowisko procedury *p2*. Schodzenie w dół po stosie odbywa się z zachowaniem tzw. *scoping rules*[2], czyli pewnych reguł omijania niektórych sekcji. Na rysunku są one oznaczone strzałkami. Proces przeszukiwania stosu będzie kontynuowany, aż do momentu znalezienia szukanego bytu lub aż do osiągnięcia „dna" stosu.

Dzięki zastosowaniu stosu środowisk do baz danych możliwe jest stworzenie takiego języka zapytań, który byłby językiem programowania, niezależnego od modelu składu danych. Zanim przedstawię stos środowisk dla bazy danych, opiszę w jaki sposób dane w obiektowej bazie będą składowane na dysku.

W przypadku relacyjnego modelu jednostkami danych są : krotka, wiersz, tabela. Natomiast w obiektowych bazach danych podstawowym elementem danych jest po prostu obiekt. W projekcie SBQLLite wyróżnione zostały trzy rodzaje obiektów:

1. *obiekt atomowy* – jest to trójka <*i,n,v*>, gdzie *i* – to unikalny wewnętrzny identyfikator nadawany przez system, *n* – nazwa zewnętrzna nadawana przez użytkownika, administratora itp., *v* – wartość atomowa, czyli wartość niepodzielna,
2. *obiekty pointerowy* – jest to trójka $<i_1, n, i_2>$. Obiekt ten jest referencją – wskaźnikiem, gdzie i_1 jest wewnętrznym identyfikatorem obiektu, i_2 - jego wartością, czyli identyfikatorem obiektu, na który wskazuje, a *n* – to nazwa zewnętrzna,
3. *obiekt kompleksowy – złożony* - reprezentowany jest przez trójkę <*i,n,T*>, gdzie *i,n* – to identyfikator i nazwa, a *T* - to zbiór dowolnych obiektów.

W [8] zdefiniowano cztery rodzaje modeli baz danych: M0, M1, M2 i M3. Najprostszy jest model M0. „W modelu M0 *skład obiektów* jest zdefiniowany jako para <*S,R*>, gdzie *S* jest zbiorem obiektów, zaś *R* jest zbiorem identyfikatorów określanych jako *identyfikatory*

2 Są to reguły zakresu wiązania nazw. Jeżeli wykonuje się procedura *p2* to powinny być widoczne nazwy zmiennych lokalnych procedury i zmiennych globalnych programu, natomiast zmienne z procedury *p1* powinny zostać ukryte na czas wykonania procedury *p2*

startowe”.[8]

Przykładową bazę danych w modelu M0 reprezentuje poniższy rysunek

S – Obiekty:

< i_1 , prac,{ < i_2 , nazwisko, „Kowalski”>,
< i_3 , zarobek, 2000>,
< i_4 , pracuje_w, i_{13} > } >,
< i_5 , prac,{ < i_6 , nazwisko, „Nowak”>,
< i_7 , zarobek, 2500>,
< i_8 , adres, { < i_9 , miasto, „Toruń”>,
< i_{10} , ulica, „Bydgoska”>,
< i_{11} , nr_domu, 15> } >,
< i_{12} , pracuje_w, i_{13} > } >,
< i_{13} , dział, { < i_{14} , nazwa, „Produkcja”>,
< i_{15} , zatrudnia, i_1 >
< i_{16} , zatrudnia, i_5 > } >

R – identyfikatory startowe:

i_1 , i_5 , i_{13}

Rys. 3.2źródło: za [8], str 148

Jak widać obiekt kompleksowy może składać się z obiektów atomowych, pointerowych, jak i kompleksowych. Złożoność obiektu może być ograniczona tylko względami logicznej konstrukcji bazy danych.

Zbiór *R* tworzą identyfikatory tych obiektów, które leżą najwyżej w hierarchii. Powyższą bazę można przedstawić w postaci trzech drzew, których korzeniami będą identyfikatory startowe: i_1 , i_5 , i_{13} , a pozostałe obiekty będą tworzyć poddrzewa (obiekt i_8) i liście.

Kolejnym modelem składu danych jest M1. Jest to rozszerzenie modelu M0 o taki pojęcia jak klasa i dziedziczenie. „Klasa jest miejscem przechowywania tych informacji dotyczących grupy obiektów, które są dla nich niezmienne, wspólne lub dotyczą całej ich populacji. Takie informacje są nazywane *inwariantami*.” [8]. Dziedziczenie jest relacją pomiędzy klasami lub pomiędzy obiektem i klasą. Każdy obiekt i każda klasa dziedzicząca z klasy nadrzędnej posiada wszystkie własności – inwarianty klasy nadrzędnej. Model M1 definiowany jest więc

jako czwórka <*S,R,KK,KO*>, gdzie *S* i *R* oznaczają to samo co w modelu M0, *KK* – to zbiór związków dziedziczenia między klasami, a *KO* – to relacje dziedziczenia między obiektami i klasami. W składzie klasa jest definiowana w postaci zwykłego obiektu, np:

< i_{50} , klasa_osoba, { < i_{51} , wiek, (...kod metody wiek...)>,
inwarianty: nazwa_obiektów=”osoba”
pole nazwisko,
pole rok_urodzenia,
pozostałe inwarianty klasy ...}>

Rys.3.3. źródło [8] str 166

Każdy obiekt dziedziczący po tej klasie posiadałby podobiekt *nazwisko* i podobiekt *rok_urodzenia*. W klasie można zdefiniować jakie mają to być podobiekty (atomowe, pointerowe, czy kompleksowe), określić typ ich wartości (string, integer) oraz ograniczenia (np. wartość minimalną, maksymalną lub wymóg unikalnej wartości). Ponadto w klasie można zdefiniować metody, które będą mogły być wykonywane na obiektach dziedziczących, w przykładzie jest to metoda *wiek*. W zbiorze *KO* znajdują się pary $<i_m, i_n>$, gdzie i_m to identyfikator obiektu, a i_n , to identyfikator klasy, po której obiekt i_m dziedziczy. W zbiorze *KK* znajdują się analogiczne pary z tym, że obydwa identyfikatory dotyczą klas. Jeśli np. *klasa_pracownik* dziedziczy po *klasa_osoba*, to każdy obiekt dziedziczący po *klasa_pracownik* będzie posiadał wszystkie inwarianty określone w obydwu klasach. Pełny przykład modelu M1 wraz z opisem znajduje się w [8] str.166-167.

Model M2 jest rozszerzeniem modelu M1 o zbiór *OO*, w którym określone są relacje dziedziczenia pomiędzy obiektami. Obiekty dziedziczące po innym obiekcie, nazywamy jego *rolami*. Weźmy np. obiekt *osoba* dziedziczący po klasie *klasa_osoba*, oraz obiekt *prac* dziedziczący po klasie *klasa_pracownik* :

< i_1 , osoba,{ < i_2 , nazwisko, „Kowalski”>, < i_3 , rok_urodzenia, 1980> } >
< i_4 , prac,{ < i_6 zarobek, 2500> } >

Rys.3.4. na podstawie [8] str 170

Jeśli obiekt *osoba* jest rolą obiektu *prac*, to wówczas dziedziczy pola *nazwisko* i *rok_urodzenia* nawet, jeśli nie został określony związek dziedziczenia między klasami *klasa_osoba* i *klasa_pracownik*. Różnica między dziedziczeniem po obiekcie, a dziedziczeniem po klasie jest taka, że w tym pierwszym przypadku dziedziczone są pola wraz

z ich wartościami. Określenie roli pozwala odwołać się do pola *nazwisko* poprzez obiekt *prac* mimo, iż nie jest ono podobiektem obiektu *prac* (rozszerzony przykład modelu M2 znajduje się w [8] str. 170-172).

Model M3 uwzględnia ponadto hermetyzację[3]. Tak jak w obiektowych językach programowania (np. Java) istnieje możliwość określenia, które pola obiektu mogą być widoczne „z zewnątrz", a które nie, tak w modelu M3 możliwe jest ukrycie pewnych własności, czy metod obiektu bazy danych. Model M3 może być rozszerzeniem modelu M1 bądź M2 poprzez dodanie do definicji klas inwariantu *lista eksportowa*, zawierającej listę nazw własności i metod, które mają być publiczne (przykład znajduje się w [8] str. 172-173).

Dla każdego modelu składu danych można zbudować stos środowisk, oczywiście odpowiednio dostosowany do potrzeb języka zapytań i specyfiki baz danych. Na stosie oprócz bytów programistycznych, takich jak dla języka programowania(metody, zmienne), znajdą się przede wszystkim obiekty bazy danych, albo raczej nazwy tych obiektów. W końcu najczęściej wykonywaną operacją na bazie danych jest zapytanie o konkretne dane. W trakcie przeszukiwania danych na stos będą wkładane i zdejmowane nowe środowiska. Uściślając, dla języków zapytań „stos środowisk składa się z sekcji odpowiadających poszczególnym środowiskom czasu wykonania. Sekcja jest *zbiorem binderów* do bytów programistycznych odpowiadającego jej środowiska... *Binder* jest parą *n(x)*, gdzie *n* może być dowolną zewnętrzną nazwą definiowaną przez programistę, użytkownika, projektanta aplikacji, projektanta bazy danych itp., zaś *x* może być dowolnym rezultatem zwracanym przez zapytanie"[8]. Nie ma przeszkód by kilka binderów miało tą samą nazwę. Szukanie danych polega na wiązaniu nazwy. Chcąc znaleźć np. obiekt *prac* o zarobkach równych 2500, najpierw na stosie szukany jest binder o nazwie *prac.* Następnie dla każdego obiektu, uzyskanego poprzez związanie nazwy, na stos dołożona zostanie nowa sekcja zawierająca bindery odnoszące się do wnętrza obiektu. Stąd trafią na wierzchu stosu znajdą się bindery do podobiektów. Potem wiązana będzie nazwa *zarobek*. Jeśli pracownik posiadał taki atrybut, nazwa związana zostanie już w pierwszej sekcji stosu. Wreszcie wartość obiektu zostanie przeczytana i porównana z wartością szukaną. Przykładowy stos środowisk dla modelu M0 przedstawionego na rysunku 3.2. wygląda następująco:

3 „Hermetyzacja jest grupowaniem elementów składowych w obrębie jednej bryły i następnie umożliwienie manipulowania tą bryłą jako całością; wiąże się z ukrywaniem części informacji dotyczącej struktury i implementacji wnętrza tej bryły"[8]. Hermetyzacja może być ortodoksyjna (na zewnątrz widoczne są tylko metody, atrybuty są ukryte) lub ortogonalna (każda własność może być prywatna- ukryta albo publiczna)

prac(i_1) (sekcja chwilowa przetwarzania)
X(i_{127}), Y(i_{128}) N(5) I(„Maria”)
...(sekcja chwilowa przetwarzania)
nazwisko(i_6), zarobek(i_7), adres(i_8), pracuje_w(i_{12})
...(sekcja chwilowa przetwarzania)
Bindery do własności nietrwałych sesji użytkownika
prac(i_1), prac(i_5), dział(i_{13})
Bindery do globalnych funkcji bibliotecznych
Bindery do własności środowiska komputerowego

Rys.3.5. źródło [8] str 183

Pierwsze cztery sekcje od dołu stosu to sekcje danych globalnych i tworzone są przy starcie systemu bazy danych. Trzecia sekcja od dołu, to sekcja bazy danych, wkładane są do dniej bindery do wszystkich obiektów startowych (ze zbioru *R*). Kolejne sekcje są dokładane do stosu w trakcie przetwarzania zapytania.

Dla menedżera nie jest istotne jak zapytanie jest ewaluowane. Interesują go jedynie wyniki zapytań, czyli żądania o dostęp do konkretnych danych, nadaje im odpowiednią kolejność i następnie wykonuje je bądź nie. Dlatego dla menedżera transakcji bardziej istotne jest jaki model składu bazy danych został wybrany i jaki jest do nich dostęp.

3.2.Przebieg działania projektu

Wersja projektu dołączona do tej pracy oparta jest na składzie danych modelu M2. Model M3 nie został zrealizowany ze względu na to, iż zastosowany język zapytań jest stylizowany na języku Python, który nie uwzględnia hermetyzacji. Dla modelu w pełni opracowane zostało dziedziczenie po obiektach. Role nie są przechowywane w zbiorze, ale pamiętane jako dodatkowy atrybut obiektu. Możliwe jest również definiowanie klas, choć nie zostały jeszcze określone związki dziedziczenia między nimi. W momencie startu programu tworzony jest „bazowy” stos środowisk. Następnie uruchamiany jest menedżer transakcji, który na podstawie pliku XML, zawierającego *back-up* bazy danych, uruchamia proces tworzenia wszystkich obiektów bazy. W trakcie pracy systemu wszystkie dane znajdują się w pamięci

operacyjnej. Dla obiektów startowych tworzone są bindery i umieszczane w sekcji bazy danych na stosie. Każdy obiekt, oprócz argumentów takich jak: identyfikator, nazwa, wartość itp., posiada informację o tym, jakie obiekty pointerowe na niego wskazują (tzw. lista pointerów zwrotnych) oraz, w przypadku obiektów kompleksowych, po jakim obiekcie dziedziczy (atrybut isARoleOf) i jakie obiekty po nim dziedziczą (lista roli). Następnie menedżer sprawdza, czy wcześniejsza praca z bazą danych została poprawnie zakończona i w razie potrzeby uruchamia proces odtwarzania. Następnie powoływana jest do życia pierwsza transakcja. Odtąd system oczekuje na zapytania od użytkownika, które będą się wykonywała w ramach transakcji. Wydanie jawnej instrukcji COMMIT lub ROLLBACK, kończy transakcję i automatycznie uruchamia następną. Również poprawne wyjście z systemu powoduje zatwierdzenie i zakończenie transakcji. Istnieje możliwość wstawiania savepoint'ów do transakcji poprzez wydanie instrukcji SAVEPOINT *nazwa_savepoint'a.* Dzięki temu, za pomocą polecenia ROLLBACK *nazwa_savepoint'a,* możliwe jest wycofanie ostatnich zmian bez anulowania całej transakcji. Wywołanie tego polecenia nie powoduje zakończenia transakcji. Jeśli użytkownik napisze zapytanie czy instrukcję, zostanie ono sparsowane, podzielone na podzapytania i przekształcone do postaci drzewa. Wierzchołki drzewa tworzą operatory algebraiczne(+,=,< itp) i niealgebraiczne (where, as, (), .) użyte w zapytaniu, natomiast liście stanowią nazwy obiektów bądź konkretne wartości (literały). Każdy wierzchołek może posiadać co najwyżej dwie gałęzie. Przykładowe drzewo syntaktyczne zapytania znajduje się w [8] na str. 204. Do wykonywania drzewa zastosowane jest podejście stosowe. Najpierw wykonywane są operacje dla podzapytania znajdującego się najgłębiej na lewej gałęzi, potem te leżące wyżej. To samo się dzieje dla prawej gałęzi. Każde wykonywane podzapytanie tworzy kolejną sekcję na stosie środowiska,a jego wynik trafia na *stos rezultatów*[4]. Operator znajdujący się wyżej na drzewie bierze ze stosu rezultatów wynik uzyskany przez swoje poddrzewo i w ich kontekście oblicza swój rezultat. W ten sposób dochodzimy do korzenia drzewa i wyniku całego zapytania. Jeśli jakaś operacja zapytania chce pracować na obiektach, menedżer transakcji musi najpierw sprawdzić, czy jest możliwy dostęp do obiektów, czyli próbuje założyć blokadę i w przypadku instrukcji modyfikujących bazę, dokonuje wpisów w logu. Jeśli założenie blokady nie jest możliwe wówczas proces użytkownika jest wstrzymywany i wznawiany dopiero w momencie uzyskania blokady. Poprawne wyjście z sytemu następuje po wydaniu instrukcji EXIT. Wówczas ostatnia

4 Jest to stos, na którym odkładane są wszelkie pośrednie i końcowe rezultaty zapytań. Jest prostym uogólnieniem stosu arytmetycznego spotykanego w implementacji języków programowania.

transakcja zostaje zatwierdzona, a następnie tworzony jest nowy *back-up* bazy danych.

3.3. Implementacja projektu

Projekt „SBQLLite" został napisany w języku Python. Język ten jest językiem skryptowym, ale posiadającym podstawowe cechy obiektowości. Został wybrany ze względu na prostotę i wygodę pisania oraz ze względu na to, że jest językiem interpretowanym, wobec tego proces testowania i usuwania błędów jest bardzo szybki i efektywny. Ponadto w języku Python zmienne nie mają typów i są tworzone w momencie nadania im wartości. Jest to cecha bardzo istotna, gdyż pracując z bazą danych nie zawsze jesteśmy w stanie określić z jakimi typami wartości elementów będziemy pracować.

Moja rola w zespole polegała na stworzeniu i wprowadzeniu transakcji do bazy danych. Kod dotyczący składu danych, gramatyki języka, budowy stosu środowisk czy procedur przetwarzania został napisany przez innych członków zespołu, przeze mnie jedynie uzupełniany o elementy niezbędne dla menedżera transakcji. Główna część kodu napisana przeze mnie znajduje się w pliku *TransManger.py*, w którym znajdują się klasy *Transaction* oraz *TransManager*, oraz *SBQLLEng.py*, w którym zaimplementowałam metody operujące na danych (m.in. *delete*, *update*, *insert* itp.).

Projekt jest dopiero w fazie prototypowej, nie jest więc w pełni funkcjonalny, ani wolny od błędów. Pewne rozwiązania są dopiero testowane, a niektóre moduły są tak zależne od siebie, że nie rozwiązany problem w jednej części uniemożliwia rozwój innej. Projekt jest ciągle rozwijany i do zespołu developerów dołączają nowi studenci.

Cały mechanizm transakcyjny zaimplementowany jest w pliku *TransManager.py*. Aby utworzyć transakcję musi zostać utworzony obiekt klasy *Transaction*. Każdy obiekt tej klasy posiada następujące atrybuty:

- *id* - identyfikator transakcji nadawany na podstawie licznika menedżera,
- *lockList* - lista, na której przechowywane są identyfikatory obiektów blokowanych przez daną transakcję,
- *waitLockList* – lista zawierająca elementy bazy danych, dla których transakcja nie uzyskała blokady,
- *operationList* - lista zapamiętująca wszystkie operacje, jakie zostały wykonane w

ramach danej transakcji. Na podstawie tych zapisów możliwe jest późniejsze wycofywanie transakcji. Operacja jest zapisana w formacie krotki (*tuple*) języka Python, w następującej postaci:

(id, old_element, new_element),

gdzie *id* jest identyfikatorem obiektu, na którym wykonywana była operacja, *old_element* jest referencją do obiektu sprzed modyfikacji, natomiast *new_element* – referencją do wersji obiektu po modyfikacji. W przypadku wstawiania nowego obiektu do bazy, wartość *old_element* będzie *'null'*, a dla operacji typu DELETE, wartość *'null'* przyjmie *new_element*. Na podstawie tego, jakie wartości kryją się pod *old_element* i *new_element* menedżer transakcji wie, czy wykonywana operacja była operacją INSERT, UPDATE, czy DELETE. W momencie wstawienia savepoint'a do transakcji, na liście operacji pojawia się jednoargumentowa krotka zawierająca nazwę savepoint'a. W sytuacji wykonania instrukcji ROLLBACK *nazwa_savepoint'a* wycofywane są wszystkie wpisy znajdujące się na liście *operationList* za krotką zawierającą podaną nazwę, następnie lista jest w tym miejscu ucinana, a „ogon" usuwany. Ostatni atrybut transakcji informuje o tym, czy transakcja jest wstrzymana, czyli czy oczekuje na blokadę. Domyślnie przyjmuje wartość 0, co oznacza, że transakcja nie jest wstrzymywana. Jeśli transakcja nie może uzyskać blokady, menedżer nadaje temu bitowi wartość 1, co uniemożliwia dalsze działanie transakcji.

W klasie *Transaction* znajdują się ponadto metody dostępu do powyższych atrybutów (*getId*, *getLocks*, *addLock*, *delWaitLock*, *addWaitLock*, *addOpList*, *getOperationList*)

Menedżer transakcji jest obiektem klasy *TransManager* i jest uruchamiany zaraz przy starcie systemu. Posiada bardzo ważne zmienne globalne jak:

- *licznik* – na podstawie tej wartości ustalany jest identyfikator każdej transakcji,
- *logfile* – zmienna plikowa, przechowująca aktualnie otwarty plik logu,
- *table* – słownik będący tablicą blokad,
- *transactions* – słownik stanowiący zbiór transakcji aktywnych,
- *recoverylist* – lista plików logu, na podstawie których wykonuje się proces odtwarzania.

Ponadto, obiekt menedżera posiada pole zawierające czas startu menedżera (używany w nazwie plików logu i *back-up*'ów) oraz odwołanie do obiektu *Eng,* dla którego menedżer został wywołany. Strukturę menedżera transakcji można podzielić na trzy podstawowe części:

1. część odpowiedzialną za załadowanie bazę danych z *back-up'u* do pamięci, zapisanie *back-up'u* i procesy odtwarzania,

2. część odpowiedzialną za manipulowanie transakcjami i dokonywanie wpisów w logu,
3. część zajmującą się blokowaniem.

Główne metody pracujące w ramach pierwszej części to:

- *loadDataBase* – metoda ta odnajduje ostatni *back-up* bazy oraz przeszukuje pliki logu w celu sprawdzenia, czy ostatnia praca z systemem została zakończona poprawnie i zapisuje do *recoverylist* te, które będą niezbędne przy odtwarzaniu,
- *saveDataBase* – metoda ta służy do wykonywania *back-up,* jest uruchamiana przy poprawnym zamknięciu bazy,
- *recovery* – jest to główna procedura odtwarzająca, parsuje pliki logu z *recoveryList* i wybiera z nich wpisy transakcji zatwierdzonych, a następnie na ich podstawie powtarza ich operacje, otwiera również bieżący log.

Druga część menedżer składa się z następujących funkcji:

- *rememberOp* – funkcja zapisujące wszystkie operacje transakcji to jej listy *operationList,*
- *drawBackOp* – funkcja wycofująca operację, pochodzącą z *operationList,*
- *rollback* – powoduje wycofanie operacji do określonego momentu zatwierdzenia, przeszukuje *operationList* w celu znalezienia nazwy savepoint'a i dla operacji znajdujących się za nią wywołuja funkcję *drawBackOp,*
- *startTrans* – tworzy nową transakcje i dokonuje odpowiedniego wpisu do logu,
- *commitTrans* - kończy transakcję zatwierdzeniem, wywołuje funkcję zdejmującą blokady i dokonuje odpowiedniego wpisu w logu,
- *abortTrans* – kończy transakcję wycofaniem, zdejmuje blokady i wycofuje zmiany spowodowane działaniem transakcji poprzez wywołanie funkcji *drawBackOp* na każdej operacji z listy *operationList ,*
- *modifyTrans* – odpowiedzialna za dokonanie wpisów modyfikacyjnych do logu przy wstawianiu, usuwaniu i zmiany wartości elementów,
- *close* – funkcja zamknięcia menedżera, powoduje wywołanie funkcji *saveDataBase* oraz zapisania w logu informacji, na podstawie których przy ponownym starcie menedżer ustawia licznik transakcji i wie, czy musi uruchomić proces odtwarzania.

Różnica między metodami *rollback*, a *abortTrans* jest taka, że w tym pierwszym przypadku wycofywana jest tylko część operacji transakcji i dla każdej z nich tworzony jest wpis modyfikacyjny w logu. Druga funkcja wycofuje wszystkie operacje, ale w logu nie tworzone są wpisy modyfikacji, tylko wpis unieważnienia transakcji

Log jest plikiem tekstowym, a jego nazwa jest konkatenacją napisu „log” i wartości czasu startu menedżera transakcji. Dzięki temu wiadomo w jakim czasie log powstawał, co może być cenną informacją. Zapisy w logu są ciągami pól oddzielonych dwukropkiem i przyjmują następującą postać:

1. przy tworzeniu transakcji – start:*numer_transakcji,*
2. przy zakończeniu transakcji – abort:*numer_transakcji*, commit:*numer_transakcji*, w zależności czy transakcja została zatwierdzona czy wycofana,
3. wpisy modyfikujące – *operacja*:*numer_transakcji*:*typ_obiektu*:*wartości_obiektu*; pole *operacja* może mieć wartość: 'u', 'd' lub 'i', w zależności od tego czy modyfikacja powstała w wyniku operacji UPDATE, DELETE czy INSERT; pole *typ_obiektu* przyjmuje wartości 1- jeśli modyfikowany był obiekt atomowy, 2 - dla obiektu kompleksowego, 3 – dla pointerów; *wartości_obiektu* to dalszy ciąg wpisu, który może być różny w zależności od typu obiektu, i tak:
   - dla obiektu atomowego:

     id: *nazwa*:*RP*:*father*:*value,*

     gdzie pola kolejno oznaczają: identyfikator obiektu, nazwa obiektu, lista identyfikatorów pointerów zwrotnych, identyfikator obiektu będącego ojcem, czyli leżącego wyżej w hierarchii(w przypadku obiektów startowych wartość jest równa 0), oraz wartość obiektu,
   - dla obiektu kompleksowego:

     id: *nazwa*:*RP*:*father*:*sub*:*is_role_of*:*roles*,

     gdzie pierwsze cztery pola są analogiczne jak poprzednio, a następne to: lista identyfikatorów podobiektów, identyfikator obiektu, po którym dany obiekt dziedziczy, lista identyfikatorów obiektów dziedziczących po danym obiekcie,
   - dla obiektu pointerowego:

     id: *nazwa*:*RP*:*father*:*id2*,

     gdzie ostatnie pole to identyfikator obiektu, na który pointer wskazuje.

Przy poprawnym zamknięciu systemu pojawia się wpis będący identyfikatorem ostatniej transakcji oraz napis 'end'. Jeśli te zapisy znajdują się w logu, to oznacza że nie trzeba odtwarzać. Identyfikator ostatniej transakcji jest potrzebny dla menedżera, by mógł kontynuować numerację dla nowo tworzonych transakcji. Jeśli zdarzyła się awaria, to numer ostatniej transakcji jest uzyskiwany w trakcie odtwarzania.

Log był konstruowany według typu powtarzanie/unieważnianie. Stąd w przypadku

wykonywania operacji najpierw dokonywany jest wpis modyfikujący do logu, następnie operacja jest wykonywana od razu bezpośrednio na elemencie bazy danych, a wpis 'commit', czy 'abort' trafia do logu na samym końcu. Jednak, ponieważ cała baza danych w trakcie pracy jest trzymana w pamięci operacyjnej, modyfikacje nie są zapisywane do pamięci trwałej, wobec czego awaria powoduje utratę modyfikacji nawet transakcji dawno zatwierdzonych. Dlatego odtwarzanie po awarii polega powtórzeniu tylko awarii zatwierdzonych, natomiast unieważnianie transakcji wycofanych lub przerwanych jest zbędne. Faktycznie więc mechanizm odtwarzania działa jak dla logu z powtarzaniem.

Trzecia część menedżera transakcji odpowiada za sterowanie współbieżnością. Co prawda aktualna wersja projektu nie jest systemem wielodostępowym i nie mogłam przetestować w pełni mechanizmu blokowania, to wydaje się być on gotowy do spełnienia swej funkcji. Zaimplementowany został mechanizm blokowania wyłącznego i wspólnego (S,X) oraz blokowanie ostrzegawcze (IS, IX). Nie została wprowadzona blokada aktualizująca, ale możliwa jest zamiany blokady na mocniejszą, o ile tylko jedna transakcja blokuje element. Dodatkowo została wprowadzona blokada SIX – jako blokada grupowa elementu. Oznacza, że element został zablokowany przez jedną transakcję w trybie wspólnym, a przez inną blokadą IX. Blokada SIX jest kompatybilna tylko z IS. Blokady są nakładane w momencie chęci dostępu do obiektu, a zdejmowane dopiero w momencie zakończenia transakcji. Zachowany jest więc protokół dwufazowego blokowania. Informacje o blokowaniu są przechowywane w tablicy blokad, zorganizowanej na zasadzie słownika, który jako klucze posiada identyfikatory elementów blokowanych. Wartości tych kluczy mają postać również słownika zawierającego następujące informacje:

- *gr_mode* – tryb grupowy blokowania,
- *wait* – bit określający, czy jakaś transakcja czeka na przydzielenie blokady na ten element,
- *list* – lista transakcji blokujących dany element lub czekających na blokadę; każdy element listy ma strukturę słownika z polami :
    - *id* – identyfikator transakcji,
    - *lock_mode* – tryb w jakim blokuje transakcja.

W wyniku testów okazało się, że blokowanie do czytania jest blokowaniem bardzo zachłannym i mocno ogranicza współbieżność. Wystarczy proste zapytanie np. *prac where nazwisko=”Kowalski”* i mamy wszystkie elementy prac zablokowane wspólnie. Przez to nie możliwa jest modyfikacja jakiegokolwiek innego obiektu *prac* bądź ich podobiektów.

Zastanawiałam się nad usunięciem blokad wspólnych. Jednak takie rozwiązanie obniża mocno poziom izolacji: możliwe wówczas byłoby czytanie brudnopisów, czy fantomów. W aktualnej wersji menedżera zastosowane jest następujące rozwiązanie: w przypadku chęci czytania danych transakcja pyta się, czy mogłaby uzyskać blokadę wspólną na elemencie. Jeśli odpowiedź jest twierdząca, transakcja uzyskuje dostęp do elementu, ale blokada wspólna nie jest nakładana. Dzięki temu unika się blokowania całych grup obiektów oraz uniemożliwia czytanie elementów modyfikowanych przez inne transakcje. W przypadku operacji typu INSERT na obiekt wstawiany nakładana jest blokada zanim jeszcze element znajdzie się w bazie. Dzięki temu nie możliwe jest czytanie fantomów. W ten sposób uzyskujemy poziom izolacji READ COMMITTED – transakcja widzi tylko swoje modyfikacje oraz te, które zostały uczynione przez transakcje zatwierdzone, nawet jeśli zatwierdzenie nastąpiło po rozpoczęciu się transakcji.

Do obsługi całego mechanizmu blokowania służą następujące metody:

- *matrix* – jest to macierz kompatybilności dla mechanizmu blokowania, jako argumenty podaje się dwa typy blokad,a wynikiem jest wartość *prawda*, jeśli blokady są ze sobą kompatybilne i *fałsz* w przeciwny przypadku,
- *lock* – jest to metody nakładająca blokadę na element. Funkcja sprawdza najpierw, czy element jest już blokowany: jeśli nie, to pojawia się nowy wpis w tablicy blokad, jeśli tak, sprawdzane jest, czy transakcja żądająca blokady nie zablokowała już tego elementu wcześniej i nie jest jedyną transakcją blokującą element. W takim przypadku menedżer zezwala na dostęp do danych ustawiając tryb blokady na silniejszy z wszystkich zgłaszanych przez transakcję. Jeśli natomiast element jest już blokowany przez inne transakcje, to najpierw sprawdzany jest bit oczekiwania. Gdy jest on równy 0, za pomocą metody *matrix* sprawdzana jest zgodność blokad i albo transakcja jest wstrzymywana albo działa dalej. Natomiast gdy bit oczekiwania jest równy 1, transakcja bez sprawdzania kompatybilności trafia na listę oczekujących. Dzięki temu mechanizm nie faworyzuje żadnego trybu blokowania, a kolejka działa jak FIFO. Zanim jakakolwiek blokada zostanie nałożona na element, następuje blokowanie ostrzegawcze „ojca”,
- *readLock* – funkcja ta sprawdza, czy element mógłby zostać zablokowany do czytania, jednak żadne zapisy w tablicy blokad nie następują,
- *unlock* – metoda służąca do zdejmowania blokady. Przy zdejmowaniu sprawdzane jest, czy jest to jedyna nałożona blokada. Wówczas usuwany jest cały wpis z tablicy blokad.

Jeśli transakcji blokujących jest więcej, wówczas usuwana jest tylko informacja o transakcji z listy. Następnie badane jest czy transakcja będąca następna w kolejce była wstrzymywana. Jeśli tak, to przeszukiwana jest cała lista transakcji w celu wznowienia niektórych z nich i ustalenia nowego trybu grupowego blokowania,

- *activateTrans* – funkcja wznawiająca działanie transakcji oczekującej na przyznanie blokady.

W wyniku testów okazało się, że blokowanie do czytania jest blokowaniem bardzo zachłannym i mocno ogranicza współbieżność. Wystarczy proste zapytanie np. *prac where nazwisko=”Kowalski”* i mamy wszystkie elementy prac zablokowane wspólnie. Przez to nie możliwa jest modyfikacja jakiegokolwiek innego obiektu *prac* bądź ich podobiektów. Zastanawiałam się nad usunięciem blokad wspólnych. Jednak takie rozwiązanie obniża mocno poziom izolacji: możliwe wówczas byłoby czytanie brudnopisów, czy fantomów. W aktualnej wersji menedżera zastosowane jest następujące rozwiązanie: w przypadku chęci czytania danych transakcja pyta się, czy mogłaby uzyskać blokadę wspólną na elemencie. Jeśli odpowiedź jest twierdząca, transakcja uzyskuje dostęp do elementu, ale blokada wspólna nie jest nakładana. Dzięki temu unika się blokowania całych grup obiektów oraz uniemożliwia czytanie elementów modyfikowanych przez inne transakcje. W przypadku operacji typu INSERT na obiekt wstawiany nakładana jest blokada zanim jeszcze element znajdzie się w bazie. Dzięki temu nie możliwe jest czytanie fantomów. W ten sposób uzyskujemy poziom izolacji READ COMMITTED – transakcja widzi tylko swoje modyfikacje oraz te, które zostały uczynione przez transakcje zatwierdzone, nawet jeśli zatwierdzenie nastąpiło po rozpoczęciu się transakcji.

Moja praca nad projektem wymagała ode mnie napisania w głównej klasie projektu *Eng* kilku metod pracujących bezpośrednio na danych i komunikujących się z menedżerem transakcji oraz oprogramowania kilku operatorów języka wewnątrz głównej funkcji przetwarzania drzewa zapytania *eval*. Są to niezbędne instrukcje potrzebne do pracy z transakcjami (*commit*, *rollback*, *savepoint*, *exit*), do pracy z obiektami (*delete*, *create_persistent*,*as*, *add_role*, ,-przecinek). Następnie musiałam zaimplementować metody, które faktycznie realizują wyżej wymienione operacje, takie jak:

- *beginTransaction* – jest funkcja żądająca od menedżera utworzenia transakcji,
- *commit* – funkcja ta wysyła żądanie do menedżera, by zatwierdził działanie podanej transakcji (dokonał wpisów w logu i zdjął blokady) oraz by utworzył nową transakcję,
- *rollback* – jeśli przy wywołaniu tej funkcji podano nazwę savepoint'a, to następuje

wycofanie części operacji przez menedżera, jeśli ten argument jest pominięty, unieważniona zostaje cała transakcja, kończona, a następnie tworzona nowa,

- *savepoint* – jest to funkcja obsługująca instrukcję języka: SAVEPOINT *nazwa_savepoint'a*, powoduje umieszczenie punktu zatwierdzenia na liście operacji *operationList* danej transakcji,
- *close* – funkcja ta obsługuje poprawne wyjście z systemu za pomocą komendy EXIT, uruchamiane są tu procedury menedżera tworzące *back-up* bazy oraz zamykające plik logu,
- *read* – metoda ta jest wywoływana przy czytaniu elementów, czyli ich ewaluacji na stosie. Zależnie od tego jak będzie wartość drugiego argumentu (0 lub 1) zastosowane zostanie blokowanie wspólne do czytanie bądź tylko sprawdzenie dostępu do elementu. Domyślnie ustawione jest ten drugi tryb, czyli przy czytaniu nie są nakładane blokady,
- *delete* – służy do usuwania obiektów bądź listy obiektów. Odbywa się w ten sposób, że najpierw usuwane są wszystkie obiekty, które po nim dziedziczą, następnie wszystkie jego podobiekty, a na końcu sam obiekt. W przypadku pointerów, zanim obiekt zostanie usunięty, modyfikowana jest lista pointerów zwrotnych obiektu, na który pointer wskazuje. Jeśli usuwany obiekt jest podobiektem innego, to z listy *subObjects* ojca, wyrzucana jest odpowiedni element. Przed zniszczeniem jakiegokolwiek obiektu zakładana jest na niego blokada wyłączna,
- *update* - funkcja wywoływana w przypadku zmiany wartości obiektu atomowego poprzez operator przypisania „:=” bądź też w przypadku zmiany roli obiektu kompleksowego (wywołanie z *add_role*). Funkcja najpierw tworzy kopię obiektu oryginalnego, na niej dokonuje zmian, a następnie podmienia obiekty w bazie,
- *add_role, del_role* – funkcja obsługującą zmianę roli obiektu, czyli określanie związków dziedziczenia za pomocą instrukcji ADD ROLE. Operacja ta jest jawną modyfikacją wartości obiektu, dlatego przy tej okazji menedżer dokonuje odpowiednich wpisów do logu,
- *createAtomic*, *createPointer*, *createComplex*, *insert* – są to funkcje obsługujące operatory *create_persistent* i *as*, wywoływane w przypadku chęci utworzenia nowego obiektu w bazie. Trzy pierwsze metody tworzą obiekt, następnie menedżer transakcji blokuje go i dokonuje wpisów w logu, a dopiero potem, za pomocą funkcji *insert,* obiekt jest wstawiany do bazy, czyli trafia na listę wszystkich obiektów i tworzone są odpowiednie bindery. Można więc powiedzieć, że zanim element znajdzie się w bazie,

już jest blokowany. Dzięki temu żadna inna transakcja nie zobaczy tego obiektu, aż transakcja wstawiająca nie zostanie zatwierdzona.

Jak wspomniałam, operacje wykonywane są od razu na oryginalnych obiektach bazy. W wielu bazach danych stosuje się mechanizm tworzenia kopii obiektów i na nich dokonywania operacji, a następnie przy zatwierdzeniu transakcji dokonuje się nadpisania obiektów bazy ich kopiami. Za tym rozwiązaniem przemawia fakt, iż w przypadku wycofania transakcji nie trzeba uruchamiać procesu wycofywania zmian, a wystarczy tylko zwolnić pamięć, w której trzymane są kopie. Drugim argumentem dla baz danych trzymających dane na dysku jest oszczędność przesyłania danych między dyskiem a pamięcią. Stosując kopie komunikacja odbywa się tylko przy odczycie i ostatecznym zapisie obiektu, a nie przy każdej modyfikacji. Ponieważ, jak na razie, nasza baza danych jest trzymana w pamięci operacyjnej, a zapis elementów na dysk dokonuje się dopiero w momencie zamknięcia systemu, drugi powód nas nie dotyczy. Poza tym kłopotliwe wydaje się być ciągłe uaktualnianie binderów na stosie środowisk tak by wiązanie nazw zwracało referencje do kopii obiektów. Dlatego zdecydowałam się na bezpośrednią pracę z obiektami. Poza tym w rzeczywistych bazach danych częściej zdarza się zatwierdzenie transakcji niż wycofanie.

Mechanizm transakcyjny zaimplementowany w prototypowej wersji projektu wydaje się spełniać wszystkie podstawowe wymagania. Jedynym jego poważnym mankamentem jest brak mechanizmów wykrywania i usuwania zakleszczeń, które niestety mogą wystąpić. Jednak dopóki nie zostanie zorganizowany wielodostęp, problem ten nie występuje. Można by również doprogramować funkcje dające możliwość wyboru poziomu izolacji, czy ręcznego nałożenia blokady np. na klasę obiektów. Myślę, że te zadania zostaną zrealizowane w miarę rozwoju projektu.

Bibliografia

1. H. Garcia-Molina, J. D. Ullman, J. Widom. „Implementacja systemów baz danych". Wydawnictwo Naukowo-Techniczne, Warszawa 2003
2. T. Connolly, C.Begg. „Systemy baz danych". Wydawnictwo RM, Warszawa 2005
3. R. Stones, N.Matthew. „Bazy danych i MySQL". Helion, Gliwice 2003
4. Software 2.0. 12/2001 str, 18-26 „Transakcyjność w bazach danych"
5. J.S. Couchman. „Oracle Certification Professional, DBO Certification Exam Guide". Osborne/McGeaw-Hill, U.S.A. California 2000
6. C.J.Date, H. Darwen. „SQL. „Omówienie standardu języka", Wydawnictwo Naukowo-Techniczne, Warszawa 2000
7. R.Elmasri, S. B. Navathe. „Wprowadzenie do systemów baz danych". Helion, Gliwice 2005
8. K. Subieta. „Teoria i Konstrukcja obiektowych języków zapytań". Wydawnictwo PJWSTK, Warszawa 2004

Strony internetowe:

1. www.postgersql.org
2. www.mysql.com

Zawartość płyty CD

1. Wersja elektroniczna pracy
   - w formacie .sxw: /tresc/praca_magisterska.sxw
   - w formacie .pdf: /tresc/praca_magisterska.pdf
2. Źródła projektu „SBQLLite”: /projekt