Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Maciej Olesiński
Nr albumu: 201077

Implementacja podstaw obiektowego systemu zarządzania bazą danych

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Sierpień 2007

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy zaprezentowano opis realizacji implementacji podstaw obiektowego systemu zarządzania bazą danych. Dokładnie przedstawiono wykorzystane struktury danych oraz algorytmy związane z operowaniem na nich, a także zastosowane podejście do klasycznych problemów bazodanowych dotyczących zapewnienia poprawnego przetwarzania transakcji. Nakreślono także możliwości związane z rozwojem powstałej warstwy danych, w celu uzyskania kompletnego SZBD.

Słowa kluczowe

obiekt, baza danych, system zarządzania bazą danych, transakcja, menedżer transakcji, dziennik transakcyjny

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2. Database Management
H.2.4. Systems

Tytuł pracy w języku angielskim

An implementation of fundamental aspects of object database management system

Spis treści

Rozdział 1

Wprowadzenie

1.1. Obiektowość w bazach danych

Z definicji obiektowy system zarządzania bazą danych jest systemem zarządzania danych wspierającym modelowanie i tworzenie danych w postaci obiektów. Powinien on także obsługiwać klasy obiektów oraz dziedziczenie własności i metod przez podklasy i ich obiekty. Nie istnieje jednak żaden szeroko uznany standard związany z obiektowymi bazami danych analogiczny do podstaw zaproponowanych w ramach artykułu Codda oferującego jasny i przejrzysty opis modelu dla relacyjnych baz danych [Codd70]. Dodatkowo nie istnieje wspólny standard języka zapytań dla obiektowych baz danych, podczas gdy relacyjne bazy danych dysponują szeroko stosowanym językiem SQL. Powstało wiele inicjatyw opracowania wspólnych rozwiązań na potrzeby modelu obiektowego, jednak nigdy nie udało się osiągnąć porozumienia w ich sprawie. Dotychczas opracowane standardy obiektowych baz danych nie uzyskały akceptacji i nie doczekały się pełnej implementacji (ODMG OQL [Odmg00] i SQL-99 [Sql99]). Spowodowane jest to faktem, iż nawet w przypadku języków programowania o podejściu obiektowym nie istnieje wspólny standard dotyczący tych języków. Każdy obiektowy język programowania bazuje na szeregu cech opisujących obiektowość, ale wykorzystane cechy nie występują we wszystkich językach.

Wraz z ogromnym rozwojem obiektowych języków programowania spodziewany był rozwój obiektowych systemów zarządzania bazami danych, który spowodowałby wyparcie systemów relacyjnych. Jednakże wysokie koszty związane z przenoszeniem kodu na całkowicie odmienne środowisko bazodanowe, zaoferowanie rozszerzeń obiektowych w ramach systemów relacyjnych tworzących obiektowo-relacyjne systemy oraz powstanie warstw pośrednich pozwalających na odwzorowanie obiektów na dane w relacyjnych bazach danych nie spowodowało rozwoju systemów w pełni obiektowych. Obecnie są one wykorzystywane głównie jako dopełnienie systemów relacyjnych lub obiektowo-relacyjnych w ramach niszowych dedykowanych rozwiązań lub w przypadku konieczności operowania na wysoce złożonych strukturach danych nie wymagających całego aparatu analitycznego dostarczanego przez model relacyjny.

Podstawowe przeszkody uniemożliwiające szerokie stosowanie obiektowych systemów zarządzania bazami danych nie wiążą się z żadnymi specyficznymi technologicznymi problemami związanymi z obiektowymi systemami, a bardziej polegają na tym iż użytkownicy baz danych przyzwyczajeni są do baz o modelu relacyjnym oraz posiadają o wiele szerszą wiedzę na ich temat. O wiele łatwiej jest wykorzystywać technologię szeroko wykorzystywaną i znaną, niż nieznaną. Dodatkowo decyzja zastosowania obiektowego systemu obarczona jest większym ryzykiem biznesowym, gdyż firmy dostarczające dedykowane obiektowe rozwiązania nie są światowymi gigantami na rynku bazodanowym.

Systemy zarządzania bazami danych o podejściu w pełni obiektowym pozwalają na operowanie na danych w postaci obiektów bezpośrednio z obiektowych języków programowania lub też wprowadzając własne języki zapytań oferujące możliwości obiektowe. Aby było to możliwe konieczne jest zdefiniowanie czym jest obiekt w ramach obiektowej bazy danych. Pojęcie obiektu najlepiej oddaje Kazimierz Subieta w swojej książce o obiektowych językach zapytań. „Obiekt jest abstrakcyjnym bytem reprezentującym lub opisującym pewną rzecz lub pojęcie reprezentowane w świecie rzeczywistym. Obiekt jest odróżnialny od innych obiektów, ma nazwę i dobrze określone granice" [Sub04].

„W odróżnieniu od modelu relacyjnego obiektowość nie zakłada konieczności określenia takiego atrybutu obiektu (lub kombinacji atrybutów), który identyfikuje go w sposób unikalny, czyli tzw. „klucza głównego" (primary key). Mówi się, że obiekt posiada swoją tożsamość (identity), tj. istnieje niezależnie od innych obiektów i od swojego aktualnego stanu. Jakkolwiek tożsamości obiektów przypisuje się niekiedy głębszy sens filozoficzny, w praktyce implementacyjnej oznacza ona, że obiekt posiada unikalny wewnętrzny identyfikator (object identifier, OID), który odróżnia go od innych obiektów. Nie mogą istnieć dwa obiekty posiadające ten sam identyfikator. Taki identyfikator jest nadawany przez system automatycznie, niezależnie od woli projektanta lub programisty. Wewnętrzny identyfikator umożliwia budowanie referencji do obiektu, w szczególności tworzenie powiązań pointerowych" [Sub04].

Nazwa obiektu w obiektowej bazie danych pozwala jasno określić jakie pojęcie ze świata rzeczywistego reprezentowane jest przez obiekt bez konieczności analizowania jego danych składowych lub odwoływania się do metamodelu, jeżeli taki istnieje. Obiekty występujące w bazie danych pod tą samą nazwą powinny stanowić reprezentacje tych samych pojęć. Nazwy w żaden sposób nie są unikalne, możliwe jest występowanie wielu składowych w ramach obiektu o tej samej nazwie. Wtedy wielokrotne wystąpienie obiektów o identycznej nazwie ma formę zbioru. Dla przykładu obiekt reprezentujący samochód może mieć cztery podobiekty reprezentujące jego koła, każde koło powinno mieć nazwę koło.

W ramach niniejszej pracy magisterskiej struktury danych zawierające niezmienny w czasie unikalne identyfikator, nazwę oraz wartość będą określane mianem obiektów. Nie zawierają one tak dobrze znanych z programowania obiektowego metod, klas oraz dziedziczenia, jednakże stanowią one podstawę do konstrukcji kompletnego rozwiązania.

1.2. Struktura systemów zarządzania bazami danych

Niezależnie od przyjętego modelu danych (obiektowy lub relacyjny), każdy system zarządzania bazami danych musi zapewniać szereg usług, aby możliwe było uznanie go za kompletny. Poza szczególnymi wyjątkami bazy danych umieszczane są w wielodostępnych środowiskach sieciowych. Oznacza to, iż uruchamiane są jako serwery, umożliwiając komunikację i dostęp do danych za pośrednictwem ustalonych protokołów. Otrzymane zapytania następnie poddawane są przetwarzaniu, optymalizacji i są kierowane do modułu wykonawczego. Nad zachowaniem bezpieczeństwa współbieżności czuwa menedżer transakcji, zezwalając na równoczesne wykonywanie wielu zleceń. Zachowanie spójności danych podlegających zmianom w obrębie transakcji także zapewniane jest poprzez mechanizm dzienników transakcyjnych. W trakcie wykonywania zapytań wykorzystywane są bufory podręczne, indeksy oraz inne struktury pomocnicze, lub też konieczne jest bezpośrednie sięganie do danych w ramach struktur dyskowych. Dodatkowo istnieje wiele innych modułów, które są w mniejszym bądź większym stopniu potrzebne do zapewnienia szeregu podstawowych usług oferowanych przez obecne systemy.

Przedstawiona implementacja podstaw obiektowego SZBD koncentruje się na usługach

niskopoziomowych dostępu do danych oraz problemach związanych z przetwarzaniem transakcji. Moduł wykonawczy zapytań oraz wszystko „ponad" nim znajduje się poza zakresem niniejszej pracy magisterskiej.

1.3. O poprzednim rozwiązaniu

W ciągu minionych dwóch lat na wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w ramach przedmiotu Systemy Zarządzania Bazami Danych oraz seminarium magisterskiego Bazy Danych 1 powstał projekt w ramach którego została stworzona kompletna implementacja bazy danych o podejściu obiektowym. Celem projektu było osiągnięcie wyników w jak najkrótszym możliwym czasie, co spowodowało iż pewne kroki projektowe zostały pominięte. Ta decyzja oraz to, że baza danych została napisana w języku C++, spowodowała wiele nawarstwiających się w czasie problemów. Uzyskane rozwiązanie, mimo bardzo bogatej funkcjonalności oraz waloru badawczego, było niestabilne oraz zawierało niezmierną ilość niekontrolowanych wycieków pamięci. Z czasem niedoprecyzowane interfejsy pomiędzy modułami składowymi systemu stały się niejasne i nieczytelne. Ten stan rzeczy skłonił autora do zaimplementowania podstaw systemu zarządzania bazą danych ponownie w ramach pracy magisterskiej z wykorzystaniem platformy z automatycznym zarządzaniem pamięcią.

1.4. .NET Framework oraz C#

Począwszy od czerwca 2000 roku, kiedy firma Microsoft zapowiedziała platformę .NET oraz nowy związany z tą platformą język programowania C#, następuje ciągły wzrost popularności tego rozwiązania. C# jest mocno otypowanym obiektowo zorientowanym językiem zaprojektowanym w celu zaoferowania twórcom optimum połączenia prostoty, wyrazistości i wydajności. Podstawowym elementem platformy .NET jest wspólne środowisko uruchomieniowe (Common Language Runtime — CLR), będący maszyną wirtualną analogiczną do maszyny wirtualnej Java (Java Virtual Machine — JVM) oraz bogaty zestaw bibliotek mogący być wykorzystany przez szereg języków programowania zaoferowanych w ramach platformy .NET, dzięki temu iż kod kompilowany jest do wspólnego języka pośredniego (Common Intermediate Language — CIL). Platforma .NET oraz język C# są ze sobą blisko związane — część funkcji C# zaprojektowana jest aby efektywnie współpracować z .NET oraz .NET zawiera funkcje efektywnie współpracujące z C# — mimo iż platforma .NET obsługuje więcej języków niż tylko C#. W swojej konstrukcji C# został opracowany czerpiąc z wielu języków programowania, ale głównie z Javy i C++.

CLR wraz z C# ma wiele podobieństw do JVM i języka programowania Java firmy Sun. Oba rozwiązania opierają się na modelu maszyny wirtualnej ukrywającej przed programistą szczegóły sprzętu komputerowego, na którym uruchamiane są programy. Oba także stosują własny kod pośredni, w przypadku rozwiązania firmy Microsoft jest to wspólny język pośredni CIL, a w przypadku Sunowskiej Javy jest to po prostu kod pośredni. Na platformie .NET kod pośredni zawsze kompilowany jest do kodu maszynowego w trybie na żądanie (Just In Time — JIT), podczas gdy Java oferuje możliwość interpretowania kodu pośredniego, kompilacji z wyprzedzeniem lub także kompilacji na żądanie. Obie platformy oferują szeroki zestaw bibliotek klas zaspokajających większość potrzeby programistycznych oraz gwarantują bezpieczeństwo działania aplikacji w ramach zarządzalnego środowiska.

Główną zaletą przemawiającą na korzyść języka C# jest możliwość wykorzystania bezpiecznych wskaźników. Mimo iż .NET Framework zawiera mechanizmy automatycznego za-

rządzania pamięcią, możliwe jest bezpośrednie odwoływanie się do określonych adresów pamięci za pomocą wskaźników dobrze znanych z C czy C++. Jedynym wymaganiem, które należy spełnić zanim możliwe jest uzyskanie i operowanie na wskaźniku jest wykorzystanie konstrukcji gwarantującej to, że wskazywany obiekt w pamięci nie zostanie w niej przemieszczony w ramach rutynowych czynności mechanizmu odśmiecania pamięci (Garbage Collection — GC). Konstrukcja ta to słowo kluczowe języka C# `fixed`.

Zyskiem z wykorzystania wskaźników jest możliwość operowania na strukturach danych zawartych w ramach zarządzalnych buforów pamięci bez potrzeby wykonywania dodatkowych kopii. Okazuje się to dość istotne gdy obiekty składowane są w ramach stron dyskowych a następnie strony te buforowane są w pamięci. Wskaźniki zezwalają na bezpośrednią manipulację wartościami struktur obiektów, lub też przemieszczaniem danych w ramach buforów.

1.5. Cel pracy

Celem niniejszej pracy magisterskiej jest stworzenie podstaw systemu zarządzania bazą danych o podejściu obiektowym, składającej się z modułów zarządzania transakcjami, dziennikami, buforami, plikami danych, a także logiki składowania obiektów. Aby możliwe było osiągnięcie w pełni funkcjonalnego SZBD, konieczne jest rozszerzenie przedstawionego rozwiązania o moduł wykonawczy dowolnego wybranego języka zapytań o podejściu obiektowym. W zależności od środowiska w jakim miałaby pracować baza danych dodatkowo mogłaby zaistnieć potrzeba implementacji aplikacji serwera, klienta oraz protokołu komunikacyjnego.

Rozdział 2

Usługi warstwy danych

Zgodnie z klasycznym modelem budowy systemu zarządzania bazą danych, dane przechowywane są trwale na dysku w ramach plików danych. Pliki te następnie podzielone są na strony dyskowe, czyli jednostki alokacji ustalonego rozmiaru, w ramach których składowane są poszczególne obiekty wraz z informacjami opisującymi daną stronę. Spójność, bezpieczeństwo oraz wielodostęp do danych zgromadzonych w bazie danych zapewniana jest poprzez moduły menedżera dzienników oraz menedżera transakcji. Czas dostępu do danych minimalizowany jest za pomocą wprowadzenia pamięci podręcznej bazy danych w ramach modułu menedżera buforów. Za rozkład danych na stronach dyskowych, wprowadzanie zmian i uaktualnianie struktury odpowiedzialny jest moduł logiki składowania.

Mimo że menedżer dzienników, transakcji oraz buforów wydają się być w pełni niezależnymi bytami, to okazuje się iż są one bardzo mocno ze sobą związane. Specyficzna implementacja jednego z nich, wymusza odpowiednie zachowanie pozostałych, a także pozwala na obsłużenie dodatkowych scenariuszy, lub wręcz ogranicza możliwą do uzyskania funkcjonalność. To powiązanie stanowi główny powód dlaczego powinny one być planowane oraz implementowane wspólne w ramach warstwy danych systemy zarządzania bazami danych.

2.1. Menedżer buforów

Wszelkie operacje na bazie danych, czyli odczytywanie danych, modyfikowanie, dodawanie oraz usuwanie, wiążą się z operacjami w pamięci na stronach pliku danych. Przeprowadzenie odczytu lub zapisu na urządzeniu dyskowym jest najdroższą operacją przy szacowaniu czasu wykonania zapytania i często czas związany z operacjami wykonywanymi na pamięci jest w pełni pomijany przy szacowaniu złożoności algorytmów bazodanowych. Baza danych minimalizuje liczbę potrzebnych operacji wejścia/wyjścia za pomocą mechanizmu zwanego menedżerem buforów, czyli blokiem pamięci wykorzystywanym do przetrzymywania kopii ostatnio używanych stron dyskowych.

Prezentowane rozwiązanie implementuje moduł buforów jako tablicę mieszającą o kluczach będących adresami stron dyskowych i wartościach będących strukturami buforów stron. Dzięki temu możliwe jest szybkie stwierdzenie czy strona znajduje się w buforze oraz uzyskania dostępu do niej.

Struktura buforująca stronę dyskową przechowuje następujące informacje:

- Czas ostatniego wykorzystania strony.
- Bufor zawierający zatwierdzoną wersję strony.

- Informacja o tym czy strona jest brudna, innymi słowy czy zatwierdzona wersja strony jest zapisana na dysku.
- Bufor mogący przechować wersję strony modyfikowaną w ramach aktywnej transakcji.
- Identyfikator aktywnej transakcji modyfikującej stronę.

Każde odwołanie do buforowanych stron odbywa się także przy użyciu mechanizmu zamków chwilowych. Oznacza to, że przy każdym pobraniu bufora reprezentującego stronę dyskową, zakładany jest na nią zamek uniemożliwiający innym wątkom operowanie na buforze reprezentującym stronę. Wprowadzenie tych zamków chwilowych jest konieczne, gdyż inaczej mogłaby powstać sytuacja gdy jedna transakcja operuje na stronie w trybie do odczytu, czyli na buforze z zatwierdzoną wersją, podczas gdy druga transakcja zatwierdza swoje zmiany na buforze z modyfikowaną wersją, podmieniając bufor odczytywany, co prowadziłoby do potrzeby przechowywania więcej niż jednej kopii.

Strony z menedżera buforów muszą być pobierane w obrębie ustalonej transakcji i w ustalonym trybie dostępu, do odczytu lub zapisu. Niezależnie od trybu, gdy w trakcie pobierania strony nie znajduje się ona w buforze, jest ona wczytywana z dysku i umieszczana w tablicy mieszającej w ramach struktury buforującej. W przypadku pobierania strony do zapisu w ramach transakcji wytwarzana jest jej kopia, która jest później zwracana w kolejnych wywołaniach pochodzących od tej transakcji. Jeżeli inna transakcja zażąda tej samej strony w trybie do zapisu zanim poprzednia się zakończyła to generowany jest wyjątek. W trakcie zatwierdzania transakcji wszystkie struktury buforujące, które zostały zmodyfikowane w jej obrębie, są modyfikowane tak, że zmodyfikowana wersja strony staje się nową zatwierdzoną wersją oraz są oznaczane jako brudne, co jest sygnałem dla wątku zapisującego dane, iż daną stronę należy zapisać na dysk. W przypadku gdy następuje próba pobrania brudnej strony do zapisu w ramach nowej transakcji, dana transakcja jest wstrzymywana i budzony jest wątek pisarza.

Zapis brudnych stron dyskowych z menedżera buforów odbywa się jedynie w obrębie dedykowanego wątku pisarza uruchamianego co konfigurowalny interwał czasowy. Jego główną rolą jest zapisanie wszystkich brudnych stron na dysk i odnotowanie w dzienniku transakcyjnym informacji o tym, iż zapis nastąpił i jaka była najstarsza zapisana strona. Dodatkowo usuwa on z bufora strony, które nie były ostatnio wykorzystywane.

2.2. Transakcje

W systemach bazodanowych wszelkie operacje wykonywane są w ramach transakcji, czyli zbioru operacji, które stanowią w istocie pewną całość i jako takie powinny być wykonane wszystkie lub żadna z nich. Warunki jakie powinny spełniać transakcje szczegółowiej opisują zasady ACID (Atomicity, Consistency, Isolation, Durability — niepodzielność, spójność, izolacja, trwałość).

2.2.1. Menedżer dzienników

Jedną z cech mechanizmu przetwarzania transakcji w bazie danych jest trwałość, czyli gwarancja że po zatwierdzeniu transakcji będzie ona na stałe zapamiętana w bazie danych i nawet awaria systemu nie spowoduje jej wycofania. Dzienniki transakcji zapewniają mechanizm pozwalający uzyskać trwałość, poprzez zapisywanie zmian wprowadzonych na stronach zanim strony te zostaną oznaczone jako brudne w trakcie zatwierdzania transakcji i przeznaczone do zapisu. Oznacza to, iż transakcja nie może zostać uznana za zatwierdzoną, póki nie

nastąpi zapis dzienników zawierających wszelkie informacje dotyczące modyfikacji przez nią wykonanych.

Dzienniki zapisywane są w postaci pliku strumieniowego zawierającego ułożone kolejno rekordy różnego typu oraz mogące charakteryzować się zmienną długością. Rekordy te odnotowują czy baza danych została bezpiecznie zamknięta, początek i koniec transakcji, zmiany wprowadzone na stronach dyskowych w ramach transakcji oraz informacje o zapisie brudnych stron przez wątek pisarza menedżera buforów. Przewijanie dziennika wstecz jest możliwe, gdyż na końcu każdego rekordu zapisywana jest jego długość, dzięki czemu przy starcie bazy danych nie istnieje konieczność przewijania całego dziennika od początku, w celu sprawdzenia czy baza została zamknięta bezpiecznie.

Jedną z integralnych części dzienników jest mechanizm różnicowania stron dyskowych pozwalający ograniczyć rozmiar danych potrzebnych do zapamiętania różnic pomiędzy stronami zmodyfikowanymi w ramach transakcji. Przedstawiona implementacja realizuje ten mechanizm za pomocą interfejsów oraz refleksji, pozwalając na pełną wymienność mechanizmu. Dodatkowo konfigurowalny parametr pozwala określić, czy rekordy powstałe w wyniku różnicowania powinny zostać poddane kompresji.

Pełna trwałość, tak jak to wcześniej zostało wspomniane, wiąże się z wymogiem synchronicznego zapisu dzienników przy zapisywaniu rekordów oznaczających zatwierdzenie transakcji oraz w momencie zapisu brudnych stron na dysk. W przypadku rekordów informujących o zapisie brudnych stron na dysk nie jest koniecznie wymuszenie synchronicznego zapisu dzienników, jednakże wprowadzenie tego wymagania znacznie upraszcza proces odtwarzania bazy danych po awarii lub wymuszonym zamknięciu, bez oczekiwania na zakończenie wszystkich aktywnych transakcji w systemie.

Charakterystyka zastosowanego rozwiązania zakłada wykorzystanie dziennika jedynie jako dziennik powtórzeń, aby było to możliwe nigdy nie następuje zapis brudnych stron na dysk zanim transakcja zostanie zatwierdzona. Oznacza to, iż zapis do dziennika następuje jedynie w trakcie zatwierdzania transakcji. Informacje o rozpoczęciu transakcji oraz wycofaniu transakcji nie są zapisywane, a zatem w ramach dziennika transakcyjnego nie pozostaje żaden ślad po transakcjach wycofanych lub transakcjach wykonywanych w momencie nastąpienia awarii. Wiąże się to ściśle z konstrukcją menedżera buforów nie posiadającego żadnych programowych ograniczeń na liczbę buforowanych stron. Jedynym ograniczeniem jest ilość dostępnej pamięci w ramach systemu operacyjnego. A zatem wszelkie transakcje wykonywane w ramach prezentowanego systemu zarządzania bazą danych muszą mieścić się w pamięci komputera, w przypadku wyczerpania pamięci następuje awaria bazy danych. Jest to rezygnacja z popularnie stosowanego mechanizmu, pozwalającego na zapis stron brudnych uczestniczących w ramach transakcji, w celu zwolnienia miejsca w menedżerze buforów na dalsze przetwarzanie tejże transakcji lub transakcji równoległych.

2.2.2. Menedżer transakcji

Niepodzielność, spójność oraz izolacja transakcji gwarantowane są w systemie zarządzania bazą danych przez moduł menedżera transakcji. Cele te realizowane są przy pomocy mechanizmu tablicy zamków, a także w powiązaniu z modułem dzienników oraz modułem menedżera buforów. Tak jak zostało wcześniej wspomniane operacje w ramach transakcji dokonywane są na kopiach stron dyskowych wewnątrz menedżera buforów, jednakże zanim zostanie przyznany dostęp do strony dla transakcji, wymagane jest otrzymanie dostępu na zamku stowarzyszonym z żądaną stroną. Każda strona dyskowa chroniona jest zamkiem zezwalającym transakcjom na dzielony dostęp do odczytu, lub wyłączny dostęp do zapisu, zgodnie z klasycznym modelem czytelników i pisarzy z dziedziny programowania współbieżnego. W przypadku

niemożności uzyskania blokady na zamku, transakcja jest wstrzymywana do czasu gdy dostęp jest możliwy.

Blokady na zamkach transakcje tworzą stosując metodę Strong Strict Two-Phase Locking (SS2PL) występującej również pod nazwą Rigorous Two-Phase Locking (R2PL). Wykorzystanie metody SS2PL oznacza, iż transakcje mogą być szeregowalne według kolejności ich zatwierdzenia [Weik01]. W ramach tej metody obowiązują dwie zasady:

1. Jeżeli transakcja T potrzebuje czytać/pisać obiekt, musi ona zażądać odpowiednio zamka do odczytu/zapisu dla tego obiektu.

2. Wszystkie zamki (do odczytu i do zapisu) uzyskane przez transakcję T w trakcie jej działania zwalniane są jedynie w momencie zatwierdzania lub wycofywania transakcji T i nigdy wcześniej.

Taki tryb działania w dużym stopniu upraszcza problemy związane ze współbieżnością, ale nie eliminuje ich w zupełności. Pozostającym problemem jest problem cyklicznego czekania, prowadzący do zakleszczenia transakcji. W najprostszym przypadku sprowadza się on do tego, iż transakcja T1 po uzyskaniu zamka na stronie X próbuje uzyskać dostęp na zamku do stronie Y, podczas gdy transakcja T2 wcześniej uzyskała dostęp na stronie Y i teraz próbuje uzyskać dostęp na zamku na stronie X. W efekcie obie transakcje, T1 i T2, oczekują wzajemnie na strony zablokowane krzyżowo i dochodzi do zakleszczenia. W celu zapobiegnięcia temu przypadkowi, prezentowana realizacja implementuje strategię czekaj-giń (wait-die). Strategia polega na tym, iż każdej transakcji przyporządkowywany jest stempel czasowy z momentu jej rozpoczęcia i następnie w przypadku próby uzyskania blokady na zamku, który jest obecnie używany przez inną transakcję, transakcja może czekać jedynie w przypadku, gdy jest starsza niż transakcja aktualnie przetrzymująca zamek. Transakcje młodsze są uśmiercane, czyli wycofywane. Wycofane transakcje mogą być uruchomione ponownie i jeżeli tak się stanie, uruchamiane są z oryginalnym stemplem czasowym, aby zapobiec zagłodzeniu, a dokładniej sytuacji kiedy ta transakcja będzie wielokrotnie uśmiercana.

Bardzo istotnym momentem dla zapewnienia zasad ACID jest moment zatwierdzania transakcji. Algorytm zatwierdzania wykonuje następujące kroki, w pierwszej kolejności zapisuje do dziennika transakcyjnego informację o tym, iż dana transakcja się rozpoczęła (zapis tej informacji w tym momencie jest możliwy dzięki temu, iż zmodyfikowane strony z menedżera buforów nigdy nie są zapisywane na dysk zanim transakcja się nie zakończy). Następnie dla każdej strony zmodyfikowanej w ramach transakcji zapisywane są do dziennika informacje o zmianach dokonanych w jej ramach, po czym zapisywany jest rekord informujący o poprawnym zakończeniu transakcji. W tym momencie wszelkie informacje są bezpiecznie zapisane do dziennika i możliwe jest trwałe naniesienie zmian na strony danych, poprzez zatwierdzenie zmian w module buforów oraz oznaczenie zmodyfikowanych stron jako brudne. Na zakończenie procesu zatwierdzania transakcji zdejmowane są blokady z zamków przez nią założonych, co zezwala innym transakcjom na dostęp do nich. W przypadku wycofania transakcji anulowane są zmiany na buforach a następnie zdejmowane są blokady z zamków założonych przez transakcję, co oznacza iż żadna informacja o wycofanej transakcji nie jest zapisywana do dzienników.

Zrealizowany menedżer transakcji zezwala na pracę transakcji na jednym z dwóch poziomów izolacji — read committed oraz serializable. Paradoksalnie zastosowanie zamków do odczytu i zapisu w przypadku hierarchicznej struktury danych pozwala uzyskać szeregowalne transakcje bez potrzeby analizy planu wykonania. Pominięcie zamków do odczytu, a zatem nie zakładanie ich oraz nie oczekiwanie na zamkach innych transakcji w trakcie odczytywania danych, oferuje poziom read committed, który nie gwarantuje powtarzalnego odczytu, ale

jest domyślnym trybem pracy transakcji na przykład w bazie danych Oracle. Jest to możliwe dzięki zastosowaniu dwuwersyjnego menedżera buforów. Dwuwersyjność polega na tym, iż w momencie otrzymania pierwszego żądania zapisu na stronie znajdującej się w menedżerze buforów tworzona jest jej kopia. Wszystkie operacje przeprowadzane w ramach transakcji piszącej na stronie nanoszone są na kopię, pozwalając pozostałym transakcjom działającym na poziomie read committed na dostęp do oryginalnej wersji strony w celu odczytu. W ramach zatwierdzenia transakcji piszącej następuje podmiana wersji oryginalnej na kopię, która staje się nową zatwierdzoną wersją strony i wszystkie kolejne odczyty przeprowadzone na stronie odbywają się na nowej wersji. Możliwość wystąpienia podmiany wersji strony powoduje brak powtarzalności odczytów, który jest charakterystyczny dla poziomu read committed.

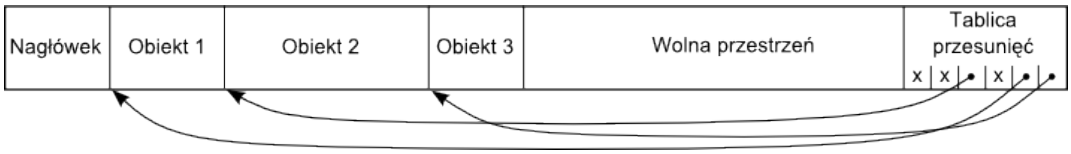
Rozdział 3

Logika składowania obiektów

Podstawa systemu zarządzania bazą danych prezentowana w niniejszej pracy magisterskiej nie posiada przedstawionych we wstępie cech obiektowej bazy danych. Dane składowane w jej obrębie nie mają nawet określonej struktury, co sprawia że ma ona charakter semistrukturalny. Jest to spowodowane tym, iż własności obiektowej bazy danych są ściśle związane z zaimplementowanym w niej językiem zapytań w ramach modułu odpowiedzialnego za wykonywanie zapytań. Zadaniem warstwy danych jest stworzenie warunków umożliwiających rozszerzenie jej o dowolny moduł wykonawczy obiektowego języka zapytań, dostarczający wsparcie dla modelowania, tworzenia i operowania na danych w postaci obiektów.

Semistrukturalna organizacja danych oznacza, iż nie istnieje wydzielony schemat opisujący format danych, zamiast tego szczegółowe informacje dotyczące danych zapisanych w ramach bazy danych związane są z tymi danymi. Podstawowe informacje związane z każdym rekordem obiektu znajdującym się w obrębie bazy danych obejmują: unikalny identyfikator obiektu, pozwalający odróżniać go od pozostałych obiektów, typ wartości zapisanej w ramach obiektu oraz nazwę obiektu. Dopuszczane wartości atomowe obejmują typy liczbowe całkowite, stałopozycyjne i zmiennopozycyjne, typ napisowy, typ reprezentujący datę i czas, typ o wartości binarnej oraz typ pozwalający przechować wskaźnik na inny rekord obiektu w ramach bazy danych. Typ obiektu jest niezmienny w czasie. Obiekt raz utworzony o konkretnym typie zachowuje ten typ do momentu usunięcia. W porównaniu do relacyjnego modelu baz danych nie istnieje możliwość przypisywania wartości pustych obiektom, czyli wartości null. Brak wartości dla obiektu o określonej nazwie reprezentowany jest poprzez brak tego obiektu w bazie danych. Poprzez nazwy obiektów oddawane jest pojęcie z rzeczywistego świata reprezentowane poprzez konkretny obiekt. Obiektom reprezentującym te same pojęcia zawsze powinny być nadawane identyczne nazwy. Powyższe założenie ogranicza ilość nazw występujących w ramach bazy danych do ilości równej ilości różnych bytów reprezentowanych w jej obrębie.

Jako model danych został wykorzystany model hierarchiczny, który jest naturalnym modelem danych dla semistrukturalnych bazy danych. W ramach tego modelu dane zorganizowane są w postaci struktury drzewiastej. Powiązanie danych pomiędzy sobą reprezentowane jest za pomocą relacji rodzic-dziecko, a mianowicie każdy rodzic może mieć wiele dzieci, ale każde dziecko ma jednego rodzica, chyba że obiekt jest korzeniem drzewa. Informacje konieczne do realizacji relacji rodzicielstwa także zapisywane są w ramach rekordu dla każdego obiektu. Koncepcyjnie, aby obiekt mógł posiadać obiekty potomne, musi on występować jako obiekt złożony. Wartością obiektu złożonego jest zbiór obiektów będących jego dziećmi. Oznacza to, iż nie może istnieć w bazie danych rekord obiektu posiadający jednocześnie wartość atomową oraz obiekty potomne. Dodatkowym wnioskiem z niniejszej organizacji danych jest to, że



Rysunek 3.1: Rozkład danych na stronie dyskowej

w przypadku usunięcia obiektu złożonego następuje jednoczesne usunięcie wszystkich obiektów w nim zagnieżdżonych. Dopuszczenie obiektów o typie wartości wskaźnikowym rozszerza model hierarchiczny do modelu o budowie grafu skierowanego, jednakże powiązanie obiektów poprzez wskaźniki nie wymusza żadnych dodatkowych własności w przypadku usuwania obiektu wskazującego lub obiektu będącego wskazywanym jak to ma miejsce w przypadku obiektów zagnieżdżonych.

Obiekty nie posiadające rodziców wspólnie określane są jako korzenie, niezależnie od tego czy są to obiekty złożone czy atomowe. Pełnią one szczególną rolę dla bazy semistrukturalnej, gdyż jedynie one dostępne są w momencie rozpoczęcia wykonania zapytania. Ponieważ każdy obiekt w bazie danych ma nazwę, możliwe jest określenie, na którym korzeniu lub korzeniach konieczne jest wykonanie zapytania. Dostęp do obiektów ukrytych w ramach struktury hierarchicznej możliwy jest jedynie w momencie gdy znany jest identyfikator żądanego obiektu. Pobieranie obiektów po nazwie dotyczy jedynie obiektów stanowiących korzenie.

3.1. Fizyczna organizacja danych

Rolę przestrzeni składowania danych w większości systemów zarządzania danych pełnią pliki dyskowe lub całe urządzenia dysków twardych bez systemu plików, ale również wtedy baza danych wytwarza własne struktury pozwalające operować na dysku analogicznie do operacji na plikach. Operacje przeprowadzane na pliku danych mają charakter operacji blokowych. Zapis i odczyt danych z pliku odbywa się w ramach spójnych obszarów danych ustalonego rozmiaru, zwanych stronami dyskowymi. W obrębie pliku danych system zarządzania bazą rezerwuje przestrzeń na dane z wyprzedzeniem, automatycznie lub na zlecenie administratora bazy danych.

Wszystkie strony dyskowe w ramach prezentowanej podstawy systemu zarządzania bazą danych mają ten sam format i służą jedynie do przetrzymywania danych, przy małym udziale metadanych, stanowiących ich skład. Podstawową informacją zliczaną do metadanych związanych ze stroną jest nagłówek strony, w ramach którego zapisane są informacje o położeniu strony w ramach pliku dyskowego, o kategorii danych i flagach opisujących stan strony, o ilości wolnego miejsca na stronie oraz o czasie ostatniej modyfikacji przeprowadzonej w ramach danych zgromadzonych na stronie.

Rekordy obiektów składowanych w ramach strony umieszczane są zawsze od początku strony, tuż za jej nagłówkiem, w spójnym bloku. Następnie znajduje się również spójny blok wolnej pamięci, a na końcu strony umieszczona jest tablica przesunięć, która zostanie dokładniej opisana w rozdziale 3.2. Blok danych oraz tablica przesunięć mogą być rozszerzane niezależnie gdyż znajdują się w odrębnych końcach strony dyskowej. Bajty zawarte w ramach wolnej przestrzeni zawsze mają wartość równą zero, co ułatwia przeprowadzanie porównanie stron dyskowych i znalezienie różnic na potrzeby dziennika transakcyjnego.

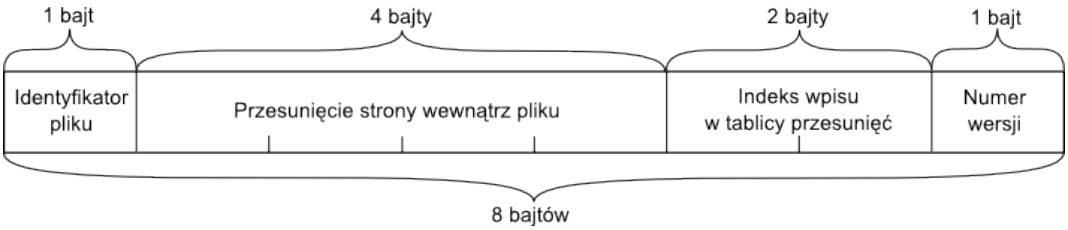
Niezależnie od typu obiektu każdy obiekt składa się z podstawowej struktury zawierającej informacje pozwalające operować na nim. Jest to długość rekordu, binder rekordu, binder re-

kordu rodzica, pole określające typ oraz pole zawierające flagi specyficzne dla rekordu. Binder to para: unikalny identyfikatora obiektu oraz identyfikator nazwy (więcej na temat kodowania nazw w postaci identyfikatorów nazw w rozdziale 3.4). Jest to związane z założeniem polegającym na tym, że zawsze z identyfikatorem obiektu występuje również nazwa obiektu, umożliwiając określenie nazwy obiektu wskazywanego przez identyfikator bez potrzeby sięgania do całego obiektu, co mogłoby być związane z pobieraniem dodatkowych stron dyskowych, podczas gdy znajomość nazwy obiektu mogłaby wykluczyć konieczność pobierania go. W zależności typu obiektu i od tego czy wartość tego typu może być zmiennego rozmiaru, następnie zapisywana jest wartość obiektu, lub struktura opisująca bufor zmiennej długości w ramach, którego przechowywana jest wartość. Do typów wymagających pamięci stałego rozmiaru zaliczane są typy liczbowe, data i czas oraz wskaźniki na inne obiekty (czyli bindery). Typy zmiennego rozmiaru to napisy, wartości binarne oraz także obiekty złożone. Mimo iż logicznie wartością obiektu złożonego powinien być zbiór obiektów w nim zawartych, fizycznie realizowane jest to jako zbiór binderów obiektów zagnieżdżonych. Zgodność realizacji fizycznej z reprezentacją logiczną zapewniana jest w ramach implementacji operacji na obiektach.

3.2. Adresacja

W każdej bazie danych koniczna jest forma adresacji rekordów, aby umożliwić ich sprawne odnajdowanie wewnątrz struktur danych na żądanie. Jednym ze schematów rozwiązywania problemu adresacji jest wykorzystanie adresacji w pełni fizycznej. Przy takim rozwiązaniu adresem rekordu jest jego adres fizyczny w ramach struktur danych, czyli dokładne położenie opisywane poprzez adres strony oraz przesunięcie w ramach tejże strony lub bezpośrednio przesunięcie w obrębie przestrzeni danych. Przemieszczanie tak adresowanych obiektów jest niemożliwe, gdyż spowodowałoby ono unieważnienie wszystkich odwołań do przemieszczonego rekordu. Sposobem na użycie adresacji fizycznej przy jednoczesnym zezwoleniu na przemieszczanie rekordów jest mechanizm odsyłaczy. W najprostszych słowach polega on na umieszczaniu informacji o nowym położeniu obiektu pod jego poprzednim adresem, zezwalając na zachowanie zewnętrznych odwołań. Odsyłacze umieszczane na poprzednich pozycjach rekordu obiektu pozostawałyby w bazie do momentu usunięcia obiektu i po pewnym czasie w przypadku gdy dane byłyby przemieszczane wielokrotnie, odszukiwanie przemieszczonych obiektów mogłoby się okazać procesem długotrwałym, gdyż istniałaby konieczność przejrzenia wszystkich jego odsyłaczy. Innym schematem rozwiązania problemu adresacji jest użycie adresacji pośredniej, poprzez wprowadzenie logicznych adresów oraz struktury przechowującej ich odwzorowanie na adresy fizyczne. To podejście zezwala na dowolne przemieszczanie obiektów w obrębie struktur danych wymagając jedynie, aby przy każdym przesunięciu rekordu obiektu, nastąpiło uaktualnienie odwzorowania. Rozdzielenie adresacji na część fizyczną i logiczną ma swoją cenę. Mianowicie przy każdym odwołaniu do rekordu obiektu konieczne jest skorzystanie ze struktury odwzorowań w celu uzyskania adresu fizycznego. Oczywiście możliwe jest wprowadzenie mechanizmów przechowujących potrzebne fragmenty struktury w pamięci podręcznej, jednakże wiąże się to z dodatkowymi mechanizmami wymagającymi oprogramowania i utrzymywania. Dodatkowo struktura odwzorowań adresów logicznych na fizyczne musiałaby w pełni obsługiwać mechanizm transakcji.

W prezentowanym rozwiązaniu wykorzystane jest połączenie adresacji fizycznej z logiczną, dodatkowo rozszerzając pojęcie adresu, w celu umożliwienia użycia go jednocześnie jako unikalnego identyfikatora obiektu. Fizyczna część adresu zawiera pola identyfikujące plik danych oraz przesunięcie strony w obrębie tego pliku. Logiczna część adresu składa się z in-



Rysunek 3.2: Pola składowe identyfikatora obiektu

deksu w tablicy przesunięć oraz numeru wersji danego adresu. Tablica przesunięć znajduje się na każdej stronie pliku danych oraz umieszczana jest na jej końcu. Pola w ramach tablicy zawierają informację o położeniu obiektów w ramach stron. Takie podejście zezwala na przemieszczanie rekordów w obrębie stron dyskowych, jednakże nie jest możliwe przeniesienie ich na inne strony dyskowe. Dodatkowo w tablicy przesunięć oprócz przesunięcia rekordu zawarte jest dodatkowe pole liczbowe, informujące który raz dany indeks jest w użyciu. Usuwając obiekt z bazy danych, w tablicy przesunięć na indeksie adresu danego obiektu, zwiększany jest numer wersji po to, aby przy ponownym wykorzystaniu tego indeksu możliwe było rozpoznanie czy też wskazuje on na ten sam obiekt co poprzednio. Gwarancja rozróżnialności jest konieczna, gdyż adresacja fizyczno-logiczna stosowana jest do unikalnej identyfikacji obiektów w bazie danych. Toteż w przypadku stosowania wskazań na obiekty potrzebny jest mechanizm pozwalający na bezbłędną identyfikację obiektów. Dodatkowo zastosowanie adresów jako identyfikatory, gwarantuje ich niezmienność w czasie. Szczegółowy wygląd struktury identyfikatora adresu fizyczno-logicznego przedstawiony jest poniżej.

Potrzeba wprowadzenia numerów wersji adresów obiektów wynika z realizacji obiektów o wartościach wskaźnikowych, czyli wskazujących na wybrany obiekt w ramach bazy danych. Wskazywany obiekt zapamiętywany poprzez zapamiętanie jego bindera jako wartość obiektu wskaźnikowego. W obrębie bindera przechowywany jest unikalny identyfikator obiektu, będący za razem adresem obiektu. Przejście po wskaźniku wiąże się z odszukaniem wskazywanego obiektu w składzie i zwróceniem referencji do niego. Dzięki numerom wersji wpisów, w trakcie pobierania wskazywanego obiektu i określania jego położenia na stronie przy pomocy tablicy przesunięć, możliwe jest zweryfikowanie czy obiekt zapisany na danym indeksie w tablicy jest wciąż wskazywanym obiektem, poprzez porównanie numerów wersji i określenie czy wskaźnik jest wciąż poprawny, czy też stał się wskaźnikiem wiszącym. Brak numerów wersji mógłby prowadzić do sytuacji, kiedy wskazywany obiekt zostałby usunięty z bazy danych, a następnie w wyniku wstawienia na jego miejscu zostałby umieszczony inny obiekt. Ponieważ nowy obiekt miałby ten sam identyfikator pliku, identyfikator strony oraz indeks przesunięcia nie możliwe byłoby wykrycie, iż nie jest to ten obiekt, który oryginalnie był wskazywany.

3.3. Operacje na obiektach

Podstawowe operacje konieczne do uzyskania pełnej funkcjonalności warstwy składowania danych w ramach obiektowej bazy danych to wstawianie lub usuwanie obiektów oraz operacje związane z modyfikacją wartości obiektów prostych i złożonych. Zostały one zaimplementowane jako szereg metod zawierających logikę związaną z powyższymi operacjami, wykorzystując menedżera buforów oraz menedżera transakcji. Dodatkowo zadaniem logiki składowania obiektów jest dbanie o zachowanie spójności bazy danych w sytuacjach tj. usuwanie obiektów złożonych, gdzie konieczne jest usunięcie całego drzewa obiektów poczynając od korzenia,

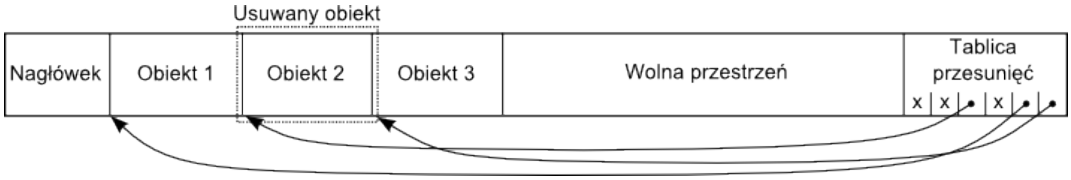
lub zmiana rodzica obiektu, kiedy trzeba sprawdzić czy też ustanowienie nowego rodzica dla obiektu nie spowoduje utworzenie cyklu rodzicielstwa.

Najprostszą operacją bazodanową jest wstawienie nowego obiektu. W momencie tworzenia nowego obiektu w bazie danych, pobierana jest strona zawierająca wystarczającą ilość wolnego miejsca, przy użyciu algorytmu opisanego w dalszej części pracy. Następnie rekord reprezentujący nowo wstawiany obiekt umieszczany jest na początku sekcji zawierającej wolną przestrzeń na stronie oraz tworzone jest odwołanie do jego pozycji wewnątrz tablicy przesunięć. Numer strony wraz z indeksem wewnątrz tablicy przesunięć oraz numerem wersji wpisu na danej pozycji w tablicy przesunięć staje się nowym unikalnym identyfikatorem utworzonego obiektu. Uzyskawszy identyfikator obiektu możliwe jest zagnieżdżenie go w ramach obiektu będącego rodzicem nowego obiektu, poprzez dodanie go do listy jego dzieci.

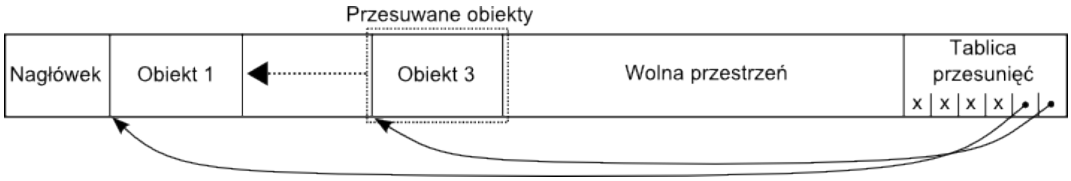
Usuwanie obiektów wymaga bardziej złożonych operacji, gdyż po pierwsze usuwany obiekt może znajdować się pomiędzy innymi obiektami na stronie danych, a wymagane jest, iż wolna przestrzeń na stronie stanowi spójny fragment umiejscowiony za danymi a przed tablicą przesunięć, która to znajduje się na końcu każdej strony. Usuwanie obiektu ze strony dyskowej realizowane jest jako przesunięcie wszystkich rekordów obiektów znajdujących się za usuwanym obiektem w taki sposób, że nadpisują one usuwany obiekt. Przestrzeń powstała w wyniku tegoż dosunięcia obiektów jest następnie zerowana. Dodatkowo w ramach tablicy przesunięć aktualizowane są przesunięcia dla wszystkich dosuniętych obiektów, przez co mimo iż zmieniło się ich położenie w obrębie strony, to ich adresy wciąż pozostają poprawne oraz w tablicy przesunięć na pozycji odpowiadającej usuwanemu obiektowi zwiększany jest numer wersji, aby w przypadku próby odwołania do usuniętego obiektu, na przykład poprzez typ wskaźnikowy, można było jednoznacznie określić, iż został on usunięty. Gdy usuwany obiekt jest obiektem złożonym, założenia spójności wymagają aby wszystkie obiekty, będące zagnieżdżone wewnątrz niego, zostały również skasowane. Proces kasowania wykonywany jest rekurencyjnie w głąb zanim nastąpi usunięcie danego obiektu złożonego. Na zakończenie operacji skasowania obiektu następuje usunięcie identyfikatora danego obiektu z listy dzieci obiektu rodzica.

Operacje na wartościach obiektów zostały zrealizowane jako trzy zestawy operacji w zależności od typu wartości, ponieważ istnieje rozróżnienie typów na typy proste o wartościach stałego rozmiaru, typy proste o wartościach zmiennej długości oraz na typy złożone, czyli po prostu obiekty złożone, których wartość stanowi lista dzieci będąca także zmiennej długości, ale charakteryzująca się specyficznymi operacjami związanymi z wyszukiwaniem, dodawaniem i kasowaniem pojedynczych wpisów. Najprostszy zestaw operacji wiąże się z typami prostymi stałego rozmiaru. W przypadku tego typu wartości, wartość zapisana jest jako część składowa rekordu obiektu. Wszelkie zmiany wartości nie wydłużają ani skracają długości rekordu obiektu, pozwalając na stosowanie operowanie nimi jedynie za pośrednictwem programistycznych wskaźników.

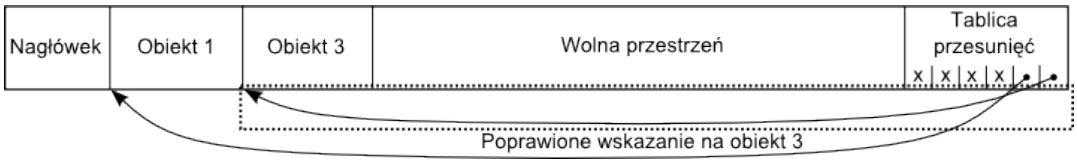
Obsługa wartości zmiennej długości stawia o wiele bardziej złożone wyzwania. Po pierwsze w przypadku wydłużania lub skracania długości wartości konieczna jest reorganizacja obiektów na stronie dyskowej. Dodatkowo możliwa jest sytuacja, kiedy długość wartości obiektu jest na tyle duża, iż nie mieści się ona na jednej stronie dyskowej i konieczne jest zastosowanie techniki zwanej łańcuchowaniem, co oznacza iż wartość jest dzielona na części i umieszczana na wielu stronach dyskowych. Ponieważ rekordy obiektów na stronach umieszczanie są kolejno po sobie, zostawiając spójny obszar z wolną przestrzenią na końcu strony, modyfikacja wartości obiektu powodująca wydłużenie bądź skrócenie jej długości jest problematyczna. Aby rozwiązać ten problem funkcje logiki składowania obiektów stosują następujący zabieg — modyfikując wartość obiektu o typie zmiennej długości, rekord obiektu jest przesuwany, tak aby stał się ostatnim rekordem na stronie i za nim znajdowała się jedynie wolna przestrzeń.



(a) Stan początkowy



(b) Przesuwanie obiektów znajdujących się za usuwanym obiektem



(c) Uaktualnianie wpisów w tablicy przesunięć dla przemieszczonych obiektów

Rysunek 3.3: Usuwanie obiektu ze strony dyskowej

Oczywiście tak samo jak w przypadku operacji usuwania obiektów, dokonywane są odpowiednie poprawki w ramach tablicy przesunięć, aby odzwierciedlała ona położenie obiektów na stronie po przesunięciu.

Tak jak zostało wcześniej wspomniane, każda strona pliku danych ma z góry ustalony i ograniczony rozmiar. W przypadku obiektów zmiennego rozmiaru, tj. obiekty złożone lub też napisy, czasami jest możliwe iż ich rozmiar jest większy powoduje, iż nie można ich pomieścić na jednej stronie. Gdy taka sytuacja zaistnieje, konieczne jest łańcuchowanie obiektów, czyli rozmieszczenie ich na więcej niż jednej stronie. Do tworzenie łańcucha wykorzystywane są strony w pełni puste i w momencie doczepienia ich do łańcucha oznaczane są jako strony łańcuchowane, co uniemożliwia zapisywanie innych danych poza łańcuchem na danej stronie. Łańcuchy mogą jedynie rosnąc i nigdy nie są skracane, gdyż jeżeli obiekt raz rozrósł się, bardziej prawdopodobnie jest iż rozrost nastąpi ponownie, niż nie.

3.4. Odwzorowywanie nazw

Wszystkie obiekty składowane w obrębie bazy danych mają w pełni określoną nazwę. Nazwa obiektu odzwierciedla pojęcie ze świata rzeczywistego reprezentowane przez obiekt. Przykładami nazw obiektów mogą być Imię dla obiektu o wartości typu napisowego "Maciej", Wzrost dla obiekty o wartości liczbowej 180, lub Osoba dla obiektu złożonego { Imię = "Maciej", Nazwisko = "Olesiński", Wzrost = 180 }. Analogią dla nazw obiektów w modelu relacyjnym są nazwy tabel. Przykładowo każdy wiersz w tabeli Osoby reprezentuje konkretną osobę, a każdy wiersz w tabeli Produkty reprezentuje konkretny produkt.

Nazwa obiektu w modelu obiektowych baz danych nie jest niczym więcej tylko prostym napisem, będącym częścią rekordu zapisanego w ramach strony dyskowej. Przechowywanie napisów w ramach struktur danych z wielu względów jest niewygodne. Przede wszystkim

napisy mają zmienną długość oraz sprawdzenie czy też dwa napisy są równe wymaga czasu proporcjonalnego do ich długości. Dodatkowo pula nazw występujących w bazie danych jest ograniczona do ilości różnych bytów przez nią reprezentowanych. Wszystkie te argumenty przemawiają za tym, aby w ramach rekordów obiektów nie przechowywać nazw jako napisów, tylko zamiast tego wprowadzić mechanizm odwzorowywania nazw na liczbowe identyfikatory nazw, których zapis wymaga stałego rozmiaru. Wprowadzenie liczbowych identyfikatorów nazw nie tylko upraszcza przechowywanie nazw, ale także zezwala na dołączenie i przechowywanie ich zawsze razem z identyfikatorami obiektów. To podejście stanowi ogromną optymalizacje, gdyż pobierając identyfikator obiektu rodzica, przeglądając listę identyfikatorów dzieci obiektu złożonego oraz przechodząc po wartości obiektu wskaźnikowego w ramach modułu wykonawczego zapytań możliwy jest dostęp do nazwy obiektu bez potrzeby pobierania całego obiektu z bazy danych.

Odwzorowanie nazw zapisywane w plikach danych ma formę meta danych niewidocznych dla użytkownika. Mimo to, wykorzystuje ono ten sam mechanizm składowania danych co normalne dane, tzn. model obiektowy. Każde pojedyncze odwzorowanie zapisywane jest jako obiekt prosty typu napisowego. W tymże obiekcie wartością jest właściwa nazwa, natomiast identyfikator nazwy umieszczony w rekordzie obiektu jest właśnie odwzorowaniem. Obiekty nazw są niewidoczne dla użytkownika dzięki temu, iż przechowywane są jako podobiekty ukrytego obiektu złożonego, odgrywającego rolę korzenia nazw. Dostęp do tego korzenia możliwy jest wyłącznie z poziomu warstwy danych, poprzez jego dobrze ustalony identyfikator. Dodatkowo strony, na których umieszczane są obiekty nazw, oznaczane są jako przynależące do kategorii nazw a nie normalnych danych, co zapobiega umieszczaniu normalnych danych na nich. Zapis odwzorowywania nazw przy użyciu tych samych mechanizmów co normalne obiekty oznacza, iż podlega on w pełni przetwarzaniu transakcji tak samo jak normalne dane. Takie podejście gwarantuje bezpieczeństwo zapisów w przypadku awarii oraz niweluje potrzebę utrzymywania dodatkowej struktury metadanych potrzebnej do przechowywania odwzorowania nazw.

W trakcie pracy systemu, odwzorowywanie nazw jest pobierane z dysku i przechowywane w pełni w pamięci operacyjnej. Struktura przechowująca je ma charakter globalny i jest niezależna od poszczególnych transakcji, gwarantując to iż odwzorowania będą identyczne w obrębie każdej pojedynczej transakcji. Konwersji podlegają nie tylko nazwy obiektów zapamiętanych w plikach danych, a także nazwy pomocnicze obiektów tymczasowych oraz zmiennych nazwanych wykorzystywanych w trakcie wykonywania zapytań w module wykonawczym. Używając tablic mieszających, pozwalających na błyskawiczną konwersję pomiędzy identyfikatorami nazw a nazwami oraz odwrotnie, możliwe jest uzyskanie nieznacznej poprawy wydajności mechanizmu wykonawczego, pozwalając mu operować na liczbach zamiast na napisach.

Identyfikatory nazw stosowane w ramach wykonywania zapytań nie są zapisywane do struktury meta danych nazw podczas pierwszego użycia, tylko przetrzymywane są jedynie w pamięci do czasu, gdy istnieje potrzeba utworzenia trwałego obiektu na dysku o konkretnej nazwie. Algorytm przeprowadzania zapisu odwzorowania zaprojektowany jest tak, aby mógł działać niezależnie od mechanizmu transakcyjnego, wykorzystując jego właściwości, w szczególności mechanizm zamków. Mianowicie w trakcie dodawania nowej nazwy do struktury meta danych nazw, zapamiętywana jest strona, której to została ona umieszczona. Próba wykorzystania nowo zapisanego odwzorowania nazwy przez kolejną transakcję wymaga uzyskania zamka na tejże stronie w celu sprawdzenia, czy transakcja pierwotnie dokonująca zapis została zatwierdzona pomyślnie i czy obiekt nazwy faktycznie został zapisany na stronie metadanych nazw. Odnalezienie obiektu na stronie oznacza pełny sukces transakcji umieszczającej go i odwzorowanie zostaje uznane za w pełni zatwierdzone. W przeciwnym

przypadku jest obiekt odwzorowania zostaje umieszczony ponownie na stronie.

Mimo, iż wykorzystanie takiego algorytmu stanowi duże ograniczenie współbieżności transakcji, to nie powoduje ono praktycznie żadnego spowolnienia pracy systemu, gdyż nowe nazwy pojawiają się niezmiernie rzadko, zazwyczaj w trakcie pierwszego wypełniania bazy danych. Odwzorowania nazw raz zatwierdzone w bazie danych nigdy nie są usuwane, czyli zapis odwzorowania następuje tylko jednokrotnie w jednej bazie danych.

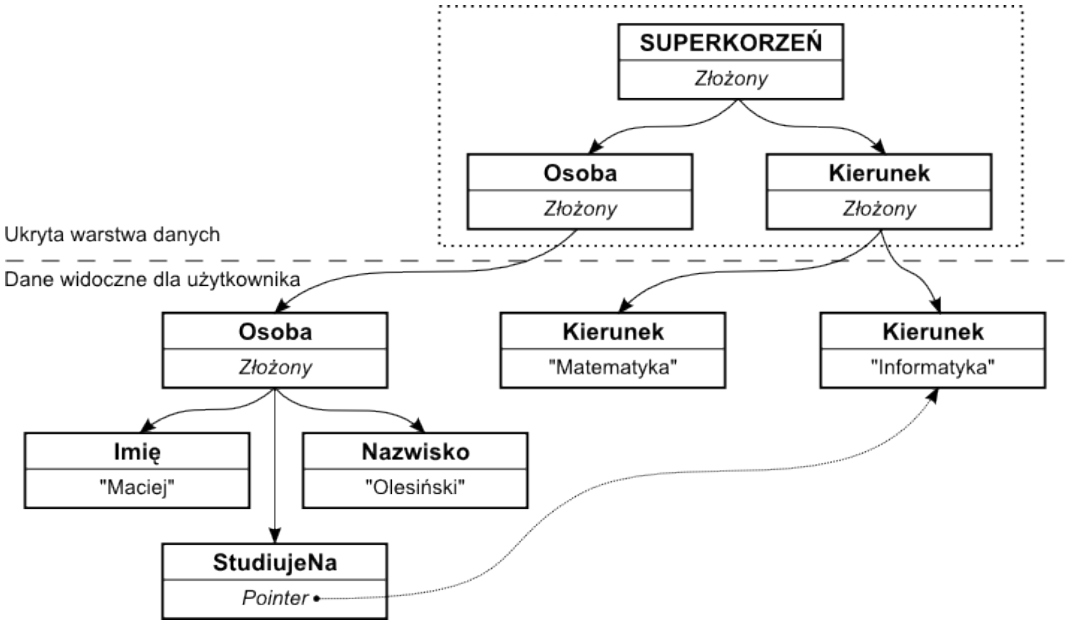
3.5. Korzenie

Podejście obiektowe w bazach danych wiąże się z hierarchicznym modelem powiązania obiektów. Dane zgromadzone mają charakter lasu, zawierającego wiele drzew, z których to każde rozpoczyna się od korzenia. Korzenie w obiektowych bazach danych pełnią podobną rolę co tabele w przypadku modelu relacyjnego. Umiejętność błyskawicznego odnajdowania wszystkich korzeni o ustalonej nazwie wewnątrz bazy danych jest niezmiernie istotna dla efektywnego wykonywania zapytań, gdyż jeżeli nie sięgamy do konkretnego obiektu wskazanego poprzez jego unikalny identyfikator obiektu, to zawsze proces pobieranie i przetwarzania danych rozpoczyna się od korzeni. Ten fakt oznacza, iż system zarządzania obiektową bazą danych powinien implementować strukturę gromadząca informacje o położeniu obiektów będących korzeniami.

Podobnie jak w przypadku mechanizmu odwzorowywania nazw nie istnieje osobna dedykowana struktura przechowująca informację o obiektach korzeniowych. Zamiast tego wykorzystywany jest ten sam mechanizm składowania danych co w przypadku normalnych obiektów. W ramach tegoż mechanizmu tworzone są dwa dodatkowe ukryte poziomy obiektów „ponad" korzeniami. Pierwszy poziom składa się z obiektu złożonego, o dobrze ustalonym identyfikatorze, nazywanego superkorzeniem. Poniżej, jako podobiekty superkorzenia umieszczane jest po jednym obiekcie złożonym dla każdej nazwy korzeni. Dopiero w ramach tych obiektów umieszczane są właściwe obiekty stanowiące korzenie dla użytkownika. Dodatkowo stosowana jest zasada, iż obiekty superkorzenia oraz korzeni nazw umieszczane są po jednym na stronie dyskowej tak jak to zostało przedstawione na poniższym rysunku. Rozwiązanie to nastawione jest na rozrost struktury korzeni tak, aby minimalizować potrzebę łańcuchowania obieków korzeniowych oraz optymalizować blokady transakcyjne zakładane w ramach tejże struktury.

Większość zapytań wysyłanych do obiektowej bazy danych, niezależnie od zastosowanego języka zapytań, w trakcie wykonywania opiera się o obiekty korzeniowe. Jednakże zazwyczaj nie istnieje potrzeba pobierania informacji o wszystkich korzeniach, a jedynie o korzeniach o kilku nazwach. Fizyczne zgrupowanie obiektów korzennych w ramach nadobiektów warstwy pośredniej umożliwia szybkie odnalezienie szukanych korzeni bez potrzeby operowania na całej strukturze korzeni, znacząco ograniczając ilość stron danych do których należy się odwołać, a tym samym ilość odczytów dyskowych.

Dodatkowym zyskiem z zastosowania opisywanej organizacji danych, wraz z ograniczeniem powodującym umieszczanie jednego obiektu struktury korzeni na jednej stronie, jest jej wspomniany wpływ na transakcyjność. Oczywiste jest, iż pełna obsługa transakcyjności wymaga stosowania wszelkich mechanizmów transakcyjnych, czyli również mechanizmu zamków. Dodanie nowego korzeniowego obiektu przez użytkownika bazy danych oznacza umieszczenie go w ramach obiektu na drugim poziomie struktury korzeni, czyli w obrębie odpowiedniego obiektu korzenia nazwy, tym samym modyfikując go. Modyfikacja wymaga założenia zamka do zapisu na stronie, na której on się znajduje.



Rysunek 3.4: Diagram struktury obiektów korzeni

3.6. Alokacja nowych stron

Dane w bazie danych zgromadzone są w ramach pliku dyskowego. Plik ten zawiera strony z danymi będące w pełni zapełnione, puste, bądź też częściowo zapełnione. W przypadku gdy wszystkie strony dyskowe w pliku zostaną zapełnione jest on automatycznie rozszerzany o dodatkowe strony dyskowe. Stopień rozszerzenia jest jednym z parametrów konfiguracyjnych warstwy danych. W trakcie działania bazy danych rozmieszczenie stron o różnym zapełnieniu w ramach pliku dyskowego nie podlega żadnym regułom, co oznacza iż w momencie wstawiania nowych obiektów do bazy danych lub wydłużania wartości obiektów łańcuchowych konieczne jest efektywne odnajdowanie stron dyskowych zawierających wystarczającą ilość wolnego miejsca do przeprowadzenia operacji. W celu rozwiązania tego problemu, często wprowadza się struktury pomocnicze przechowujące listę wolnych stron, indeks „śmieci", czy też tablicę zajętości stron. Zastosowanie dodatkowych struktur oznaczałoby wprowadzenie stron zawierających inną organizację wewnętrzną niż strony z danymi oraz wymagałoby zapewnienia im spójności za pomocą mechanizmów transakcyjnych. Tak jak można było zaobserwować w przypadku odwzorowywania nazw oraz korzeni celem autora było ograniczenie struktur danych jedynie do struktur przechowujących rekordy reprezentujące obiekty, co oznacza rezygnację z dodatkowych struktur zarządzania przestrzenią dyskową i zastosowanie alternatywnego rozwiązania.

Wykorzystany algorytm pobierania stron dyskowych dysponujących wystarczającą ilością miejsca potrzebną do zrealizowania konkretnej operacji na bazie danych nie opiera się na żadnej strukturze pomocniczej. Dodatkowo żadne informacje dotyczące zajętości stron dyskowych nie są zapamiętywane pomiędzy kolejnymi uruchomieniami systemu zarządzania bazą danych. Schemat działania opiera się na sekwencyjnym przeszukiwaniu dyskowego pliku z danymi oraz zapamiętywaniu miejsca wewnątrz pliku, od którego rozpoczyna się grupa stron nie w pełni zajętych. Oznacza to iż mechanizm odnajdowania pustych stron gromadzi potrzebne mu dane podczas pracy systemu.

Dokładny algorytm stosuje rozróżnienie kategorii stron dyskowych na strony zwykłe, strony zawierające struktury superkorzeni oraz strony zawierające obiekty stanowiące odwzorowanie nazw na identyfikatory liczbowe. Dla każdej z tych kategorii zapamiętywany jest indeks pierwszej strony nie będącej w pełni zapełnionej. Poszukiwanie strony dysponującej odpowiednią ilością wolnego miejsca rozpoczyna się zawsze od zapamiętanego indeksu dla danej kategorii. Począwszy od tego indeksu następuje kolejno sprawdzanie zajętości stron. Sprawdzanie odbywa się dwuetapowo. Pierwszy etap sprawdzenia dostępności wolnego miejsca ogranicza się jedynie do pobrania strony z menedżera buforów bez uprzedniego zażądania zamka na niej. Dopiero w przypadku stwierdzenia spełnienia kryteriów wyszukiwania przez daną stronę następuje próba uzyskania zamka do zapisu na wytypowanej stronie i ponowne sprawdzenie kryteriów. W przypadku niemożności natychmiastowego uzyskania zamka do zapisu, gdyż wytypowana strona jest zablokowana w ramach innej toczącej się transakcji, lub uzyskaniu innych wyników niż w trakcie pierwszego sprawdzenia, strona ta jest pomijana i następuje przejście do kolejnej.

Głównym atutem prezentowanej strategii jest brak potrzeby utrzymywania dodatkowych struktur danych, których celem byłby opis zajętości poszczególnych stron w ramach pliku danych. Dodatkowo niniejsza strategia została zaprojektowana z myślą o minimalizacji konfliktów transakcyjnych, poprzez zastosowanie systemu podwójnej weryfikacji przydatności strony, gdzie pierwsze sprawdzenie odbywa się z pominięciem blokad transakcyjnych. Wykorzystanie niniejszego algorytmu oznacza konieczność odnalezienia wolnych stron po każdorazowym uruchomieniu serwera bazodanowego, jednakże gdy już to nastąpi to pobieranie kolejnych wolnych stron powinno odbywać się średnio w czasie stałym.

Innym aspektem związanym z mechanizmem alokacji i pobierania nowych stron jest jego wpływ na rozmieszczenie danych w pliku bazy danych. Prezentowane rozwiązanie charakteryzuje się rozmieszczaniem danych wstawianych razem na stronach dyskowych umieszczonych blisko siebie. W trakcie rozmieszczania nowych obiektów nie jest uwzględniane położenie obiektu będącego rodzicem nowo tworzonego. Określenie zysków bądź strat wynikających z wybrania niniejszej strategii jest niemalże niemożliwe, gdyż wszystko zależy od rodzaju i sposobu działania modułu wykonawczego. Jeden moduł wykonawczy może pobierać obiekty ze struktury drzewiastej zgodnie z porządkiem odwiedzania węzłów w głąb, podczas gdy inna implementacja może pobierać obiekty w porządku w szerz. Oznacza to iż w zależności od wyboru strategii przeszukiwania drzewa obiektów, należałoby stosować odmienny algorytm rozmieszczania obiektów na dysku w celu zapewnienia najlepszego czasu dostępu.

Rozdział 4

Interfejs dla mechanizmu wykonawczego

Kod powstały w ramach niniejszej pracy magisterskiej obejmuje podstawowe funkcje przechowywania i operowania na danych w ramach obiektowego systemu zarządzania bazą danych. W jego skład wchodzi logika przechowywania obiektów na dysku, menedżer buforów oraz menedżer transakcji wraz z menedżerem dzienników. Wszystkie te elementy wspólnie stanowią zamkniętą całość umożliwiającą mechanizmowi wykonawczemu współpracowanie z danymi za pomocą prostego interfejsu. Prezentowane rozwiązanie jednakże nie dostarcza żadnej konkretnej implementacji modułu wykonawczego dla zapytań, tylko oferuje prosty interfejs umożliwiający pracę na drzewie obiektów w bazie danych w uniwersalny sposób.

Podstawową klasą w ramach interfejsów dostępowych jest klasa `DataLayer`. Odpowiedzialna jest ona za przechowywanie całej logiki wchodzącej w skład warstwy danych oraz odpowiednie zarządzanie menedżerami. Mimo to udostępnia ona publicznie jedynie trzy metody. Dwie realizujące uruchomienie warstwy danych oraz jej zatrzymanie oraz trzecią pozwalającą rozpocząć nową transakcję w ramach bazy danych oraz otrzymać instancję obiektu klasy `Transaction`. Wszelkie operacje na danych następnie są wykonywane przy pośrednictwie klasy `Transaction`. Ten sposób realizacji warstwy dostępowej oznacza, iż niezależnie od rodzaju żądania skierowanego do danych zgromadzonych w bazie, konieczne jest pośrednictwo mechanizmu transakcyjnego. W momencie wywołania funkcji rozpoczynającej transakcje możliwe jest określenie poziomu izolacji dla nowo tworzonej transakcji, w przypadku nie określenia tegoż poziomu, stosowany jest domyślny poziom izolacji ustawiony w ramach pliku konfiguracyjnego.

Obiekt transakcji udostępnia szereg metod związanych z kontrolowaniem przebiegu transakcji reprezentowanej przez tenże obiekt. W ich skład wchodzą metody pozwalające na wycofanie, zatwierdzenie lub ponowne uruchomienie transakcji w przypadku wystąpienia zakleszczenia. Dodatkowo obiekt transakcji udostępnia metody realizujące odwzorowanie z napisów reprezentujących nazwy na liczbowe identyfikatory nazw w ramach globalnej struktury odwzorowania nazw wewnątrz bazy danych. Konieczne jest stosowanie tych metod, gdyż wszystkie pozostałe metody udostępniane w ramach interfejsu dla mechanizmu wykonawczego oczekują liczbowych identyfikatorów nazw, zamiast nazw w formie napisów. Główne funkcje obiektu transakcji wiążą się jednak z metodami umożliwiającymi operowanie na obiektach zgromadzonych w bazie danych. Dostępne operacje obejmują pobieranie obiektów poprzez wskazanie bindera (struktury łączącej identyfikator nazwy oraz unikalny identyfikator obiektu), tworzenie nowych obiektów oraz pobieranie wszystkich obiektów korzennych lub jedynie obiektów o określonym identyfikatorze nazwy. Nowo utworzone obiekty za pośrednictwem

obiektu transakcji tworzone są jako obiekty korzenne.

Wszystkie obiekty pobierane z bazy danych, niezależnie od ich typu, reprezentowane są poprzez instancje klasy typu `ObjectPointer`. Każda instancja tejże klasy reprezentuje odrębny obiekt bazodanowy. W jej skład wchodzą własności oraz metody pozwalające uzyskiwać informacje oraz przeprowadzać operacje na konkretnym obiekcie bazodanowym, a w przypadku pracy z obiektem złożonym, możliwe jest pobranie kolekcji obiektów w nim zagnieżdżonych. Możliwe do uzyskania informacje o obiekcie obejmują jego unikalny identyfikator, identyfikator nazwy, nazwę, typ wartości oraz wskazanie na obiekt będący rodzicem. Dostęp do wartości obiektu możliwy jest poprzez wiele otypowanych własności specyficznych dla konkretnego typu obiektu. Operacje związane z obiektem to metoda pozwalająca usunąć dany obiekt wraz ze wszystkimi obiektami zagnieżdżonymi oraz metoda pozwalająca tworzyć nowe obiekty zagnieżdżone w ramach danego obiektu złożonego. Należy w tym momencie zauważyć, iż tak samo jak obiekt transakcji, instancja klasy reprezentującej złożony obiekt bazodanowy, pozawalają na tworzenie nowych obiektów, jednakże w przypadku transakcji nowo utworzone obiekty stają się korzeniami, podczas gdy tworzenie nowych obiektów przy pośrednictwie klasy `ObjectPointer`, tworzy obiekty zagnieżdżone.

Wskaźniki lub referencje do obiektów bazodanowych w ramach interfejsu reprezentowane zawsze są przez pary unikalnych identyfikatorów obiektów, będących za razem adresami fizycznymi obiektów, oraz identyfikatorów nazw. Pary te zwane są binderami. Podstawową zaletą łączenia adresów z nazwami i przechowywanie ich razem w trakcie przetwarzania i pobierania obiektów jest to, iż możliwe jest natychmiastowe określenie nazwy obiektu w dowolnym momencie wykonywania zapytania, bez konieczności sięgania do bazy danych i pobierania całych obiektów. Jest to niezmiernie przydatne w przypadku języków zapytań mających charakter ścieżkowy, kiedy wykonanie zapytania sprowadza się do przejścia w głąb po gałęziach o określonych nazwach i nie istnieje konieczność przeglądania wszystkich synów każdego odwiedzonego węzła.

Rozdział 5

Konfiguracja

Do konfiguracji warstwy danych wykorzystane zostały standardowe mechanizmy oferowane przez platformę .NET. Oznacza to, iż konfiguracja aplikacji przechowywana i odczytywana jest z pliku w formacie XML. Schemat pliku konfiguracyjnego jest ściśle określony w przypadku parametrów konfiguracyjnych środowiska .NET, jednakże możliwe jest dowolne rozszerzanie schematu do potrzeb aplikacji poprzez dodawanie dodatkowych sekcji konfiguracyjnych. Plik ten, zgodnie z konwencją obowiązującą w ramach .NET Framework, umieszczany jest w katalogu z aplikacją pod nazwą identyczną do nazwy programu wykonawczego, rozszerzoną o końcówkę „.config". Tak umieszczony plik pozwala uzyskać dostęp do zawartych w nim ustawień w trakcie działania aplikacji, zgodnie z jej potrzebami.

Podstawowa konfiguracja warstwy danych odbywa się w ramach sekcji diagnostics, log, transaction oraz store. Zadaniem dodatkowo występującej sekcji configSections jest rozszerzenie standardowego schematu konfiguracji programów opartych na technologii .NET o dodatkowe specyficzne dla nich sekcje konfiguracyjne oraz wskazanie klas odpowiedzialnych za ich obsługę. Poniżej zamieszczony został przykładowy plik konfiguracyjny dla powstałej warstwy danych.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
      <section name="diagnostics"
         type="Loxim.Configuration.DiagnosticsSection, Loxim.Utility"/>
      <section name="log"
         type="Loxim.Configuration.LogSection, Loxim.DataLayer"/>
      <section name="transaction"
         type="Loxim.Configuration.TransactionSection, Loxim.DataLayer"/>
      <section name="store"
         type="Loxim.Configuration.StoreSection, Loxim.DataLayer"/>
  </configSections>

  <diagnostics
      fileName="Messages.log"
      bufferSize="8192"
      level="Debug"
      consoleOutput="true"
   />
```

```
   <log
      fileName="Primary.xlf"
      differentiator="Loxim.Log.CompareDifferentiator, Loxim.DataLayer"
      compress="false"
   />

   <transaction
      isolationLevel="Serializable"
   />

   <store>
      <buffer
         lingerTime="00:01:00"
         flushInterval="00:00:30"
      />

      <storage
         fileName="Primary.xdf"
         fileGrowth="0.2"
         pageSize="1024"
         allocation="Sequential"
         fillFactor="0.8"
         encoding="utf-8"
      />
   </store>
</configuration>
```

Najistotniejszą a za razem najbardziej złożoną sekcją jest sekcja `store`. Składa się ona z dwóch podsekcji zawierających szereg parametrów pozwalających wpływać na działanie menedżera buforów oraz logiki składowania obiektów. W ramach ustawień bufora możliwe jest określenie czasu jak długo nieużywane czyste strony powinny być przechowywane w buforze (`lingerTime`) oraz czasu co jaki uruchamiany jest wątek odpowiedzialny za zapis brudnych stron na dysk (`flushInterval`). W sekcji konfiguracji warstwy danych oprócz ustawienia nazwy pliku bazy danych (`fileName`), możliwe jest określenie przyrostu wielkości pliku w momencie, gdy plik danych zostanie w pełni zapełniony (`fileGrowth`) oraz ustalenie rozmiaru strony (`pageSize`). Rozmiar strony, mimo iż jest konfigurowalny, nie może zostać zmieniony po utworzeniu pliku bazy danych, gdyż nie istnieją żadne mechanizmy realizujące konwersje obecnie już zaalokowanych stron do nowego rozmiaru. Dodatkowo istnieje parametr konfiguracyjny `fillFactor` określający poziom zapełnienia strony, po którego przekroczeniu żaden nowo utworzony obiekt nie może zostać na niej umieszczony. Rodzaj kodowania stosowany do przechowywania wartości napisowych wybierany jest za pomocą parametru `encoding`. Dostępne są wszystkie kodowania wspierane przez platformę .NET Framework miedzy innymi ASCII, UTF-7, UTF-8, Unicode, UTF-32 oraz wiele innych.

W ramach sekcji `log`, odpowiadającej za konfigurację modułu dzienników transakcyjnych, możliwe jest określenie nazwy pliku dziennika (`fileName`), nazwy typu klasy implementującej mechanizm wyliczania różnic pomiędzy stronami (`differentiator`) oraz czy wyliczone różnice powinny być poddawane kompresji przed umieszczeniem w dzienniku (`compress`). Dostępne są dwa typy mechanizmów porównywania stron: `CompareDifferentiator` znajdujący ciągi bajtów wstawionych, usuniętych oraz zmienionych oraz `StoreDifferentiator`

przechowujący kopię całych stron oryginalnych oraz zmienionych.

Jedynym parametrem konfiguracyjnym dostępnym w ramach menedżera transakcji jest ustawienie określające domyślny poziom izolacji dla nowo tworzonych transakcji. Natomiast sekcja `diagnostics` zawiera szereg parametrów odpowiedzialnych za działanie mechanizmu zapisywania i wyświetlania wiadomości diagnostycznych.

Rozdział 6

Modele M0, M1, M2 i M3

W ciągu minionych lat powstało wiele prób stworzenia wspólnego modelu dla obiektowych baz danych. Dwa największe i najbardziej znane projekty to ODMG OQL [Odmg00] i SQL-99 [Sql99]. Niestety żadne z tych rozwiązań nie zdobyło dużej popularności, głównie z powodu niesłychanej złożoności zaproponowanego modelu. W książce *Teoria i konstrukcja obiektowych języków zapytań* [Sub04] Kazimierz Subieta proponuje inne spojrzenie na problem konstrukcji modeli obiektowych baz danych. Wprowadza on szereg modeli, rozpoczynający się od najprostszego modelu, a następnie każdy kolejny model stanowi rozszerzenie poprzedniego zachowując jego własności.

M0 jest najprostszym modelem. Zawiera on złożone struktury hierarchiczne obiektów oraz zapewnia możliwość tworzenia powiązań wskaźnikowych pomiędzy obiektami. Nie zajmuje się za to klasami, rolami, interfejsami oraz dziedziczeniem. Za jego pomocą możliwe jest wyrażenie danych relacyjnych, zagnieżdżonych relacji oraz danych semistrukturalnych. Do osiągnięcia tych celów konieczne jest zastosowanie wewnętrznych identyfikatorów obiektów, nazw obiektów oraz wartości obiektów o określonym typie. Model M0 jest modelem w pełni wspieranym przez rozwiązanie prezentowane w niniejszej pracy magisterskiej.

W ramach modelu M1, rozszerzającego model M0, wprowadzone zostają klasy, dziedziczenie statyczne pomiędzy klasami wraz z wielokrotnym dziedziczeniem. Dodanie klas oznacza, iż również pojawiają się metody i procedury. Aby możliwe było wprowadzenie wsparcia dla tego modelu konieczna jest praca implementacyjna dotycząca mechanizmów klas i funkcji. Klasy powinny być reprezentowane przy wykorzystaniu możliwości obiektów złożonych oferowanych w ramach modelu M0. Konieczne jest dodane nowego typu obiektu dla klas, w pełni zachowującego właściwości obiektu złożonego, ale pozwalającego w łatwy sposób odróżniać obiekty klasy od zwykłych obiektów. Informacja o tym jakie klasy implementuje obiekt powinna być zapisywana wraz z tym obiektem, jednakże klasa niekoniecznie musi znać obiekty będące jej implementacjami, a nawet nie powinna. Jedynie obiekty złożone powinny móc implementować klasy i informacje o klasach implementowanych przez nie powinny być zapisywane w ramach wartości obiektu, gdyż niestety obiekt może implementować wiele klas, a więc lista zaimplementowanych klas ma zmienną długość. Wartości obiektów złożonych stanowią listę par binderów wraz z informacją o tym czy dany binder jest obiektem zagnieżdżonym, czy też rolą. Możliwe jest wprowadzenie kolejnych wartości, pozwalających przechowywać na tej liście bindery zaimplementowanych klas przez obiekt. Wszelkie dane, o które rozszerzany byłby obiekt złożony, wymagające pamięci stałego rozmiaru powinny być umieszczane w ramach struktury `ComplexEntry`, która wchodzi w skład każdego obiektu złożonego. Realizacja metod klasowych nie wymaga znaczących zmian w warstwie danych. Metody powinny zostać zaimplementowane albo jako odmiana binarnego typu danych lub jako wartości tekstowe,

wprowadzając nowy typ specyficzny dla metod, ale zachowując mechanikę wybranego podstawowego typu. Wykorzystanie typu binarnego pozwoliłoby zapisywać metody w postaci nie skompilowanej lub też zachowywać drzewo powstałe w wyniku analizy treść metody w zależności od potrzeb modułu wykonawczego, gdyż to moduł wykonawczy jest odpowiedzialny za konkretną implementację wsparcia proceduralnego. Warstwa danych jedynie ma służyć składowaniu metody.

Model M2 wprowadza dodatkowo pojęcie dynamicznych ról. Dynamiczne role pozwalają zmieniać na żądanie tożsamość obiektu. Mimo, iż opis logiki składowania obiektów nie porusza tego tematu, prezentowane rozwiązanie zawiera wsparcie dla dynamicznych ról. Realizacja ról jest analogiczna do relacji rodzic-dzieci, jednakże dotyczy jedynie obiektów złożonych. Każdy obiekt złożony może mieć dowolnie wiele ról, ale każdy obiekt może być co najwyżej rolą jednego obiektu.

Hermetyzacja jest jedynym nowym rozszerzeniem oferowanym przez model M3. W ramach zastosowania hermetyzacji następuje wprowadzenie podziału części składowych na publiczne i prywatne. Ten podział dotyczy relacji rodzic-dzieci, gdyż powiązanie obiektów z ich podobiektami jest realizowane w tym miejscu. Implementacyjnie należałoby rozszerzyć wartości listy binderów obiektów potomnych w ramach obiektu złożonego o dodatkowe pole określające poziom dostępu do konkretnej składowej.

Rozdział 7

Podsumowanie

Podstawy obiektowego systemu zarządzania bazą danych zrealizowane w ramach niniejszej pracy magisterskiej stanowią kompletny i spójny blok oprogramowania. Wykorzystanie i zaimplementowanie modułów składowych tj. menedżer buforów, transakcji oraz dzienników w ramach architektury warstwy danych pozwoliło uzyskać wydajne i bezpieczne transakcyjnie rozwiązanie. Oferując przejrzysty obiektowy interfejs dostępu do danych, przedstawione rozwiązanie, umożliwia proste rozszerzenie go o moduł wykonawczy zapytań dowolnego języka zapytań. Natomiast zwarta postać warstwy danych wraz z usługami osadzonymi w niej nie narzuca żadnych kłopotliwych wymagań na aplikację serwera, ale niekoniecznie musi to być serwer z dostępem sieciowym, w ramach której miałaby ona funkcjonować. Wszystko to razem pozawala niewielkim kosztem przekształcić wynik niniejszej pracy magisterskiej w pełni funkcjonalny system zarządzania bazą danych o podejściu obiektowym, zawierający wszystkie moduły występujące w dużych komercyjnych bazach danych.

Zastosowana reprezentacja danych oraz konstrukcja interfejsu dla warstwy wykonawczej oferuje elastyczny model współpracy z danymi. Możliwe jest rozszerzenie i kontynuacja niniejszego projektu w celu zaimplementowania dowolnych języków zapytań poczynając od XPath, przez język SBQL wykorzystujący stosowe podejście do pracy z obiektami, po rozwiązania pozwalające na korzystanie z danych zgromadzonych w obiektowej bazie danych bezpośrednio wewnątrz obiektowego języka programowania tj. C# czy Java, przy użyciu różnego rodzaju warstw pośrednich. Dla każdego z wymienionych języków zapytań lub też metodologii dostępu można zaimplementować specyficzne egzekutory pracujące na wspólnej warstwie danych. Wytworzone moduły wykonawcze mogłyby pracować równocześnie, niezależnie obsługując wielu klientów zgodnie z ich potrzebami.

Wykorzystanie środowiska zarządzalnego do budowy podstawy SZBD ma także dużo zalet związanych z badawczym charakterem pracy. Środowisko zarządzalne .NET Framework oferując bogaty zestaw klas zgromadzonych w bibliotekach systemowych, pozwala programiście skupić się jedynie na najważniejszych aspektach tworzonego oprogramowania oraz algorytmów w nim stosowanych, znacznie zmniejszając czas potrzebny do wykonania dodatkowego modułu i odciążając go od konieczności dbania o szczegóły związane z zarządzaniem pamięcią.

Dodatek A

Zawartość płyty kompaktowej

Katalog	*Opis*
\thesis	Katalog zawiera tekst źródłowy niniejszego pliku w formacie LaTeX.
\thesis\figures	Katalog zawiera pliki źródłowe obrazków wykorzystanych w pracy magisterskiej. Dla każdego obrazka istnieje plik źródłowy w formacie SVG oraz dwa pliki wynikowe odpowiednio w formatach EPS i PNG.
\source\Loxim.DataLayer	W tym katalogu umieszczone zostało właściwe oprogramowanie wykonane w ramach pracy magisterskiej. Znajdują się tam pliki i katalogi składające się na warstwę danych podstaw systemu zarządzania bazą danych. Wszystkie trzy podkatalogi umieszczone w katalogu source zawierają dodatkowo pliki projektów Visual Studio 2005.
\source\Loxim.Utility	Zawiera pliki z pomocniczym kodem dla warstwy danych.
\source\Server	W katalogu znajduje się kod źródłowy programu zawierającego szereg testów dla powstałej wartwy danych.
\bin	Katalog zawiera szereg plików binarnych powstałych z kompilacji kodu źródłowego z katalogów source.

Dodatek B

Przykłady wykorzystania

Poradnik instalacji

Do uruchamiania lub kompilacji testowych programów konieczna jest obecność środowiska uruchomieniowego platformy .NET Framework w wersji 2.0. Większość dzisiejszych instalacji systemu Windows posiada preinstalowaną wersję tego oprogramowania lub też zostało ono uprzednio pobrane z witryny Windows Update. W przypadku gdy na komputerze nie jest zainstalowane oprogramowanie .NET, konieczne jest pobranie wersji uruchomieniowej lub pełniejszego pakietu dla programistów, wzbogaconego o przykłady oraz bogatą dokumentację, z witryny Windows Update lub bezpośrednio używając jednego z poniższych odnośników. Możliwe jest również wykorzystanie otwartego projektu implementacji alternatywnej platformy uruchomieniowej Mono, mającej za cel pełną kompatybilność z .NET, jednakże niniejszy poradnik nie obejmuje tego tematu.

- Pakiet podstawowy (zawiera również kompilator)
  http://www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5

- Pakiet wzbogacony o przykłady i dokumentację
  http://www.microsoft.com/downloads/details.aspx?FamilyID=fe6f2099-b7b4-4f47-a244-c96d69c35decn

Następnie konieczny jest plik z kodem testowym. W tym pliku umieszczamy wywołania warstwy danych oraz instrukcje pozwalające na korzystanie z jej funkcji. Poniżej zamieszczono listing prostego programu tworzącego instancje składu danych oraz tworzącego jeden obiekt prosty o wartości liczbowej 5 i wypisującego informacje o wszystkich korzeniach w bazie danych. Kod został napisany w języku C# oraz powinien zostać zapisany w pliku `Test.cs`.

```
using System;
using Loxim;
using Loxim.Store;
using Loxim.Transactions;

public class Test
{
   static void Main(string[] args)
   {
```

```
      DataLayer dataLayer = new DataLayer();
      dataLayer.Start();

      Transaction t = dataLayer.BeginTransaction();

      t.CreateObject(t["kilof"], 5);

      foreach (Binder binder in t.GetRoots())
         Console.WriteLine("binder={0} name={1}", binder, t[binder.NameID]);

      t.Commit();

      dataLayer.Stop();
   }
}
```

Kolejnym etapem jest kompilacja. Do kompilowania programów w języku C# służy kompilator `csc.exe`. Dostarczany jest on z każdą wersją platformy .NET Framework i jest umieszczany w katalogu `%SystemRoot%\Microsoft.NET\Framework\`*wersja*, gdzie numer wersji dla 2.0 to `v2.0.50727`. Aby kompilacja mogła się powieść konieczne są jeszcze biblioteki warstwy danych `Loxim.DataLayer.dll` oraz `Loxim.Utility.dll`, które należy umieścić w katalogu z plikiem testowym. Dodatkowo jeżeli kompilujemy w trybie debug, możemy umieścić w katalogu pliki pdb zawierające symbole dla tych bibliotek, które są pomocne w przypadku występowania problemów. Proces kompilacji uruchamiany jest poleceniem:

```
%SystemRoot%\Microsoft.NET\Framework\v2.0.50727\csc /debug /t:exe
      /r:Loxim.DataLayer.dll,Loxim.Utility.dll /out:Test.exe Test.cs
```

Jeżeli kompilacja zakończy się sukcesem, w katalogu pojawi się nowy plik `Test.exe`. Zanim zostanie on uruchomiony konieczne jest jeszcze upewnienie się, iż istnieje domyślny plik konfiguracyjny dla tego pliku. Domyślne pliki konfiguracyjne zawsze mają nazwę taką jak programy, tylko dodatkowo mają umieszczone rozszerzenie „.config", czyli dla programu testowego pełna nazwa pliku konfiguracyjnego to `Test.exe.config`. Przykładowy plik konfiguracyjny wymagany przez warstwę danych można odnaleźć w katalogu `\source\Server`. Należy go przekopiować i nadać mu właściwą nazwę. Po dokonaniu ostatniego kroku możliwe jest uruchomienie aplikacji testowej.

Podstawowy program

Do wykorzystania warstwy danych powstałej w ramach niniejszej pracy magisterskiej, konieczne jest utworzenie instancji obiektu `DataLayer`. Następnie należy wywołać metodę uruchamiającą warstwę danych. Po tych przygotowaniach możliwe jest rozpoczynanie nowych transakcji otrzymując obiekt `Transaction` po wywołaniu metody `BeginTransaction`. Obiekt transakcji zawiera funkcje pozwalające na tworzenie nowych obiektów będących korzeniami oraz pobierania obiektów korzeniowych obecnie istniejących w bazie danych. W poniższym przykładzie za pomocą obiektu transakcji został utworzony obiekt złożony reprezentujący osobę. Następnie do tego obiektu zostały dodane podobiekty imię, nazwisko, data urodzenia, wzrost oraz dwa numery telefonów. Dodawanie podobiektów realizowane jest poprzez

wywoływanie metody `CreateObject` na obiekcie, zamiast na transakcji. Także został utworzony podobiekt złożony adres, składający się z kilku prostych podobiektów. Osobno został utworzony obiekt studenta i został ustawiony jako rola obiektu osoba przy pomocy własności `RoleOf`. Po wykonaniu wszystkich operacji transakcja została zatwierdzona, a następnie cały skład danych został zatrzymany.

```
using System;
using Loxim;

public class Test
{
   static void Main(string[] args)
   {
      // Tworzenie i uruchamianie warstwy danych
      DataLayer dataLayer = new DataLayer();
      dataLayer.Start();

      // Otwieranie nowej transakcji
      Transaction t = dataLayer.BeginTransaction();

      // Tworzenie obiektu złożonego "person" oraz prostych podobiektów
      ObjectPointer person = t.CreateObject(t["person"]);
      person.CreateObject(t["firstName"], "Maksymilian");
      person.CreateObject(t["lastName"], "Ochudzki");
      person.CreateObject(t["birthDate"], DateTime.Parse("1977-03-21"));
      person.CreateObject(t["height"], 178);
      person.CreateObject(t["telephone"], "555-247-725");
      person.CreateObject(t["telephone"], "555-015-381");

      // Tworzenie podobiektu złożonego
      ObjectPointer address = person.CreateObject(t["address"]);
      address.CreateObject(t["street"], "Bracka");
      address.CreateObject(t["number"], 7);
      address.CreateObject(t["postalCode"], "05-635");

      // Tworzenie obiektu student i przypisanie go jako rola obiektu person
      ObjectPointer student = t.CreateObject(t["student"]);
      student.CreateObject(t["indexNumber"], 199287);
      student.RoleOf = person.Binder;

      // Zatwierdzenie transakcji
      t.Commit();

      // Zatrzymywanie warstwy danych
      dataLayer.Stop();
   }
}
```

Zagnieżdżanie obiektów oraz zmiana rodzica

Wszystkie obiekty składowane w ramach bazy danych albo są korzeniami, albo są zagnieżdżone w ramach obiektów złożonych. Obiekty raz utworzone jako korzenie lub jako podobiekty innych obiektów nie muszą na zawsze pozostać w tej roli. Możliwa jest zmiana rodzica obiektu poprzez przypisanie bindera innego obiektu lub pustego bindera na własność `Parent` w dowolny sposób. Jednakże tak, aby nie powstał cykl na relacji rodzicielstwa, co uniemożliwiałoby dostęp do obiektów znajdujących się w ramach tego cyklu, gdyż nie istniałby korzeń w jego obrębie. W ramach poniższego przykładu w pierwszej części tworzone są trzy obiekty. Obiekt korzeniowy `a`, do którego następnie dodawany jest podobiekt `b`, a do `b` dodawany jest obiekt `c`. Po dokonaniu podmiany rodziców dla obiektów `b` i `c` w bazie danych obiekt `a` i `c` są korzeniami oraz obiekt `c` jest podobiektem obiektu `a`. W drugiej części przykładu przedstawiony został scenariusz demonstrujący kontrolę poprawności przypisań rodziców zapobiegającą tworzeniu cykli.

```
public void Nesting(Transaction t)
{
   ObjectPointer a, b, c, d;

   // Tworzenie obiektów zagnieżdżonych
   a = t.CreateObject(t["a"]);
   b = a.CreateObject(t["b"]);
   c = b.CreateObject(t["c"]);

   // b przestaje być podobiektem a i staje się korzeniem
   b.Parent = Binder.Empty;
   // c staje się podobiektem c
   c.Parent = a.Binder;

   try
   {
      // Tworzenie obiektów zagnieżdżonych
      a = t.CreateObject(t["a"]);
      b = a.CreateObject(t["b"]);
      c = b.CreateObject(t["c"]);

      // Próba zmiany rodzica powodująca cykl
      a.Parent = c.Binder;
   }
   catch (NestingException)
   {
      Console.WriteLine("*** Wykryto niedozwolony cykl! ***");
   }
}
```

Eksport zawartości bazy danych do pliku XML

Poniżej przedstawiony został przykład kodu pozwalający na wygenerowanie pliku w formacie XML zawierającego informację o obiektach znajdujących się w bazie danych. Wykorzystuje

on standardowe klasy platformy .NET Framework do celu tworzenia plików XML w prosty sposób. Właściwe przetwarzanie rozpoczyna się od pobrania wszystkich korzeni za pomocą metody `GetRoots`, a następnie rekurencyjnie wywoływana jest funkcja `XmlDumpRecursion` zapisująca informację o danym obiekcie. W tejże funkcji generowane są podstawowe elementy i atrybuty XML opisujące obiekt oraz w zależności od typu przetwarzanego obiektu następuje zejście rekurencyjne do obiektów zagnieżdżonych i wskazywanych lub wypisanie wartości obiektów prostych. Dodatkowo przykład ilustruje sposób wykorzystania mechanizmu ról, zachowującego analogiczny interfejs do relacji rodzicielstwa. Tak jak możliwe jest otrzymanie informacji o dzieciach obiektu złożonego poprzez własność `Children`, tak informacje o rolach można uzyskać poprzez `Roles`.

```
using System;
using System.IO;
using System.Xml;

public void XmlDump(Transaction t, string fileName)
{
   XmlWriterSettings s = new XmlWriterSettings();
   s.Indent = true;
   s.IndentChars = "\t";

   StreamWriter w = File.CreateText(fileName);
   XmlWriter x = XmlWriter.Create(w, s);

   try
   {
      x.WriteStartDocument();
      x.WriteStartElement("root");

      // Wywołanie rekurencyjne dla każdego obiektu będącego korzeniem
      foreach (Binder b in t.GetRoots())
         XmlDumpRecursion(t, x, t.GetObject(b));

      x.WriteEndElement();
      x.WriteEndDocument();
   }
   finally
   {
      x.Close();
      w.Close();
   }
}

private void XmlDumpRecursion(Transaction t, XmlWriter x, ObjectPointer o)
{
   // Utworzenie elementu o nazwie obiektu
   x.WriteStartElement(o.Name);
   // Oraz dodawanie podstawowych atrybutów
   x.WriteAttributeString("type", o.Type.ToString());
```

```
   w.WriteAttributeString("binder", o.Binder.ToString());

   // Jeżeli obiekt jest złożony
   if (o.Type == ObjectType.Complex)
   {
      // Dla każdego podobiektu wywołanie rekurencyjne
      foreach (Binder b in o.Children)
         XmlDumpRecursion(t, x, t.GetObject(b));

      // Dla każdej roli utworzenie elementu role
      foreach (Binder b in o.Roles)
      {
         x.WriteStartElement("role");
         XmlDumpRecursion(t, x, t.GetObject(b));
         x.WriteEndElement();
      }
   }
   // Jeżeli obiekt jest wskaźnikiem
   else if (o.Type == ObjectType.Pointer)
      // Wywołanie dla wskazywanego obiektu
      XmlDumpRecursion(t, x, t.GetObject(o.PointerValue));
   // Obiekt jest prosty, wypisanie wartości
   else
      x.WriteValue(o.Value);

   // Zamknięcie elemenmtu obiektu
   x.WriteEndElement();
}
```

Bibliografia

[Bun97] Peter Buneman, *Semistructured Data*, Symposium on Principles of Database Systems, Pages: 117–121, 1997

[Codd70] E.F Codd, *A relational model for large shared data banks*, Communications of the ACM, Volume 13, Issue 6, Pages: 377–387, 1970

[Date95] C.J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Company Inc., 1995

[Mol00] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, *Database System Implementation*, Prentice-Hall, 2000

[Odmg00] R. G. G. Cattell, Douglas K. Barry, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000

[Sql99] Peter Gulutzan, Trudy Pelzer *SQL-99 Complete, Really*, CMP Books, 1999

[Sub04] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo PJWSTK, 2004

[Weik01] Gerhard Weikum, Gottfried Vossen, *Transactional Information Systems*, Elsevier, 2001