POLITECHNIKA BIAŁOSTOCKA
WYDZIAŁ INFORMATYKI
KATEDRA OPROGRAMOWANIA

Marek Grześ
Nr albumu 32254

Przetwarzanie rozproszonych zapytań w heterogenicznym środowisku baz danych

Praca magisterska
napisana pod kierunkiem
dr inż. Marka Krętowskiego

Białystok 2003

Białystok, dn. 23 czerwca 2003 roku

Politechnika Białostocka
Wydział Informatyki

Marek Grześ

Oświadczenie

Przyjmuję do wiadomości, że Wydział Informatyki Politechniki Białostockiej nabywa prawa autorskie dotyczące rezultatów i oprogramowania wytworzonego w ramach przygotowania pracy magisterskiej (autor: Marek Grześ, tytuł: "Przetwarzanie rozproszonych zapytań w heterogenicznym środowisku baz danych", promotor: dr inż. Marek Krętowski). Ich publikacja i wykorzystanie wymaga zgody Dziekana Wydziału Informatyki Politechniki Białostockiej.

Podpis magistranta

.....

Podstawa prawna: Ustawa z dnia 4.02.1994 o prawie autorskim i prawach pokrewnych (Dz. U. z 1994 r Nr 24).

Art. 1. Przedmiotem prawa autorskiego jest każdy przejaw działalności twórczej o indywidualnym charakterze, ustalony w jakiejkolwiek postaci, niezależnie od wartości, przeznaczenia i sposobu wyrażenia (utwór).

Art. 2. W szczególności przedmiotem prawa autorskiego są utwory:

1. wyrażone słowem, symbolami matematycznymi, znakami graficznymi (literackie, publicystyczne, naukowe, kartograficzne oraz programy komputerowe).

Art. 3. Utwór jest przedmiotem prawa autorskiego od chwili ustalenia, chociażby miał postać nieukończoną.

Art. 4. Ochrona przysługuje twórcy niezależnie od spełnienia jakichkolwiek formalności.

Ustalenie w rozumieniu tej ustawy następuje w chwili opublikowania, złożenia w formie opisanej, lub wygłoszenia publicznego (w formie wykładu, referatu, seminarium itp.).

Pisanie nie polega na umiejętności pisania,
lecz na umiejętności wykreślania tego, co źle napisane.

Antoni Czechow

Spis treści

Rozdział 1

Wprowadzenie

Systemy komputerowe przeżywające ciągłą ewolucję osiągnęły już poziom rozwoju, który pozwala zaspokoić większość potrzeb efektywnego przetwarzania danych. Niewątpliwie ogromne znaczenie miały tu osiągnięcia lat osiemdziesiątych, szczególnie opracowanie i upowszechnienie mikroprocesorów. Głównie dzięki temu osiągnięciu komputery pojawiły się w domach prywatnych oraz przedsiębiorstwach małej i średniej wielkości. Nie kosztowały już tyle co duże komputery mainframe. Kolejnym istotnym osiągnięciem w dziedzinie informatyki było rozpowszechnienie sieci komputerowych, szczególnie szybkich sieci lokalnych. Stały się one infrastrukturą do tworzenia systemów rozproszonych.

Dynamiczny rozwój komputerów oraz rywalizacja ich producentów prowadzą do zróżnicowania platform sprzętowych i programowych, a przez to do wynikającej z tego powodu niekompatybilności. W chwili obecnej, kiedy na rynku komputerowym istnieje duża różnorodność systemów, począwszy od różnych odmian systemu Windows, przez systemy UNIX i LINUX, po systemy serii MAC OS X, niezwykle istotną okazuje się konieczność ich integracji. Potrzeba ta wynika w wielu przypadkach z naturalnych praw rządzących światem biznesu. Fuzje firm czy wchłanianie mniejszych przedsiębiorstw przez większe prowadzą często do konieczności zaadoptowania istniejącego sprzętu komputerowego i oprogramowania, których likwidacja jest ekonomicznie nieuzasadniona. Potrzeba integracji systemów komputerowych jest następstwem ich gwałtownego rozwoju i masowego wykorzystania oraz stanowi wyzwanie stojące przed przemysłem.

Integracja heterogenicznych systemów będzie tu rozważana w kontekście rozproszonego przetwarzania danych, które umożliwiają coraz szybsze sieci komputerowe, w szczególności sieć Internet. Analiza systemów rozproszonych będzie dotyczyła rozproszonych baz danych ze szczególnym uwzględnieniem przetwarzania zapytań.

Potrzeba zajmowania się rozproszonymi bazami danych wynika z konieczności przechowywania i przetwarzania coraz większych ilości danych, a także z problemów w zarządzaniu przedsiębiorstwami. Przedsiębiorstwa są często rozproszone. Rozproszenie to może być widziane zarówno fizycznie, jako różne lokalizacje ich oddziałów, lub logicznie w postaci podziałów organizacyjnych. W takim przypadku najkorzyst-

niej jest przechowywać dane w miejscu ich powstawania i wykorzystania. Systemy rozproszone zyskują w tym momencie na znaczeniu, pozwalając odzwierciedlić strukturę przedsiębiorstwa i przyczyniając się do poprawy wydajności przetwarzania. Dane związane z określonym oddziałem mogą być przechowywane w nim lokalnie, a system rozproszonej bazy danych pozwala na globalny dostęp do wszystkich zasobów.

Poza względami organizacyjnymi na korzyść rozproszonego przetwarzania danych przemawiają również względy techniczne. Wykorzystując zbiór komputerów, można uzyskać lepszy współczynnik ceny do wydajności niż odpowiadający mu jeden komputer, np. mainframe. Tego typu zbiór komputerów może mieć większą moc obliczeniową niż moc jaką kiedykolwiek osiągnie pojedynczy procesor.

Rozproszone bazy danych warto także analizować w kontekście hurtowni danych. Hurtownie danych optymalizowane są pod kątem zapytań operujących na ogromnych ilościach danych. Często dla każdego zapytania indywidualnie tworzone są indeksy pozwalające na skrócenie czasu wydobycia informacji. Warto się tu zastanowić, czy skoro wiele przypadków analizy danych pochodzących z hurtowni danych wymaga szczegółowego projektowania indeksów, to czy nie można uzyskać poprawy wydajności rozmieszczając dane na różnych komputerach. Przy takim rozmieszczeniu zapytanie może wykonywać się równolegle na wielu maszynach.

1.1 Cel pracy

Celem pracy jest stworzenie oprogramowania pozwalającego na rozproszone przetwarzanie zapytań w środowisku heterogenicznych baz danych. Jego realizacja poprzedzona zostanie próbą ugruntowania i podsumowania wiedzy w zakresie sposobów komunikacji międzyprocesowej w sieci, a także w zakresie rozproszonych baz danych, w szczególności przetwarzania rozproszonych zapytań. Rozważania te mają być podstawą do przedstawienia koncepcji i projektu, a następnie wykonania implementacji oprogramowania. Będą ukierunkowane na ukazanie standardów opartych o język XML *(ang. Extensible Markup Language)* jako sposobu na zapewnienie współpracy różnych systemów w środowisku rozproszonym w sposób prostszy i bardziej naturalny niż istniejące dotychczas technologie. Rozważania te mają służyć dobraniu odpowiednich technik, które zagwarantują heterogeniczność baz danych tworzonego systemu. Analiza baz danych ukierunkowana na metody dostępu do danych, z uwzględnieniem różnych modeli danych, ma pokazać istniejące obecnie standardy. Celem pracy jest również przedstawienie rozproszonych baz danych, które ma stanowić podsumowanie wiedzy, a zarazem wstęp teoretyczny do realizacji przetwarzania rozproszonych zapytań. Realizacja postawionego w pracy celu projektowo - programistycznego, oprócz wyzwania natury intelektualnej, ma służyć pokazaniu, że rozproszone przetwarzanie danych, szczególnie przetwarzanie rozproszonych zapytań, jest godną uwagi dyscypliną informatyki. Jej rozwijanie i uprawianie już w chwili obecnej przynosi korzyści, a często nawet gwarantuje jedyną drogę rozwiązania

pewnych problemów.

1.2 Układ pracy

Organizacja pracy jest następująca. W rozdziale drugim zawarto analizę metod i technik rozproszonego przetwarzania danych z uwzględnieniem ich stopniowego rozwoju. W rozdziale trzecim dokonano analizy metod i technik dostępu do baz danych. Rozdział czwarty wprowadza w tematykę rozproszonych baz danych i przedstawia ich główne cechy i założenia projektowe. Rozdział piąty szczegółowo ukazuje problem przetwarzania zapytań. Stanowi to podstawy do zrealizowania oprogramowania opisanego w rozdziale szóstym. Wyniki testów stworzonego systemu zebrano w rozdziale siódmym, a podsumowanie pracy stanowi rozdział ósmy. W dodatkach dokonano przeglądu literatury oraz wyjaśniono podstawowe pojęcia, akronimy i skróty.

Rozdział 2

Sieciowe mechanizmy komunikacji międzyprocesowej

Analiza zagadnienia przetwarzania rozproszonego zostanie poprzedzona wyjaśnieniem kilku kluczowych pojęć. Jest to szczególnie istotne ze względu na fakt, iż systemy rozproszone w literaturze często są odmiennie definiowane. W [19] podano następujące określenie: *"W systemie rozproszonym [...], inaczej - luźno powiązanym, procesory nie dzielą pamięci ani zegara. Zamiast tego każdy procesor ma własną pamięć lokalną. Procesory komunikują się ze sobą za pomocą różnych sieci komunikacyjnych"*. Podobne określenie systemów rozproszonych znajduje się w [11, str. 21]: *"System rozproszony określamy jako zbiór samodzielnych komputerów połączonych za pomocą sieci, wyposażonej w oprogramowanie zaprojektowane z myślą o utworzeniu zintegrowanego środowiska obliczeniowego"*. Prowadzone w pracy dalsze rozważania będą dotyczyły systemów rozproszonych zdefiniowanych jak wyżej, tzn. rozumianych jako samodzielne komputery połączone za pomocą sieci, przy czym ze względu na istnienie opóźnień powodowanych przez sieć, czas komunikacji między nimi jest uważany za istotny.

Z przetwarzaniem rozproszonym wiążą się pojęcia przetwarzania współbieżnego i równoległego. W [4] przetwarzanie współbieżne definiuje się następująco: *"Program współbieżny jest zbiorem zwykłych programów sekwencyjnych wykonywanych abstrakcyjnie równolegle"*. W przytoczonej definicji użyto określenia - abstrakcyjnie - w celu wyraźnego odróżnienia przetwarzania współbieżnego od przetwarzania równoległego. Przetwarzanie współbieżne może odbywać się przy użyciu jednego procesora z abstrakcyjnie jednoczesnym, współbieżnym wykonywaniem programów, w szczególności wątków lub procesów. Nieco szerzej pojęcie to zdefiniowano w [21, str. 19]: *"Mówimy, że dwa procesy są współbieżne, jeśli jeden z nich rozpoczyna się przed zakończeniem drugiego"*. Określenie to jest na tyle ogólne, że w jego myśl współbieżnymi są dowolne dwa procesy działające na dwóch dowolnych maszynach nie mających ze sobą połączenia. Z kolei przetwarzanie równoległe odbywa się z wykorzystaniem wielu procesorów, które pozwalają na równoczesne[1] wykonywanie

[1]Pomijany jest tu fakt, iż według szczególnej teorii względności Einsteina nie istnieje czas ab-

wielu procesów, ale przy istotnym założeniu, że czas komunikacji między współpracującymi procesorami jest pomijalnie mały. Mają one w tym przypadku najczęściej wspólną pamięć i zegar. Dotyczy to głównie maszyn wieloprocesowych.

Z systemami rozproszonymi związane jest pojęcie przezroczystości tych systemów jako cechy, która powoduje, iż użytkownik widzi taki system jako system scentralizowany, tzn. istnienie sieci komputerowej i wielu komputerów jest przed nim ukryte [12, str. 666]. Warto tu zwrócić uwagę, iż owa przezroczystość może występować na różnych poziomach abstrakcji. Otóż najczęściej pojawia się ona na poziomie końcowego użytkownika oprogramowania, gdzie odpowiedzialność za przezroczystość ponosi programista, twórca konkretnej aplikacji, wykorzystującej techniki przetwarzania rozproszonego. Znacznie rzadziej występują sytuacje, kiedy już przed programistą system rozproszony jest przezroczysty. Dotyczy to przede wszystkim rozproszonych systemów operacyjnych. Omawiane w pracy mechanizmy gwarantują najczęściej przezroczystość dopiero na poziomie końcowego użytkownika aplikacji.

2.1 Wady i zalety rozproszonego przetwarzania danych

Podejmując rozważania i analizę rozproszonego przetwarzania danych, należy zastanowić się jeszcze nad tym, jakie mogą płynąć z niego korzyści i z jakimi problemami można się zetknąć. Znaczenie przetwarzania rozproszonego podkreśla dzisiejsza świetność Internetu będącego z jednej strony rynkiem zbytu na technologie rozproszonego przetwarzania danych, z drugiej zaś inspiracją do ich rozwoju. Najważniejsze zalety systemów rozproszonych, mające także odzwierciedlenie we właściwościach Internetu, przedstawiają się następująco:

- Zbiór mikroprocesorów może mieć lepszy współczynnik ceny do wydajności niż odpowiadający mu jeden procesor w dużym komputerze mainframe [45, str. 20 i nast.].
- System rozproszony może mieć większą moc obliczeniową[2] niż moc jaką kiedykolwiek osiągną systemy scentralizowane.
- Niekiedy zastosowanie obliczeń równoległych lub rozproszonych jest jedynym sposobem uzyskania rozwiązania zadania numerycznego w akceptowalnym czasie, np. symulacji zjawiska fizycznego w czasie krótszym niż nastąpi rzeczywiste zdarzenie. W [26, str. 13 i nast.] znajduje się analiza, z której wyciągane

solutny.

[2]Terminem tym określana jest szybkość jednostki centralnej systemu komputerowego wyrażona w milionach rozkazów na sekundę (ang. Millions of Instructions Per Second - MIPS) lub milionach operacji zmiennopozycyjnych wykonywanych w ciągu sekundy (ang. Millions of Floating Point Operations Per Second - MFLOPS)[38].

są wnioski, że obliczenie prognozy pogody dla Polski, rozwiązując układ równań różniczkowych cząstkowych Naviera - Stokesa, z pełną dokładnością i w podanym czasie na komputerze z jednym procesorem jest niemożliwe przy założeniu, że największą osiągalną prędkością w przyrodzie jest prędkość światła.

- Komputery wchodzące w skład systemu rozproszonego mogą zachowywać lokalną autonomię. Każdy z nich może mieć swoją politykę bezpieczeństwa, ustawienia i udostępniane zasoby [46, str. 11].

- Systemy rozproszone gwarantują przyrostowy rozwój. Możliwe jest stopniowe dodawanie nowych komputerów zwiększających zasoby systemu.

- Z wyżej wymienionej właściwości wynika również większa niezawodność, gdyż przyrostowy rozwój umożliwia także usuwanie komputerów z systemu bez powodowania jego awarii. Jest to jedno z założeń Internetu, który był projektowany z możliwością funkcjonowania w wyniku częściowego zniszczenia.

- Komputery wchodzące w skład systemu rozproszonego mogą dzielić między sobą posiadane zasoby [46, str. 11].

Systemy rozproszone mają również swoje wady i ograniczenia. Jako najważniejsze z nich można wymienić:

- Stosunkowo nowe podejście, mało doświadczeń w zakresie budowy i eksploatacji takich systemów, brak oprogramowania [45, str. 24].

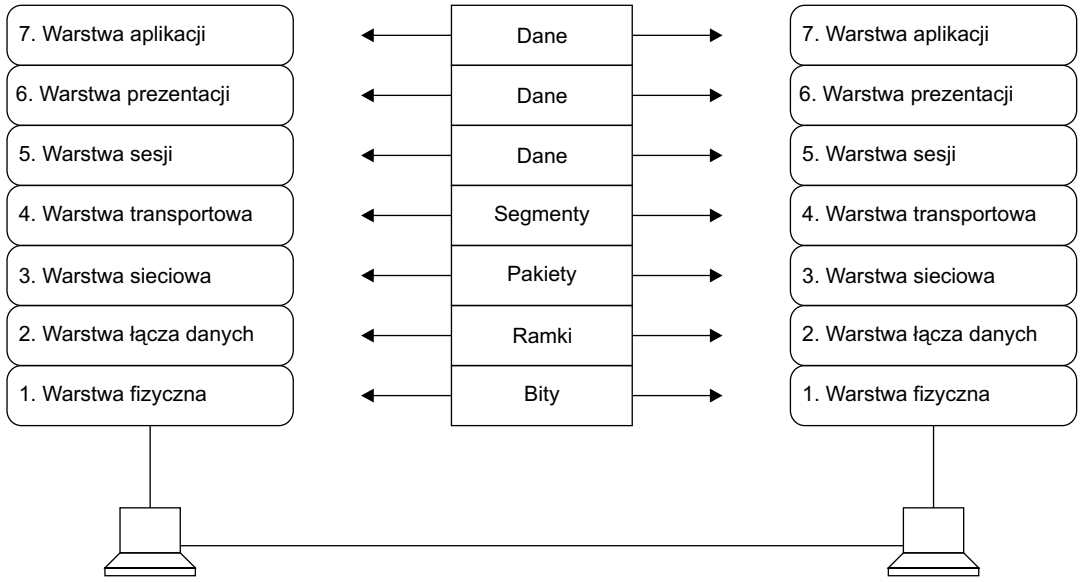
- Tworzenie systemów rozproszonych nie jest łatwe. Taki system musi być odporny na błędy jakie mogą pojawić się na komputerach wchodzących w skład systemu rozproszonego [46, str. 12].

- Jakość działania systemu rozproszonego silnie zależy od niezawodności i przepustowości sieci komunikacyjnej. Jest to szczególnie zauważalne w funkcjonowaniu Internetu.

- Autoryzacja dostępu jest trudniejsza do zaimplementowania, szczególnie w Internecie, do którego jest obecnie dostęp z każdego zakątka świata.

2.2 Znaczenie sieci komputerowych

W pracy przyjmuje się, że system rozproszony stanowią komputery połączone siecią komputerową. Istotna okazuje się tu potrzeba komunikacji między odległymi maszynami. Lokalne sieci komputerowe umożliwiają łączenie w obrębie budynków nawet setek komputerów w taki sposób, że niewielkie ilości informacji mogą być przesyłane między maszynami w czasie tysięcznych i krótszych części sekundy. Pozwoliły one na tworzenie szybkich, zintegrowanych środowisk obliczeniowych.



Rysunek 2.1: Warstwowy model komunikacji sieciowej według modelu OSI

Sieci komputerowe, biorąc pod uwagę zasoby fizyczne, składają się z mediów transmisyjnych, różnego typu urządzeń aktywnych oraz urządzeń nadawczych i odbiorczych. Sam sprzęt nie wystarcza jeszcze do realizacji efektywnej komunikacji. Do tego celu potrzebne są standardy i protokoły, czyli zestawy reguł definiujących sposoby komunikowania się i wymiany danych. Istotnym osiągnięciem na drodze określania standardów było opracowanie zaleceń przez Międzynarodową Organizację Normalizacyjną *(ang. International Organization for Standardization)* zwanych Modelem Wzorcowym Połączeń w Systemach Otwartych *(ang. Open System Interconnection Reference Model)* [45, str. 56]. Opracowanie tego modelu miało kluczowe znaczenie dla rozwoju sieci. Komunikację sieciową sprowadza on do abstrakcyjnego modelu warstwowego (rys. 2.1), w którym poszczególne warstwy, czasem wyróżnione w sprzęcie i oprogramowaniu w postaci modułów, są odpowiedzialne za realizację określonego podzbioru funkcji na rzecz innych sąsiednich modułów.

Model odniesienia ISO/OSI składa się z siedmiu warstw. Z punktu widzenia rozważanych w pracy zagadnień warto zwrócić uwagę na warstwę trzecią i czwartą. Warstwa trzecia, zwana warstwą sieciową, odpowiada za umożliwienie transmisji między dwoma dowolnymi węzłami, o ile istnieje między nimi połączenie fizyczne. Stosowane są w tej warstwie protokoły takie jak protokół IP *(ang. Internet Protocol)* i IPX *(ang. Internetwork Packet Exchange)*. Z kolei protokoły warstwy czwartej, zwanej warstwą transportową, mają na celu realizację niezawodnego transportu. Przykładem jest protokół TCP *(ang. Transmission Control Protocol)* i SPX *(ang. Sequenced Packet Exchange)*. Protokoły TCP i IP są jednymi z protokołów wchodzących w skład stosu protokołów TCP/IP. Warto zaznaczyć, że TCP/IP nie jest pojedynczym produktem. Jest to uogólniona nazwa całej rodziny protokołów i opro-

gramowania udostępniającego szereg usług sieciowych.

Dwie warstwy wymienione wyżej są godne uwagi podczas rozważań nad zagadnieniem systemów rozproszonych, gdyż z punktu widzenia abstrakcji programistycznych adresowanie maszyn w sieci odbywa się z wykorzystaniem warstwy sieciowej, zaś realizacja transportu i, co najważniejsze, adresowanie zasobów w ramach jednej maszyny zapewnia warstwa transportowa. W przypadku protokołu IP adresowanie odbywa się z wykorzystaniem ciągle jeszcze trzydziesto-dwu[3] bitowych adresów internetowych. Warstwa transportowa używa liczb całkowitych zapisywanych na szesnastu bitach zwanych numerami portów. Połączenie adresu warstwy sieciowej i numeru portu warstwy transportowej prowadzi do abstrakcji zwanej gniazdem.

Omawiana wyżej rodzina protokołów TCP/IP jest najczęściej stosowana do realizacji usług warstwy sieciowej i transportowej w istniejącej sieci Internet. Oprócz protokołów rodziny TCP/IP podobne funkcje mogą realizować inne protokoły o znacznie już mniejszym znaczeniu wynikającym z faktu, iż są to często protokoły firmowe. Można tu wymienić protokół komunikacyjny DECnet stosowany w sieciach lokalnych do transmisji pomiędzy komputerami produkcji firmy Digital Equipment Corporation; protokół NETBIOS *(ang. Network Basic Input/Output System)* stworzony w roku 1984 przez firmę IBM i wypełniający funkcje warstwy sieciowej, transportowej i sesji; protokół XNS *(ang. Xerox Network System)* będący podstawą protokołów IPX/SPX firmy Novell i zaprojektowany przez firmę Xerox w późnych latach siedemdziesiątych czy też protokoły stosu protokołów rozproszonego systemu operacyjnego Banyan VINES *(ang. Banyan Virtual Integrated Network Service)* firmy Banyan.

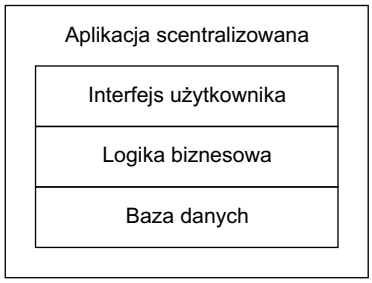
Omawiane protokoły zapewniają komunikację z odległymi komputerami w sposób niezawodny (warstwa transportowa), o ile istnieje między nimi jakiekolwiek połączenie fizyczne (warstwa sieciowa). W istocie protokoły te stanowią podwaliny do budowy rozproszonych systemów, umożliwiając im komunikację w środowisku rozproszonych maszyn.

2.3 Architektury systemów informatycznych

Istotny wpływ na architektury tworzonych obecnie systemów informatycznych miało powstanie i rozwój sieci komputerowych. Wskutek tego powstające obecnie oprogramowanie jest w większości ukierunkowane na obsługę i wykorzystanie sieci. W naturalny sposób narzuca to pewne schematy na tworzone aplikacje. Analiza konkretnych architektur poprzedzona zostanie wyjaśnieniem kluczowych pojęć mających znaczenie w prowadzonych dalej rozważaniach.

Jak wiadomo działanie aplikacji informatycznych w mniejszym lub większym stopniu polega na przetwarzaniu danych. W odniesieniu do przetwarzania aplikacyjnego stosowane jest pojęcie logiki biznesowej. Logika biznesowa obejmuje mecha-

[3]Nowa specyfikacja protokołu IP w wersji szóstej, tzw. IPv6, przewiduje adresy internetowe zapisywane już na 128 bitach, co znaczenie zwiększy przestrzeń adresową Internetu.



Rysunek 2.2: Struktura aplikacji scentralizowanej

nizmy i elementy systemu informatycznego, których rola polega na przetwarzaniu danych i organizowaniu właściwej strategii działania. Logika biznesowa nie obejmuje swoim zasięgiem zarządzania bazą danych i interakcji z użytkownikiem systemu. Pojęcie logiki biznesowej jest tu o tyle istotne, gdyż miejsce jej realizacji w systemach komputerowych prowadzi do ich podziału omawianego niżej.

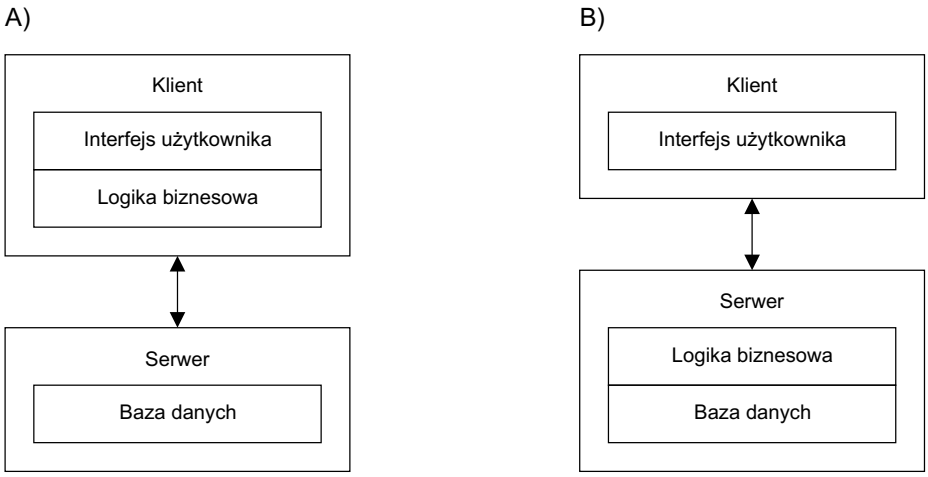
Poruszane zagadnienia wymagają jeszcze wyjaśnienia pojęcia komponentu. W literaturze można znaleźć wiele prób formalnego sprecyzowania znaczenia tego terminu. Interesującą propozycję podano w [8, str. 42] *"Komponent to fizyczna, wymienna część systemu, która wykorzystuje i realizuje pewien zbiór interfejsów"*. W [44, str. 38] zwrócono uwagę na fakt, że: *"komponent oprogramowania jest jednostką montażową"*. Uwzględniając przytoczone wyżej sformułowanie, używane w pracy pojęcie komponentu będzie dotyczyło programowych jednostek montażowych realizujących pewne zbiory interfejsów, z których budowane są systemy informatyczne.

2.3.1 Architektura autonomiczna

Cechą charakterystyczną tej architektury jest to, że całe przetwarzanie skupione jest w ramach jednej aplikacji, która, jak każda dobrze zaprojektowana z punktu widzenia inżynierii oprogramowania [24], może mieć wewnętrzną strukturę modułową i składać się z wielu jednostek montażowych w postaci różnego rodzaju komponentów. Wszystkie one stanowią jedną, nierozerwalnie powiązaną całość. Aplikacja dostarcza mechanizmów interakcji z użytkownikiem, dostępu do danych i ich przechowywania oraz realizuje zadania logiki biznesowej (rys. 2.2).

Zaletą tego modelu jest prostota polegająca na skupieniu całego przetwarzania w jednym miejscu i eliminacja potrzeby synchronizacji dostępu. Konieczność dostarczenia wszystkich mechanizmów obsługi powoduje jednak, że aplikacje tego typu są z reguły znacznie skomplikowane pod względem budowy wewnętrznej.

Omawiana architektura oprócz samodzielnych aplikacji pisanych obecnie głównie z myślą o komputerach osobistych, choć nie tylko, dotyczy również scentralizowanych systemów pracujących na maszynach mainframe. W tego typu systemach wszystkie dane i aplikacje znajdują się na jednym komputerze. Dostęp do nich uzyskuje się z wykorzystaniem terminali.



Rysunek 2.3: Struktura aplikacji klient-serwer z grubym A) i cienkim B) klientem

2.3.2 Architektura klient - serwer

Model klient - serwer jest ważnym czynnikiem kształtującym strukturę nie tylko systemów rozproszonych, ale ogólnego pojęcia systemów informatycznych. U podstaw tego modelu leży wyeksponowanie w budowie systemu zasobów[4] zwanych serwerami, oferujących usługi użytkownikom nazywanym klientami. Istotą modelu klient - serwer jest istnienie zasobów świadczących usługi oraz klientów korzystających z tych usług, przy ważnym założeniu, że dostawcy usług są elementami biernymi, tzn. świadczą usługi jedynie na żądanie klientów.

Architektura klient - serwer nazywana jest architekturą dwuwarstwową. Jedną z warstw stanowi jeden bądź więcej serwerów, zaś drugą warstwę tworzą aplikacje klienckie. Taki podział umożliwia tworzenie systemów informatycznych o właściwościach zależnych od tego, która warstwa będzie odpowiadać za przechowywanie danych, która za logikę biznesową, a która za dostarczenie mechanizmów interfejsu użytkownika. W praktyce stosowane podejścia są zróżnicowane względem umiejscowienia logiki biznesowej. Prowadzi to do dwóch koncepcji strony klienta w architekturze klient - serwer. Wyróżnia się tu pojęcia cienkiego i grubego klienta, które zilustrowano na rys. 2.3. Cienki klient jest rodzajem architektury aplikacji klienckiej, której rola polega jedynie na zapewnieniu interakcji z użytkownikiem. Gruby klient oprócz mechanizmów kontaktu z użytkownikiem zawiera również elementy logiki biznesowej lub nawet całą logikę biznesową.

Obie koncepcje mają swoje wady i zalety. Zastosowanie grubego klienta stwarza niedogodności w przypadku aktualizacji systemu. Wprowadzanie zmian w logice przetwarzania pociąga za sobą konieczność wymiany aplikacji klienckich na wszyst-

[4]Zastosowane tu określenie zasobu ma szerokie znaczenie zasobów systemów komputerowych, gdyż, w odniesieniu do architektury klient - serwer, kooperować mogą procesy, wątki, a nawet na innym poziomie uogólnienia także komponenty programistyczne czy nawet całe komputery.

kich stanowiskach, na których pracują - najczęściej bez możliwości równoległego funkcjonowania starej i nowej wersji. Systemy z grubym klientem mają także znacznie większe wymagania sprzętowe, gdyż logika biznesowa znajduje się po stronie klienta, który jest przez to znacznie rozbudowany. Nie bez ograniczeń jest również rozwiązanie z cienkim klientem. Skupienie logiki biznesowej po stronie serwera podnosi jego wymagania sprzętowe. Ponieważ klient w tym przypadku pełni rolę interfejsu użytkownika, konieczne są częste odwołania do serwera, co może prowadzić do znacznego obciążenia sieci. Wybór odpowiedniej architektury zależy od zastosowania i powinien odbywać się w oparciu o starannie przeprowadzoną analizę.

2.3.3 Architektura wielowarstwowa

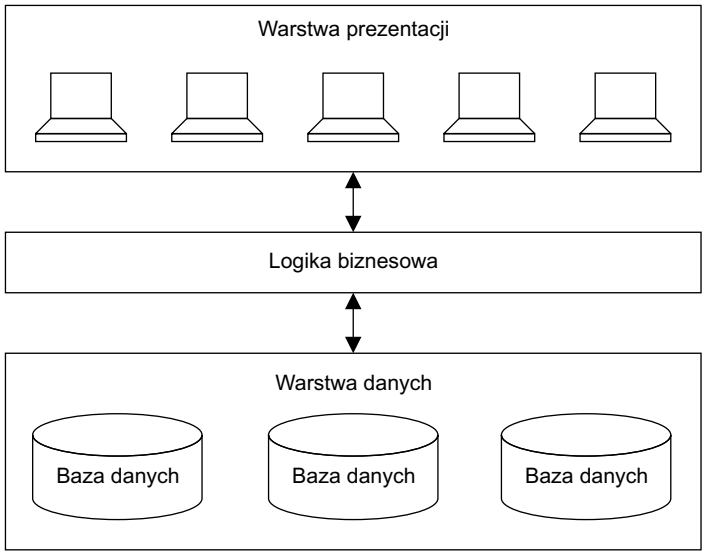
Dziel i rządź *(łac. Divide et Imperia)* - owa starożytna, ciągle niestety aktualna mądrość polityczna, jest wykorzystywana do rozwiązywania problemów programistycznych. Opiera się ona na założeniu, że złożony problem da się podzielić na mniejsze podproblemy, z których każdy będzie znacznie łatwiejszy do rozwiązania. Pozwala to nie tylko łatwiej budować złożone systemy, ale także nimi zarządzać i konserwować.

Aplikacja w architekturze wielowarstwowej składa się z co najmniej trzech warstw oraz punktów dostępu do warstw w postaci dobrze zdefiniowanych interfejsów między nimi. Zwykle warstwy składają się z jednego lub kilku komponentów programistycznych, których zastosowanie daje możliwość realizacji modułowości oprogramowania w standardzie binarnym. Podział na warstwy jest bardzo wydajnym sposobem nie tylko projektowania i wytwarzania oprogramowania, ale także, co jest szczególnie ważne, jego konserwacji, gdyż każda warstwa odpowiada za oddzielne zadanie wykonywane przez aplikację [48, str. 280 i nast.].

W przypadku najczęściej występującej architektury trójwarstwowej wyróżniane są następujące warstwy (rys. 2.4):

- Warstwa danych, która służy do przechowywania danych we wchodzących w jej skład bazach danych.
- Warstwa logiczna, która zawiera komponenty logiki biznesowej.
- Warstwa prezentacji, która stanowi interfejs użytkownika.

Architektura wielowarstwowa jest doskonale adaptowana przez systemy rozproszone, szczególnie w środowisku internetowym, gdzie rozdzielenie logiki biznesowej od warstwy prezentacji siłą rzeczy prowadzi do rozwiązania cienkiego klienta. Środowisko internetowe zapewnia ku temu powszechnie używane i sprawdzone mechanizmy. Dotychczas masowo wykorzystuje się tu język HTML *(ang. Hyper Text Markup Language)* ze wsparciem elementów dynamicznych takich jak DHTML *(ang. Dynamic HTML)*, aplety języka Java czy komponenety ActiveX. Logika biznesowa jest zlokalizowana na serwerze, wskutek tego strategia aplikacji jest łatwa w modyfikacji i zawsze aktualna dla wszystkich klientów. Umożliwia to projektantom opóźnienie decyzji, gdzie i jakie usługi powinny być realizowane oraz pozwala na przyrostowe



Rysunek 2.4: Ogólna struktura aplikacji internetowej w architekturze trójwarstwowej

dodawanie funkcjonalności do systemu. Architektura ta wykazuje również liczne korzyści związane z obsługą i dostępem do baz danych. Wymiana bazy danych czy też zmiana jej schematu jest znacznie ułatwiona, ponieważ tylko warstwa logiki biznesowej korzysta z warstwy danych. O ewentualnej zmianie motoru bazy danych i związanych z tym sterowników strona kliencka może nawet nie mieć pojęcia, a całość może odbywać się bez przerywania pracy aplikacji klienckich. Cecha ta jest istotna również ze względów bezpieczeństwa. Bezpieczeństwo może być również podniesione przez zabezpieczenie części systemu, np. warstwy danych, ścianą ogniową.

Systemy informatyczne w architekturze trójwarstwowej mogą stanowić rozwiązanie problemu przenośności i wieloplatformowości. Zastosowanie cienkiego klienta w postaci przeglądarki WWW, łączącego się z warstwą logiki biznesowej przez protokół HTTP *(ang. Hyper Text Transfer Protocol)*, pozwala rozpowszechniać system na dowolną platformę. Dodatkowym atutem są możliwości integracji istniejącego oprogramowania, którego likwidacja jest często ekonomicznie nieuzasadniona.

Z powstaniem architektury trójwarstwowej wiąże się pojęcie serwera aplikacji. Serwery aplikacji są w tym ujęciu wyróżnionymi w oprogramowaniu jednostkami, których zadaniem jest zagwarantowanie środowiska uruchomieniowego komponentom warstwy logiki biznesowej. Ponieważ warstwa ta jest najbardziej skomplikowanym elementem systemu informatycznego w architekturze trójwarstwowej, komponenty wchodzące w jej skład są często uruchamiane w środowisku serwera aplikacji. Serwery aplikacji projektowane są jako środowiska wykonawcze dla wybranych technologii komponentowych. Jako najbardziej znane technologie tego typu można wymienić COM+ *(ang. Component Object Model)* i EJB *(ang. Enterprise Java Beans)*. Serwery aplikacji dostarczają dodatkowych mechanizmów kontroli przetwarzania, do

których należy aktywacja w chwili żądania *(ang. Just In Time Activation)* polegająca na tym, że zasób jest wznawiany w chwili wystąpienia żądania klienta. Kiedy klient przez określony czas nie wywołuje metod obiektu, taki obiekt jest niezauważalnie zwalniany przez serwer aplikacji. Gwarantuje to oszczędność zasobów po stronie serwera, kiedy z usług jego komponentów korzystają naraz setki klientów. W ramach usług serwerów aplikacji można tu wymienić również przechowywanie puli obiektów do ponownego wykorzystania (np. otwartych połączeń do bazy danych) czy też możliwość realizacji przetwarzania transakcyjnego w środowisku rozproszonych maszyn [13, str. 18 i nast.].

2.4 Techniki komunikacji sieciowej między procesami

Samo istnienie sieci komputerowych i protokołów komunikacyjnych nie gwarantuje jeszcze łatwego ich wykorzystania z poziomu środowisk programistycznych. Do realizacji tego celu wymagany jest interfejs programistyczny. Dopiero wtedy zapewniona zostanie możliwość komunikacji aplikacji znajdujących się nie tylko w różnych przestrzeniach adresowych, ale szczególnie na różnych maszynach.

2.4.1 Interfejs gniazd

Programowanie sieciowe, w szczególności w środowisku internetowym, polega najczęściej na wykorzystywaniu usług warstwy transportowej modelu odniesienia OSI. Aby uczynić je możliwie prostym, został wprowadzony specjalny interfejs programistyczny służący do komunikowania się z warstwą transportową i warstwami niższymi. Na interfejs ten składają się tzw. gniazda i funkcje do operowania na nich. Termin gniazdo pojawia się w kilku różnych kontekstach. W [42, str. 69] podano np., że *"Adres IP oraz numer portu, są często nazywane gniazdem"*. W kontekście aplikacyjnych interfejsów programistycznych *(ang. Application Programming Interface - API)* nazywa się je interfejsem gniazdowym, interfejsem gniazd lub po prostu gniazdami. W [42, str. 29] używa się też *"terminu gniazd TCP jako synonimu pojęcia punkt końcowy protokołu TCP"*. W prowadzonych w pracy rozważaniach gniazda rozpatrywane są ze szczególnym uwzględnieniem ich roli jako interfejsu programistycznego.

Interfejs gniazd powstał dla systemu UNIX BSD *(ang. Berkeley Software Distribution)* i jest prostym rozwinięciem uniksowego modelu dostępu do plików. Identyfikator gniazda jest traktowany jako deskryptor pliku niskiego poziomu, dlatego dla pewnych operacji można używać takich samych funkcji jak do operacji na plikach. W interfejsie gniazd systemu Windows nie ma takiej analogii gniazdo - plik jak w systemie UNIX. Problem pochodzi z Windows 3.1, w którym za obsługę plików odpowiada system DOS. W systemie DOS nie ma zaimplementowanej obsługi gniazd. Dopiero programy korzystające z interfejsu programistycznego Win32 mogą trakto-

wać gniazda jak zwykłe uchwyty[5], co pozwala zastosować do nich systemowe funkcje (`CreateFile, FileRead, FileWrite, CloseHandle`) obsługi plików [26, str. 138] [25].

Standardowe interfejsy gniazd udostępniane bezpośrednio przez interfejsy programistyczne systemów operacyjnych są często obudowywane bibliotekami obiektowymi. Tego typu implementacje to między innymi pakiet `java.net` dla języka Java, zestawy klas dla środowisk Borland Delphi i Borland C++ oraz klas MFC *(ang. Microsoft Foundation Classes)* Winsock. Wykorzystanie tych narzędzi sprowadza pracę z gniazdami do poziomu programowania obiektowego.

Gniazda są podstawowym mechanizmem komunikacji w sieci komputerowej i zapewniają elastyczną, wystarczającą w ogólnych przypadkach komunikację. Praca z gniazdami sprowadza się do przesyłania ciągów bajtów między nimi. Znaczenie przesyłanej informacji musi być precyzyjnie uzgodnione między komunikującymi się stronami. Wykorzystanie gniazd jest proste i szybkie jedynie w przypadku istniejących protokołów warstw wyższych i implementujących je aplikacji. W przypadku technologii tworzenia systemów rozproszonych gniazda mają wiele ograniczeń. Wymagają one sprzęgu między klientem i serwerem na poziomie aplikacji, reguł określających kodowanie i dekodowanie wymienianych komunikatów, obsługi błędów, nie mówiąc już o skalowalności czy odzyskiwaniu danych po awariach. W wyniku usystematyzowania tych problemów i ich rozwiązań powstawały nowe technologie, czasem ustanawiano protokoły warstw wyższych umożliwiające efektywniejsze techniki realizacji systemów rozproszonych, ale u swoich podstaw mające często gniazda, które dają możliwość niezawodnej wymiany danych.

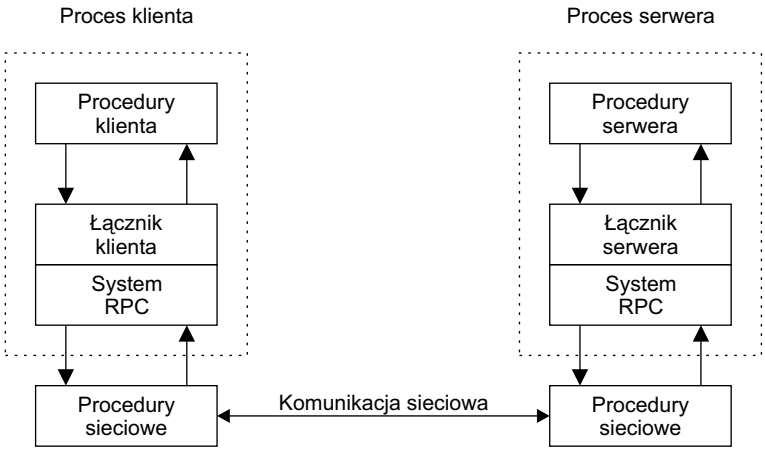
2.4.2 Zdalne wywołanie procedur

Opisywany w poprzednim podpunkcie interfejs gniazd jest w [42, str. 480] zaliczany do metod *"jawnego programowania sieciowego"*, gdzie komunikacja sieciowa nie jest ukryta przed programistą. Alternatywną metodą tworzenia rozproszonych programów użytkowych jest stosowanie *"niejawnego programowania sieciowego"* [42, str. 481]. W tej kategorii w bieżącym podpunkcie omówiony zostanie mechanizm zdalnego wywoływania procedur *(ang. Remote Procedure Call - RPC)* oraz jego obiektowe odpowiedniki w podpunktach kolejnych. Cechą charakteryzującą to podejście jest ukrycie sieciowych operacji wejścia-wyjścia przed programistą dzięki zastosowaniu semantyki wywołań procedur.

Mechanizm działania

Istotą działania RPC w realizacji obliczeń rozproszonych jest umożliwienie programom wykonywania procedur znajdujących się na innych maszynach. Gdy proces

[5]Uchwyt (ang. handler) jest trzydziesto-dwu bitową wartością, która identyfikuje zasoby przydzielane procesom. Identyfikator gniazda w interfejsie Win32 ma właściwie typ HANDLE, lecz dla upodobnienia do biblioteki z systemu UNIX nazwano go SOCKET.



Rysunek 2.5: Zdalne wywołanie procedury przedstawione w znacznym stopniu ogólności [45, str. 487]

na maszynie A wywołuje procedurę na maszynie B, to następuje jego zawieszenie (możliwa jest też praca w trybie asynchronicznym bez blokowania), a wykonanie procedury przebiega na maszynie B. Informacje od procesu wywołującego do wywoływanego są przenoszone przez sieć za pomocą parametrów, a wracają w postaci wyników procedury [45, str. 93]. Za interpretację ciągów bajtów przesyłanych najczęściej z użyciem gniazd między współpracującymi maszynami odpowiadają mechanizmy wewnętrzne RPC. Programista nie ma do czynienia z żadnym przekazywaniem komunikatów ani wejściem-wyjściem.

Problem zdalnego wywołania procedur jest znacznie bardziej skomplikowany niż wywołania wewnątrzprocesowe. Argumenty wywołania funkcji i jej parametry aktualne nie mogą być w standardowy sposób odłożone na stos, jak również wynik nie może być zwrócony tą samą metodą. Muszą być wykorzystane mechanizmy pośrednie w postaci łączników, które zapewnią komunikację sieciową. Wyjaśnienia wymaga tu pojęcie łącznika. Terminem tym określa się funkcje generowane automatycznie przez system obsługi RPC i dołączane do programów klienta i serwera RPC. Ich zadaniem jest zapewnienie połączenia w sensie składni i semantyki języka programowania między kodem stworzonym przez programistę a systemem RPC. Architekturę mechanizmu zdalnych wywołań przedstawia rys. 2.5.

W przypadku zdalnego wywołania procedur kluczową rolę odgrywa łącznik. Kiedy klient wywołuje procedurę zdalną, jego działania lokalne sprowadzają się do wywołania procedury jego łącznika. Zadaniem łącznika jest upakowanie parametrów w komunikacie wyjściowym i przekazanie żądania wywołania zdalnej procedury do łącznika serwera. Dopiero łącznik serwera po otrzymaniu zlecenia i parametrów wywołania ze strony klienta wywołuje odpowiednią metodę serwera. Po zakończeniu działania wykonywanej procedury na serwerze zwrócone parametry są upakowywane w komunikacie wyjściowym do łącznika klienta. Łącznik klienta po otrzymaniu

odpowiedzi ze strony maszyny serwera przekazuje wyniki do wywołującego modułu klienta. Mechanizm ten jest przezroczysty dla programisty, tzn. identycznie wygląda wywołanie metod lokalnych, jak i zdalnych, przy ukrytej implementacji komunikacji sieciowej, która przy użyciu łączników może być realizowana z wykorzystaniem dowolnych technik komunikacji sieciowej. Może to być zarówno bezpośrednie stosowanie interfejsu gniazd, jak również protokołów warstw wyższych takich jak HTTP.

Rola interfejsu

Łączniki, aby mogły komunikować się między sobą, muszą posiadać wspólny język opisu sposobu komunikacji. Językiem tym jest najczęściej język definiowania interfejsów *(ang. Interface Definition Language - IDL)*. W modelu RPC spoiwem łączącym jest definicja interfejsu. Stanowi ona kontrakt między serwerem a jego klientami, określający usługi oferowane klientom przez serwer. Materialną reprezentacją tego kontraktu jest plik definicji interfejsu pisany często w języku IDL. Na podstawie tego pliku zawierającego definicje zdalnych metod generowany jest kod łączników dla aplikacji RPC. Takie podejście z zastosowaniem punktu odniesienia w postaci pliku definicji interfejsu umożliwia prowadzenie dialogu komunikujących się stron we wspólnym języku.

Zewnętrzna reprezentacja danych

Generowanie łączników klienta i serwera na podstawie pliku definicji interfejsu pomaga rozwiązać problem odmiennych formatów danych. W przypadku, gdy komunikujące się stacje mają odmienne architektury programowe lub sprzętowe dane mogą być kodowane na różne sposoby. Na przykład typy proste w niektórych językach programowania mogą mieć różne rozmiary (np. typ `long` w języku C może mieć 32 lub 64 bity). Odmienne mogą być też uporządkowania bajtów (np. mniejsze wyżej lub mniejsze niżej). Pakiet Sun RPC[6] rozwiązuje ten problem, stosując do opisu danych oraz ich kodowania standard zewnętrznej reprezentacji danych XDR *(ang. External Data Representation)*. Wszystkie dane wymieniane między stacjami w trakcie komunikacji są kodowane do standardu XDR. Zawiera on elementy języka, które są używane w plikach definicji interfejsu do opisu i kodowania danych [42, str. 515]. XDR został wbudowany w pakiet Sun RPC, dzięki czemu łączniki klienta i serwera utworzone na podstawie pliku definicji interfejsu zawierają jego implementację, którą stosują do kodowania przesyłanych przez sieć danych.

Usługi nazewnicze

Definicja interfejsu określa nazwę usługi, za pomocą której klienci mogą się do niej odwoływać. Nazwa ta może mieć format tekstowy, ale niekiedy wykorzystuje się

[6]Sun RPC jest implementacją RPC stworzoną i rozwijaną przez firmę Sun. Pierwsza wersja tego projektu opublikowana została w roku 1985.

UUID[7] *(ang. Universal Unique IDentifier)* - 128 bitowy identyfikator. Nazwy interfejsów umożliwiają klientom odnalezienie identyfikatora komunikacyjnego szukanej usługi na serwerze. Postać identyfikatora komunikacyjnego zależy od środowiska. W systemach wykorzystujących stos protokołów TCP/IP jest to internetowy adres komputera oraz numer portu.

Jedna z metod lokalizacji usługi przez klienta polega na wmontowaniu klientowi na stałe sieciowego adresu serwera i portu programu. Trudność w takim podejściu wynika ze skrajnej nieelastyczności. Przeniesienie serwera, jego zwielokrotnienie lub zmiana interfejsu pociąga za sobą konieczność wyszukania i ponownego skompilowania wielu programów [45, str. 103]. Aby unikać tych problemów, obecnie większość systemów rozproszonych do kontaktowania klientów z serwerami stosuje tzw. wiązanie dynamiczne.

Wiązanie dynamiczne w systemach rozproszonych realizuje się z wykorzystaniem serwera nazw, który jest oddzielną usługą utrzymującą tablicę odwzorowań nazw usług na porty serwerów. Przykładem serwerów nazw są serwery systemu DNS *(ang. Domain Name System)* stosujące hierarchiczną strukturę adresowania odwzorowującą nazwy domen na adresy internetowe.

Metoda eksportowania i importowania interfejsów jest bardzo elastyczna. Można na przykład za jej pomocą obsługiwać wiele serwerów dostarczających takich samych interfejsów. Serwer nazw może wedle własnych potrzeb losowo rozpraszać klientów po serwerach w celu wyrównania obciążeń. Może także okresowo odpytywać serwery, automatycznie wyrejestrowując każdy z nich, który nie nadeśle prawidłowej odpowiedzi. Pomaga to tolerować uszkodzenia. Ponadto serwer nazw może asystować przy sprawdzaniu tożsamości.

Opisana forma dynamicznego wiązania ma również swoje wady. Eksportowanie i importowanie interfejsów zajmuje czas. Ponieważ żywot wielu procesów klientów jest krótkotrwały, a jednocześnie są one ciągle rozpoczynane na nowo, więc nakład pracy związany z odwzorowywaniem nazw interfejsów może okazać się znaczący [45, str. 106].

Standard RPC w wersji Sun RPC stosuje wiązanie dynamiczne polegające na wykorzystaniu programu odwzorowującego identyfikatory interfejsów na odpowiadające im numery portów. Programem tym, w zależności od systemu operacyjnego, jest `portmap` lub `rpcbind` [42, str. 495].

Różnice RPC w porównaniu z wywołaniem lokalnym

Zarówno wewnętrzna struktura RPC, która jest ukryta przed programistą, jak i praktyczne wykorzystanie różnią się znacznie od tradycyjnych, lokalnych wywołań.

[7]Konsorcjum OSF (ang. Open Software Foundation) w specyfikacji DCE (ang. Distributed Computing Environment) opisującej zasady komunikowania się komputerów definiuje identyfikatory UUID do jednoznacznego określania poszczególnych komponentów. Microsoft nazywa je GUID (ang. Globally Unique Identifier), a często także CLSID (Class ID) lub IID (Interface ID), w zależności od zastosowania. Wszystkie te określenia dotyczą UUID. Użycie konkretnego typu zależy od zastosowania.

Podstawowe różnice to:

- Obsługa błędów - konieczność obsługi błędów zdalnego serwera lub sieci.
- Widoczność zmiennych globalnych - ponieważ klient nie ma dostępu do przestrzeni adresowej serwera, nie jest możliwe przekazywanie parametrów przez zmienne globalne.
- Osiągi - zdalne wywołania procedur działają wolniej niż wywołania lokalne.
- Sprawdzanie tożsamości - ponieważ zdalne wywołania procedur mogą być przesyłane w sieciach narażonych na włamanie, sprawdzanie tożsamości może być konieczne.
- Przekazywanie parametrów musi uwzględniać różne architektury sprzętowe i systemowe współpracujących programów.

Choć technika RPC jest krokiem naprzód w porównaniu z "surowym" przekazywaniem komunikatów, nie jest wolna od własnych problemów. Odpowiedni serwer należy zlokalizować. Wskaźniki oraz złożone struktury danych są trudne do przekazywania. Dokładna semantyka RPC nie jest jasna, ponieważ klienci i serwery mogą ulegać awariom niezależnie od siebie. Realizacja wydajnego zdalnego wywołania procedur jest niełatwa.

2.4.3 Usługi kolejkowania komunikatów

Zarówno w omawianym wyżej interfejsie gniazd, zdalnym wywołaniu procedur, jak i większości innych metod zdalnej komunikacji obie strony biorące udział w połączeniu muszą być aktywne w tym samym czasie. Wszystkie komponenty wchodzące w skład rozproszonej aplikacji muszą być aktywne przez cały czas działania lub przynajmniej w momentach odwołań do nich. Pozbycie się tych wymagań zapewniają technologie JMS *(ang. Java Message Service)* oraz MSMQ *(ang. Microsoft Message Queue)*. Są to protokoły pozwalające aplikacjom na wzajemne przesyłanie komunikatów. Podstawową właściwością tych protokołów i jednocześnie różnicą w stosunku do innych metod komunikacji międzyprocesowej jest możliwość wysyłania komunikatu do procesu, który w danej chwili nie jest dostępny, z powodu np. jego wyłączenia lub problemów z siecią. Mechanizmy wykonawcze MSMQ i JMS kolejkują komunikaty wysyłane do nieaktywnych odbiorców, aby dostarczyć je w późniejszym terminie. Dzięki temu komunikacja nie wymaga dostępności obu stron w tym samym czasie. Pozwala to tworzyć systemy rozproszone, w których aplikacja może działać, kiedy nie wszystkie komponenty są aktywne [49, str. 305].

W omawianej wcześniej technologii RPC komunikacja wymaga kontraktu w postaci definicji interfejsu. Technologie MSMQ i JMS eliminują również to wymaganie. Odbiorca i nadawca nie muszą nic o sobie wiedzieć. Jedyną informacją potrzebną do

komunikacji jest format przekazywanych komunikatów i nazwa kolejki (komunikaty przesyłane są przez kolejki komunikatów identyfikowane przez nazwę tekstową) [23].

Na potrzeby aplikacji wymagających dużej pewności poprawnego działania JMS wspiera mechanizm potwierdzeń dostarczenia komunikatu dokładnie raz.

Znaczenie omówionych technologii MSMQ i JMS potwierdza fakt, iż znajdują one wykorzystanie w warstwie logiki biznesowej trójwarstwowego modelu aplikacji. JMS jest integralną częścią platformy J2EE *(ang. Java 2 Platform Enterprise Edition)* umożliwiającą komunikację komponentów EJB i nie tylko. MSMQ jest natomiast wykorzystywana przez technologię COM+ [13, str. 22].

2.4.4 Zdalne wywołanie metod

RPC sprowadziło komunikację sieciową do poziomu wywołania procedur. Aby dopasować semantykę wywoływania metod obiektów do potrzeb rozproszonego przetwarzania, opracowano technikę zwaną zdalnym wywołaniem metod.

Programowanie obiektowe na drodze rozwoju systemów rozproszonych

Obiektowość jest jednym z najnowszych paradygmatów, który spotkał się z bardzo dobrym przyjęciem ze strony przemysłu programistycznego.

W językach obiektowych realizowane są trzy podstawowe idee tego paradygmatu:

- Enkapsulacja - ukrywanie szczegółów implementacyjnych obiektów.
- Dziedziczenie - możliwość wykorzystania istniejących obiektów do tworzenia nowych, bardziej wyspecjalizowanych.
- Polimorfizm - możliwość umieszczenia w kodzie różnych zachowań w zależności od tego, jaki obiekt został użyty [13, str. 7].

Niewątpliwie programowanie obiektowe może stanowić punkt wyjścia do rozważań na temat rozproszonych systemów obiektowych, a przede wszystkim ułatwić zrozumienie mechanizmów nimi rządzących. Z tego powodu dalsze przedstawianie ewolucji systemów rozproszonych warto poprzedzić analizą ograniczeń programowania obiektowego. Najważniejsze z nich to:

- Jawność szczegółów implementacji przed użytkownikiem.
- Konieczność rekompilacji wszystkich modułów korzystających z danej klasy w przypadku zmiany jej implementacji.
- Dziedziczenie zapewnia ponowną używalność na poziomie kodu źródłowego, ale nie na poziomie kodu binarnego.
- Brak możliwości korzystania z obiektów przez inne języki programowania.
- Brak możliwości uruchomienia obiektu w innej przestrzeni adresowej lub na innej maszynie.

Związek zdalnego wywołania metod z ideą komponentu

Ostatnie lata przyniosły dynamiczny rozwój technologii komponentowych. Zaliczyć można do nich również omawiane niżej technologie zdalnego wywołania metod. Warto na wstępie jednak określić ich związek z pojęciem komponentu. Jak podaje [44, str. 38] *"Komponent oprogramowania jest jednostką montażową, której interfejsy są określone w drodze umów i której kontekstowe zależności są wyłącznie jawne. Komponent oprogramowania może być instalowany niezależnie i podlega zestawieniu przez osoby postronne"*. Powyższe stwierdzenie nie porusza aspektu rozproszenia, gdyż komponenty mogą, lecz nie muszą być rozproszone, ale często dzięki rozproszeniu łatwiej zapewnić możliwość ich niezależnego budowania i montowania. W myśl tego stwierdzenia można przyjąć, że zarówno RPC, jak i technologie zdalnego wywołania metod są technologiami komponentowymi, gdyż ich elementy są binarnymi jednostkami montażowymi, które, dysponując kontraktem w postaci interfejsu, mogą wykorzystywać osoby postronne. Daje to możliwość budowania systemów informatycznych z binarnych komponentów rozproszonych w sieci komputerowej.

COM/DCOM

Component Object Model jest specyfikacją opisującą współpracę obiektów i ich klientów poprzez interfejsy w binarnym standardzie. Jest też implementacją zwaną biblioteką COM. Implementacja ta jest dostarczona przez biblioteki (DLL w Microsoft Windows), które zwierają:

- Zbiór podstawowych funkcji API umożliwiających tworzenie aplikacji COM, zarówno klientów jak i serwerów - dla klientów dostarcza podstawowych funkcji tworzenia obiektów, dla serwerów możliwości udostępniania tych obiektów.
- Implementacje lokalnych usług, dzięki którym COM określa na podstawie identyfikatora klasy, jaki serwer implementuje tą klasę i gdzie ten serwer jest zlokalizowany.
- Przezroczyste wywoływanie procedur, gdy obiekt pracuje w lokalnym lub odległym serwerze, włączając w to implementację protokołu sieciowego.
- Mechanizm pozwalający aplikacji na kontrolę tego, jak pamięć jest alokowana w procesie.

COM jest również metodą pozwalającą na komunikację między komponentami programowymi. DCOM *(ang. Distributed COM)* będący rozszerzeniem COM jest binarnym i sieciowym standardem, który umożliwia komunikację przez sieć.

CORBA

Common Object Request Broker Architecture jest specyfikacją stworzoną w 1990 roku przez Object Management Group[8], która definiuje magistralę obiektową, umożliwiając integrację i zarządzanie obiektami zdefiniowanymi w większości obiektowych języków programowania. Środowisko CORBA zapewnia model niejawnego programowania sieciowego. Umożliwia wywoływanie metod zdalnych obiektów rezydujących w sieci w taki sposób, jakby były one obiektami lokalnymi. Standard CORBA nie dotyczy żadnej wyszczególnionej implementacji. Definiuje tylko specyfikację, na podstawie której tworzone są z założenia kompatybilne implementacje standardu.

Java/RMI

Remote Method Invocation jest rozproszoną technologią obiektową, która rozszerza model języka Java o możliwość wywoływania metod obiektów przez sieć. RMI pozwala obiektom z jednej JVM *(ang. Java Virtual Machine)* na przezroczyste wywoływanie metod obiektów z innej, zlokalizowanej na innym komputerze maszyny wirtualnej języka Java. Jest to homogeniczne środowisko do budowania rozproszonych aplikacji. Zostało opracowane do działania tylko w środowisku języka Java (nowe implementacje mogą też współpracować z obiektami CORBA pisanymi w innych językach). Dzięki użyciu jednego języka, RMI może w pełni wykorzystywać jego cechy i zalety, takie jak mechanizm wyjątków i szeregowania obiektów [6].

Analiza porównawcza technologii COM/DCOM, CORBA i Java/RMI

Wymienione wyżej technologie rozszerzają model programowania obiektowego o pojęcie rozproszonych obiektów, tzn. pozwalają na wywoływanie metod obiektów znajdujących się na innych maszynach. Odmienna realizacja tego celu przez twórców trzech porównywanych koncepcji prowadzi do różnic zarówno z punktu widzenia architektonicznego, jak i programistycznego.

Omawiane technologie, jako standardy sieciowe, definiują w warstwie aplikacyjnej własne protokoły komunikacji sieciowej. W modelu DCOM stosowane jest rozszerzenie RPC, zwane Object RPC, które rozszerza standardowe mechanizmy RPC o możliwość odwołań oraz reprezentację referencji do zdalnych obiektów. Dla środowiska CORBA zdefiniowano dedykowany protokół IIOP *(ang. Internet Inter Object Request Broker Protocol)*. Również środowisko Java/RMI posiada własny protokół JRMP *(ang. Java Remote Method Protocol)*. Zastosowanie specjalizowanych protokołów dla każdej z omawianych technologii praktycznie uniemożliwia ich współpracę na poziomie protokołów komunikacyjnych.

Idea realizacji uzdalniania obiektów jest w trzech omawianych modelach podobna, ponieważ analogicznie jak w RPC po stronie klienta i serwera obecne są łącz-

[8]Ponad 800 czołowych firm zajmujących się oprogramowaniem, utworzyło organizację o nazwie Object Management Group, która przyjęła za cel ustalenie infrastruktury dla oprogramowania obiektowego.

niki. W celu wywołania metody zdalnej, klient odwołuje się do lokalnego łącznika. Dopiero łącznik przy pomocy odpowiedniego protokołu sieciowego kontaktuje się z łącznikiem serwera. Łącznik serwera ma kontakt z właściwym obiektem.

Z punktu widzenia programistycznego w omawianych technologiach do współpracy pomiędzy klientem a serwerem potrzebny jest kontrakt w postaci definicji interfejsu. Zarówno DCOM jak i CORBA nie zależą od języka programowania, definiując interfejsy w języku IDL. Zdefiniowany w tym języku interfejs może być przy użyciu odpowiednich narzędzi przekształcony na kody źródłowe stosownych języków (dotyczy to głównie łączników) służące do implementacji zarówno serwerów, jak i klientów. W modelu DCOM zastosowano jeszcze pewne rozszerzenie, gdzie kod języka IDL może być skompilowany do jego binarnej wersji zwanej biblioteką typów. Biblioteka typów może być w automatyczny sposób wykorzystywana przez różne środowiska programistyczne, umożliwiając korzystanie z komponentów COM i DCOM. Java/RMI jako środowisko homogeniczne pozwala definiować interfejs bezpośrednio i tylko w języku Java.

Możliwości języków definiowania interfejsów poszczególnych środowisk są różne. Zarówno CORBA jak i Java/RMI umożliwiają wielokrotne dziedzicznie na poziomie interfejsu. Z kolei DCOM umożliwia implementację wielu interfejsów przez jeden obiekt. Do nawigacji pomiędzy interfejsami w ramach jednego obiektu służy wówczas metoda `QueryInterface`. Inna istotna różnica polega na podejściu do obsługi błędów. CORBA i Java/RMI pozwalają na obiektową obsługę błędów poprzez mechanizm wyjątków przesyłanych przez sieć. W modelu DCOM obsługa zarówno błędów systemowych, jak i błędów metod obiektów użytkowych odbywa się przez zwracane przez te metody wartości typu `HRESULT`. Jest to 32-bitowa wartość wskazująca powodzenie lub porażkę wraz z kodem określającym ich przyczynę.

Po stronie klienta obiekty reprezentowane są przez referencje do ich interfejsów. Aplikacja kliencka posiadająca referencję do interfejsu może wywoływać jego metody zdalne w taki sposób, jakby były one metodami obiektów lokalnych. Ponieważ w modelu DCOM jeden obiekt może implementować kilka interfejsów, klient przy użyciu systemowej funkcji COM `QueryInterface` może przełączać się między interfejsami danego obiektu COM.

Implementacja komponentu serwera w językach obiektowych polega na wyprowadzeniu klasy pochodnej z interfejsu (w języku C++ interfejs jest reprezentowany przez klasę abstrakcyjną) i zaimplementowaniu jego metod. Ponieważ po stronie klienta znany jest tylko interfejs, na zasadzie polimorfizmu programowania obiektowego, klient uzyskuje dostęp do wewnętrznej implementacji obiektu. Ukazuje to duży związek rozproszonych metod obiektowych z klasyczną koncepcją obiektów.

Porównanie omawianych technologii zostało dodatkowo przedstawione w tab. 2.1.

Cecha	CORBA	DCOM	Java/RMI
Protokoły komunikacji sieciowej	IIOP	ORPC	JRMP
Metody lokalizacji implementacji obiektów	ORB (Object Request Broker)	SCM (Service Control Manager)	JVM (Java Virtual Machine)
Metody aktywacji obiektów zdalnych.	Object Adapter	SCM (Service Control Manager)	JVM (Java Virtual Machine)
Unikatowe identyfikatory interfejsów	Nazwa interfejsu	128 bitowy (UUID) identyfikator interfejsu	Nazwa interfejsu
Realizacja przekazywania parametrów przez wartość i referencję	Typy zdefiniowane jako interfejsy - przez referencję; pozostałe typy - przez wartość	Język IDL pozwala na wybór sposobu	Obiekty implementujące interfejs Remote - przez referencję. Obiekty implementujące interfejs Seriazable i typy proste - przez wartość
Usługi katalogowe	Implementation Repository	Rejestr systemowy	RMIRegistry
Niezależność od języka programowania	Tak	Tak	Nie, tylko Java
Odśmiecanie pamięci (ang. garbage collection)	Nie	Tak	Tak
Implementacja	Każdy obiekt dziedziczy z `CORBA.object.`	Każdy obiekt implementuje `IUnknown`	Każdy obiekt implementuje `java.rmi.Remote`
Wielokrotne dziedziczenie na poziomie interfejsu	Tak	Nie	Tak
Implementacja wielu interfejsów przez obiekt	Nie	Tak; nawigacja między interfejsami przy użyciu metody `QueryInterface`	Nie, ale można zastosować dziedziczenie po wielu interfejsach [44, str. 206]
Język definiowania interfejsu	IDL	IDL	Bezpośrednio i tylko język Java
Obsługa błędów	Wyjątki przesyłane przez sieć	Kod błędu typu `HRESULT` zwracany przez metody systemowe i użytkowe	Wyjątki przesyłane przez sieć

Tabela 2.1: Porównanie technologii CORBA, DCOM i Java/RMI [20]

Omówione wyżej technologie są doskonałymi przykładami technologii komponentowych. Umożliwiają budowę aplikacji z rozproszonych w sieci komponentów binarnych. Jest to bardzo istotna właściwość, gdyż prowadzi do wyeliminowania konieczności rekompilacji wszystkich modułów w przypadku zmiany kodu źródłowego w jednym z nich.

2.4.5 Simple Object Access Protocol

Z dokonanych wcześniej analiz wynika, że podstawowym z punktu widzenia programistycznego mechanizmem tworzenia systemów rozproszonych jest interfejs gniazd. W ogólnych przypadkach zapewnia on wystarczającą komunikację wymagającą niestety interpretacji przesyłanych ciągów bajtów. Niemniej jednak gniazda są podstawowym mechanizmem niezawodnej komunikacji sieciowej, który stanowi grunt do budowy bardziej złożonych architektur programistycznych. Pewne ułatwienie programowania rozproszonego stwarza koncepcja RPC, która sprowadza komunikację sieciową do poziomu wywołania procedur. W dobie rosnącej popularności języków obiektowych zrodziła się potrzeba komunikacji na poziomie obiektów programowych rezydujących na różnych maszynach. Było to bodźcem do rozszerzenia zdalnego wywołania procedur do poziomu zdalnego wywołania metod obiektów. Dodatkowy wpływ rozwijającej się koncepcji komponentu doprowadził do powstania kilku obszernych specyfikacji magistral obiektowych. Dotyczy to omawianych wyżej technologii COM/DCOM, CORBA i Java/RMI. Koncepcje te mają jednak pewne wady ograniczające niekiedy możliwości ich wykorzystania. Do uruchomienia aplikacji użytkowych wymagają często dodatkowych środowisk wykonawczych. Wprawdzie w przypadku Java/RMI konieczność użycia maszyny wirtualnej Javy wynika z charakteru języka, jednak wymagają one najczęściej dodatkowego serwera usług katalogowych. DCOM posiada zintegrowane z systemem Windows środowisko wykonawcze, które w przypadku przeniesienia na inną platformę również musi być przeniesione. Integracja tych technologii też jest praktycznie niemożliwa, a przeniesienie na inną platformę czy wykorzystanie innych języków programowania wymaga często dodatkowego oprogramowania. Poważnym ograniczeniem ich zastosowania w obliczu pojawiających się wymagań i dążeń działania Internetu jest też brak przystosowania do działania w sieciach rozległych, gdzie barierą okazują się być ściany ogniowe *(ang. firewalls)*.

Koncepcja Web Services

Coraz częściej pojawiają się koncepcje uczynienia z Internetu zintegrowanego środowiska bez szwów z jednoczesnym podnoszeniem poziomu zabezpieczeń w dostępie do sieci lokalnych. Zastosowania biznesowe wymagają dostępu do danych bez względu na położenie, rodzaj sprzętu, platformę czy aplikację.

Całkowicie nowe, ale zarazem długo oczekiwane możliwości wykorzystania w globalnej sieci Internet wykazuje koncepcja Web Services, która polega na udostępnianiu programistycznych interfejsów w sieci WWW. Specyfikuje ona protokół oparty o język XML, który aplikacje sieciowe mogą wykorzystać do komunikacji między sobą.

Język XML, jak wskazuje jego nazwa, jest rozszerzalnym językiem znaczników. Rozszerzalnym, ponieważ w przeciwieństwie do języka HTML, w którym znaczniki są zdefiniowane, można tworzyć własne według określonych potrzeb. W efekcie język XML stwarza możliwość tworzenia na jego podstawie nowych, bardziej specjalizowanych języków. Należy do nich język HTML, który jest szczególnym przypadkiem języka XML.

Pojawienie się języka XML ułatwiło wymianę danych pomiędzy systemami pracującymi w różnych środowiskach, ponieważ dokumenty XML są plikami tekstowymi, które mogą być przetwarzane na dowolnych platformach. Uniwersalność tego języka stwarza atrakcyjny sposób przekazywania informacji pomiędzy systemami. Programiści mogą używać różnych systemów operacyjnych, różnych języków, a możliwość współdziałania takich systemów zostanie zagwarantowana. W podobny sposób jak niegdyś udostępniono w sieci WWW dane w formacie HTML, tak obecnie programy mogą wymieniać dane XML, między innymi przez interfejsy tworzone zgodnie z protokołem SOAP *(ang. Simple Object Access Protocol)*. SOAP jest opartym na języku XML protokołem ciągle jeszcze opracowywanym przez XML Protocol Working Group[9]. Jego celem jest zapewnienie możliwości komunikowania się aplikacji Web Services. SOAP definiuje format jednokierunkowych wiadomości, które w szczególnym przypadku są użyteczne w realizacji zdalnego wywołania procedur w postaci dialogu zamówienie - odpowiedź. SOAP nie jest związany z żadnym protokołem transportowym, chociaż HTTP jest najbardziej popularny. Nie jest również związany z żadnym szczególnym systemem operacyjnym czy językiem programowania, dzięki temu klient i serwer takiego dialogu może teoretycznie pracować na dowolnej platformie i może być napisany w dowolnym języku. Wystarczy możliwość zinterpretowania komunikatów SOAP zapisanych w języku XML.

Przykładowy dialog przebiegu zdalnego wywołania procedury w oparciu o protokół SOAP ilustrują rys. 2.6 i rys. 2.7. Na rys. 2.6 zobrazowano zapytanie SOAP, które w tym przypadku niesie informację wywołania zdalnej procedury o nazwie `GetLastTradePrice` z parametrem `DEF`. Rys. 2.7 obrazuje odpowiedź niosącą wynik wywołania funkcji `GetLastTradePrice`, którego wartość wynosi `34.5`. Komunikaty SOAP są obecnie przenoszone najczęściej przez protokół HTTP. Może on być w naturalny sposób zaadoptowany do przenoszenia wiadomości SOAP, a przez co może stanowić warstwę transportową zdalnego wywołania procedur.

Protokół SOAP tworzy standard wymiany informacji między komunikującymi się stronami, w którym szczegóły interpretacji komunikatów opisuje WSDL *(ang. Web Services Definition Language)*. WSDL również jest opartą na języku XML specyfikacją, która służy do opisu usług Web Services dostarczanych przez protokół

[9]XML Protocol Working Group jest grupą powołaną przez organizację World Wide Web Consortium - W3C.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
```

Nagłówek HTTP

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DEF</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Komunikat SOAP

Rysunek 2.6: Zapytanie SOAP przenoszone w komunikacie HTTP

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
```

Nagłówek HTTP

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Komunikat SOAP

Rysunek 2.7: Odpowiedź SOAP przenoszona w komunikacie HTTP

SOAP. WSDL pozwala opisywać usługę, a SOAP tworzyć tę usługę. Takie podejście pozwala na dynamiczne tworzenie aplikacji z rozproszonych komponentów Web Services.

WSDL pozwala opisywać pojedynczą usługę, co sprowadza się do opisu interfejsu SOAP. Twórcy Web Services zauważyli potrzebę łączenia kilku prostych usług w celu otrzymania złożonych wyników. Powołano kolejną grupę roboczą, która ma zaprojektować język do komponowania i opisywania powiązań między Web Services.

Programowanie zastosowań SOAP

Chociaż protokół SOAP bazuje na języku XML, to jego wykorzystanie nie wymaga z reguły operowania przez programistę na komunikatach XML. Istniejące obecnie narzędzia zwalniają tworzącego aplikacje z jakiegokolwiek kontaktu z językiem XML, a komunikację sieciową sprowadzają do zdalnego wywołania procedur lub metod obiektów. Oczywiście możliwie jest też użycie protokołu SOAP, przetwarzając komunikaty

XML, takie np., które przedstawiono na rys. 2.6 i rys. 2.7. Jest to metoda, która nie wymaga żadnego dodatkowego oprogramowania, ale jest zarazem najbardziej pracochłonna dla programisty. Niżej omówione zostaną trzy metody programowania SOAP.

Plik definicji interfejsu Jedna z koncepcji realizacji SOAP zakłada wykorzystanie pliku definicji interfejsu, który w tym przypadku pełni identyczną rolę jak w omawianych wcześniej RPC, CORBA, etc. Stanowi on kontrakt pomiędzy klientem a serwerem umożliwiający ich współpracę. Na jego podstawie tworzone są funkcje odpowiedzialne za kodowanie i dekodowanie komunikatów SOAP. Przykładową realizację tej koncepcji reprezentuje pakiet gSOAP [15]. Pakiet ten jest przeznaczony do wykorzystania w środowiskach języków C i C++, dlatego plik definicji interfejsu tworzony jest w języku C. Przykładowa definicja interfejsu SOAP może mieć w tym przypadku postać:

```
int ns1__getQuote(char *symbol, float &Result);
```

Na podstawie pliku definicji interfejsu program o nazwie `soapcpp2` generuje kody funkcji, które po skompilowaniu mogą być konsolidowane z aplikacjami klienta i serwera. Funkcje te zapewniają kodowanie i dekodowanie komunikatów SOAP. Strona serwera wymaga jedynie zaimplementowania funkcji określonych definicją interfejsu. Przykładowa aplikacja strony klienta wywołująca zadeklarowaną wyżej funkcję może mieć postać:

```
#include "soapH.h"
int main(){
    struct soap soap;
    float quote;
    soap_init(&soap);
    if(soap_call_ns1__getQuote(&soap,
        "http://services.net:80/soap","","AOL",quote)== SOAP_OK)
        cout << "Current AOL Stock Quote = " << quote;
    else soap_print_fault(&soap, stderr);
    soap_end(&soap);
}
```

Bezpośrednie tworzenie serwera Opisany wyżej sposób programowania SOAP wyraźnie rozdziela fazę tworzenia interfejsu od fazy implementacji serwera. Istnieją środowiska programistyczne, które pozwalają na bezpośrednie tworzenie zdalnych funkcji. Można tu wymienić środowisko Visual Studio .Net, gdzie tworzenie zdalnych metod usług SOAP sprowadza się do zaimplementowania metod klasy, która w przypadku języka C# musi być klasą potomną klasy `System.Web.Services.WebService`. Podobnie sytuacja wygląda w środowisku C++ Builder 6.0, chociaż tam zachowano koncepcję interfejsu i klasa implementująca metody SOAP dziedziczy z klasy abstrakcyjnej będącej definicją interfejsu.

Wykorzystanie języka WSDL Zasadnicze zalety protokołu SOAP płyną z połączenia jego funkcjonalności z językiem WSDL. Środowiska programistyczne, które pozwalają na tworzenie serwerów SOAP, stwarzają też możliwość uzyskania opisu w języku WSDL implementowanego przez serwer interfejsu. Plik WSDL może być plikiem statycznym zapisanym na dysku lub może być też generowany dynamicznie i wysyłany na żądanie przez serwer implementujący usługę SOAP. Tworzenie aplikacji strony klienta sprowadza się w takiej sytuacji do pobrania pliku WSDL opisującego daną usługę na podstawie, którego odpowiednie narzędzia mogą wygenerować funkcje realizujące zdalne wywołania. Dla przykładowej zdalnej funkcji:

```
double  add(double a, double b);
```

stworzonej w C++ Builder 6.0 najistotniejsze dla bieżących rozważań fragmenty pliku WSDL stworzonego przez to środowisko mają postać:

```
<message name="addRequest">
    <part name="a" type="xs:double" />
    <part name="b" type="xs:double" />
</message>
<message name="addResponse">
    <part name="return" type="xs:double" />
</message>
<portType name="IMySOAPTest">
    <operation name="add">
        <input message="tns:addRequest" />
        <output message="tns:addResponse" />
    </operation>
</portType>
```

Analizując zamieszczony kod WSDL można zauważyć, że zawarta w nim informacja opisująca funkcję `add` wystarcza do wygenerowania kodujących i dekodujących komunikaty XML funkcji strony klienta.

Możliwości Web Services

Wykorzystanie do transportu protokołu HTTP, kiedy trudno już znaleźć platformę, która nie wspiera tego protokołu, pozwala łatwo integrować oprogramowanie z wykorzystaniem Web Servieces. Umożliwia to współpracę oprogramowania uruchamianego na wszystkich rodzajach sprzętu, od komputerów PC po maszyny mainframe i różnego rodzaju urządzenia mobilne. Komunikacja przez port 80, który standardowo wykorzystuje protokół HTTP jest praktycznie zawsze dopuszczana przez ściany ogniowe, które w tym momencie nie są przeszkodą dla Web Services. Web Services eliminują również potrzebę stosowania specjalizowanego środowiska wykonawczego, ponieważ do ich tworzenia wykorzystuje się techniki dynamicznego generowania

kodu języka HTML po stronie serwera, co w najprostszym przypadku (kiedy wykorzystuje się standardowe mechanizmy CGI - *ang. Common Gateway Interface*) ogranicza się do zapotrzebowania jedynie na serwer WWW.

Zastosowania Web Services szybko doprowadziły do wniosku, że ich realizacja jest sensowna, kiedy potencjalni klienci mogą znaleźć informacje na ich temat. Na tej podstawie rozpoczęto prace nad UDDI *(ang. Universal Description Discovery and Integration)*, czyli specyfikacją opisującą rejestry katalogujące przedsiębiorstwa i usługi Web Services udostępniane przez te przedsiębiorstwa.

Web Services oraz wykorzystanie języka XML jest nowym kierunkiem rozwoju metod i technik budowy aplikacji a także wymiany informacji przez sieć, który może przyczynić się co całkowitej zmiany oblicza dzisiejszego Internetu.

Rozdział 3

Metody i techniki dostępu do baz danych

W niniejszym rozdziale przedstawione zostaną metody i techniki dostępu do baz danych, które są istotne w świetle rozważań na temat rozproszonych baz danych omawianych w rozdziale kolejnym. Warto tu podkreślić, że interfejsem najniższego poziomu w dostępie do danych jest najczęściej system zarządzania bazą danych *(ang. Data Base Management System - DBMS)*.

3.1 Typy baz danych

Systemy bazodanowe i metody dostępu do danych są znacznie zróżnicowane w zależności od typu bazy danych, w szczególności od modelu danych. Poniżej przedstawiono przegląd podstawowych typów baz danych z uwzględnieniem ich głównych właściwości, mających wpływ na sposoby dostępu do danych i ich prezentacji w aplikacjach użytkowych.

3.1.1 Plikowe bazy danych

Podstawową strukturą służącą do trwałego przechowywania danych w postaci elektronicznej jest plik. Jeżeli plik ma stanowić bazę danych, to dane w nim zapisane powinny być ujęte w dobrze określoną i sztywną strukturę. Struktura ta powinna odzwierciedlać semantykę przechowywanego w bazie modelu rzeczywistości oraz umożliwiać szybki dostęp do danych. Najprostszym modelem takiej struktury jest przyjęcie podziału na tzw. rekordy zapisywane jako wiersze pliku tekstowego, dzielące się z kolei na określoną liczbę pól zapisywanych jako części wiersza oddzielone od siebie umownie przyjętym znakiem lub sekwencją znaków [9]. Poza prostotą i czytelnością przedstawionego modelu danych zaletą jest możliwość bezpośredniego dostępu do danych za pomocą dowolnego edytora tekstu. Dotyczy to np. wykonania nietypowej operacji, której nie przewidziano projektując narzędzia do manipulowa-

nia danymi. System UNIX oferuje liczne programy, takie jak `sed`, `grep` czy `awk`, umożliwiające efektywne przetwarzanie plików tekstowych.

Plikowe bazy danych mają szereg słabości, które często uniemożliwiają ich wykorzystanie. Z powodu braku systemu DBMS i faktu bezpośredniego dostępu do danych odpowiedzialność za integrację rekordów z założonym wzorcem ponosi aplikacja korzystająca z takiej bazy danych. Trudności powoduje również zapewnienie możliwości równoczesnej modyfikacji jednego zbioru danych przez różne osoby czy programy bez naruszania integralności danych.

3.1.2 Relacyjne bazy danych

Relacyjny model danych został opracowany przez E.F. Codd'a, który w 1970 roku opublikował pracę *"Relacyjny model logiczny dla dużych wielodostępowych banków danych" (ang. A Relational Model for Large Shared Data Banks)* [5, str. 31]. Zaprezentował w niej założenia relacyjnego modelu baz danych, opartego na gałęziach matematyki (teorii mnogości, rachunku predykatów, algebrze relacji).

W modelu relacyjnym dane gromadzone są w tabelach, z których każda ma stałą ilość kolumn i dowolną ilość wierszy. Wiersze tabel określane są mianem krotek, a kolumny z poszczególnych wierszy mianem atrybutów. Już pierwsza postać normalna[1] modelu relacyjnego narzuca warunek, aby typy kolumn były wyłącznie typami prostymi. Tabelaryczne przechowywanie danych ma odzwierciedlenie w dostępie do nich oraz w sposobie ich prezentowania. Wyniki zapytań podawane są w formie tabel, które mogą zawierać wiele krotek, jedną krotkę lub w szczególności pojedynczą wartość będącą atrybutem wiersza.

Na sposoby dostępu i operowania na danych ma wpływ fakt, iż wszelkie przetwarzanie oparte jest na wartościach pól w krotkach. Krotki nie posiadają uniwersalnego, niezmiennego w czasie identyfikatora (odzwierciedlającego ich fizyczne położenie w pliku lub plikach) dostępnego z poziomu języków dostępu do danych. Z tego powodu przeglądanie wyników zapytań wymaga dodatkowych mechanizmów, umożliwiających przeglądanie wiersz po wierszu.

Wykorzystując zasady algebry relacyjnej, opracowano podstawy języka baz danych SQL *(ang. Structured Query Language)*. Różni się on od typowych języków programowania już w założeniach (język algorytmiczny określa jak dany temat zrealizować, a SQL - co należy wykonać). SQL był pierwotnie zaprojektowany jako język zapytań. Obecnie określany jest jako język baz danych, służący także do definiowania danych i operowania nimi [5, str. 79]. Jako język baz danych SQL stał się standardowym interfejsem dla relacyjnych baz danych.

[1] Model relacyjny obejmuje również właściwą sobie teorię projektowania. Teoria ta jest nazywana normalizacją relacji i prowadzi do kolejnych postaci normalnych.

3.1.3 Obiektowe bazy danych

Na powstanie i rozwój obiektowych baz danych bezsprzecznie wpływ miał sukces obiektowych języków programowania. Z drugiej zaś strony, mimo popularności relacyjnych baz danych, zaczęto szukać alternatywnych rozwiązań z powodu ograniczeń modelu relacyjnego. Najbardziej znaczące wymagania dotyczą definiowania własnych typów danych i operowania na nich, braku obsługi ogólnych typów danych występujących w językach programowania czy wreszcie braku typów obiektowych pozwalających lepiej modelować rzeczywistość. Cechy obiektowe występujące w bazach danych umożliwią dostęp do innego typu danych niż ma to miejsce w bazach relacyjnych. Wynikiem zapytania może tu być struktura, literał, obiekt lub zbiór obiektów.

Ponadto istotne z punktu widzenia dostępu do danych jest to, że tworzone obiekty otrzymują unikalne identyfikatory (związane z ich fizycznym położeniem, do których jest możliwy dostęp z poziomu języków dostępu do danych) niezmienne w czasie, które mogą być wykorzystywane przez inne obiekty w celu definiowania powiązań z tymi obiektami. Identyfikatory te są zwykle konwertowane na wskaźnik do pamięci w chwili wczytania do pamięci konkretnego obiektu. Pozwala to skrócić czas dostępu do obiektu, gdy jest on obecny w pamięci operacyjnej [28].

Dostęp do obiektowych baz danych realizowany jest przez dwie koncepcje interfejsów użytkownika. Pierwsza, podobnie jak w bazach relacyjnych, polega na definiowaniu specjalnego języka baz danych. W tym przypadku odpowiednikiem języka SQL z modelu relacyjnego jest obiektowy język zapytań OQL *(ang. Object Query Language)* [27]. OQL stanowi próbę standaryzacji zorientowanych obiektowo języków zapytań, w której starano się połączyć elementy programowania wysokiego poziomu SQL z paradygmatami programowania obiektowego [47, str. 467]. Druga koncepcja interfejsów użytkownika w bazach obiektowych rozszerza obiektowe języki programowania o instrukcje konieczne do obsługi bazy danych. To podejście, ograniczające się w danym przypadku do jednego języka, może zapewniać efektywniejsze przetwarzanie i eliminuje konieczność uczenia się przez programistę innych języków [27].

3.1.4 Obiektowo-relacyjne bazy danych

Obiektowo - relacyjne bazy danych są wynikiem ostrożnej ewolucji systemów relacyjnych w kierunku obiektowych. Jest to powodowane chęcią wprowadzenia do modelu relacyjnego cech obiektowości, takich jak klasy, metody, dziedziczenie czy hermetyzacja [28]. Obiektowo - relacyjne bazy danych można zdefiniować formalnie jako bazy relacyjne z rozszerzeniem obiektowym, gdzie obiekty mogą być przechowywane w bazie danych, a ich metody mogą być wywoływane bezpośrednio z języków baz danych. Próbą standaryzacji tej koncepcji jest język SQL3.

W omawianych wyżej obiektowych bazach danych pojęcie relacji nie jest tak istotne jak w modelach relacyjnych. Natomiast w SQL3 relacja jest nadal podstawo-

wym pojęciem bazy danych. Język SQL3 rozszerza model relacyjny o dopuszczenie bardziej złożonych typów krotek oraz dziedzin atrybutów relacji. Dlatego obiekty w SQL3 mogą być dwojakiego rodzaju. Mogą to być obiekty wierszowe, które są po prostu krotkami oraz abstrakcyjne typy danych, które są w istocie obiektami używanymi jako składowe krotek [47, str. 499].

Model dostępu do danych w bazach obiektowo - relacyjnych odbiega znacznie od istniejącego w systemach obiektowych ze względu na brak odwzorowania między obiektami z języka programowania, a obiektami czy tabelami w bazie. Tutaj tłumaczenie wciąż spoczywa na programiście. Z drugiej jednak strony dostęp do danych nie ogranicza się już tylko do tabel zawierających proste typy danych [28] . Model obiektowo - relacyjny pozwala dodatkowo na dostęp do obiektów, jak również tabel zawierających dane złożonych typów, w szczególności typów obiektowych.

3.1.5 Bazy danych dokumentów XML

Wraz z powstaniem języka XML i rosnącą ilością jego zastosowań szczególnie w dziedzinie przechowywania i przetwarzania danych pojawiły się koncepcje baz danych, które pozwalają gromadzić dane bezpośrednio w formacie języka XML. Obecnie istnieje wiele projektów baz danych realizujących tą koncepcję. Jako przykłady można wymienić Xindce [2], X-hive [51] czy eXist [16]. Zasadnicze cechy tego typu baz danych to:

- możliwość składowania danych w formacie XML, obejmująca procesy ich dodawania i pobierania,
- realizacja zapytań do bazy danych, które pozwalają przeszukiwać zgromadzone w bazie pliki XML; zapytania te są realizowane z wykorzystaniem języka XPath [52],
- możliwość edycji danych XML przechowywanych w bazie, bez konieczności pobierania z niej całych dokumentów; w przypadku Xindce realizowane jest to przy użyciu języka XUpdate [53],
- możliwość definiowania indeksów na elementach i wartościach atrybutów przechowywanych dokumentów XML.

Dane w formacie XML są bez żadnych konwersji, składowane bezpośrednio w bazie danych. Próby konwersji danych XML, np. do modelu relacyjnego, mogą prowadzić do skomplikowanych zależności między tabelami, chociaż z drugiej strony konwersja odwrotna jest procesem prostym i naturalnym. Przewagę nad konwersją do obiektowego modelu danych wykazują tego typu bazy danych również w szerokiej niezależności danych w formacie XML od języka programowania, kiedy to bazy obiektowe są często ściśle powiązane z określonym językiem.

3.2 Interfejsy programistyczne dostępu do baz danych

Przeprowadzone wyżej rozważania w sposób ogólny i abstrakcyjny z punktu widzenia rozwiązań programistycznych obrazują, do jakiego typu danych umożliwiają dostęp różne rodzaje systemów bazodanowych. Niniejszy podrozdział dotyczy technik programistycznych dostępu do szeroko rozumianych źródeł danych. Wyróżnione jest tu pojęcie źródła danych, jako instancji nie zawsze tożsamej z bazą danych, gdyż końcowa aplikacja nie musi koniecznie korzystać bezpośrednio z systemu zarządzania bazą danych, lecz z jednego lub kilku programowych modułów pośrednich. Na tym etapie rozważań nie jest istotny fakt, co jest źródłem danych, ważne jest to, jak dostęp do danych może być realizowany w aplikacji końcowej. Źródła danych będą omawiane w następnym podrozdziale.

3.2.1 Dostęp bezpośrednio w języku programowania

Najbardziej naturalny dostęp do danych występuje w przypadku, gdy istnieje bezpośrednie odwzorowanie języka programowania na język źródła danych. W tej sytuacji nie są potrzebne żadne dodatkowe interfejsy. Rozwiązanie to jest często stosowane w plikowych bazach danych, jak również w bazach obiektowych. Brak dodatkowych interfejsów jest postrzegany jako sposób na podniesienie efektywności przetwarzania, szczególnie w złożonych aplikacjach. Tego typu systemy zarządzania bazą danych budowane są często w postaci bibliotek, które mogą być konsolidowane bezpośrednio z kodem aplikacji korzystającej z takiej bazy danych. Przykładem biblioteki tego typu jest baza danych Berkeley DB.

Ze względu na popularność modelu relacyjnego i wywodzącego się z tego modelu języka SQL, najczęściej stosuje się metody dostępu do relacyjnych baz danych wykorzystujące interfejs w postaci języka baz danych. Dalej omawiane, na przykładzie języka SQL, metody dotyczą wykorzystania języków baz danych. Koncentrują uwagę na zagadnieniu, jak dopasować SQL do środowiska programistycznego.

3.2.2 Interpretowany SQL

Najprostszym sposobem korzystania z SQL jest tryb interpretowany, w którym polecenia SQL przekazywane są poprzez interfejs linii poleceń. Stosowane jest do tego celu oprogramowanie pełniące rolę terminala dostępu bezpośrednio do systemu zarządzania bazą danych lub do pośrednich źródeł danych. Oprogramowanie to może działać zarówno lokalnie jak też poprzez sieć. Jego działanie opiera się na pobieraniu zapytań wprowadzanych przez użytkownika z linii poleceń lub ze wskazanych plików skryptowych, kierowaniu ich do odpowiedniego systemu bazy danych oraz odbieraniu odpowiedzi i prezentowaniu jej użytkownikowi.

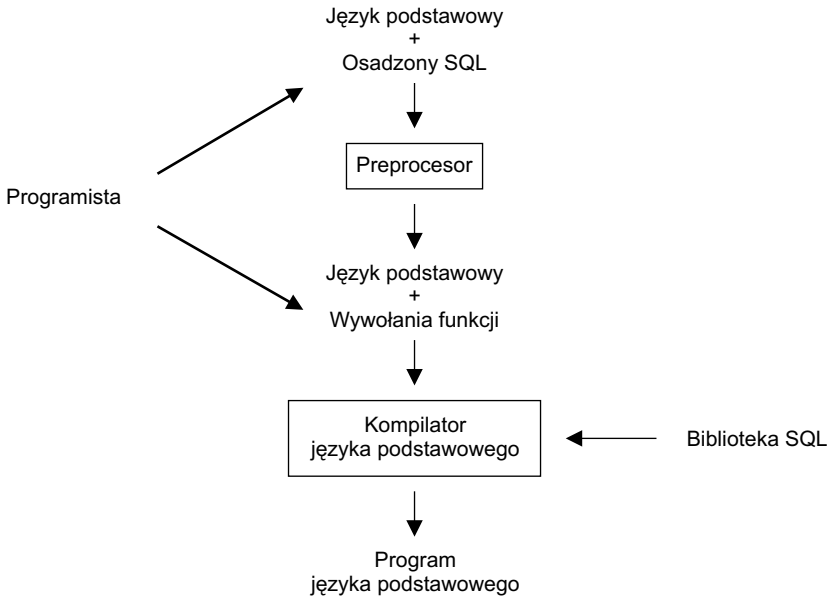
Zaletą tego rozwiązania jest możliwość przetwarzania wsadowego, kiedy korzysta się z plików skryptowych. Dodatkowo możliwe jest wykorzystanie terminala języka

SQL z poziomu innych aplikacji poprzez np. mechanizm potoków.

Interfejsy wiersza poleceń są używane przez programistów głównie do testowania zapytań SQL, używanych z poziomu innych metod dostępu. Dodatkowo umożliwiają sprawne prowadzenie podstawowych czynności administracyjnych [5, str. 121].

3.2.3 Osadzony SQL

Rzadko się zdarza, że wykorzystuje się interpreter SQL, który pozwala na bezpośrednie wprowadzanie i wykonywanie zapytań w tym języku. W praktyce instrukcje SQL stanowią fragmenty większego programu. Sposób łączenia kodu SQL z językami ogólnego przeznaczenia, określany jest mianem osadzonego SQL *(ang. embedded SQL)*. W kodzie wykorzystywanego języka programowania bezpośrednio wstawiane są instrukcje SQL. Rys. 3.1 prezentuje typowy sposób programowania z wykorzystaniem osadzonych instrukcji SQL.



Rysunek 3.1: Przetwarzanie programów z osadzonymi instrukcjami SQL [47, str. 416]

Programista tworzy program w standardowej formie, ale z wykorzystaniem osadzonych instrukcji SQL, które nie należą do języka podstawowego. Tak napisany program jest analizowany przez preprocesor, który powoduje zamianę kodu SQL na kod zapisany w języku podstawowym. Działanie preprocesora sprowadza się najczęściej do zamiany tekstu instrukcji SQL na wywołania funkcji w języku podstawowym. Na schemacie pokazano, że zamiast osadzać kod SQL w programie programista może bezpośrednio w języku podstawowym wywołać funkcję, która uruchamia SQL. Podejście to będzie przedmiotem rozważań następnego punktu.

Po zakończeniu analizy programu przez preprocesor może on zostać skompilowany w standardowy sposób, gdyż zawiera już tylko instrukcje języka podstawowego i wywołania funkcji SQL, których definicji na ogół dostarczają dostawcy systemów baz danych [47, str. 416].

3.2.4 Interfejsy poziomu wywołań

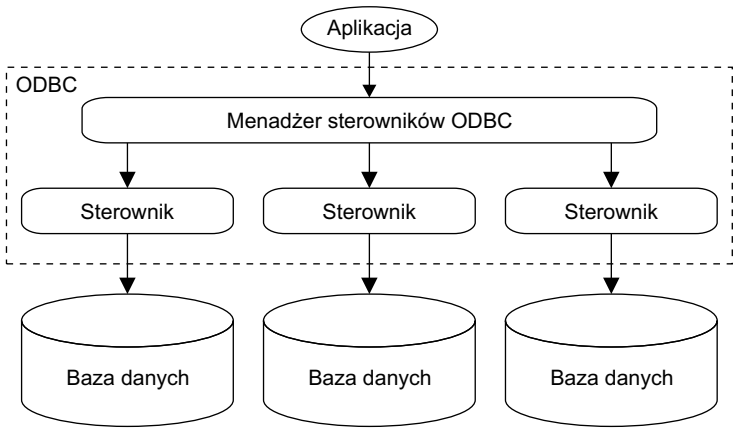
Na rys. 3.1 omawianym w poprzednim punkcie pokazano, że programista może uzyskać dostęp do bazy danych bezpośrednio w języku podstawowym, wykorzystując funkcje, które uruchamiają mechanizmy komunikacji ze źródłem danych. Zamiast umieszczać instrukcje SQL w kodzie aplikacji, umieszcza się ciąg wywołań podprogramów, które wysyłają komunikaty do danego źródła danych. Tego typu sposób tworzenia aplikacji pozwala kompilować program bez potrzeby użycia preprocesora. Dostawcy baz danych na ogół dołączają do systemu biblioteki z niezbędnymi funkcjami. Następnie funkcje, które implementują mechanizmy dostępu do baz danych za pomocą języka SQL, można wywoływać w programach zapisanych w językach podstawowych. Wykorzystanie języka SQL w tych wywołaniach, polega na używaniu instrukcji tego języka jako parametrów.

Omawiany interfejs poziomu wywołań jest najczęściej zarówno dostarczanym przez twórców baz danych, jak i używanym przez programistów interfejsem programistycznym dostępu do baz danych. Nie licząc narzędzi wiersza poleceń, interfejsy poziomu wywołań są na ogół pierwszym mechanizmem dostępu w każdej nowo powstającej bazie danych. Również omawiane niżej technologie, dostarczające pośredniego dostępu do baz danych, wykorzystują najczęściej ideę interfejsów poziomu wywołań [5, str. 124].

3.3 Źródła danych

Poprzedni podrozdział przedstawia rozwiązania programistyczne, jakie pozwalają końcowym aplikacjom łączyć się ze źródłami danych. W niniejszym skupiono uwagę nad tym co, oprócz samej bazy danych, może być źródłem danych aplikacji. Omówione zostaną najczęściej używane mechanizmy.

Każdy system bazodanowy ma własne mechanizmy dostępu do danych, które są zwykle na tyle specyficzne, że ograniczają zastosowanie tylko do jednego systemu. W czasach, kiedy nie istniały komputery PC, a oprogramowanie tworzone było głównie z myślą o komputerach mainframe, nie było to jeszcze znaczącym problemem. Rozwój oprogramowania dla komputerów osobistych przyczynił się do konieczności ujednolicenia dostępu zarówno do baz danych różnych typów, jak i różnych baz danych tego samego typu. W ten sposób zaczęły powstawać mechanizmy pośredniczące w dostępie do danych, występujące między aplikacją końcową a bazą danych.



Rysunek 3.2: Schemat architektury ODBC

3.3.1 ODBC

ODBC *(ang. Open Data Base Connectivity)* zostało zaprojektowane jako próba utworzenia standardu jednolitego dostępu do baz danych. Przy jego opracowaniu uczestniczyło wiele firm. W wyniku powstał jednolity sposób wymiany danych pomiędzy aplikacją użytkową a DBMS, uznany za standard przemysłowy, który używa języka SQL jako języka baz danych. Zasadnicza korzyść jaka wynika ze stosowania omawianego standardu polega na tym, że aplikacja ma dostęp do różnych DBMS na poziomie raz napisanego, skompilowanego i konsolidowanego kodu aplikacji. Aby móc to zapewnić, twórcy ODBC musieli rozwiązać dwa problemy. Po pierwsze, ODBC musiało udostępniać programistom baz danych ogólny interfejs programistyczny, który byłby uniwersalnym mechanizmem komunikacji z używanym DBMS. Po drugie, należało zapewnić ODBC możliwość porozumiewania się z każdym DBMS. Osiągnięto to przez zastosowanie sterowników i menedżera sterowników ODBC. Sterowniki są modułami budowanymi w sposób dedykowany do obsługi poszczególnych baz danych. Menedżer sterowników jest pierwszą warstwą architektury ODBC, która otrzymuje wywołania bezpośrednio z aplikacji użytkowych. Wywołania te kierowane są przez menedżera sterowników do odpowiednich sterowników baz danych, które są uruchamiane w chwili żądania i mogą być w sposób niewidoczny dla końcowej aplikacji zamieniane. Tak więc sterownik ODBC jest to jedyna warstwa kodu mająca jakąkolwiek wiedzę na temat wewnętrznego działania DBMS. Omawiana architektura ODBC przedstawiona została na rys. 3.2.

Wykorzystanie ODBC jako warstwy pośredniej może, w niektórych sytuacjach wiązać się ze spadkiem wydajności, gdyż wymusza ono rezygnację ze specyficznych rozwiązań stosowanych w konkretnych systemach bazodanowych.

3.3.2 OLE DB

Kolejną powszechnie stosowaną techniką udostępniania danych jest OLE DB *(ang. Object Linking and Embedding)*. OLE DB zostało stworzone przez Microsoft w oparciu o model COM. Zalety tego modelu, takie jak niezależność od języka programowania czy inne korzyści wynikające z tego, iż jest to standard binarny, przenoszą się na utworzone na jego podstawie OLE DB, które jest zbiorem interfejsów COM opracowanych w celu umożliwienia jednolitego dostępu do danych.

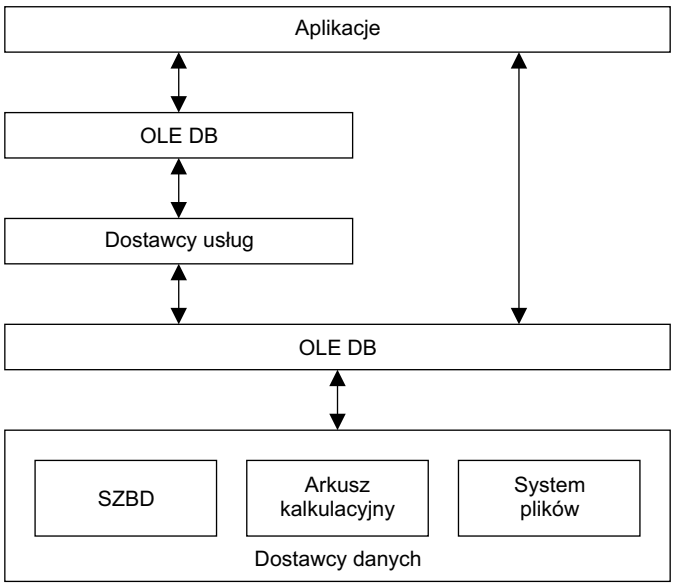
Twórcy OLE DB podobnie jak twórcy ODBC mieli do rozwiązania dwie grupy zagadnień. Pierwszą, polegającą na zapewnieniu interfejsu programistycznego, rozwiązał model COM, dostarczając standardu definiowania interfejsów. Druga, podobnie jak w ODBC, polegała na dostarczeniu mechanizmów porozumiewania się z DBMS lub innymi źródłami danych. Wykorzystanie powszechnie stosowanego w systemie Windows modelu COM zapewniło uniwersalny dostęp do danych różnego formatu, w tym danych przechowywanych w tradycyjnych systemach bazodanowych, danych tekstowych, danych zapisanych w arkuszach kalkulacyjnych, wiadomości e-mail, i innych. W ODBC kluczową rolę w porozumiewaniu się z bazami danych odgrywały dedykowane sterowniki. W OLE DB podobną rolę odgrywają dostawcy danych. Są to moduły programowe, których zadaniem jest udostępnianie dowolnego typu danych (sprecyzowanego w danym zastosowaniu) w formacie tabelarycznym, czyli w formacie posiadającym wiersze i rekordy. Ponieważ wiele aplikacji w systemie Windows bazuje na modelu COM, możliwe jest bezpośrednie wykorzystanie ich komponentów do realizacji dostawców danych OLE DB, które też są oparte na standardzie COM. Zapobiega to tworzeniu redundantnych baz danych (np. komponent COM klienta pocztowego, pobierający i przechowujący wiadomości e-mail może stać się dostawcą danych OLE DB).

Rys. 3.3 obrazuje ogólny schemat architektury OLE DB. Pokazano na nim, że oprócz dostawców danych mogą być wykorzystywane również moduły zwane dostawcami usług, które nie mają bezpośredniego dostępu do danych. Zadaniem dostawcy usług jest zapewnienie dodatkowych funkcji dostępu do danych, np. integracji kilku źródeł danych i prezentacji ich jako jednego źródła.

Zbudowanie OLE DB w oparciu o standard COM, zagwarantowało dużą efektywność, niezależność od języka programowania i dostęp do różnego typu danych.

3.3.3 JDBC

Z problemem zapewnienia jednolitego dostępu do baz danych zetknęli się również twórcy środowiska dla języka Java. Jak wiadomo język ten tworzono z myślą zapewnienia przenaszalności jego kodu, nie tylko źródłowego, na różne platformy. Z tego powodu naturalną była potrzeba zapewnienia tej przenośności także w przypadku technik dostępu do baz danych. Przedstawione wcześniej dwa mechanizmy skupiają się głównie wokół systemu Windows (choć standard ODBC jest implementowany również na innych platformach; np. systemie UNIX).



Rysunek 3.3: Budowa OLE DB

Dla zastosowań platformy języka Java przedsiębiorstwo Sun Microsystems opracowało zestaw standardowych bibliotek API tego języka, które nazwano JDBC *(ang. Java DataBase Connectivity)*. JDBC jest zestawem klas i interfejsów, które pozwalają aplikacjom w Javie komunikować się za pomocą instrukcji SQL z dowolną, relacyjną bazą danych.

Podobnie jak dwie omawiane wyżej technologie, również JDBC ma dwa główne zbiory interfejsów. Pierwszy jest przeznaczony dla twórców aplikacji użytkowych, natomiast drugi służy do pisania sterowników dla określonych systemów baz danych. Sterowniki tworzone są w postaci programowych modułów, umożliwiających połączenie systemu JDBC z określonym DBMS [14, str. 17 i nast.]. Raz napisana aplikacja użytkowa bazująca na JDBC może być przenoszona na różne platformy, jak też może korzystać z różnych baz danych, bez potrzeby zmiany kodu źródłowego a nawet ponownej kompilacji. Możliwe jest to dzięki idei niezależności języka Java od platformy.

Tak jak było to przedstawione wcześniej, omawiane technologie mogą stanowić dla aplikacji pośrednie źródło danych. W praktyce stanowią one też często źródła danych względem siebie. Z omawianych mechanizmów pierwszy rozwinął się standard ODBC, dla którego stworzono sterowniki do wielu baz danych. Z tego powodu ekonomicznie uzasadnione było wykorzystanie tych sterowników przez udostępnienie źródeł danych w standardzie ODBC implementacjom OLE DB i JDBC.

3.4 Metody dostępu do źródeł danych

Wykorzystanie omawianych wyżej interfejsów w praktyce często okazuje się trudne i skomplikowane. Dotyczy do głównie technologii ODBC i OLE DB. Aby stworzyć prosty program nawiązujący połączenie z bazą danych, wykonujący na niej zapytanie i pobierający wyniki, programista musi w przypadku ODBC napisać około stu linii kodu. Podobnie sytuacja wygląda w przypadku OLE DB, gdzie programista musi operować wieloma interfejsami COM.

W wyniku trudności w wykorzystaniu omawianych wyżej mechanizmów w praktyce są one często interfejsami do wykorzystania przez narzędzia programistyczne, a nie bezpośrednio przez programistów. Narzędzia te generują automatycznie kody źródłowe wykorzystujące interfejsy dostępu do baz danych.

Na bazie ODBC i OLE DB powstało kilka mechanizmów wyższego poziomu. Zaliczyć tu można wspomaganie obsługi ODBC ze strony klas biblioteki MFC. Wykorzystanie programowania obiektowego w połączeniu z narzędziami do automatycznego generowania kodu obsługi ODBC znacznie ułatwia stosowanie tego mechanizmu. Również dla OLE DB wprowadzono wysokopoziomową osłonę w postaci ADO *(ang. Active Data Objects)*. ADO jest łatwym w użyciu interfejsem do OLE DB, ponieważ dla tego mechanizmu stworzono również narzędzia wizualnego projektowania i generowania kodu podstawowych operacji.

Choć programowanie interfejsu JDBC nie jest tak skomplikowane jak ODBC czy OLE DB, to na jego podstawach powstają również alternatywne interfejsy wyższego poziomu. Należy tu wymienić koncepcję osadzonych instrukcji SQL w kod języka Java. Standard ten jest nazywany SQLJ *(ang. SQL for Java)* i realizuje koncepcję osadzonych instrukcji SQL omawianych w punkcie poprzednim. Cechą charakterystyczną SQLJ jest to, że działa on w oparciu o uniwersalne metody dostępu do danych jakie stwarza JDBC przez co jest niezależny od źródła danych.

Inna koncepcja wspomagania dostępu do źródeł danych JDBC wykorzystuje bezpośrednie odwzorowania tablic relacyjnych baz danych na klasy języka Java. W tym obiektowo - relacyjnym odwzorowaniu tabela definiuje klasę, każdy wiersz tabeli staje się instancją klasy (czyli obiektem), a każda wartość atrybutu w kolumnie wiersza odpowiada atrybutowi instancji (obiektu).

3.5 Kierunki rozwoju technik dostępu do baz danych

Aktualne prace nad sposobami dostępu do baz danych dotyczą w znacznej mierze zastosowań języka XML. Język XML w połączeniu ze standardami powstającymi na bazie jego zastosowań i koncepcją Web Services może stać się powszechnie stosowanym formatem i sposobem wymiany danych. Już od kilku lat wiodący producenci systemów baz danych starają się zapewnić dostęp do danych mających format XML. Rodzi to potrzebę opracowywania technik importu danych z XML do bazy danych

i odwrotnie.

Potrzeba organizacji wymiany danych między aplikacjami pisanymi w dowolnych środowiskach programistycznych, pracującymi na dowolnych platformach programowych i sprzętowych stała się przyczyną powstania takich rozwiązań jak Web Services. Zapewniają one jednakowy dostęp do danych pochodzących nie tylko z baz danych, ale także z różnego rodzaju urządzeń mobilnych, bezpośrednio z systemu plików czy usług poczty elektronicznej.

Zastosowanie języka XML jako formatu wymiany danych widać w głównym produkcie dostępu do danych ADO.NET firmy Microsoft. W ADO.NET podstawowym formatem wymiany danych między aplikacjami jest XML. Aplikacje mogą wymieniać zbiory danych w postaci strumieni danych XML. Zastosowanie XML i wykorzystanie jego podstawowej cechy jaką jest rozszerzalność, pozwala w prosty sposób opisywać różne typy danych (nawet obiektowe). Dla porównania w ADO współdzielenie danych wymaga zastosowania mechanizmów szeregowania COM, co ogranicza np. możliwe typy danych do typów COM i wymaga konwersji do tych typów [10] .

Poruszone tu podejście dostępu do danych z wykorzystaniem XML wynika z zasadniczej cechy tego języka, jaką jest rozszerzalność. Pozwala to opisywać z jego użyciem różne typy danych i odwzorowywać różne modele danych. Z drugiej strony ta sama rozszerzalność może doprowadzić do chaosu w zastosowaniach i rodzących się opisach formatów danych w języku XML. Dlatego warto tu podkreślić konieczność standaryzowania zastosowań XML, a jako przykład takiego podejścia rokującego stworzenie ogólnie przestrzeganego standardu, można wymienić specyfikację SOAP i bazujące na niej Web Services.

Rozdział 4

Rozproszone bazy danych

Rozproszone bazy danych stanowią jedną z głównych realizacji idei systemów rozproszonych. Określenie definiujące tego typu bazy danych znajduje się w [12, str. 660]: *"Pełne wsparcie rozproszonej bazy danych oznacza, że pojedyncza aplikacja może swobodnie operować na danych rozrzuconych w różnych bazach danych, zarządzanych przez różne DBMS, posadowionych na wielu komputerach o odmiennych systemach operacyjnych i połączonych ze sobą rozmaitymi sieciami komunikacyjnymi. Termin swobodnie oznacza tu, że z logicznego punktu widzenia aplikacja realizuje się tak, jakby wszystkie dane były zarządzane tylko jednym DBMS, działającym na jednej maszynie."*. Przytoczona tu definicja wykazuje wiele podobieństw z określeniami systemów rozproszonych, które zostały przedstawione w rozdziale drugim. Dotyczy to przede wszystkim własności działania tych systemów na wielu samodzielnych komputerach komunikujących się za pośrednictwem sieci komputerowych. Określenie o podobnej treści przedstawiono w [35, str. 8]: *"Rozproszoną bazę danych można zdefiniować jako logicznie zintegrowane kolekcje współdzielonych danych, które są fizycznie rozproszone w węzłach sieci komputerowej."*. Uwypuklona tu logiczna integracja danych pozwala, aby z punktu widzenia użytkownika system rozproszony wyglądał dokładnie tak samo jak zwykły system [12, str. 663]. Użytkownicy są tu rozumiani jako osoby operujące na danych, dla nich rozproszona baza danych wydaje się być pojedynczą, jednolitą bazą danych.

4.1 Ogólna klasyfikacja rozproszonych baz danych

Jedna z możliwych klasyfikacji rozproszonych baz danych wynika z określenia zawartego w [5, str. 130]: *"Rozproszony system baz danych to system baz danych, w którym występuje rozłożenie danych przez ich fragmentaryzację (podział) lub replikację do różnych konfiguracji sprzętowych i programistycznych na ogół rozmieszczonych w różnych (geograficznie) miejscach organizacji."*. Stwierdzenie to podkreśla dwie zasadnicze możliwości rozpraszania danych, którymi są fragmentaryzacja i replikacja. Te dwie koncepcje, wytyczające zasadnicze kierunki rozwoju rozproszonych baz danych, wymagają często zupełnie innych rozwiązań projektowych i prowadzą do

osiągnięcia odmiennych celów.

W literaturze spotykane są też inne podziały podejmujące próby definicji typów rozproszonych baz danych. W [35, str. 11] zaproponowano, aby każdy z rozproszonych systemów bazodanowych kwalifikować jako jeden z dwóch typów:

- homogeniczny,
- heterogeniczny.

W systemach homogenicznych wszystkie bazy danych wchodzące w skład systemu rozproszonego mają jednakowe DBMS, zaś rozproszony system heterogeniczny jest w stanie integrować różne DBMS, szczególnie różniące się pod względem konfiguracji sprzętowych i programowych. W [30, str. 27] podano, że termin heterogeniczne bazy danych używany jest często zamiennie z określeniami bazy federacyjne i systemy multi-bazodanowe *(ang. multi-database)*.

Jeszcze inna klasyfikacja baz danych możliwa jest ze względu na sposób wykorzystania danych. Można tu wyróżnić systemy OLTP *(ang. On Line Transaction Processing)*, których bieżące przetwarzanie transakcji obejmuje takie zastosowania jak: systemy rezerwacji, obsługę punktów sprzedaży, oprogramowanie giełdowych stacji roboczych - ogólnie zastosowania, gdzie zachodzą częste, współbieżne aktualizacje i pobieranie zazwyczaj małych ilości informacji. Drugim typem są systemy OLAP *(ang. On Line Analytical Processing)*, do których zaliczane są hurtownie danych *(ang. data warehouse)*. Hurtownie danych przechowują informacje o wybranych aspektach działalności organizacji przeznaczone do późniejszych analiz, np. w celach strategicznych. Systemy te są projektowane najczęściej po kątem optymalizacji zapytań zwracających lub przetwarzających duże ilości danych.

4.2 Zagadnienia projektowe rozproszonych baz danych

Rozproszone bazy danych w porównaniu z systemami scentralizowanymi wymagają rozpatrzenia kilku dodatkowych kwestii projektowych. Konieczne staje się zapewnienie logicznej integracji danych fizycznie rozproszonych w węzłach systemu. Istotne jest tu uwzględnienie wpływu sieci komputerowej.

Poniżej omówionych zostanie kilka problemów projektowych związanych z kwestią rozproszonego charakteru tych systemów.

4.2.1 Przetwarzanie zapytań

Optymalizacja zapytań, której celem jest zminimalizowanie czasu wykonania zapytania lub, co jest często równoważne, zmniejszenie wykorzystania zasobów systemu, jest jednym z głównych problemów projektowych baz danych. W przypadku rozproszonych baz danych jest to zabieg trudniejszy ze względu na większą liczbę

parametrów (m.in. czas komunikacji w sieci komputerowej, czas wykonania operacji wejścia/wyjścia, różny czas obliczeń na poszczególnych węzłach) wpływających na wydajność zapytań rozproszonych. Języki zapytań, takie jak SQL, pozwalają jedynie na określenie, jaką informację użytkownik chce uzyskać, nie definiując, w jaki sposób ma to być zrealizowane [35, str. 31]. Część systemu DBMS wykonująca zapytania określana jest mianem procesora zapytań. W celu realizacji zapytań rozproszonych, procesor zapytań wykonuje trzy zasadnicze zabiegi:

- lokalizację danych,
- optymalizację globalną,
- optymalizację lokalną.

Lokalizacja danych polega na transformacji, czasem z wykorzystaniem algebry relacji[1], zapytania wejściowego do systemu rozproszonego na równoważny zbiór zapytań do baz danych przechowywanych w różnych węzłach sieci. Dla powstałych w ten sposób cząstkowych zapytań w etapie globalnej optymalizacji generowany jest optymalny plan wykonań poszczególnych fragmentów. Podejmowane są decyzje o kolejności przetwarzania zapytań, a także o danych przesyłanych między węzłami rozproszonego systemu. Optymalizacja lokalna z kolei przeprowadzana jest oddzielnie na każdym stanowisku zaangażowanym w realizację zapytania.

W [12, str. 673] przedstawiono analizę przykładowego rozproszonego zapytania, gdzie uzyskano wyniki, w których najwolniejsza technika jest dwa miliony razy wolniejsza od najlepszej. Realizacja tego zapytania polega na złączeniu trzech tabel. Różnice czasów wykonania zależą od tego w jakim kierunku dane przesyłane są między węzłami. Zamiast przesyłać do węzła B tabelę z węzła A liczącą 100000 rekordów korzystniej jest przesłać liczącą 10 rekordów tabelę z węzła B do węzła A. Przykład ten podkreśla znaczenie fazy optymalizacji rozproszonych zapytań.

4.2.2 Zarządzanie katalogiem

W rozproszonych bazach danych w nieco szerszym znaczeniu w stosunku do baz scentralizowanych występuje pojęcie słownika danych, który oprócz metadanych, czyli danych o danych (informacji o relacjach, perspektywach, indeksach, użytkownikach), zawiera także informacje kontrolne pozwalające na funkcjonowanie w rozproszeniu. Słownik danych określany jest czasem terminem katalogu [12, str. 676].

Znaczenie katalogu i jego właściwości są zróżnicowane w zależności od typu bazy danych, do jakiej się stosują. W replikujących bazach danych zagadnienia związane z opisem relacji, perspektyw, więzów integralności itp. są najczęściej przechowywane na każdym z węzłów systemu niezależnie, tak jak w klasycznym systemie DBMS. Inaczej problem wygląda w bazach, w których występuje rozproszenie danych na

[1] Algebra relacji jest to zbiór operatorów, które działają na całych relacjach i dają w wyniku również całą relację.

różne węzły poprzez dzielenie tabel. Tego typu podejście wymaga rozproszonej realizacji takich kwestii jak integralność danych.

W teorii systemów rozproszonych można odnaleźć dwie zasadniczo różniące się ideologicznie realizacje katalogu. Jest to rozwiązanie scentralizowane, w którym katalog jest przechowywany w jednym, centralnym miejscu oraz przeciwne do niego rozwiązanie w pełni rozproszone. W tym przypadku nie ma centralnego miejsca, którego awaria mogłaby zablokować cały system. Między tymi dwoma skrajnymi realizacjami istnieją różne rozwiązania pośrednie [12, str. 676]:

- pełna replikacja - kopia całego katalogu jest przechowywana na każdym węźle.
- katalogi lokalne z kopią centralną - wszystkie węzły w sposób rozproszony przechowują własną część katalogu; łączne zasoby wszystkich katalogów są dodatkowo przechowywane w jednym centralnym miejscu.

Każde z tych rozwiązań charakteryzuje się pewnymi ograniczeniami. Rozwiązania scentralizowane powodują uzależnienie od węzła zarządzającego katalogiem. W realizacjach rozproszonych z kolei operacje zdalne są często czasochłonne.

4.2.3 Przekazywanie aktualizacji

Przekazywanie aktualizacji jest zasadniczym problemem rozproszonych baz danych z replikacją. Dotyczy to nie tylko tego, aby aktualizację dowolnego obiektu przesłać do wszystkich jego przechowywanych kopii, ale także, aby uczynić to efektywnie, często z dodatkowymi ograniczeniami, tzn. zachować integralność danych, nie obciążać źródeł danych procesami replikacji, zapewnić lokalną autonomię czy prostotę zarządzania. Stosowane obecnie algorytmy replikacji danych w środowisku rozproszonym można, uwzględniając synchronizację replikowanych danych, zakwalifikować do jednego z dwóch modeli rozpraszania danych:

- synchronicznego,
- asynchronicznego.

Model synchroniczny realizowany jest najczęściej przez zastosowanie mechanizmów rozproszonych transakcji (omawianych w punkcie kolejnym) w celu zapewnienia integralności danych na wszystkich węzłach. Zatwierdzanie transakcji może nastąpić dopiero wtedy, gdy dane na wszystkich węzłach systemu zostaną uaktualnione. Wadą tego rozwiązania jest to, iż w przypadku niedostępności choćby jednego z węzłów, np. z powodu awarii sieci komputerowej, nie może dojść do zatwierdzenia transakcji na pozostałych węzłach. W modelu asynchronicznym replikacja może odbywać się niezależnie od transakcji z możliwością wystąpienia opóźnień.

Proces replikacji może odbywać się z wykorzystaniem różnych koncepcji technicznych. Jedno z pierwszych rozwiązań replikacji wykorzystuje metody stosowane przy tworzeniu kopii zapasowych danych. Wyróżnia się tu dwa podejścia:

- kopia danych na poziomie systemu plików - wszystkie pliki z danymi są fizycznie kopiowane,
- zrzut instrukcji SQL *(ang. dump and reload)* - cała zawartość bazy danych zamieniana jest na sekwencję instrukcji w języku SQL, których wykonanie pozwala odtworzyć bieżący stan bazy danych.

Techniki te są najczęściej stosowane w mniejszych systemach DBMS (np. w bazach danych MySQL [36] i PostgreSQL [39]) jako środki tworzenia kopii zapasowych. Warto tu podkreślić, że replikacja z wykorzystaniem zrzutu instrukcji SQL może prowadzić do powstania kopii danych o mniejszych rozmiarach niż kopie na poziomie systemu plików. Wynika to z faktu, iż w tym przypadku nie jest kopiowana zawartość indeksów, a tylko informacja (w postaci instrukcji SQL) jak je stworzyć.

Innym rozwiązaniem jest replikacja migawkowa *(ang. snapshot replication)*. Polega ona na tym, że do wszystkich węzłów pobierających dane przesyłane są obrazy całej bazy danych lub wybranych jej fragmentów (tabel, zbiorów tabel). Przy każdym uaktualnieniu danych następuje przesłanie całej ich zawartości. Replikacja tego typu prowadzi do powstania kopii tylko do odczytu, które nie mogą być zmieniane. Jest to replikacja z jednokierunkową propagacją zmian.

Kolejne rozwiązanie wykorzystuje mechanizmy stosowane do zapewnienia integralności danych, czyli wyzwalacze skojarzone z określonymi zdarzeniami w bazie danych. Mogą być one wykorzystywane do inicjowania procedur replikacji danych. W przeciwieństwie do procedur omawianych wcześniej, wykorzystanie wyzwalaczy znajduje zastosowanie przy replikacji zarówno synchronicznej, jak i asynchronicznej. Jednym z głównych problemów związanych z wykorzystaniem wyzwalaczy jest to, że powodują one dodatkowe obciążenie bazy danych związane z realizacją replikacji. Muszą się one znajdować w bazie danych i być przez nią wywoływane. Należy tu zaznaczyć, że wyzwalacze pełnią tu rolę mechanizmu powiadomienia o zdarzeniu, konieczności aktualizacji stanu systemu. Mogą być tu stosowane dowolne mechanizmy przesyłania aktualizacji, takie jak sekwencje instrukcji SQL wykonanych od czasu ostatniej aktualizacji (takie podejście zastosowano przy realizacji replikacji w bazie danych MySQL, jednakże tam aktualizacja danych nie jest inicjowana przez wyzwalacz, lecz jest sterowana zegarem o definiowanym interwale czasowym).

W formie przykładu realizacji replikacji danych przedstawione zostaną techniki stosowane w Microsoft SQL Server 2000. Procesy replikacji danych w tym przypadku dotyczą wymiany danych między dwoma zasadniczymi rodzajami węzłów. Są to serwery publikujące *(ang. publisher)* oraz prenumerujące *(ang. subscriber)*. Serwery publikujące udostępniają serwerom prenumerującym dane do aktualizacji. Współpraca między serwerami może odbywać się na jeden ze sposobów:

- akcja przesyłania danych inicjowana jest przez serwer prenumerujący *(ang. pull subscription)*,
- serwer publikujący ustanawia proces aktualizacji danych *(ang. push subscription)*.

Przebieg procesu aktualizacji w SQL Serwer 2000 dobywać się może na jeden z omówionych niżej sposobów wybieranych w zależności od zastosowania.

- replikacja migawkowa
  Ten typ replikacji polega na wysyłaniu przez serwer publikujący obrazów z zawartością całej bazy danych. Kopie danych wędrują do wszystkich serwerów w ustalonych chwilach. Serwery prenumerujące są w ten sposób odświeżane kompletną kopią danych. Rozwiązanie to znajduje zastosowanie tam, gdzie dane nie zmieniają się często, dopuszczalna jest czasowa niespójność danych, zasoby są nieduże i węzły systemu mogą być czasowo niedostępne.

- replikacja transakcyjna *(ang. transactional replication)*
  Idea tego typu replikacji tkwi w zamknięciu całego procesu aktualizacji w granicach jednej transakcji. Transakcja zmieniająca dane na serwerze publikującym powoduje zainicjowanie transakcji w serwerach prenumerujących. Rozwiązanie to stosowane jest w przypadkach, kiedy wymagana jest propagacja zmian danych na wszystkie serwery w jak najkrótszym czasie oraz aktualizacje powinny być atomowe.

- replikacja połączeniowa *(ang. merge replication)*
  Ten rodzaj replikacji pozwala na autonomiczną pracę węzłów systemu. Przechowują one wówczas niezależne zbiory danych. Całość zasobów jest łączona jedynie okresowo lub na żądanie. Proces ten wymaga mechanizmów rozwiązywania konfliktów danych. Są tu dostępne metody domyślne, jak i konfigurowalne przez użytkownika, uwzględniające np. priorytety węzłów lub danych. Ten typ replikacji nadaje się do zastosowań, gdzie w węzłach systemu przechowywane są dane w pewien sposób niezależne, np. dane przechowywane w węzłach różnych oddziałów przedsiębiorstwa dotyczące tylko danego oddziału. Łączenie danych tego typu może przebiegać całkowicie bezkonfliktowo.

4.2.4 Kontrola wykonania transakcji

Transakcje są jednym z podstawowych mechanizmów związanych z przetwarzaniem baz danych. Pojęcie to zostało w [12, str. 423] określone jako *"logiczna jednostka pracy"*, a jej rolę w przetwarzaniu danych podkreślają własności niepodzielności (dla świata zewnętrznego przebiega ona w sposób niepodzielny), spójności (nie narusza niezmienników), izolacji (bieżąca transakcja nie koliduje z żadną inną) i trwałości (po zakończeniu transakcji wprowadzone zmiany są trwałe). Zapewnienie tych wymagań, szczególnie w środowisku rozproszonym, jest zabiegiem skomplikowanym i wymaga przeprowadzenia złożonych analiz. W niniejszym punkcie przedstawione zostaną najpopularniejsze algorytmy realizacji transakcji.

W prowadzonej niżej analizie użyte zostaną pojęcia koordynatora i uczestnika transakcji, które są tu odpowiednio elementami zarządzającymi przebiegiem transakcji i uczestniczącymi w niej. Prosty sposób zatwierdzania transakcji stanowi pro-

tokół jednofazowego, niepodzielnego zatwierdzania transakcji *(ang. one - phase atomic commit protocol)* [45, str. 542]. Sposób działania tego protokołu polega na komunikowaniu przez koordynatora żądania zatwierdzenia bądź cofnięcia transakcji wszystkim węzłom biorącym w niej udział. Wysyłanie komunikatu ponawiane jest do momentu, aż wszystkie węzły potwierdzą, że spełniły żądanie. Rozwiązanie to nie nadaje się jednak do praktycznego wykorzystania, ponieważ nie pozwala wszystkim węzłom uczestniczyć w podjęciu decyzji o zatwierdzeniu bądź odrzuceniu transakcji. Braki tego algorytmu eliminuje dwufazowy protokół zatwierdzania *(ang. two - phase commit protocol - 2PC)*, który pozwala wszystkim węzłom uczestniczyć w podejmowaniu decyzji o zatwierdzeniu transakcji (dowolny węzeł może przyczynić się do odrzucenia transakcji). Możliwe jest to dzięki dodaniu drugiej fazy zatwierdzania, która pozwala przeprowadzić zapytanie do wszystkich węzłów o gotowość do zatwierdzenia transakcji. Obie fazy protokołu przedstawiają się następująco:

- Koordynator wysyła do wszystkich uczestników komunikaty z pytaniem o potwierdzenie gotowości do przeprowadzenia zatwierdzenia transakcji (N - 1 komunikatów). Uczestnicy wysyłają pozytywne bądź negatywne odpowiedzi (N - 1 komunikatów).

- Koordynator wysyła do wszystkich uczestników polecenie zatwierdzenia lub cofnięcia transakcji (N - 1 komunikatów). Po wykonaniu zadania uczestnicy wysyłają potwierdzenie do koordynatora (N -1 komunikatów).

Protokół dwufazowego zatwierdzania mimo pewnych wad jest obecnie najczęściej stosowanym protokołem zatwierdzania transakcji w systemach rozproszonych, w szczególności w rozproszonych bazach danych. Najczęściej wymienianą wadą tego protokołu jest to, iż jest on protokołem blokującym, tzn. w przypadku awarii koordynatora dowolny uczestnik może zostać zablokowany. Prowadzi to do niepotrzebnego blokowania zasobów danego uczestnika przed dostępem przez inne transakcje nawet w sytuacjach, kiedy przetwarzanie bieżącej transakcji na pozostałych węzłach zostało zakończone. W przypadku, kiedy koordynator ulegnie trwałej awarii, uczestnicy mogą "nigdy" nie zakończyć rozproszonych transakcji, a zasoby mogą zostać zablokowane "na zawsze".

Kolejnym rozwiązaniem zatwierdzania transakcji jest protokół trójfazowy *(ang. three - phase commit protocol - 3PC)*. Jego główna cecha w porównaniu z omawianym wyżej protokołem dwufazowym polega na tym, że jest to protokół nieblokujący (tzn. nie blokuje uczestników transakcji). Możliwość realizacji tego protokołu w trybie nieblokującym zapewniona została dzięki podziałowi fazy drugiej protokołu 2PC na dwie nowe fazy.

Nieblokujące zatwierdzanie transakcji w protokole 3PC wymaga zastosowania dodatkowego elementu - elekcji koordynatora. Pozbycie się blokującego czekania przez uczestników zatwierdzania transakcji w przypadku awarii koordynatora odbywa się tu przez wybór nowego koordynatora.

4.2.5 Odtwarzanie stanu bazy danych po awariach

Prawidłowe przeprowadzenie wykonania transakcji wymaga rozwiązania problemów odtwarzania stanu bazy danych po awariach. Na tym podłożu transakcja odgrywa rolę jednostki odtwarzania, poza wspomnianą wcześniej logiczną jednostką pracy. Przywracanie stanu baz danych po awariach odbywa się zwykle na poziomie transakcji. Prowadzi to do konieczności rozważenia w implementacji transakcji mechanizmów odtwarzania, które pozwolą cofnąć transakcję. W [45, str. 186] zostały przestawione dwa najczęściej stosowane rozwiązania:

- Prywatna przestrzeń robocza
  W tym przypadku na potrzeby każdej rozpoczynanej transakcji tworzona jest kopia danych (dotyczy to najczęściej tylko tych danych, które będą modyfikowane przez transakcję). Całe przetwarzanie w obrębie transakcji odbywa się na kopii danych. Dopiero jeśli transakcja jest zatwierdzona, dane zmodyfikowane przez transakcję zostają zamienione z danymi pierwotnymi (unika się tu oczywiście kopiowania danych na rzecz operacji na wskaźnikach).

- Rejestr zapisów wyprzedzających *(ang. write-ahead log)*
  W tej metodzie możliwe jest modyfikowanie plików w miejscach ich występowania. Tym razem przed każdą zmianą danych następuje zapisanie informacji do rejestru zapisów wyprzedzających, przechowywanych w pamięci trwałej. Zapisywany rekord dziennika zawiera najczęściej następujące informacje:

  – identyfikator transakcji,
  – typ operacji,
  – dane wykorzystywane przez transakcję do wykonania operacji,
  – pierwotne wartości danych,
  – nowe wartości danych.

  Zmiany w plikach z danymi dokonywane są dopiero po zapisaniu powyższych informacji do dziennika. Na podstawie dziennika zdarzeń można wycofać transakcję lub rekonstruować bazę po awariach.

4.2.6 Kontrola współbieżnego dostępu

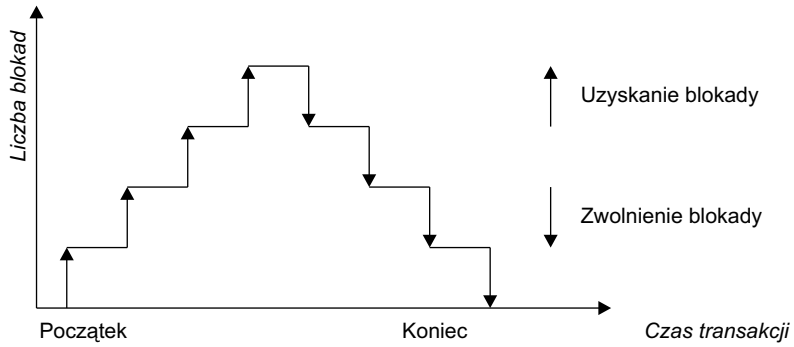
Wykonywanie transakcji powinno odbywać się w taki sposób, aby mogły być one realizowane współbieżnie, nie przeszkadzając sobie nawzajem. Jednym z podstawowych mechanizmów kontroli współbieżności jest nakładanie blokad. Polega to na tym, że transakcja, która chce odczytać bądź zapisać obiekt, musi najpierw zablokować dostęp do niego przed innymi transakcjami. Obsługę blokowania zapewnia najczęściej zarządca, który może być scentralizowany lub rozproszony w postaci zarządców lokalnych na poszczególnych maszynach. Schemat blokowania jest często

rozszerzony o zróżnicowanie blokad nakładanych na obiekty. W najprostszym przypadku można wyróżnić blokady tylko do odczytu i zapisu. Z zagadnieniem blokad związane jest też pojęcie ziarnistości zajmowania, które dotyczy doboru jednostki zajmowania obiektów. W przypadku relacyjnych baz danych jednostką tę może stanowić, np. relacja, wiersz lub atrybut.

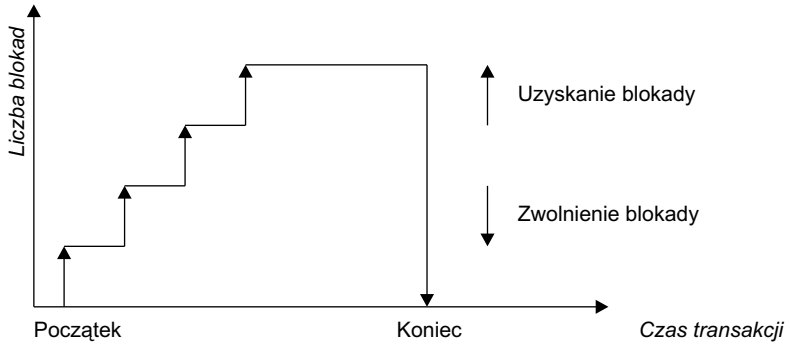
Mechanizmy kontroli dostępu z uwagi na występowanie współbieżności wymagają spełnienia warunku, aby program współbieżny był poprawny przy wszystkich przeplotach [4]. Na podłoże współbieżności transakcji przenosi się to w postaci wymagania, aby przeplatane wykonanie transakcji dało takie same rezultaty, jak wykonanie po jednej na raz. Przedstawione tu kryterium kontroli poprawności współbieżności jest nazywane szeregowalnością. Aby zagwarantować spełnienie warunku szeregowalności zakładanie i zdejmowanie blokach nie może obywać się w sposób dowolny, bo może to prowadzić do niespójności i zakleszczeń. Rozwiązanie tego problemu przedstawia twierdzenie Eswarna dotyczące dwufazowego blokowania *(ang. two - phase locking - 2PL)*, które w [12, str.452] zostało sformułowane następująco: *"Jeżeli wszystkie transakcje spełniają protokół dwufazowego blokowania, to wszystkie przeplatane porządki są szeregowalne"*. Dwufazowe blokowanie można opisać następująco:

- Każda transakcja, która chce rozpocząć działanie na obiekcie musi założyć na nim blokadę.
- Po zwolnieniu chociaż jednej blokady transakcja nie może już więcej zakładać blokad.

Nakładanie blokad zgodnie z tym schematem pozwala zagwarantować szeregowalność transakcji i wyróżnia dwie fazy blokowania. Pierwsza faza obejmuje nakładanie przez transakcję blokad, druga faza rozpoczyna się w momencie zdjęcia pierwszej z nich. Obrazują to rys. 4.1 i rys. 4.2.



Rysunek 4.1: Stopniowe zwalnianie blokad [37]



Rysunek 4.2: Zachowanie blokad do końca transakcji [37]

Rys. 4.1 przedstawia sytuację, kiedy transakcja zwalnia blokady stopniowo, na drugim transakcja zachowuje blokady do momentu zakończenia działania [37].

Omówienie metod blokowania warto rozszerzyć o technikę kontroli współbieżności zwaną blokowaniem wielowariantowym, którą zaproponowano w [3]. Rozwiązanie to pozwala wyeliminować konieczność czekania przez transakcję na operacje czytania, co jest szczególnie istotne w rozproszonych bazach danych. Idea pomysłu tkwi w tym, aby można było na danym obiekcie jednocześnie wykonywać dowolną liczbę operacji czytania i tylko jedną pisania. Rozwiązanie to zgodnie z [12, str. 462] przedstawia się następująco:

- Jeżeli transakcja T2 chce czytać obiekt, do którego aktualnie ma dostęp transakcja T1, to otrzymuje ona dostęp do poprzednio zatwierdzonej wersji tego obiektu. Taka wersja jest przechowywana zwykle w dzienniku transakcji w celu ewentualnego odtworzenia.

- Jeżeli transakcja T2 chce modyfikować obiekt, do którego aktualnie ma dostęp do odczytu transakcja T1, to transakcja T2 otrzymuje do niego dostęp, podczas gdy transakcja T1 zachowuje dostęp do swojej własnej wersji tego obiektu (która teraz jest już poprzednią wersją).

Stosowanie mechanizmów blokowania, jako rozwiązania kontroli dostępu we współbieżnym przetwarzaniu transakcyjnym, prowadzi z kolei do problemu zakleszczeń *(ang. deadlock)*. Ogólnie w przetwarzaniu współbieżnym, jak i rozproszonym, pojęcie to jest rozumiane jako stan, w którym co najmniej dwa procesy czekają wzajemnie na zajście zdarzenia powodowanego przez drugi z nich. Zakleszczenie transakcji w [12, str. 450] przestawiono następująco: *"Zakleszczenie jest to sytuacja, w której dwie lub więcej transakcji znajdują się jednocześnie w stanie wzajemnego oczekiwania na uwolnienie blokady, aby dalsze operacje z tych transakcji mogły być wykonane"*. Warto tu zaznaczyć, że algorytm dwufazowego blokowania też może prowadzić do zakleszczeń.

Szczegółowo omówione algorytmy rozwiązywania problemu zakleszczeń zarówno w systemach scentralizowanych, jak i rozproszonych znajdują się w [12, str. 194

i nast.]. Przedstawiono tam algorytmy nie tylko wykrywania zakleszczeń, ale także zapobiegania ich występowaniu. Jednym z najczęściej omawianych algorytmów wykrywania zakleszczeń jest algorytm, którego idea opiera się na poszukiwaniu cykli w grafie poszukiwań. Graf poszukiwań jest grafem zorientowanym, w którym wierzchołki odpowiadają procesom (transakcjom) a krawędzi oczekiwaniom na zwolnienie zasobu.

Rozdział 5

Przetwarzanie rozproszonych zapytań

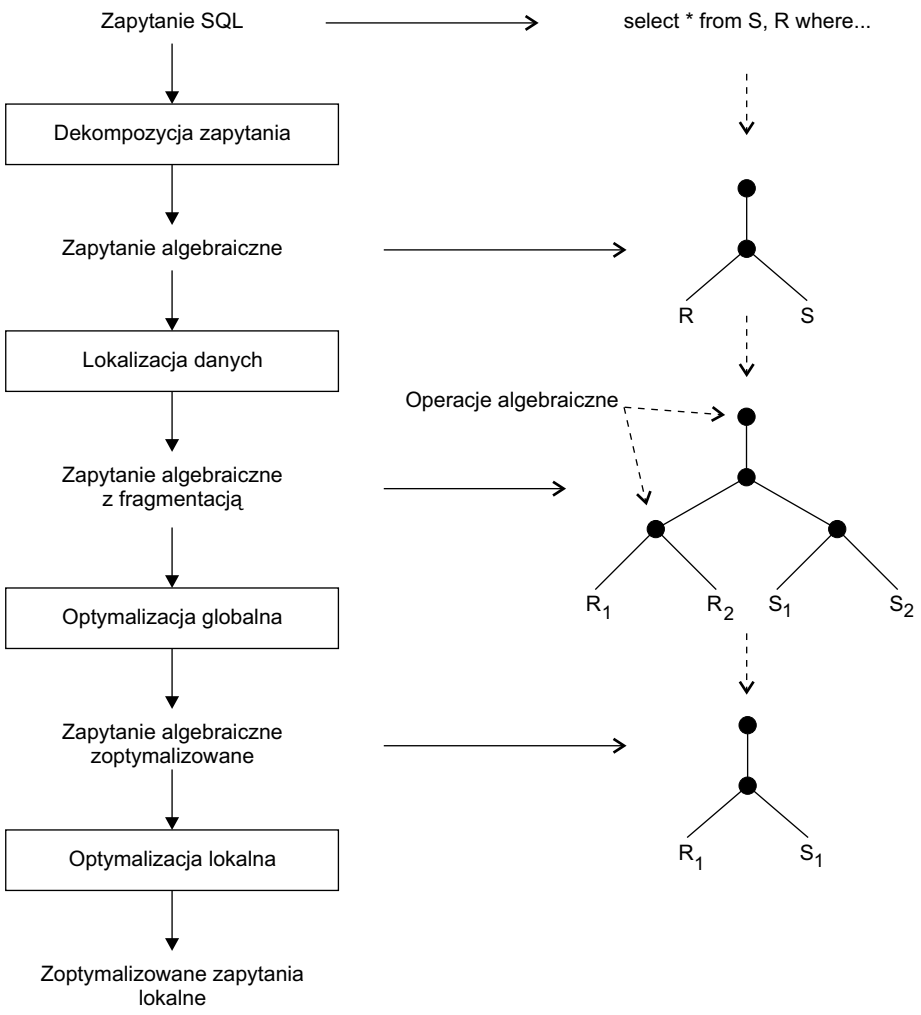
Podstawową funkcjonalnością każdego systemu baz danych jest umożliwienie użytkownikowi zadawania zapytań w języku wysokiego poziomu. Korzystając np. z języka SQL, można określić co się chce uzyskać, a system DBMS, w szczególności jego element zwany procesorem zapytań, znajdzie odpowiedni scenariusz realizacji zapytania.

Procesor zapytań ma krytyczne znaczenie dla wydajności systemów baz danych. Dotyczy to zarówno systemów scentralizowanych, jak i rozproszonych, przy czym w drugim przypadku jego realizacja jest znacznie bardziej skomplikowana ze względu na większą liczbę czynników, które należy wziąć pod uwagę.

Przedstawione rozważania odnoszą się do relacyjnych baz danych. Rola procesora zapytań polega w nich na przekształceniu zapytania wyrażonego w języku wysokiego poziomu, takim jak SQL, do sekwencji operacji na relacjach. Zapytanie wyrażone z pomocą algebry relacji jest następnie poddawane optymalizacji według kryteriów omówionych w rozdziale czwartym (t.j. wg. czasu całkowitego lub czasu odpowiedzi). Ostatnią fazę działania procesora zapytań stanowi wykonanie zapytania.

5.1 Ogólny schemat przetwarzania rozproszonych zapytań

Problem przetwarzania rozproszonych zapytań można sprowadzić do ogólnego modelu warstwowego (rys. 5.1) [46, str. 352]. Model ten odzwierciedla sposób działania procesora zapytań.



Rysunek 5.1: Warstwowy schemat przetwarzania rozproszonych zapytań [46, str. 352]

5.1.1 Dekompozycja zapytania

Warstwa pierwsza dekomponuje zapytanie wyrażone w języku wysokiego poziomu do zapytania wyrażonego za pomocą algebry relacji, które dalej będzie nazywane zapytaniem algebraicznym. Postacią maszynową zapytania algebraicznego jest drzewo operacji algebraicznych.

Zanim zapytanie wejściowe przekształcone zostanie do postaci zapytania algebraicznego poddawane jest ono dodatkowym zabiegom: normalizacji, analizie i eliminacji redundancji.

Normalizacja

Celem normalizacji jest transformacja zapytania do postaci znormalizowanej, która może ułatwić dalsze przetwarzanie. Zabieg ten dotyczy najczęściej warunków logicznych klauzuli WHERE zapytania. Normalna postać koniunkcyjna *(ang. conjunctive normal form)*:

$$(p_{11} \vee p_{12} \vee \ldots p_{1n}) \wedge \ldots \wedge (p_{m1} \vee p_{m2} \vee \ldots p_{mk})$$

może być np. zamieniona na normalną postać alternatywną *(ang. disjunctive normal form)*:

$$(q_{11} \wedge q_{12} \wedge \ldots q_{1l}) \vee \ldots \vee (q_{s1} \wedge q_{s2} \wedge \ldots q_{st})$$

Czynność ta pozwala zdekomponować zapytanie na kilka niezależnych koniunkcyjnych podzapytań, których wyniki łączone są operatorem sumy zbiorów.

Analiza

Analiza ma na celu sprawdzenie poprawności zapytania. Jest ono poddawane kontroli typów oraz sprawdzeniu poprawności semantycznej. Jeśli jeden z atrybutów lub nazwa relacji nie są zdefiniowane w schemacie relacji lub kiedy zachodzi sprzeczność między operacją a typem atrybutu do jakiego ją zastosowano, to takie zapytanie jest odrzucane.

Eliminacja redundancji

Formuły logiczne zawarte w zapytaniach mogą niekiedy zawierać redundantne predykaty. Bezpośrednie wykonywanie zapytań z takimi predykatami może prowadzić do tego, że pewne czynności będę wykonywały się kilkakrotnie, bądź będą wykonywały się zupełnie niepotrzebnie. Eliminacja redundancji odbywa się z wykorzystaniem reguł, których przykłady podano niżej:

$$\begin{aligned} p \wedge p &\Leftrightarrow p \\ p \vee p &\Leftrightarrow p \\ p \vee true &\Leftrightarrow p \\ p \wedge \neg p &\Leftrightarrow false \\ p \vee \neg p &\Leftrightarrow true \\ p_1 \wedge (p_1 \vee p_2) &\Leftrightarrow p_1 \end{aligned}$$

Dekompozycja

Po wykonaniu przedstawionych wyżej czynności wstępnych następuje dekompozycja zapytania do zapytania algebraicznego reprezentowanego przez drzewo operacji algebraicznych. Transformacja ta może być przeprowadzona w następujący sposób:

1. Najpierw tworzone są liście drzewa z relacji zawartych w klauzuli FROM zapytania.

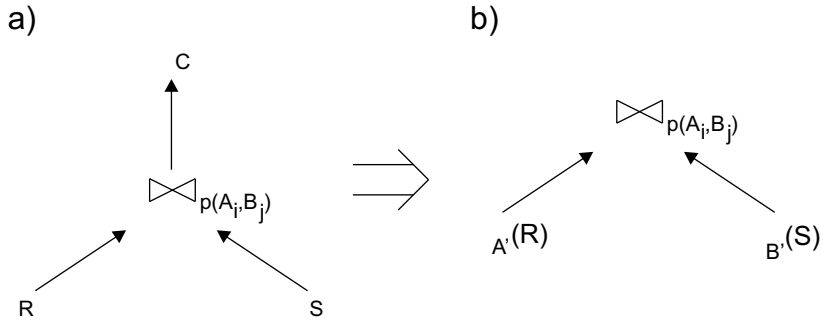
2. Następnie tworzony jest korzeń drzewa z projekcją zawartą w klauzuli SELECT.

3. Ostatnim etapem jest transformacja pozostałych operacji do operacji algebraicznych. Powstałe operacje wstawiane są w wewnętrznych węzłach tworzonego drzewa.

Otrzymane w powyższy sposób zapytanie algebraiczne może być już na tym etapie (jeszcze przed wykonaniem lokalizacji danych) poddane wstępnej optymalizacji. Wykorzystując jedną z sześciu reguł transformacji relacyjnych zawartych w [37, str. 196 i 197] o postaci:

$$\Pi_C(R \bowtie_{p(A_i,B_j)} S) \Leftrightarrow \Pi_{A'}(R) \bowtie_{p(A_i,B_j)} \Pi_{B'}(S)$$

gdzie Π jest operacją projekcji; C, A' i B' są zbiorami atrybutów takimi, że $C = A' \cup B'$; R i S są to relacje, których zbiory atrybutów to odpowiednio A i B; $\bowtie_{p(A_i,B_j)}$ jest złączeniem relacji z warunkiem p obejmującym atrybuty A_i i B_j, można zapytanie reprezentowane przez drzewo operacji algebraicznych z rys. 5.2a zastąpić drzewem przedstawionym na rys. 5.2b. Drzewo uzyskane w ten sposób stanowi bardziej optymalny scenariusz realizacji zapytania, ponieważ operacje projekcji wykonują się przed operacją złączenia.



Rysunek 5.2: Optymalizacja zapytania: a) drzewo przed optymalizacją, b) drzewo po optymalizacji

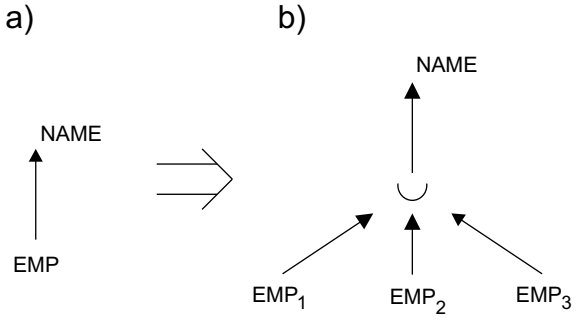
5.1.2 Lokalizacja danych

Zapytanie algebraiczne utworzone w poprzednim kroku odnosi się do całych relacji. Ponieważ relacje w rozpatrywanej bazie danych są rozproszone, konieczne jest, aby zapytanie algebraiczne wyrazić nie w kategoriach całych relacji, ale ich fizycznych fragmentów. Czynność ta wykonywana jest przez warstwę lokalizacji danych.

Bezpośredni algorytm przeprowadzenia lokalizacji danych polega na zastąpieniu każdej relacji operacją złożenia tej relacji z fizycznych fragmentów rozproszonych po węzłach systemu. Podejście to może być realizowane przez zastępowanie liści drzewa operacji algebraicznych poddrzewami reprezentującymi złożenie relacji z fragmentów. Dla podziału relacji $EMP(ID, NAME, MGR)$ przedstawionego w przykładzie 5.1:

$$\begin{aligned} EMP_1 &= \sigma_{ID\leqslant 3}(EMP) \\ EMP_2 &= \sigma_{3<ID\leqslant 6}(EMP) \\ EMP_3 &= \sigma_{ID>6}(EMP) \end{aligned} \tag{5.1}$$

gdzie σ jest operacją selekcji, zapytanie algebraiczne z rys 5.3a, które uwzględnia podział danych może być wyrażone tak jak na rys 5.3b.



Rysunek 5.3: Zapytanie: a) bez lokalizacji danych, b) z lokalizacją danych

Zaprezentowane tu rozwiązanie może być niewystarczające, gdyż często pozostają jeszcze nieefektywności, które należy usunąć. W tym celu stosuje się techniki redukcyjne, które generują prostsze i bardziej zoptymalizowane zapytania.

Redukcja fragmentacji poziomych

Fragmentacja pozioma (przykład 5.1) opiera się na operacji selekcji. Pozwala to na przeprowadzenie optymalizacji w przypadku, kiedy relacja podzielona poziomo jest poddawana selekcji. Przykładowe zapytanie:

```
SELECT * FROM EMP WHERE ID=5
```

wyrażone zapytaniem algebraicznym $\sigma_{ID=5}(E_1 \cup E_2 \cup E_3)$ może być zoptymalizowane do postaci $\sigma_{ID=5}(E_2)$.

Podobne możliwości optymalizacji wykazuje zastosowanie operacji złączenia. Uwzględniając podział relacji $SAL(ID, SALARY, TAX)$ według przykładu 5.2

$$\begin{aligned} SAL_1 &= \sigma_{ID\leqslant 3}(SAL) \\ SAL_2 &= \sigma_{ID>3}(SAL) \end{aligned} \tag{5.2}$$

zapytanie o treści

```
SELECT * FROM EMP, SAL WHERE EMP.ID = SAL.ID
```

można z postaci $(EMP_1 \cup EMP_2 \cup EMP_3) \bowtie_{ID} (SAL_1 \cup SAL_2)$, przy założeniu, że relacje EMP_1, SAL_1 oraz EMP_2, EMP_3 i SAL_2 znajdują się odpowiednio na tym samym komputerze, zoptymalizować redukując je do postaci $(EMP_1 \bowtie_{ID} SAL_1) \cup (EMP_2 \bowtie_{ID} SAL_2) \cup (EMP_3 \bowtie_{ID} SAL_2)$.

Redukcja fragmentacji pionowych

Fragmentacja pionowa rozprasza relacje, bazując na projekcji ich atrybutów. Rekonstrukcja logicznych relacji odbywa się w tym przypadku z wykorzystaniem operatora złączenia (przykład 5.3).

$$\begin{aligned} EMP_1 &= \Pi_{ID,NAME}(EMP) \\ EMP_2 &= \Pi_{ID,MGR}(EMP) \\ EMP &= EMP_1 \bowtie_{ID} EMP_2 \end{aligned} \tag{5.3}$$

Podobnie jak w przypadku fragmentacji poziomych zapytania wyrażone na fragmentach pionowych mogą być redukowane przez usuwanie poddrzew produkujących bezużyteczne fragmenty relacji. Przykładem jest zapytanie:

```
SELECT NAME FROM EMP
```

wyrażone na relacji EMP podzielonej zgodnie z przykładem 5.3, które z postaci $\Pi_{NAME}(EMP_1 \bowtie_{ID} EMP_2)$ może być zredukowane do $\Pi_{NAME}(EMP_1)$.

5.1.3 Optymalizacja globalna

Zapytanie algebraiczne uzyskane w wyniku działania dwóch omówionych wyżej faz i reprezentowane przez drzewo operacji algebraicznych stanowi tylko jeden z wielu scenariuszy wykonania zapytania wejściowego. Znalezienie drzewa, które reprezentuje optymalny porządek wykonania operacji na danych, stanowi zadanie optymalizatora zapytań. Jak podaje [37, str. 210] czynność ta jest trudna do rozwiązania obliczeniowo *(ang. computationally intractable)*. Znalezienie optymalnego planu jest problemem klasy NP *(ang. Non-deterministic Polynomial-time hard)* [29, str. 2], dlatego celem optymalizacji jest znalezienie rozwiązania bliskiego optimum lub co najmniej uniknięcie złych strategii.

Zagadnienia optymalizacji wymagają określenia kryterium względem, którego problem może być optymalizowany. W przypadku baz danych wyróżnia się dwa modele kosztu:

- koszt całkowity *(ang. total cost)* - koszt jest sumą kosztów wykorzystania wszystkich komponentów realizujących zapytanie;
- czas odpowiedzi *(ang. response time)* - koszt stanowi czas upływający od inicjacji do zakończenia zapytania.

W scentralizowanych bazach danych funkcja kosztu uwzględnia czas operacji wejścia-wyjścia oraz wykorzystania procesora. W bazach rozproszonych bierze się pod uwagę dodatkowo czas komunikacji w sieci komputerowej.

Istotnym czynnikiem optymalizacji zapytań (nie tylko rozproszonych) jest jak najdokładniejsza wiedza o rozmiarach przetwarzanych relacji wejściowych, a także relacji będących wynikami operacji pośrednich. W tym celu gromadzi się informacje statystyczne o stanie bazy danych lub pobiera się aktualne informacje bezpośrednio z niej. Dane statystyczne są często podstawą określania liczności pośrednich wyników operacji na relacjach. Dla tych wyników stosuje się także techniki przewidywania ich liczności. W [37, str. 213] podano wzory pozwalające wyznaczać liczności relacji wynikowych dla operacji algebry relacji.

5.1.4 Optymalizacja lokalna

Każde zapytanie lokalne jest optymalizowane w ramach bazy danych, na której jest ono wykonywane. Optymalizacja ta odbywa się z wykorzystaniem algorytmów przetwarzania zapytań stosowanych w systemach scentralizowanych.

Wydajność przetwarzania zapytań pobierających dane bezpośrednio z miejsc ich fizycznego składowania (tak jak w przypadku zapytań lokalnych) zależy w znacznym stopniu od właściwie prowadzonych czynności administracyjnych oraz właściwie przeprowadzonego etapu projektowania. Znaczenie tych czynników podkreśla przykład konfiguracji przedstawiony w [34, str. 116 i nast], gdzie w opisie projektowania fizycznego układu bazy danych w oparciu o DBMS Oracle 8*i* podano, że najbardziej optymalne w tym przypadku jest rozłożenie danych na 22 dyskach twardych (przedstawiane są również rozwiązania bardziej oszczędne ze względu na liczbę dysków, ale mimo to wymaganych jest co najmniej siedem dysków twardych). Zabieg ten ma na celu zrównoleglenie dostępu do danych, kiedy znajdują się one na różnych dyskach. Szczególnie efektywne może być przechowywanie na różnych dyskach indeksów i danych, do których się one odnoszą. Organizowanie dostępu do danych obejmuje również bardzo istotny element jakim jest zastosowanie pamięci podręcznej.

5.2 Optymalizacja rozproszonych zapytań

Faza optymalizacji ma duże znaczenie w całym procesie przetwarzania zapytań. Istnieje wiele różnych algorytmów realizujących to zadanie. Przed omówieniem koncepcji kilku z nich podane zostaną dwie metody ich klasyfikacji. Pierwsza dzieli techniki optymalizacji zapytań względem zależności między etapem optymalizacji a wykonaniem zapytania:

- statyczne - optymalizacja przeprowadzana jest przed rozpoczęciem wykonania zapytania; konieczna jest estymacja rozmiarów pośrednich wyników, ale raz przeprowadzony etap optymalizacji może być wykorzystywany wielokrotnie;

- dynamiczne - optymalizacja odbywa się już w trakcie wykonania zapytania; możliwe jest uwzględnienie pośrednich rozmiarów relacji;
- hybrydowe - istnieje wiele koncepcji łączenia cech algorytmów statycznych i dynamicznych; jedna z nich wykorzystuje optymalizację statyczną przed rozpoczęciem wykonania, a jeżeli w jego trakcie błąd przewidywanych rozmiarów danych przekroczy określony próg to przeprowadzana jest korekta.

W [29, str. 3 i 4] podano inną klasyfikację algorytmów optymalizacji zapytań:

- przeszukiwanie wyczerpujące - wszystkie algorytmy tego typu mają wykładniczą złożoność czasową i pamięciową, ale gwarantują znalezienie najlepszego planu względem zadanego modelu kosztu; realizują one sprawdzanie całej przestrzeni rozwiązań;
- heurystyki - są to algorytmy, które bazując na określonych heurystykach mają z reguły wielomianową złożoność czasową; dają rozwiązania, które mogą być mniej optymalne niż te uzyskane wyszukiwaniem wyczerpującym, ale są często jedynym sposobem uzyskania rozwiązania w akceptowalnym czasie, kiedy rozmiar danych jest zbyt duży dla algorytmów wyczerpujących;
- algorytmy probabilistyczne - zasadniczą zaletą algorytmów tej klasy jest stała złożoność pamięciowa; czas działania większości z nich nie może być przewidziany, ponieważ są one niedeterministyczne.

5.2.1 Algorytm rozproszonego systemu INGRES

Koncepcja dynamicznej optymalizacji rozproszonych zapytań przedstawiona zostanie na przykładzie algorytmu z rozproszonego systemu INGRES.

Idea tego algorytmu opiera się na dzieleniu zapytania wejściowego na podzapytania. Wykonuje on najpierw operacje jednoargumentowe i próbuje następnie zminimalizować rozmiary wyników pośrednich, porządkując operacje binarne.

Dekompozycja zapytania odbywa się z wykorzystaniem dwóch podstawowych technik: odrywania *(ang. detachment)* i podstawiania *(ang. substitution)*.

Odrywanie

Odrywanie jest pierwszym krokiem wykonywanym przez procesor zapytań. Polega ono na dekompozycji zapytania q na dwa podzapytania q' i q'', gdzie zapis $q' \leftarrow q''$ oznacza, że q' jest wykonywane wcześniej, a jego rezultat jest wykorzystywany przez q''. Przebieg odrywania ilustruje przykład:
q :

```
SELECT EMP.NAME
FROM EMP, SAL, PROJ
```

```
WHERE EMP.ID = SAL.ID
AND SAL.IP = PROJ.IP
AND PROJ.SUB = 'CAD'
```

q' :

```
SELECT PROJ.IP INTO IPROJ
FROM PROJ
WHERE PROJ.SUB = 'CAD'
```

q'' :

```
SELECT EMP.NAME
FROM EMP, SAL, IPROJ
WHERE EMP.ID = SAL.ID
AND SAL.IP = IPROJ.IP
```

Odrywanie polega na wydzieleniu podzapytania odwołującego się tylko do jednej tabeli. W przedstawionym przykładzie, wykonanie podzapytania na relacji $PROJ$ pozwala zredukować rozmiar dalej przetwarzanych danych.

Podstawianie

Zapytania wieloargumentowe, do których nie można więcej zastosować reguły odrywania są konwertowane do zapytań jednoargumentowych. Odbywa się to przez podstawianie wartości krotek. W zapytaniu n-argumentowym zmienne jednej z relacji są zastępowane przez ich wartości. Powstaje w ten sposób zbiór zapytań $(n-1)$-argumentowych. Liczność tego zbioru równa jest liczbie krotek relacji, która poddana została procesowi podstawiania. Ogólnie koncepcję tą można przedstawić jako zastąpienie zapytania $q(V_1, V_2, \ldots, V_n)$ przez zbiór zapytań $\{q_i(V_1, V_2, \ldots, t_{ij}, \ldots, V_n), t_{ij} \in R_k\}$, gdzie V_i są to atrybuty ze zbioru relacji, których dotyczy zapytanie; $i = 1 \ldots n$ jest liczbą atrybutów; $k = 1 \ldots m$ jest liczbą relacji R_k zapytania taką, że $m \leqslant n$; t_{ij} jest j - tą wartością atrybutu V_i z czego wynika, że $j = 1 \ldots card(V_i)$.

Zakładając, że relacja SAL ma dwie krotki, w których $ID = S1$ albo $ID = S2$, realizacja podstawiania może wyglądać jak na poniższym przykładzie.
q :

```
SELECT EMP.NAME
FROM EMP, SAL
WHERE SAL.ID=EMP.ID
```

q_1 :

```
SELECT EMP.NAME
FROM EMP
WHERE EMP.ID = 'S1'
```

q_2 :

```
SELECT EMP.NAME
FROM EMP
WHERE EMP.ID = 'S2'
```

Omawiany proces podstawiania musi być odpowiednio optymalizowany. Najpierw wybierane są relacje, które powodują najmniejsze wyniki pośrednie.

Algorytm INGRES

Input: MVQ zapytanie wieloargumentowe
Output: wynik zapytania
begin
  for each OVQ_i **in** MVQ **do** (1)
    run(OVQ_i) {wykonanie zapytania jednoargumentowego}
  end-for
  {zastąpienie MVQ przez n nieredukowalnych zapytań}
  $MVQ'\_list \leftarrow$ REDUCE(MVQ) (2)
  while $n \neq 0$ **do** (3)
    {wybranie nieredukowalnego zapytania z najmniejszymi fragmentami nie mającego poprzedników}
    $MVQ' \leftarrow$ SELECT_QUERY(MVQ'_list) (3.1)
    {wyznaczenie fragmentów do przesłania i węzła przetwarzającego zapytanie MVQ'}
    Fragment-site-list$\leftarrow$ SELECT_STRATEGY(MVQ') (3.2)
    {przesłanie wybranych fragmentów do wyznaczonych węzłów}
    for each pair(F,S) **in** Fragment-site-list **do** (3.3)
      move fragment F to site S
    end-for
    run(MVQ') (3.4)
    $n \leftarrow n - 1$
  end-while {wyjściem algorytmu jest wynik ostatniego MVQ'}
end.

Rysunek 5.4: Algorytm INGRES [37, str. 232]

Symbole MVQ i OVQ_i oznaczają odpowiednio zapytanie wielo i jednoargumentowe. W kroku (1) przeprowadzane jest odrywanie i wykonanie zapytań jednoargumentowych, jakie dają się wyłonić w zapytaniu wejściowym. Kolejnym etapem jest redukcja zapytania w kroku (2) na n nieredukowalnych zapytań. Zapytania te są następnie wykonywane w krokach (3.1) do (3.4). W kroku (3.1) wybierane jest kolejne zapytanie do przetwarzania, które spowoduje powstanie najmniejszych wyników pośrednich. W kroku (3.2) tworzona jest strategia wykonania wybranego zapytania.

Strategia ta opisywana jest za pomocą par F i S, gdzie F oznacza fragment relacji, który musi być przeniesiony na stronę S. Krok (3.3) przenosi wszystkie fragmenty zgodnie ze strukturą par F i S, a w kroku (3.4) następuje już wykonanie zapytania. Jeżeli pozostają w tym momencie jeszcze jakieś niewykonane zapytania to algorytm wraca do punktu (3) i wykonuje kolejną iterację.

Optymalizacja dynamiczna jest korzystna ze względu na fakt, że jest znany aktualny rozmiar danych. Dla każdego wywołania zapytania, cały proces optymalizacji jest powtarzany od nowa.

Omawiany tu algorytm systemu INGRES może w zależności od zastosowanej funkcji kosztu optymalizować czas całkowity lub czas odpowiedzi.

5.2.2 Algorytm systemu R*

System R* wykonuje statyczną optymalizację zapytań, bazując na przeszukiwaniu całej przestrzeni rozwiązań. Na wejściu do tego algorytmu podawane jest drzewo algebry relacyjnej uzyskane z dekompozycji zapytania SQL. Na wyjściu zwracane jest drzewo, które stanowi optymalny plan wykonania zapytania.

input: QT drzewo algebraiczne zapytania na relacjach $R_1, \ldots, R_n$
output: drzewo algebraiczne z minimalnym kosztem
begin
  for each relacja $R_i \in QT$ **do** (1)
    {wyznaczenie kosztu metod dostępu do relacji R_i}
    for each ścieżka dostępu AP_{ij} **to** R_i **do** (1.1)
      cost (AP_{ij}) {oszacowanie kosztu}
    end-for
    {wyznaczenie metody dostępu z minimalnym kosztem do relacji R_i}
    $best\_AP_i \leftarrow AP_{ij}$ (1.2)
  end-for
  {wyznaczenie kosztu wszystkich możliwych złączeń relacji}
  for each porządek $(R_{1i}, R_{2i}, \ldots, R_{ni})$ **with** $i = 1, \ldots, n!$ **do** (2)
    utwórz strategię $(\ldots((best\_AP_{ti} \bowtie R_{2i}) \bowtie R_{3i}) \bowtie \ldots \bowtie R_{ni})$
    wyznacz koszt strategii
  end-for
  {wybranie najlepszego uporządkowania złączeń}
  output $\leftarrow$ strategia z minimalnym kosztem (2.1)
  for each węzeł k przechowujący relacje z QT **do** (3)
    $LS_k \leftarrow$ strategia lokalna(strategy, k)
    send(LS_k, site k) {każda strategia lokalna jest optymalizowana w węźle k}
  end-for
end.

Rysunek 5.5: Algorytm R* [37, str. 236]

Optymalizator wyznacza wartość kosztu dla każdego sprawdzanego drzewa. Na wyjściu zwracane jest to drzewo, dla którego koszt jest najmniejszy. Sprawdzane drzewa uzyskiwane są z permutacji złączeń wykonywanych na n relacjach zapytania. Strategie, w których pojawiają się iloczyny kartezjańskie są eliminowane od razu, ponieważ prowadzą one z reguły do powstawania większych wyników pośrednich.

Optymalizacja w przedstawionym na rys. 5.5 algorytmie składa się z trzech etapów reprezentowanych przez trzy jego główne pętle (1), (2) i (3).

- W pierwszej kolejności (1) dla każdej relacji występującej w zapytaniu wyszukiwana jest najlepsza metoda dostępu (tzn. z minimalnym kosztem). Duże znaczenie mają tu indeksy. Pętla (1.1) szacuje koszt wszystkich sposobów dostępu do relacji R_i. Instrukcja (1.2) pozostawia metodę dostępu o najmniejszym koszcie.

- W kolejnym etapie (2) dla każdej relacji R_t estymowany jest koszt najlepszego uporządkowania operacji złączenia, kiedy dana relacja R_t jest pobierana jako pierwsza najlepszą metodą wyznaczoną w kroku poprzednim. Najlepsze pod względem kosztu uporządkowanie stanowi najkorzystniejszy plan wykonania wybierany instrukcją (2.1). Uporządkowania, które prowadzą do operacji iloczynu kartezjańskiego kosztem operacji złączenia są eliminowane od razu jeszcze w pętli (2). Sytuację tego typu obrazuje przykład:

  ```
  SELECT EMP.NAME
  FROM EMP, SAL, PROJ
  WHERE EMP.ID = SAL.ID
  AND SAL.IP = PROJ.IP
  AND PROJ.SUB = 'CAD'
  ```

  Zakładając, że tabela EMP ma indeks na ID, SAL ma indeks na IP, a $PROJ$ na IP i SUB optymalny scenariusz wykonania zapytania wygląda następująco: $((PROJ \bowtie SAL) \bowtie EMP)$. Natychmiast odrzucony powinien być scenariusz $((EMP \times PROJ) \bowtie SAL)$, który zawiera iloczyn kartezjański.

- Ostatnim etapem (3) jest wyznaczenie zadań dla lokalnych baz danych. Każda z nich po otrzymaniu do realizacji swojej części zapytania generuje lokalny plan dostępu do danych.

Optymalizacja statyczna wykonywana przed rozpoczęciem realizacji zapytania musi uwzględniać informacje statystyczne o rozmiarach danych. Raz przeprowadzona optymalizacja może być wykorzystywana przy wielokrotnych jego wywołaniach.

Omawiany tu algorytm systemu R* optymalizuje koszt całkowity wykonania zapytania, włączając przetwarzanie lokalne i koszty komunikacji.

5.2.3 Algorytm zachłanny

Przy rozwiązywaniu problemu optymalizacji zapytań zastosowanie znajduje także algorytm zachłanny *(ang. greedy)*. Ponieważ nie przeszukuje on całej przestrzeni rozwiązań, działa znacznie szybciej niż algorytmy wyczerpujące. Z reguły nie prowadzi on do uzyskania najlepszego rozwiązania, ale w porównaniu z algorytmem wyczerpującym może okazać się jedyną metodą uzyskania rozwiązania w akceptowalnym czasie. Ogólną jego postać przedstawia rys. 5.6.

input: QT drzewo algebraiczne zapytania na relacjach $R_1, \ldots, R_n$
output: wyznaczony plan wykonania
begin
  for each relation $R_i \in QT$ **do** (1)
    {wyznaczenie kosztu metod dostępu do relacji R_i}
    for each access path AP_{ij} **to** R_i **do**
      cost (AP_{ij}) {oszacowanie kosztu}
    end-for
    {wyznaczenie optymalnego planu dostępu do relacji R_i}
    optPlan($\{R_i\}$) $\leftarrow AP_{ij}$
  end-for
  toDo = $\{R_1, \ldots, R_n\}$
  {budowanie rozwiązania wybierając najlepsze rozwiązanie danej iteracji}
  for $i = 1$ **to** $n - 1$ **do** (2)
    {znalezienie takiej pary $O, I \in$ toDo, że złączenie jej relacji daje minimalny wynik, ze wszystkich par znajdujących się w toDo}
    find $O, I \in$ toDo **such that** $P =$ {min. koszt wszystkich par} (2.1)
    {zastąpienie znalezionej pary O, I przez nową relację τ będącą wynikiem złączenia O oraz I}
    toDo = toDo - $\{O, I\} \cup \{\tau\}$ (2.2)
    {uwzględnienie złączenia jako blok budujący rozwiązanie}
    optPlan($\{\tau\}$) = $\{P\}$ (2.3)
    delete(optPlan(O)) (2.4)
    delete(optPlan(I)) (2.5)
  end-for
  return optPlan($\{R_1, \ldots, R_n\}$) (3)
end.

Rysunek 5.6: Algorytm zachłanny [29, str. 10]

Algorytm ten tworzy plan wykonania zapytania od dołu do góry drzewa *(ang. bottom up way)*. Podobnie jak w algorytmie systemu R* pierwszą fazą działania (1) jest określenie najlepszej metody dostępu dla każdej z relacji występującej w zapytaniu. W fazie drugiej (2) w każdym przebiegu pętli wybierana jest (2.1) najlepsza do wykonania w danym momencie operacja złączenia. Relacje wybrane do złączenia

są w krokach (2.2) do (2.5) zastępowane przez to złączenie, które jest dalej uwzględniane jako blok budujący rozwiązanie końcowe.

5.2.4 Programowanie dynamiczne i interaktywne programowanie dynamiczne w optymalizacji zapytań

Algorytm systemu R* w formie przedstawionej na rys. 5.5 w sposób wyczerpujący przeszukuje przestrzeń rozwiązań. Dzięki wykorzystaniu programowania dynamicznego algorytm ten może być zrealizowany w sposób efektywniejszy. Zmodyfikowana (usprawniona) wersja tego algorytmu przedstawiona jest na rys. 5.7.

input: QT drzewo algebraiczne zapytania na relacjach $R_1, \ldots, R_n$
output: drzewo algebraiczne z minimalnym kosztem
begin
  for each relation $R_i \in QT$ **do** (1)
    {wyznaczenie kosztu metod dostępu do relacji R_i}
    for each access path AP_{ij} **to** R_i **do**
      cost (AP_{ij}) {oszacowanie kosztu}
    end-for
    {wyznaczenie optymalnego planu dostępu do relacji R_i}
    optPlan($\{R_i\}$) $\leftarrow AP_{ij}$
  end-for
  {tworzenie bloków budujących o rozmiarze i}
  for $i = 2$ **to** n **do** (2)
    {znajdowanie uporządkowań wszystkich podzbiorów i - elementowych zbioru relacji $R_1, \ldots, R_n$}
    for all $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do** (2.1)
    optPlan(S) = $\emptyset$
      {tworzenie uporządkowań w zbiorze relacji S}
      for all $O \subset S$ **do** (2.2)
        optPlan(S)=optPlan(S) $\cup$ joinPlans(optPlan(O), optPlan($S-O$))
        {odrzucenie niekorzystnych planów}
        prunePlans(optPlan(S))
      end-for
    end-for
  end-for
  return optPlan($\{R_1, \ldots, R_n\}$)
end.

Rysunek 5.7: Algorytm oparty na programowaniu dynamicznym [30, str. 8]

Algorytm ten przeszukuje przestrzeń rozwiązań, budując rozwiązanie w kierunku od dołu do góry. Rozważa wszystkie możliwe uporządkowania operacji złączenia

relacji zapytania. Podobnie jak w omawianych wyżej algorytmach ten również rozpoczyna działanie od określenia najlepszych metod dostępu do każdej z relacji występującej w zapytaniu (1). Zgodnie z drugą fazą algorytmu (2), w pierwszej kolejności wyznaczane są złączenia par tabel w oparciu o najlepsze metody dostępu do danych z tych tabel. Dalsze przetwarzanie obejmuje znajdowanie najlepszych uporządkowań operacji złączenia dla trójek tabel, w oparciu o dwójki i metody dostępu do pojedynczych tabel. Postępowanie to jest kontynuowane do momentu, aż zostanie znalezione najlepsze uporządkowanie operacji złączenia n tabel w oparciu o wszystkie uporządkowania do liczby $n-1$ tabel. W każdym kroku tej fazy tworzone są coraz bardziej złożone fragmenty rozwiązania w oparciu o prostsze fragmenty uzyskane wcześniej. Dla zadanej wartości i pętla (2.1) przebiega wszystkie i - elementowe podzbiory S zbioru relacji $R_1, \ldots, R_n$. Dla każdego z tych podzbiorów pętla (2.2) generuje plany złączeń relacji z danego podzbioru i pozostawia najlepsze z nich.

Omawiany tu algorytm pozwala znaleźć najlepsze rozwiązanie, lecz wykładniczy koszt czasowy i pamięciowy prowadzą do istotnego obciążenia w przypadku bardziej złożonych zapytań. Dotyczy to szczególnie zapytań odwołujących się do dużej liczby tabel. Inne algorytmy, które mają dużo mniejszą złożoność, nie są w stanie dać zadowalającego rozwiązania, tzn. odpowiednio bliskiego optimum. Rozwiązanie łączące cechy algorytmu zachłannego i algorytmu wykorzystującego programowanie dynamiczne stanowi algorytm interaktywnego programowania dynamicznego *(ang. Interactive Dynamic Programming - IDP)*. Dzięki połączeniu dwóch koncepcji algorytm IDP daje dla prostych zapytań, tak jak algorytm wyczerpujący, rozwiązanie najlepsze. Jeśli zapytanie jest zbyt złożone dla algorytmu programowania dynamicznego, IDP daje rozwiązanie znacząco lepsze niż inne algorytmy przeznaczone do tego typu zapytań.

Algorytm IDP (rys. 5.8) działa zasadniczo w ten sam sposób co algorytm wykorzystujący programowanie dynamiczne. Różnicą jest to, że IDP uwzględnia ograniczoność zasobów komputera (np. pamięci) oraz to, że użytkownik lub program może chcieć ograniczyć czas przeznaczony na optymalizację zapytania.

Algorytm IDP zakłada, że komputer ma wystarczającą ilość pamięci, aby pomieścić uporządkowania operacji złączenia do liczby k, ale nie większej. W takim razie zapytanie o n relacjach, w którym $n > k$ nie może być optymalizowane na bazie programowania dynamicznego, gdyż wymagane są większe zasoby lub konieczne jest stronicowanie pamięci na dysk. W takim przypadku algorytm IDP znajduje najpierw uporządkowania złączeń do liczby k korzystając z programowania dynamicznego. Realizowane jest to w pętli (2.1), która działa tak jak pętla (2) z rys. 5.7. Różnicą jest to, że po osiągnięciu granicy k algorytm IDP zamiast szukać uporządkowania dla relacji w liczbie $k+1$, co spowodowałoby jego załamanie, znajduje sekwencją instrukcji (2.2) najlepsze z dotychczas uzyskanych rozwiązań dla k relacji i uwzględnienia je jako blok budujący rozwiązanie końcowe. W kolejnych iteracjach tworzone są uporządkowania $k+1, k+2, \ldots$ Działania te są podobne do algorytmu zachłannego, w którym bloki budujące tworzone są z dwóch relacji. W algorytmie IDP tworzy się je z k relacji. Wynika z tego, że dla $k = 2$ algorytm IDP zachowuje

się jak algorytm zachłanny.

input: QT drzewo algebraiczne zapytania na relacjach $R_1, \ldots, R_n$, rozmiar bloku k
output: wyznaczony plan wykonania
begin
  for each relation $R_i \in QT$ **do** (1)
    for each access path AP_{ij} **to** R_i **do**
      cost (AP_{ij})
    end-for
    optPlan($\{R_i\}$) $\leftarrow AP_{ij}$
  end-for
  toDo = $\{R_1, \ldots, R_n\}$
  while |toDo| > 1 **do** (2)
    k = min{k, |toDo|}
    {tworzenie bloków budujących o rozmiarze k}
    for $i = 2$ **to** k **do** (2.1)
      for all $S \subseteq$ toDo **such that** $|S| = i$ **do**
        optPlan(S) = $\emptyset$
        for all $O \subset S$ **do**
        optPlan(S)=optPlan(S) $\cup$ joinPlans(optPlan(O), optPlan($S - O$))
        prunePlans(optPlan(S))
        end-for
      end-for
    end-for
    {znalezienie takiego bloku V, którego koszt P jest najmniejszy}
    find P, V **with** $P \in$ optPlan(V), $V \subseteq$ toDo, $|V| = k$ **such that** (2.2)
      P = {min. koszt wszystkich uporządkowań k-elemntowych zbiorów}
    {zastąpienie zbioru V przez nową relację τ będącą wynikiem złączenia relacji z tego zbioru względem uporządkowania dającego koszt P}
    optPlan($\{\tau\}$) = $\{P\}$
    {uwzględnienie τ jako blok budujący rozwiąznie}
    toDo = toDo $-V \cup \{\tau\}$
    for all $O \subseteq V$ **do** delete(optPlan(O))
  end-while
  prunePlans(optPlan($R_1, \ldots, R_n$))
  return optPlan($R_1, \ldots, R_n$)
end.

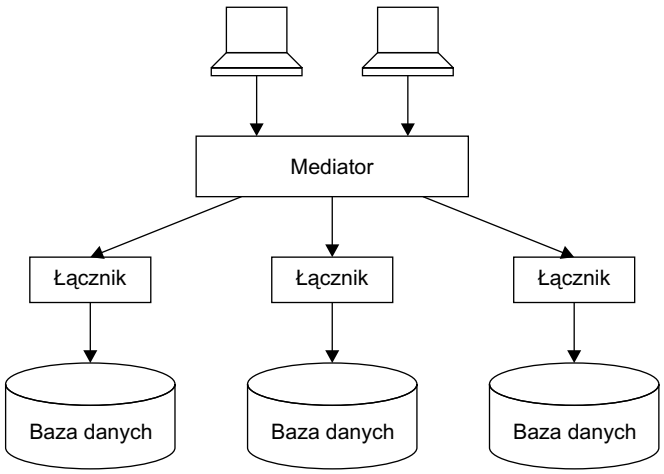
Rysunek 5.8: Algorytm IDP [29, str. 12]

Działanie algorytmu IDP w znaczącym stopniu zależy od wartości k. Może być ona ustalona zarówno przez użytkownika w celu np. zredukowania czasu optymalizacji, jak też algorytm IDP może sam znaleźć górną granicę k na podstawie ograniczenia zasobów. Oczywiście im wyższe k tym większe zapotrzebowanie na zasoby i

tym lepszy jest plan zapytania generowany przez ten algorytm, ponieważ zbliża się on wtedy do algorytmu wyczerpującego.

5.3 Przetwarzanie zapytań w systemach heterogenicznych

Heterogeniczne bazy danych są szczególnym przypadkiem baz rozproszonych. Przetwarzanie zapytań w tego typu bazach danych nie różni się znacząco od mówionego wyżej ogólnego schematu przetwarzania zapytań w bazach homogenicznych. Systemy heterogeniczne posiadają zwykle dodatkową warstwę oprogramowania (rys. 5.9) zwanego mediatorem [30] lub integratorem [37], które pozwala użytkownikowi widzieć rozproszone bazy danych jako jedną i spójną całość. Mediator pobiera zapytania, dokonuje optymalizacji globalnej i odwołuje się do poszczególnych baz danych, które najczęściej mają własne procesory zapytań. Lokalne bazy danych mogą w ten sposób zachowywać autonomię. Główną cechą heterogenicznych baz danych jest również to, że poszczególne bazy danych takiego systemu mogą mieć różne modele danych. W takim przypadku konieczne jest istnienie łączników pomiędzy systemami DBMS a mediatorem (rys. 5.9).



Rysunek 5.9: Znaczenie łącznika w heterogenicznej bazie danych

Wymienione wyżej cechy heterogenicznych baz danych mają swoje odzwierciedlenie w możliwościach przetwarzania zapytań.

- Z faktu, iż poszczególne bazy danych zachowują lokalną autonomię oraz mogą mieć różne modele danych wynikają ograniczenia możliwości przesyłania danych między nimi. Zabiegi tego typu są często wykorzystywane w przetwarzaniu rozproszonych zapytań i pozwalają uzyskać znaczącą poprawę wydajności. Do zabiegów tych można zaliczyć operację `semi-join`. Jest ona sposobem

realizacji złączenia, który unika przesyłania całych relacji pomiędzy węzłami systemu. Idea pomysłu tkwi w tym, aby przesłać z węzła A do węzła B tylko te kolumny, które występują w warunku złączenia. Wówczas po wybraniu w węźle B odpowiadających warunkowi krotek (ograniczając tym samym ich liczbę), można je przysłać do węzła A i tam dokonać tego złączenia.

- Ponieważ bazy danych posiadają własne procesory zapytań ich możliwości lokalnej optymalizacji mogą być inne i często ukryte przed mediatorem. Z tego powodu mediator musi często estymować koszty odwołań do lokalnych baz danych. Przykładowe trzy rozwiązania tego problemu przedstawia [30, str. 32. i nast].

- Optymalizacja lokalna sprowadza się do przetwarzania zapytań w bazach scentralizowanych. Obejmuje wszystkie etapy przetwarzania zapytań począwszy od analizy leksykalnej treści zapytania.

Rozdział 6

Polaris - projekt i implementacja

Niniejszy rozdział przedstawia oprogramowanie stworzone w ramach pracy. W oparciu o zawarte w rozdziałach poprzednich analizy uzasadnia wybór zastosowanych technologii i algorytmów realizacji postawionego celu projektowego.

6.1 Cel i zakres projektu

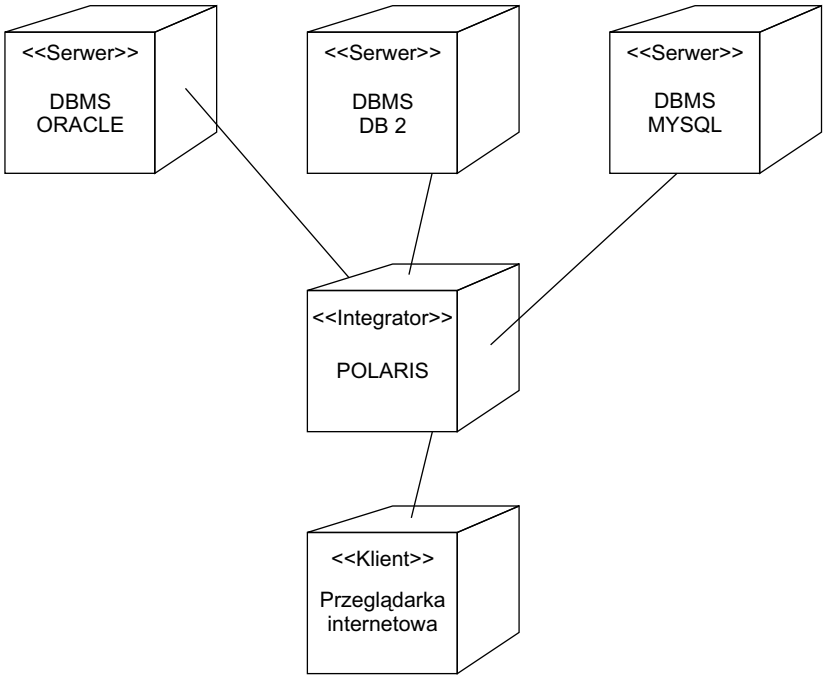
Celem projektu jest stworzenie oprogramowania umożliwiającego realizację systemów informatycznych składających się z rozproszonych baz danych. Przyjęto wobec niego dwa zasadnicze założenia:

- powinno ono gwarantować dostęp do heterogenicznych baz danych
- wykonanie zapytań do rozproszonych baz danych powinno wyglądać tak samo jak do baz scentralizowanych z ukrytą przed użytkownikiem komunikacją sieciową

Spełnienie drugiego z tych założeń prowadzi do konieczności przetwarzania zapytań w ramach aplikacji integrującej rozproszone bazy danych. Oprogramowanie realizujące to zadanie i będące zarazem kluczowym elementem systemu określone zostało terminem Polaris[1]. Jego umiejscowienie w działającym systemie obrazuje rys. 6.1.

Zakres pracy obejmuje przedstawienie projektu Polaris, wykonanie jego implementacji oraz przeprowadzenie testów z wykorzystaniem różnych baz danych.

[1] Jest to łaciński termin opisujący ostatnią gwiazdę Małej Niedźwiedzicy (łac. Ursa Minor) zwaną Gwiazdą Polarną lub Biegunową. Gwiazdozbiór ten, odkryty przez astronoma Talesa, znalazł praktyczne zastosowanie w żegludze, jako swego rodzaju kompas. Nazwa Polaris jako nazwa projektu informatycznego ma tu metaforyczne znaczenie oprogramowania pozwalającego, jak kompas, poruszać się w świecie rozproszonych baz danych.



Rysunek 6.1: Diagram wdrożenia aplikacji Polaris wyrażony w notacji UML

6.2 Realizacja komunikacji w środowisku heterogenicznych baz danych

Realizacja rozproszonej bazy danych wymaga zagwarantowania komunikacji sieciowej. W rozdziale drugim pracy omówiono techniki pozwalające na zapewnienie komunikacji w systemach rozproszonych. Na podstawie przeprowadzonych analiz wybrana zostanie metoda komunikacji w realizowanej rozproszonej bazie danych.

Jako mechanizm najniższego poziomu zaprezentowane zostały gniazda. Dają one ogromne możliwości realizacji komunikacji sieciowej między procesami, lecz zarazem wymagają największego wkładu pracy ze strony programisty. Umożliwiają przesyłanie ciągów bajtów, których interpretacja musi być zaimplementowana przez programistę danego zastosowania. Analizie poddano również mechanizmy sprowadzające się do wywołań procedur i metod obiektów. Techniki te wprowadziły znaczne ułatwienie programowania zastosowań sieciowych, w niektórych przypadkach programista wywołując metodę może nawet nie być świadom tego, że jej wykonanie będzie zdalne. Podstawowe założenia i cel tych technik są zbliżone. Nie mniej jednak występują znaczne różnice w możliwościach ich wykorzystania. Implementacje RPC tworzone są z reguły na jedną platformę uniemożliwiając w ten sposób interoperacyjność. Java/RMI ogranicza się do języka Java i wymaga maszyny wirtualnej tego języka dla danej platformy. CORBA zapewnia niezależność od języka, jednak dla każdego z języków wymagana jest oddzielna implementacja standardu. W DCOM

możliwość wykorzystania ogranicza się głównie do systemu Windows za wyjątkiem kilku implementacji środowiska uruchomieniowego COM na inne platformy.

Na potrzeby realizacji komunikacji Polaris z bazami danych wykorzystano implementację standardu SOAP. Jako mechanizm oparty na przemysłowych standardach w postaci języka XML i protokołu HTTP[2] pozwala on uzyskać interoperacyjność praktycznie między wszystkimi urządzeniami pracującymi w sieci Internet. Warunkiem na to, aby urządzenia mogły współpracować ze sobą zgodnie ze specyfikacją SOAP jest możliwość tworzenia klientów i serwerów protokołu HTTP dla tych urządzeń. Dodatkową korzyścią wynikającą z zastosowania protokołu SOAP jest całkowita niezależność od języka implementacji. Strona serwera może być tworzona w dowolnym języku programowania, o ile pozwala on na dynamiczne generowanie kodu języka HTML. W najprostszym przypadku może być zastosowany interfejs CGI, który wymaga od języka programowania, aby umożliwiał on pisanie na standardowe wyjście, czytanie ze standardowego wejścia i dostęp do zmiennych środowiskowych. Strona klienta wymaga, aby język posiadał implementację interfejsu gniazd.

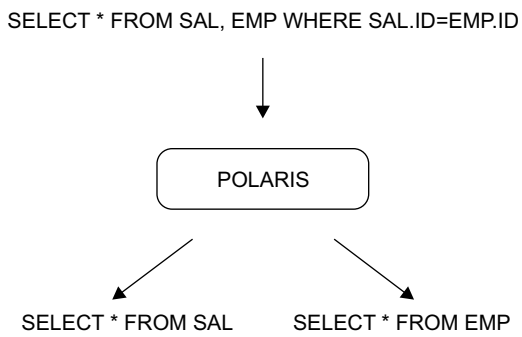
Cecha interoperacyjności protokołu SOAP ma szczególne znaczenie w przypadku rozproszonej bazy danych. Dzięki wykorzystaniu tego protokołu możliwe jest integrowanie baz danych, pracujących pod kontrolą różnych systemów operacyjnych, z interfejsem dostępu do danych w odmiennych standardach i przeznaczonych dla różnych języków. Ważną cechą standardu SOAP, którego komunikaty przenoszone są przy użyciu protokołu HTTP jest to, że taki mechanizm komunikacji jest odporny na większość ścian ogniowych (port 80 jest w większości przypadków otwarty, choć zdarza się, że blokowane są komunikaty protokołu HTTP, zawierające pole SOAPAction).

6.3 Procesor zapytań

Realizacja Polaris ma umożliwić tworzenie systemów, które pozwolą wykonywać zapytania SQL, pobierające w sposób przezroczysty dane z tablic ulokowanych w różnych bazach danych na różnych komputerach. Projekt zakłada, że tablice te będą znajdowały się w bazach danych, do których dostęp będzie realizowany również w języku SQL. Rolę Polaris w tym ujęciu ilustruje przykładowe zapytanie na rys. 6.2. Scenariusz przedstawiony na tym rysunku zakłada, że tablice EMP i SAL znajdują się na różnych komputerach. Polaris otrzymując żądanie wykonania zapytania SQL, którego realizacja wymaga przeanalizowania danych z obu tablic, wykonuje dwa odwołania w języku SQL do baz danych. Otrzymane w ten sposób dane musi przetworzyć, aby zagwarantować spełnienie warunku zapytania wejściowego.

Zgodnie z ustaleniami poczynionymi w punkcie poprzednim komunikacja pomiędzy elementami składowymi systemu, w szczególności pomiędzy bazami danych a

[2]Specyfikacja dopuszcza realizację transportu komunikatów SOAP również z wykorzystaniem innych protokołów, np. pocztowych. W chwili obecnej stosuje się niemal wyłącznie protokół HTTP dlatego dalsze rozważania odbywają się w świetle wykorzystania tego właśnie protokołu.



Rysunek 6.2: Rola Polarisa w przetwarzaniu zapytań

Polarisem, jest realizowana w oparciu o implementację protokołu SOAP. Pozwala to zapewnić interoperacyjność systemu, umożliwiając współpracę różnych baz danych. Zastosowanie protokołu SOAP pozwala osiągnąć daleko idącą niezależność od platformy, systemu bazy danych, metody dostępu do danych i języków programowania.

6.3.1 Specyfikacja składni obsługiwanego języka SQL

SQL implementowany w ramach Polaris opiera się na specyfikacji tego języka w wersji SQL92. Do realizacji wybrano podstawowe instrukcje, których specyfikację w notacji BNF przedstawiono poniżej.

```
query_spec ::= "select" [ "distinct" | "all" ] select_list
table_exp [ order_by_clause ]

select_list ::= "*" | select_sublist { "," select_sublist }
select_sublist ::= [ qualifier "." ] column_name | qualifier "." "*"
column_name ::= IDENT
qualifier ::= IDENT

table_exp ::= from_clause [ where_clause ]
from_clause ::= "from" table_ref { "," table_ref }
table_ref ::= table_name [ [ "as" ] correlation_name ]
table_name ::= IDENT
correlation_name ::= IDENT

order_by_clause ::= "order" "by" sort_spec_list
sort_spec_list ::= sort_spec { "," sort_spec }
sort_spec ::= column_name | INT [ "asc" | "desc" ]

where_clause ::= "where" search_condition
search_condition ::= boolean_term | search_condition
```

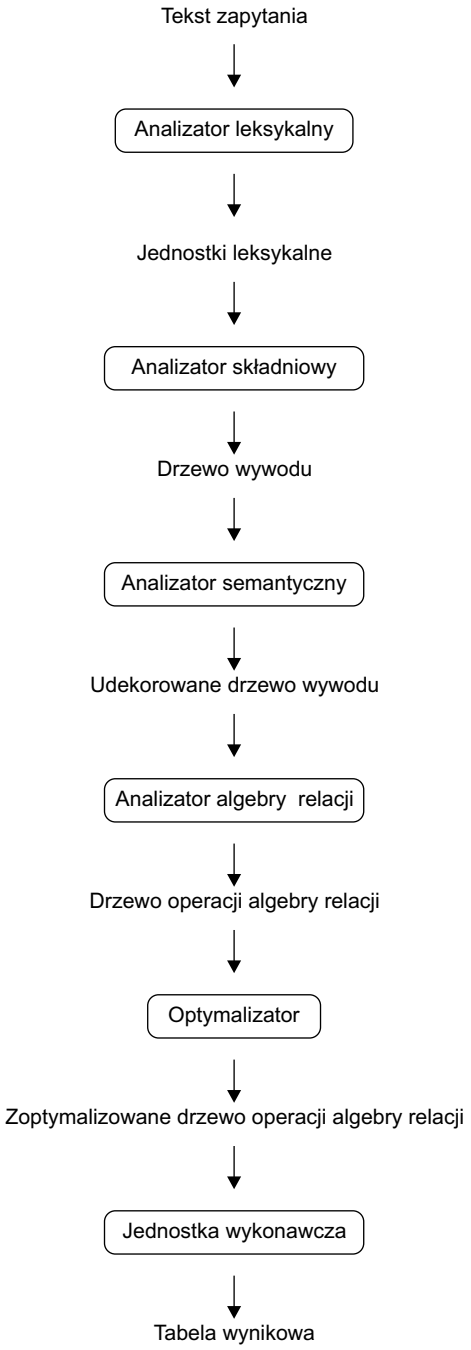
```
"or" boolean_term
boolean_term ::= boolean_factor | boolean_term "and" boolean_factor
boolean_factor ::= [ "not" ] boolean_test

boolean_test ::= comp_predicate | null_predicate | like_predicate
| bool_predicate | "(" search_condition ")"

bool_predicate ::= column_ref "is" [ "not" ] truth_value
truth_value ::= "true" | "false"

comp_predicate ::= value_exp  comp_op value_exp
comp_op ::= "=" | "<>" | "<" | ">" | "<=" | ">="

like_predicate ::= column_ref "like" STRING
null_predicate ::= column_ref "is" ["not"] "null"

value_exp ::= num_value_exp | string_value_exp |
datetime_value_exp | column_ref

num_value_exp ::= term | num_value_exp "+" term |
num_value_exp "-" term
term ::= factor | term "*" factor | term "/" factor
factor ::= [ "+"|"-" ] num_primary

num_primary ::= INT | REAL | column_ref
string_value_exp ::= STRING | column_ref
datetime_value_exp ::= DATE_STRING | TIME_STRING | DATETIME_STRING |
column_ref
column_ref ::= [ qualifier "." ] column_name
```

Symbolem startowym jest `query_spec`, zaś wszystkie symbole oznaczone kapitalikami są symbolami terminalnymi. Ich postać definiują poniższe wyrażenia regularne:

```
TIME_STRING     [0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}
DATE_STRING     [0-9]{4}-[0-9]{1,2}-[0-9]{1,2}
DATETIME_STRING [0-9]{4}-[0-9]{1,2}-[0-9]{1,2}[ ][0-9]{1,2}:
                [0-9]{1,2}:[0-9]{1,2}
IDENT           [A-Za-z][A-Za-z0-9_]*
INT             [0-9]+
REAL            [0-9]+[eE][+-]?[0-9]+ | [0-9]+\.[0-9]*[eE][+-]?[0-9]+ |
                [0-9]+\.[0-9]* |  \.[0-9]+ |  \.[0-9]*[eE][+-]?[0-9]+
STRING          [^'\n]*
```

Wybrana do implementacji część specyfikacji języka SQL obejmuje operację `SELECT` z obsługą takich elementów jak sortowanie, eliminacje powtórzeń krotek,

aliasy tabel, większość warunków klauzuli `WHERE`. Dotyczy to porównań wartości logicznych, liczbowych i tekstowych, a także obsługi wyrażeń arytmetycznych.



Rysunek 6.3: Fazy przetwarzania zapytań

Przykładowe zapytania:

```
SELECT DISTINCT * FROM SAL
SELECT * FROM EMP as E, SAL as S WHERE S.ID=E.ID
SELECT * FROM EMP ORDER BY NAME
SELECT ID FROM SAL WHERE SALARY*0.9=2000*2/3
SELECT NAME, SURNAME FROM EMP WHERE NAME LIKE 'SMITH'
```

6.3.2 Realizacja przetwarzania rozproszonych zapytań

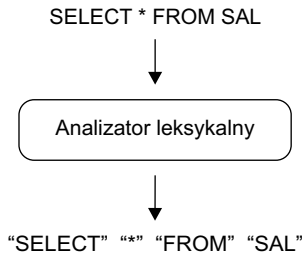
W przetwarzaniu realizowanym przez procesor zapytań można wyróżnić omawiane niżej etapy, które dodatkowo zostały zobrazowane na rys. 6.3.

Analiza leksykalna zapytania

Zadaniem analizy leksykalnej jest rozpoznawanie różnych rodzajów elementów w strumieniu wejściowym, którym w tym przypadku dostarczane jest zapytanie w języku SQL. Analizator leksykalny odczytuje dane wejściowe i zapewnia rozróżnienie, który składnik został napotkany (rys. 6.4).

Dane wejściowe: ciąg znaków reprezentujący zapytanie w języku SQL

Dane wyjściowe: ciąg rozpoznanych symboli leksykalnych



Rysunek 6.4: Przykład działania analizatora leksykalnego

Realizacja fazy analizy leksykalnej odbywa się z wykorzystaniem generatora analizatorów leksykalnych o nazwie `flex` [17]. Jest to program, który dla podanego w postaci wyrażeń regularnych opisu jednostek leksykalnych generuje skaner rozpoznający te jednostki.

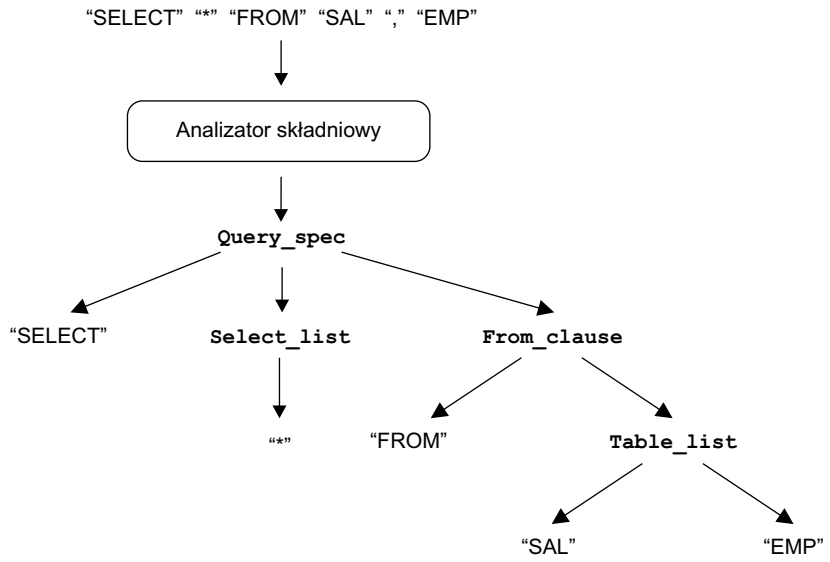
Analiza składniowa zapytania

Drugim zadaniem jest rozpoznanie struktury gramatycznej zdania składającego się z symboli leksykalnych uzyskanych w kroku poprzednim (przykład abstrahujący

od szczegółów gramatyki, ilustruje rys. 6.5). W opisywanym przypadku gramatyka zdefiniowana została specyfikacją w notacji BNF przedstawioną w podpunkcie poprzednim.

Dane wejściowe: ciąg symboli leksykalnych

Dane wyjściowe: drzewo wywodu obrazujące gramatyczny rozbiór podanego ciągu symboli leksykalnych



Rysunek 6.5: Przykład działania analizatora składniowego

Fazę analizy składniowej zrealizowano z wykorzystaniem generatora analizatorów składniowych o nazwie `bison` [7]. Jest to program, który konwertuje opis gramatyki bezkontekstowej na program w języku C rozpoznający tą gramatykę.

Analiza semantyczna

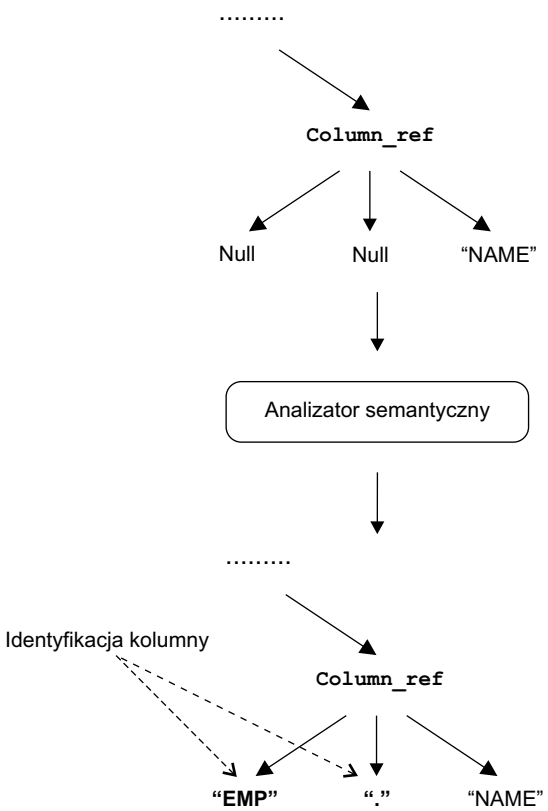
Wygenerowane w poprzednim kroku drzewo wywodu jest poprawne pod względem syntaktycznym. Pozostaje jeszcze zweryfikowanie czy zapytanie przestrzega zasad semantyki języka SQL. Na tym etapie jest to kontrola statyczna w odróżnieniu od kontroli dynamicznej, która następuje podczas wykonania programu (w tym przypadku zapytania) [1, str. 325]. Możliwości statycznej analizy semantycznej w przypadku języka SQL są znacznie ograniczone w porównaniu z klasycznymi językami programowania. Np. kontrola typów może być w pewnych przypadkach realizowana dopiero w fazie wykonania, gdyż często od konkretnych danych zgromadzonych w bazie danych zależy czy w danym przypadku wystąpi wartość atomowa czy cała relacja. Obrazuje to przykład poniżej:

```
SELECT * FROM EMP
WHERE ID = (SELECT ID FROM EMP WHERE NAME='BROWN')
```

Błąd wystąpi jedynie wtedy, kiedy podzapytanie zwróci więcej niż jedną wartość. Jest to oczywiście niemożliwe do stwierdzenia przed rozpoczęciem wykonania zapytania [47, str. 307].

Dane wejściowe: drzewo wywodu

Dane wyjściowe: udekorowane drzewo wywodu (rys. 6.6)



Rysunek 6.6: Przykład działania analizatora semantycznego

Polaris przeprowadza następujące działania weryfikacji semantyki i przygotowania do dalszego przetwarzania:

- sprawdzenie jednoznaczności odwołań do kolumn i tabel,
- sprawdzenie przynależności kolumn do wyspecyfikowanych relacji,
- sprawdzenie istnienia relacji,

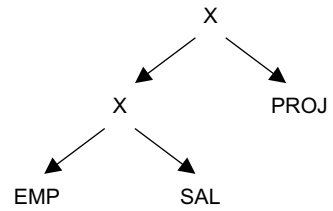
- dodanie aliasów do wszystkich odwołań do kolumn; w przypadku tabel, dla których nie wyspecyfikowano aliasu, domyślnym aliasem staje się nazwa tabeli.

Transformacja drzewa wywodu do drzewa operacji algebry relacji

Wykonywanie zapytań zapisanych w języku SQL może odbywać się na wiele, równoważnych sposobów. W [47, str. 299] przedstawiono trzy koncepcje: pętle zagnieżdżone, przypisania równoległe i przejście do postaci algebry relacji. W projektowanej aplikacji zastosowano rozwiązanie trzecie, które jest najczęściej stosowane w praktyce i stwarza szerokie możliwości optymalizacji szczególnie w przypadku zapytań rozproszonych.

Zadaniem omawianej fazy przetwarzania procesora zapytań jest transformacja drzewa wywodu do drzewa operacji algebry relacji. Kierując się koncepcją zaproponowaną w [37, str. 194-195] proces ten można opisać następująco:

- Pierwszym krokiem jest utworzenie liści drzewa, przez umieszczenie w nich relacji z klauzuli `FROM` zapytania. Tworzone liście są łączone wierzchołkami zawierającymi operację iloczynu kartezjańskiego. Wynik tego kroku może mieć postać jak na rys. 6.7, gdzie w liściach zawarto relacje z bazy danych.

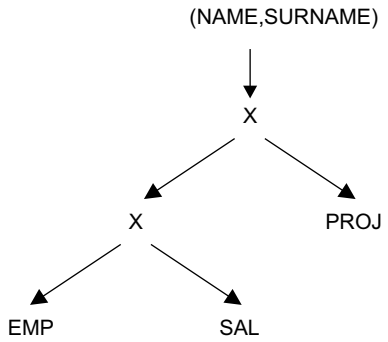


Rysunek 6.7: Przykład wykonania pierwszego kroku tworzenia drzewa operacji algebraicznych

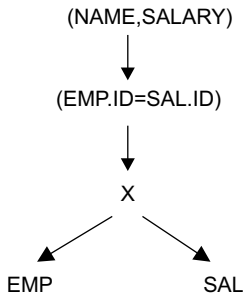
- W drugim kroku tworzony jest korzeń drzewa zawierający operację rzutowania na atrybuty wymienione w klauzuli `SELECT`. W wyniku wykonania tego kroku drzewo może mieć postać jaką obrazuje rys. 6.8.

- W trzecim kroku przetwarzana jest klauzula `WHERE`, która jest tłumaczona na sekwencję operacji relacyjnych wstawianych w węzłach wewnętrznych drzewa. Obejmuje to operacje selekcji, sumy, przecięcia i złączenia.

Przykładowe drzewo operacji algebry relacji dla zapytania `SELECT NAME, SALARY FROM EMP, SAL WHERE EMP.ID=SAL.ID` może mieć postać jaką obrazuje rys. 6.9.



Rysunek 6.8: Przykład wykonania drugiego kroku tworzenia drzewa operacji algebraicznych



Rysunek 6.9: Przykład drzewa operacji algebry relacji

Dane wejściowe: udekorowane drzewo wywodu

Dane wyjściowe: drzewo operacji algebry relacji

Optymalizacja i wykonanie zapytania reprezentowanego przez drzewo operacji algebry relacji

Drzewo operacji algebry relacji generowane w kroku poprzednim reprezentuje jedną z wielu strategii wykonania zapytania. Ważnym etapem przetwarzania zapytań jest wybranie takiej strategii, która będzie najbardziej optymalna pod względem określonych kryteriów. W projektowanym systemie kryterium optymalizacji są:

- minimalizacja czasu odpowiedzi,
- minimalizacja liczby połączeń z lokalnymi bazami danych,
- maksymalizacja liczby operacji algebry relacji wykonywanych przez lokalne bazy danych.

Proces optymalizacji obejmuje manipulację rozmieszczeniem i łączenie węzłów w drzewie, tak aby w jak największym stopniu sprostać tym wymogom. Realizowane jest przez algorytm przedstawiony na rys. 6.10.

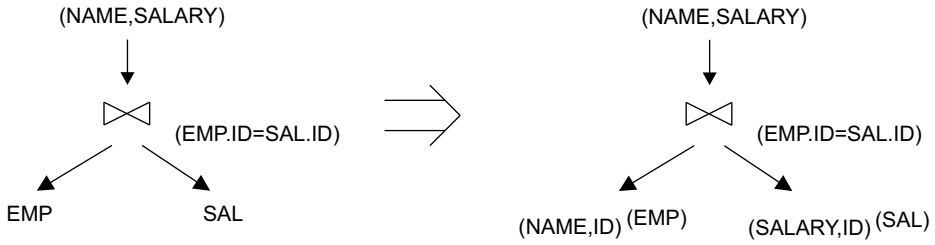
input: drzewo algebraiczne zapytania
output: wynik zapytania
begin
{optymalizacja statyczna}
{przesunięcie projekcji}
projection_down() (1)
{przesunięcie selekcji}
optimize_select() (2)
{zamiana iloczynów kartezjańskich na złączenia}
cart_to_join() (3)
{łączenie liści drzewa}
merge_sons() (4)
{usuwanie redundancji}
rm_duplicates_from_cart() (5)
{optymalizacja dynamiczna}
{porządkowanie złączeń i wykonanie zapytania}
execute_algebraic_tree() (6)
end.

Rysunek 6.10: Algorytm optymalizacji zapytań

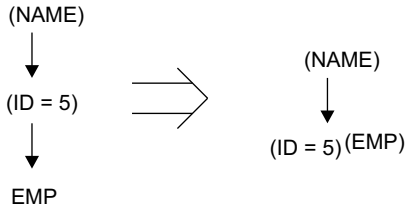
Algorytmy optymalizacji zapytań omawiane w rozdziale piątym dotyczyły kwestii porządkowania złączeń relacji. Realizacja kompletnej implementacji wymaga wykonania dodatkowych zabiegów, które w połączeniu z omawianymi wcześniej algorytmami stanowią rozwiązanie zaprojektowane na potrzeby przetwarzania zapytań w aplikacji Polaris. Jego działanie składa się z dwóch głównych etapów optymalizacji:

- statycznej,
- dynamicznej.

Optymalizacja statyczna W tej fazie drzewo przygotowywane jest do pobrania danych w możliwie najefektywniejszy sposób. Realizowane jest to w krokach (1) do (5). W kroku (1) następuje umieszczenie w liściach drzewa projekcji na te kolumny, które występują w projekcjach, selekcjach i innych operacjach na drodze od korzenia do danego liścia. Wynik działania tej fazy ilustruje rys. 6.11. W kroku (2) wszystkie operacje selekcji, w których występuje odwołanie tylko do jednej tabeli, są przesuwane do wszystkich liści poddrzewa będącego poniżej węzła z tą operacją. Jest ona dodawana do tych liści, które odwołują się do tej samej tabeli (rys. 6.12). Kolejny

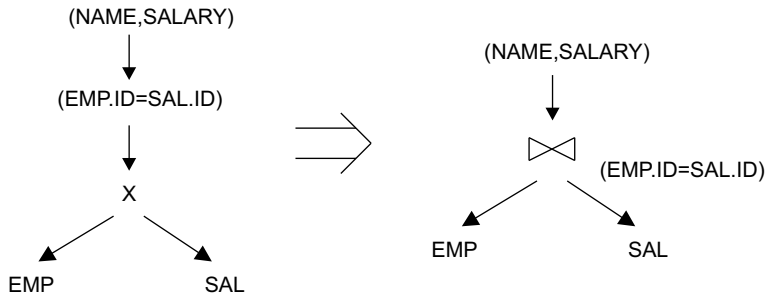


Rysunek 6.11: Przykład przesunięcia projekcji



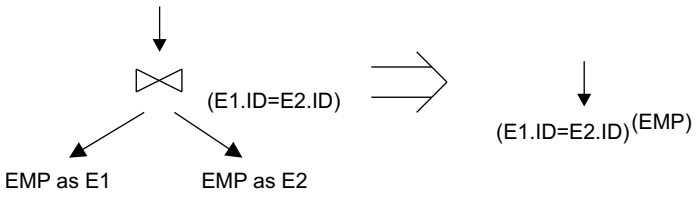
Rysunek 6.12: Przykład przesunięcia selekcji jednoargumetnowej

krok (3) zamienia, o ile to możliwe, iloczyny kartezjańskie na złączenia. Operacje selekcji odwołujące się do dwóch różnych tabel przesuwane są niżej w drzewie do odpowiednich iloczynów kartezjańskich. W efekcie iloczyny kartezjańskie stają się złączeniami naturalnymi lub złączeniami teta (rys. 6.13). W kroku (4) zachodzi łą-

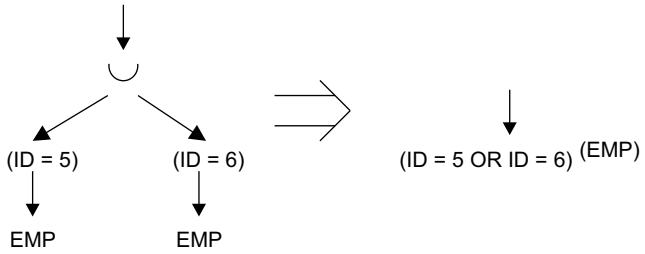


Rysunek 6.13: Przykład zamiany iloczynu kartezjańskiego na złączenie

czenie liści drzewa odwołujących się do tej samej tabeli i będących synami operacji złączenia, iloczynu kartezjańskiego lub sumy relacji. Przykłady działania tej fazy na synach operacji złączenia i sumy ilustrują odpowiednio rys. 6.14 i rys. 6.15. Ostatnim elementem optymalizacji statycznej jest usunięcie redundancji w kroku (5). Polega to na usunięciu redundantnych predykatów (rys. 6.16).



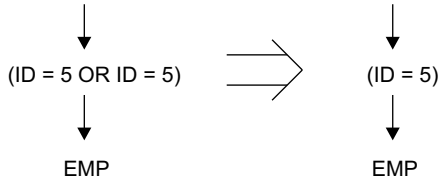
Rysunek 6.14: Przykład łączenia synów złączenia



Rysunek 6.15: Przykład łączenia synów sumy

Optymalizacja dynamiczna Przeprowadzana jest na etapie wykonania zapytania, które rozpoczyna się równoległym (zmniejszenie czasu odpowiedzi) pobraniem danych z miejsc ich przechowywania. Odwołania do baz danych stanowią liście drzewa, a dodatkowe warunki umieszczone w nich w etapie optymalizacji statycznej pozwalają w znaczny sposób ograniczyć rozmiar danych do dalszego przetwarzania. Po wydobyciu danych wykonywane są złączenia tabel, których kolejność ustalana jest według algorytmu zachłannego, ale opartego o rzeczywiste rozmiary relacji. Spośród n relacji do złączenia w każdym kroku wybierana jest para, w której iloczyn liczności jej relacji jest najmniejszy.

Po wykonaniu złączeń realizowane są pozostałe operacje algebry relacji zwarte w przetwarzanym drzewie. Zgodnie z porządkiem przetwarzania zapytań reprezentowanych przez drzewo algebry relacji, wynik przetwarzania w korzeniu drzewa stanowi tabela będąca wynikiem zapytania.



Rysunek 6.16: Przykład usuwania redundancji

Prezentacja wyników

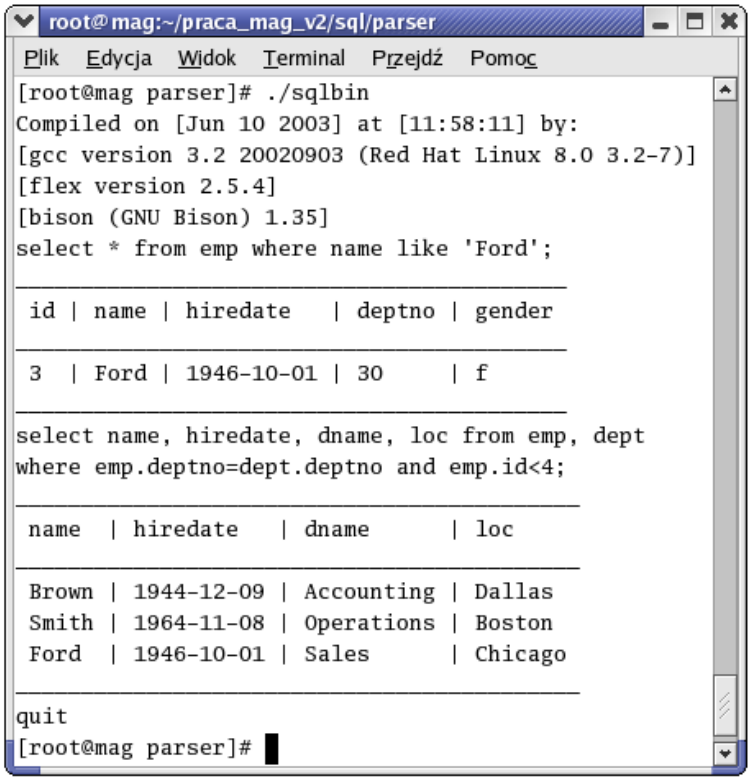
Ostatnim etapem przetwarzania jest prezentacja uzyskanych wyników. Tabela danych utworzona w kroku opisanym wyżej jest drukowana na standardowe wyjście terminala tekstowego lub wyświetlana w oknie terminala graficznego.

6.4 Polaris - możliwości wykorzystania

Polaris realizuje dwie z omawianych w rozdziale trzecim metody dostępu do danych. Pierwszą jest wykorzystanie go jako interpretera języka SQL w postaci terminala graficznego lub tekstowego. Drugą jest możliwość wykorzystania go jako źródło danych z interfejsem programistycznym do zastosowań aplikacyjnych.

Terminal tekstowy

Polaris w tej wersji jest aplikacją tekstową, która ze standardowego wejścia pobiera tekst zapytania SQL. Wynik tego rozproszonego zapytania drukowany jest na standardowe wyjście. Pozwala to na szerokie możliwości wykorzystania go zarówno w formie terminala wiersza poleceń, jak również poprzez wywołania przez inne programy z zastosowaniem mechanizmu potoków do komunikacji. Rys. 6.17 przedstawia przykładową sesję terminala tekstowego.

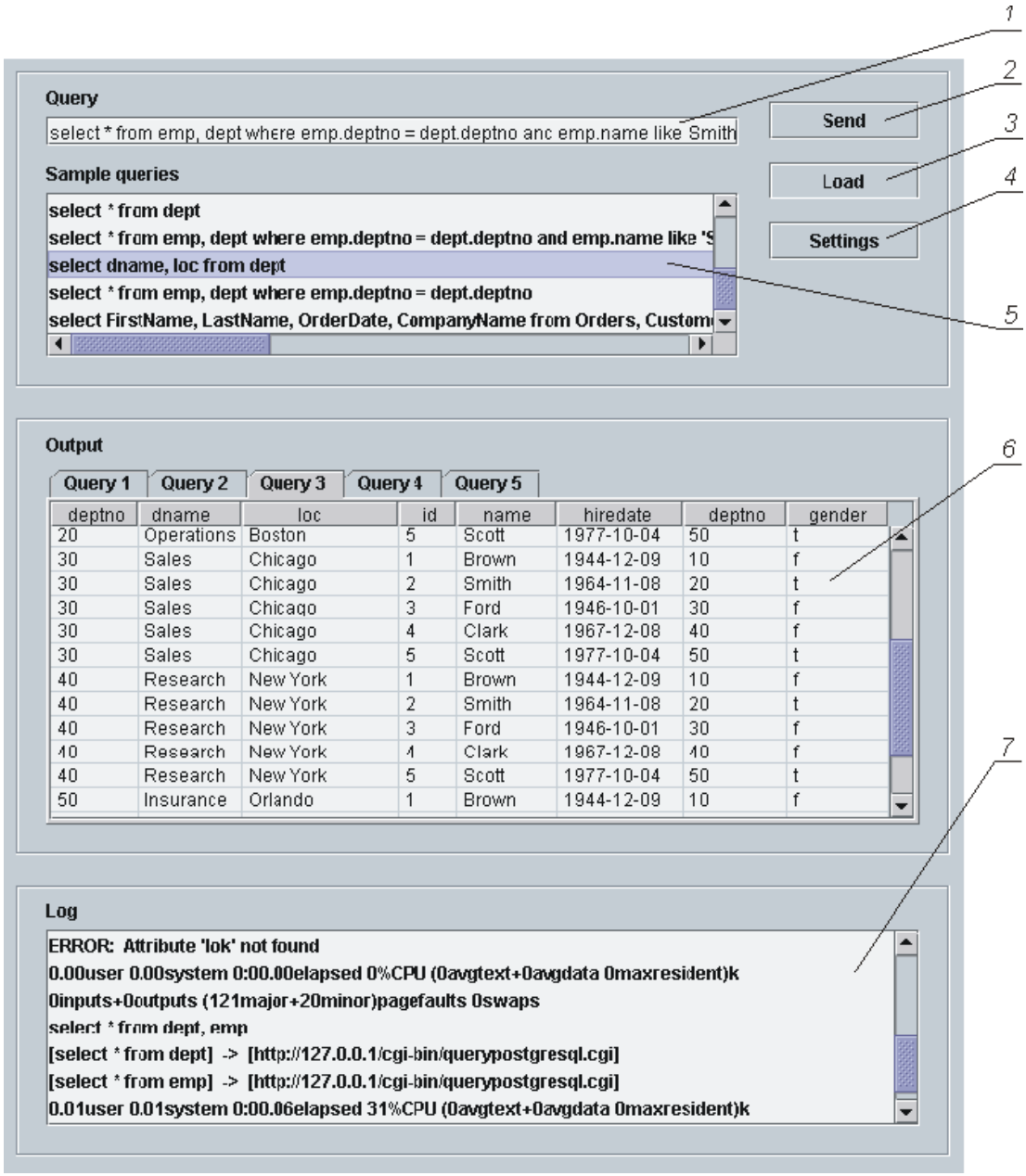


Rysunek 6.17: Przykładowa sesja terminala tekstowego

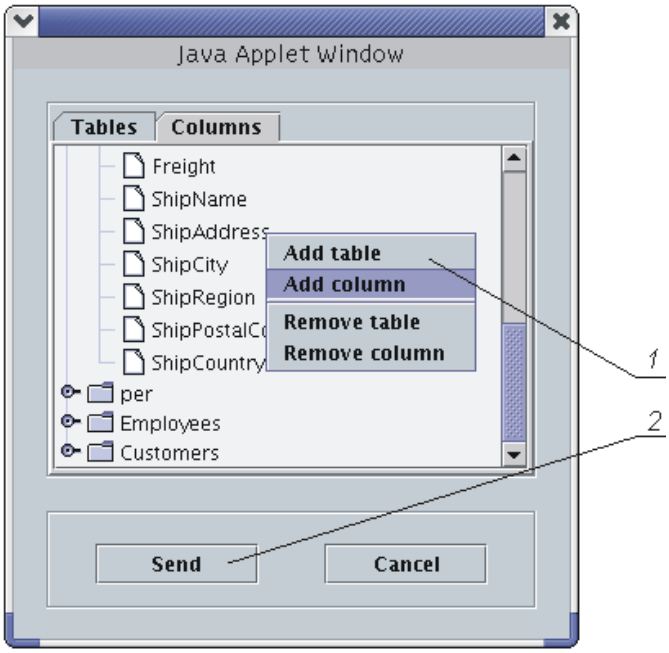
Terminal graficzny

W celu zapewnienia graficznego interfejsu stworzone zostało oprogramowanie pośredniczące w dialogu między użytkownikiem a Polarisem. Oprogramowanie to zrealizowano jako aplet języka Java. Oprócz funkcji użytkowych pozwala ono również na administrowanie systemem. Przykładowa sesja pracy z terminalem graficznym została przedstawiona na rys. 6.18.

Terminal ten pozwala na wykonywanie rozproszonych zapytań, które wprowadzane są w oknie edycyjnym 1 (rys. 6.18). Przykładowe zapytania umieszczono w liście 5 i można je kopiować do pola 1 przy użyciu przycisku 3. Wykonanie zapytania inicjowane jest przyciskiem 2. Kończy się ono zapisaniem wyniku do jednej z tabel w polu 6 i informacji diagnostycznych w dzienniku 7. Informacje te obejmują zarówno błędy wykonania jak też czas realizacji zapytania i pokazują odwołania do poszczególnych baz danych. Przycisk 4 uruchamia okno dialogowe (rys. 6.19) pozwalające konfigurować system rozproszonych baz danych. Dotyczy to rozłożenia tabel w węzłach systemu oraz kolumn w ramach tych tabel. Z wykorzystaniem menu kontekstowego 1 można wykonywać podstawowe czynności edycyjne, dodawanie oraz usuwanie tabel i kolumn. Akceptacja zmian przyciskiem 2 powoduje zapisanie nowej konfiguracji na serwerze, na którym pracuje Polaris.



Rysunek 6.18: Przykładowa sesja terminala graficznego



Rysunek 6.19: Konfiguracja systemu

Interfejs programistyczny

Dostęp do danych wymagany jest zazwyczaj z poziomu aplikacji użytkowych. Z tego powodu Polaris realizowany jest również jako biblioteka dołączana dynamicznie w systemie LINUX. Pozwala to na wykorzystanie go jako interfejsu programistycznego w aplikacjach użytkowych. W tym celu biblioteka eksportuje funkcję:

```
int distr_query(FILE* in, FILE* out);
```

Parametr `in` wskazuje strumień, z którego funkcja ma wczytać zapytanie. Parametr `out` wskazuje strumień wyjściowy, do którego funkcja ma zapisać wynik. Ustawienie na `NULL` tych parametrów powoduje przyjęcie wartości domyślnych, tzn. standardowego wejścia dla parametru `in` i standardowego wyjścia dla parametru `out`. Niżej przedstawiono przykładowy program wykorzystujący Polaris jako interfejs programistyczny.

```
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
    void *handle;
    int (*distr_query)(FILE* in, FILE* out);
    char *error;
```

```
    handle = dlopen ("/usr/lib/libsqlbin.so", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }

    distr_query = dlsym(handle, "distr_query");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
     }

    (*distr_query)(NULL/*stdin*/,NULL/*stdout*/);

    dlclose(handle);
}
```

Program ten przy użyciu funkcji `dlopen` ładuje Polaris w postaci biblioteki dynamicznej znajdującej się w pliku `/usr/lib/libsqlbin.so` i uzyskuje z niego wskaźnik do funkcji realizującej zapytanie. Wywołanie tej funkcji powoduje wczytanie zapytania ze standardowego wejścia i zapisanie wyniku na standardowe wyjście.

Rozdział 7

Praktyczne wykorzystanie Polaris

Niniejszy rozdział ilustruje sposoby wykorzystania stworzonego oprogramowania do przetwarzania zapytań w systemach rozproszonych. Najpierw omówiona zostanie przykładowa konfiguracja heterogenicznego systemu rozproszonego i pokazana rola Polaris w zapewnieniu przezroczystego interfejsu dostępu do baz danych. Następnie przedstawione zostanie porównanie szybkości wykonania tego samego zapytania w wersji scentralizowanej (wszystkie tabele na jednym komputerze) i rozproszonej (tabele na różnych komputerach).

7.1 Przetwarzanie zapytań w systemie heterogenicznym

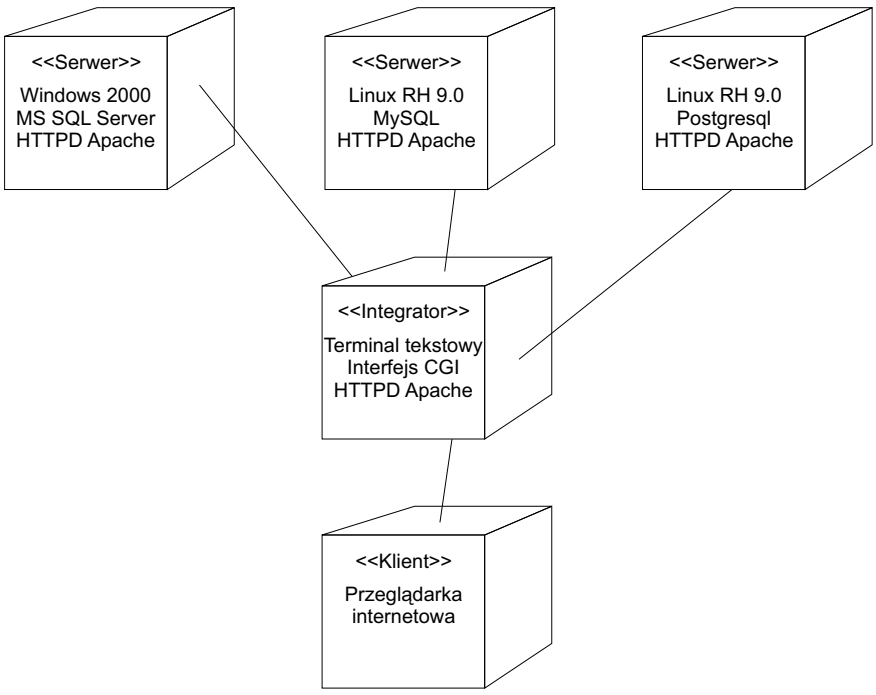
Aby zaprezentować w praktyce, jak Polaris pozwala na współpracę różnych baz danych pracujących pod kontrolą różnych systemów operacyjnych przygotowany został system rozproszony w omówionej poniżej konfiguracji.

Rozłożenie węzłów, czyli fizycznych zasobów obliczeniowych, przedstawiono na rys. 7.1. Lokalne bazy danych uruchomione zostały na trzech komputerach, z których jeden pracował pod kontrolą systemu operacyjnego Windows 2000, a pozostałe dwa systemu Linux Red Hat 9.0. W każdej z tych baz danych umieszczono jedną z trzech tabel przykładowej bazy danych o schemacie przedstawionym na rys. 7.2. Omawianą konfigurację sprzętu i oprogramowania zebrano w tab. 7.1.

	System operacyjny	**Baza danych**	**Tabela**
Serwer 1	Windows 2000	MS SQL Server	Orders
Serwer 2	Linux RH 9.0	MySQL	Customers
Serwer 3	Linux RH 9.0	Postgresql	Employees

Tabela 7.1: Konfiguracja baz danych

Polaris został zainstalowany na oddzielnym komputerze. Mógł być użyty bez-



Rysunek 7.1: Konfiguracja testowego systemu

pośrednio jako terminal tekstowy lub wywoływany przez skrypt CGI pozwalał na wykorzystanie interfejsu graficznego w przeglądarce internetowej.

Rozważamy wykonanie przykładowego zapytania wymagającego pobrania danych z trzech tabel umieszczonych na różnych komputerach w różnych bazach danych:

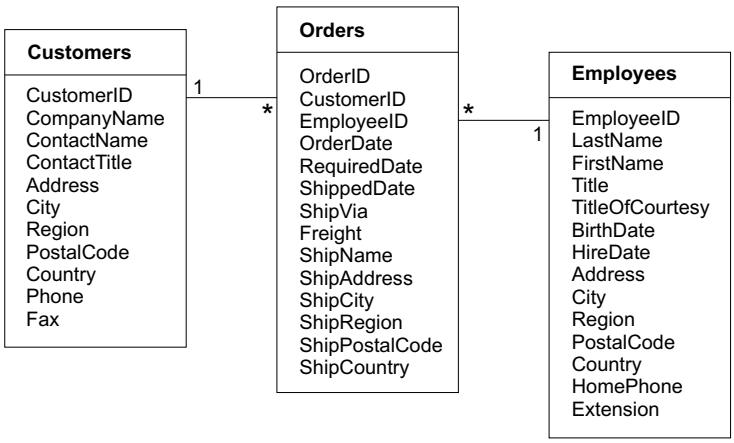
```
select ShipName, ShipCity, ShipCountry, OrderDate, CompanyName, Phone
from Orders, Customers, Employees
where Orders.CustomerID=Customers.CustomerID
and Orders.EmployeeID=Employees.EmployeeID
and Employees.EmployeeID=5;
```

Zapytanie to, wymagające złączenia trzech tabel w celu pobrania danych o klientach i zamówieniach realizowanych przez pracownika o identyfikatorze 5, zostało zdekomponowane przez Polaris do następujących trzech zapytań do baz danych:

- zapytanie do bazy danych MS SQL Server

  ```
  select Orders.OrderDate, Orders.ShipCountry, Orders.ShipCity,
  Orders.ShipName, Orders.CustomerID, Orders.EmployeeID
  from Orders
  ```

- zapytanie do bazy danych MySQL

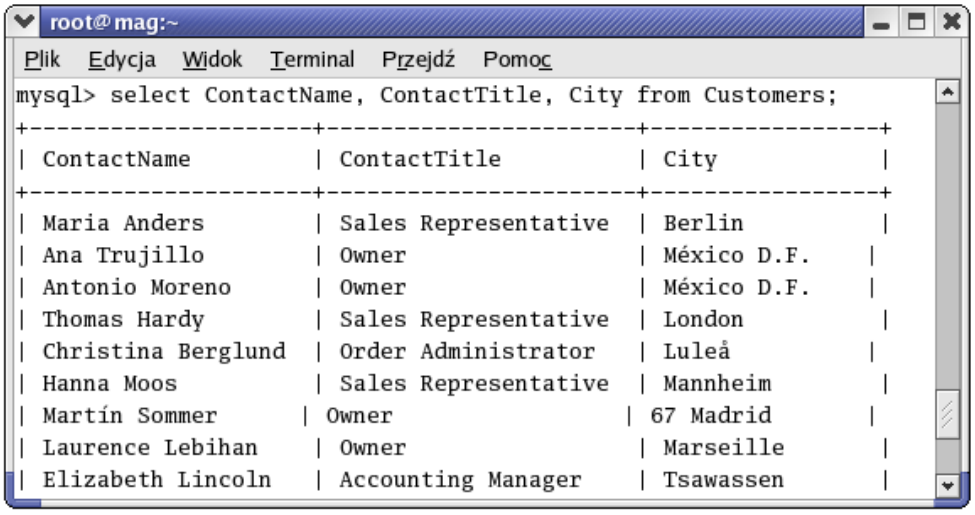


Rysunek 7.2: Schemat bazy danych

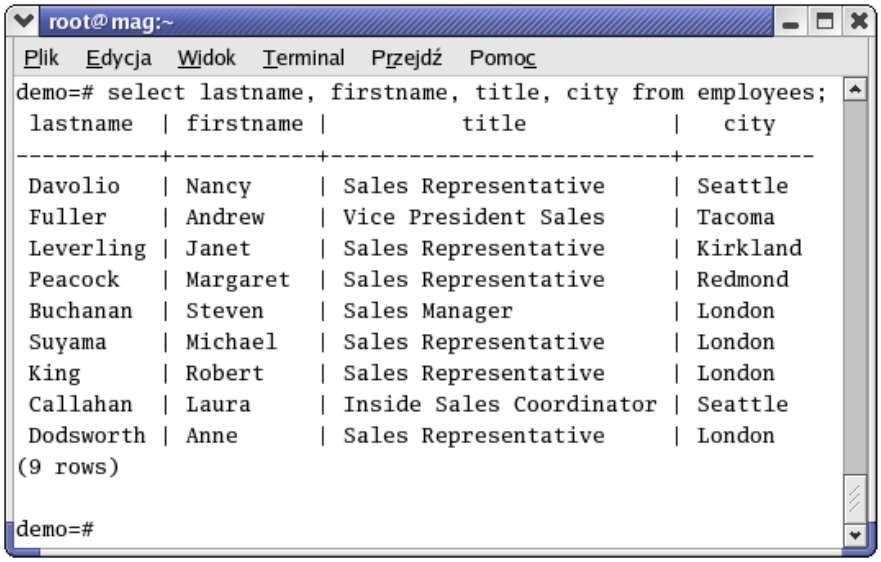
```
select Customers.Phone, Customers.CompanyName,
Customers.CustomerID
from Customers
```

- zapytanie do bazy danych Postgresql

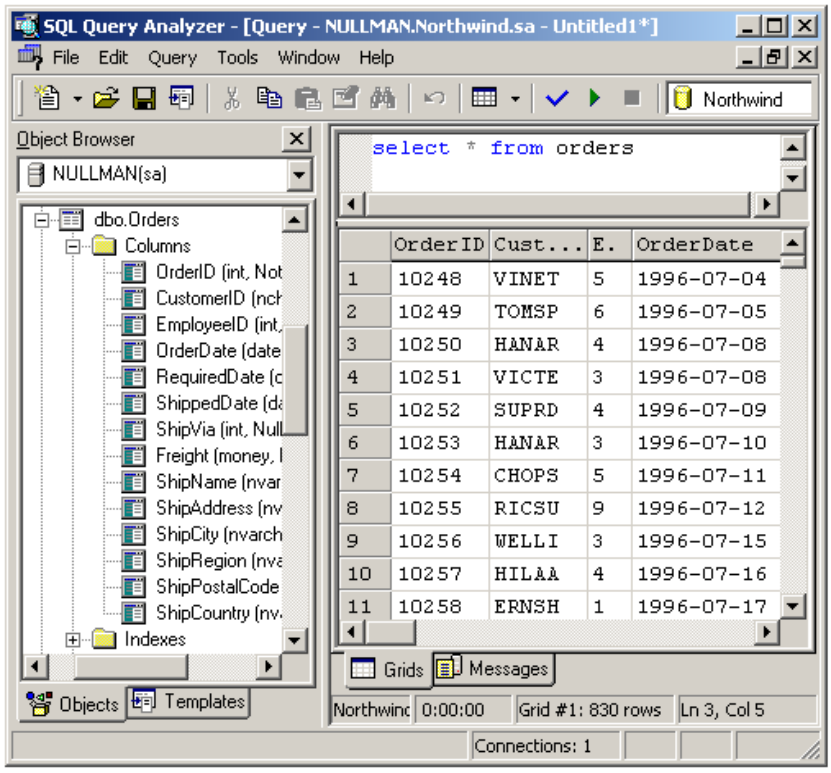
```
select Employees.EmployeeID
from Employees
where Employees.EmployeeID=5
```



Rysunek 7.3: Tabela Customers z bazy danych MySql



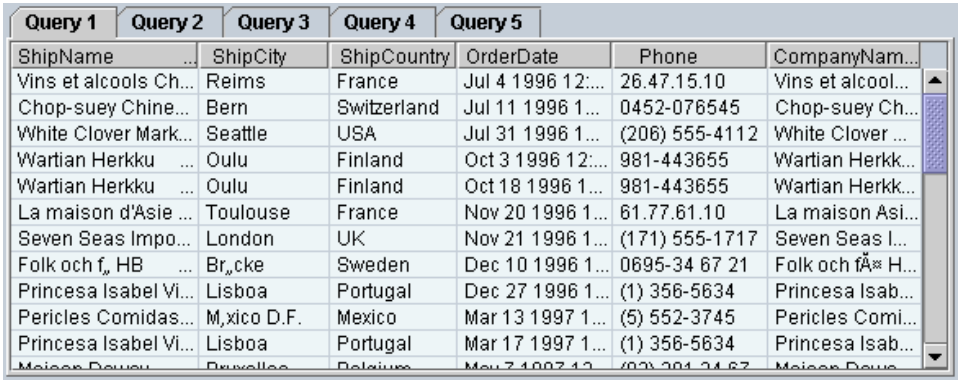
Rysunek 7.4: Tabela Employees z bazy danych Postgresql



Rysunek 7.5: Tabela Orders z bazy danych MS SQL Server

Każde z tych zapytań posłużyło do pobrania danych z tabel zlokalizowanych na

różnych komputerach. Tabele te ilustrują odpowiednio rys. 7.3, 7.4 i 7.5. W przypadku odwołania do bazy danych Postgresql zapytanie zawierało warunek selekcji, co pozwoliło, zgodnie z wymaganiami zapytania wejściowego, ograniczyć rozmiar pobieranych danych. Wyniki tych zapytań w celu spełnienia warunków zapytania wejściowego poddawane były dalszemu przetwarzaniu już w ramach Polaris. Polegało to na złączeniu trzech tabel pobranych z odległych lokalizacji. Wynik tego działania przedstawia rys. 7.6.



Rysunek 7.6: Prezentacja w oknie interfejsu graficznego Polaris wyniku rozpatrywanego zapytania rozproszonego

7.2 Efektywność przetwarzania zapytań rozproszonych

Sprawdzenie efektywności zapytań rozproszonych wykonywanych przy użyciu Polaris polegało na porównaniu czasów wykonania zapytania SQL w dwóch wersjach: scentralizowanej i rozproszonej. Zapytanie to miało postać:

```
select * from tab1, tab2, tab3
where tab1.id1 = tab2.id1
and tab3.id3 = tab2.id3
and tab1.id1 < 10
and tab3.id3 < 10
and tab2.id1 < 10
and tab2.id3 < 10;
```

Było to testowe zapytanie do bazy danych o wygenerowanej losowo zawartości. Istotą jego poza łączeniem trzech tabel było to, że proces ten odbywał się na wyselekcjonowanej zawartości tych tabel.

Testy przeprowadzono na jednakowych komputerach PC wyposażonych w procesory Intel Celeron o częstotliwości pracy 633MHz i pamięć po 192MB każdy. Wykorzystano system operacyjny Linux Red Hat 7.3 i bazę danych Postgresql w wersji

7.2.1 oraz serwer HTTP Apache w wersji 1.3.23. Oprogramowanie to było używane z domyślną konfiguracją, tzn. taką jaka występowała po zainstalowaniu.

W konfiguracji scentralizowanej trzy testowe tabele zostały umieszczone w tej samej bazie danych Postgresql. Zapytanie scentralizowane wykonano z użyciem terminala tekstowego `psql` umożliwiającego dostęp do tej bazy danych. Zapytanie rozproszone wykonywał Polaris. Dane do realizacji zapytania pobierał on z trzech baz danych znajdujących się na oddzielnych komputerach. Na każdej z tych baz znajdowała się jedna z tabel testowego schematu.

Do pomiaru czasu wykorzystano program `time` dostępny w systemie Linux. Pozwala on uzyskać informacje o zużyciu przez dany program zasobów systemu komputerowego, w szczególności uzyskać czas upływający pomiędzy wywołaniem a zakończeniem działania procesu. Wywołania testowe miały postać:

```
time psql demo < query.sql > /dev/null
time ./sqlbin demo < query.sql > /dev/null
```

Pierwsze z tych wywołań przedstawia test scentralizowany a drugie rozproszony. Zastosowanie w obu wywołaniach przekierowania na strumień `/dev/null` służyło wyeliminowaniu czasu wypisania wyniku na standardowe wyjście.

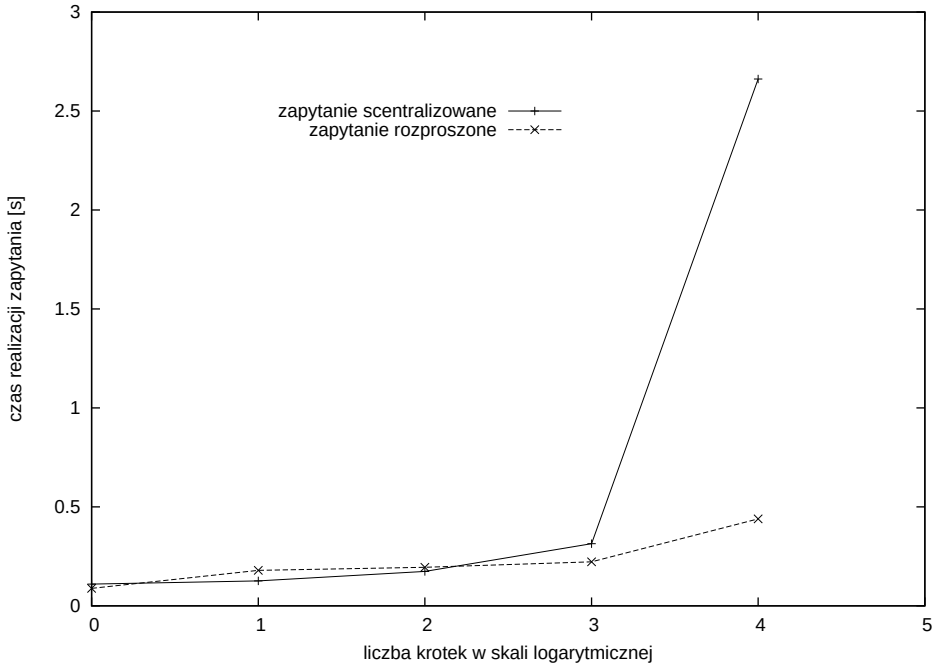
Wyniki obu testów dla liczności krotek tabel od 0 do 10000 reprezentuje tab. 7.2. Na rys. 7.7 przedstawiono dodatkowo porównanie czasów średnich. Badania przepro-

Liczba krotek	**Przetwarzanie scentralizowane** [s]			**Przetwarzanie rozproszone** [s]		
	Test 1	Test 2	Test 3	Test 1	Test 2	Test 3
0	0.136	0.099	0.094	0.083	0.093	0.089
10	0.141	0.126	0.112	0.177	0.176	0.184
100	0.168	0.201	0.152	0.208	0.192	0.184
1000	0.476	0.245	0.222	0.231	0.219	0.217
10000	6.236	0.965	0.784	0.444	0.438	0.437

Tabela 7.2: Wyniki testu szybkości

wadzono na pięciu różnych rozmiarach bazy danych. Dla każdego z tych rozmiarów przeprowadzono testy wykonania zapytania rozproszonego i scentralizowanego. W tabeli przedstawiono po trzy wyniki każdego z wariantów.

Dla rozmiaru danych o liczności krotek od 0 do 1000 porównywane scenariusze mają podobne czasy. Świadczy to na korzyść Polaris przez to, że czas propagacji komunikacji sieciowej nie wykazał znaczącego opóźnienia. Zróżnicowanie wyników nastąpiło przy dziesięciu tysiącach krotek. W tym przypadku Polaris wykazywał co najmniej dwa razy krótszy czas realizacji zapytania. Możliwe to było dzięki zrównolegleniu procesu wydobywania małej porcji danych z niezależnych komputerów. Zainteresowanie może budzić wynik pierwszego testu zapytania scentralizowanego



Rysunek 7.7: Porównanie średnich czasów realizacji zapytania scentralizowanego i rozproszonego

przy tej liczności. Czas 6.236 sekundy uzyskano bezpośrednio po umieszczeniu danych w bazie danych. Wyniki kolejnych wywołań świadczą o optymalizacji wykonywanej przez Postgresql, włączając wykorzystanie pamięci podręcznej. W celu potwierdzenia powyższej hipotezy powtórzono ostatni test wariantu scentralizowanego dla dziesięciu tysięcy krotek. Drugi test dał podobne wyniki. Pierwsze wywołanie zapytania trwało 5.157 sekundy a pozostałe oscylowały wokół wartości średniej 0.8 sekundy. Przy uwzględnieniu tej wartości, czas realizacji zapytania przez Polaris dla 10000 krotek był dwukrotnie mniejszy.

Rozdział 8

Podsumowanie

Praca poruszyła kilka zagadnień z obszernej dziedziny jaką są systemy rozproszone. Są one z pewnością godną uwagi dyscypliną informatyki znajdującą szerokie zastosowania zarówno w nauce, jak i przemyśle. Ich rolę w dzisiejszym świecie doniośle podkreślają słowa autorów książki [11]:

> *„Można dyskutować, czy rozproszone systemy komputerowe są dziś równie ważne dla społeczeństwa jak mosty, samoloty i aparatura medyczna [...]"*

Cel pracy - umożliwienie wykonywania rozproszonych zapytań w środowisku heterogenicznych baz danych - został osiągnięty. Wynikiem prac jest oprogramowanie, które pozwala wykonywać zapytania SQL wymagające pobrania danych z rozproszonych w sieci komputerowej baz danych działających pod kontrolą różnych systemów operacyjnych. Projekt i implementacja oprogramowania poprzedzone zostały analizą metod i technik niezbędnych do realizacji postawionych celów. Dokonano obszernego przeglądu sposobów komunikacji międzyprocesowej w sieciach komputerowych. Jako istotne w realizacji rozproszonego przetwarzania informacji omówiono techniki programistyczne realizacji dostępu do baz danych. Przeprowadzono analizę rozproszonych baz danych, ze szczególnym uwzględnieniem przetwarzania i optymalizacji zapytań. Stało się to podstawą teoretyczną do opracowania procesora zapytań w projektowanej aplikacji Polaris.

Stworzone oprogramowanie pozwala na realizację w pełni funkcjonalnych konfiguracji rozproszonych, heterogenicznych baz danych. Przykładowy, działający system tego typu omówiono w rozdziale siódmym. Pozwolił on zintegrować trzy różne bazy danych pracujące na dwóch całkowicie odmiennych systemach operacyjnych. Ponadto przeprowadzone testy pokazały, że rozproszone przetwarzanie zapytań z wykorzystaniem Polaris może być w pewnych sytuacjach rozwiązaniem problemu szybkiego dostępu do danych.

Istnieje wiele możliwości rozwoju opracowanego systemu. Do najbardziej istotnych można zaliczyć zaimplementowanie dodatkowych instrukcji SQL oraz prowadzenie dalszych badań w zakresie optymalizacji zapytań z ewentualnym rozszerzeniem problemu na obiektowe bazy danych.

Dodatek A

Algebra działań na relacjach

Selekcja - $\sigma(R)$ W wyniku jej zastosowania do relacji R powstaje nowa relacja, do której należy pewien podzbiór krotek relacji R.

Rzutowanie - $\Pi(R)$ Tworzy nową relację, która powstaje z relacji R przez usunięcie z niej pewnych kolumn.

Iloczyn kartezjański - $R \times S$ Daje relację składającą się ze wszystkich możliwych krotek, będących kombinacjami dwóch krotek, po jednej z każdej z wskazanych relacji.

Suma - $R \cup S$ Daje w wyniku relację składającą się ze wszystkich krotek, występujących w jednej z obu wskazanych relacjach.

Przecięcie - $R \cap S$ Przecięcie relacji R i S jest taką relacją, do której należą tylko te elementy, które występują zarówno w relacji R, jak i S.

Różnica - $R - S$ Daje w wyniku relację składającą się ze wszystkich krotek, występujących w pierwszej i nie występujących w drugiej relacji.

Złączenie teta - $R \bowtie_p S$ Wynik iloczynu kartezjańskiego ograniczony do podzbioru krotek spełniających warunek p złączenia.

Złączenie naturalne - $R \bowtie S$ Wynik iloczynu kartezjańskiego ograniczony do krotek, które mają równe wartości dla wspólnych atrybutów pochodzących z łączonych relacji.

Dodatek B

Słownik

B.1 Podstawowe terminy

Dziedziczenie Możliwość wykorzystania istniejących obiektów do tworzenia nowych, bardziej specjalistycznych.

Deskryptor Deskryptory plików są to liczby całkowite używane przez system do identyfikacji plików, z których korzysta konkretny proces. Kiedy system otwiera istniejący plik lub tworzy nowy, uzyskiwany jest deskryptor pliku, który następnie może być użyty podczas czytania z pliku lub pisania do niego [43, str. 29].

Drzewo operacji algebraicznych *(ang. relational algebraic tree)* Jest drzewem, w którym każdy liść jest relacją przechowywaną w bazie danych, a pozostałe wierzchołki drzewa zawierają operacje algebry relacji. Sekwencja przetwarzania przebiega od liści do korzenia, który reprezentuje odpowiedź na zapytanie. W trakcie przetwarzania wierzchołki nie będące liśćmi zawierają relacje pośrednie uzyskane z użyciem operatorów algebry relacji z tych wierzchołków [37, str. 194].

Drzewo wywodu Inaczej drzewo wyprowadzenia, drzewo składniowe, opisuje składniową strukturę wejścia, obrazuje jak z symbolu startowego można wyprowadzić napis w danym języku. Formalną i szczegółową definicję drzewa wywodu można znaleźć np. w [1, str. 28].

Enkapsulacja Ukrywanie szczegółów implementacyjnych obiektów.

Gniazdo Interfejs programów użytkowych przeznaczony do programowania usług sieciowych; określenie opisujące połączenie adresu IP oraz numeru portu; punkt końcowy protokołów warstwy transportowej modelu OSI [42, str. 29 i 69].

Interfejs Powierzchnia stanowiąca wspólną granicę pomiędzy przyległymi obszarami; punkt, w którym współpracują niezależne systemy lub odrębne grupy; urządzenie lub system dzięki, któremu współdziałanie to zachodzi. Zestaw operacji, które wyznaczają usługi oferowane przez klasę lub komponent [8, str. 140].

Komponent Fizyczna, wymienna część systemu, która realizuje pewien zbiór interfejsów [8, str. 140].

Logika biznesowa Logika biznesowa obejmuje mechanizmy i elementy aplikacji, których rola polega na przetwarzaniu danych zgodnie z założonym działaniem aplikacji oraz organizowaniu właściwej strategii działania. Logika biznesowa nie obejmuje swoim zasięgiem zarządzania bazą danych i interakcji z użytkownikiem systemu.

Maszyna Turinga Abstrakcyjna maszyna obliczeniowa złożona z głowicy czytająco-piszącej oraz potencjalnie nieskończonej, posegmentowanej taśmy zawierającej symbole (np. liczby lub operatory działań). Głowica może wykonywać skończony zbiór instrukcji: analizuje zawartość segmentu taśmy i w zależności od własnego stanu i obserwowanego symbolu przesuwa się wzdłuż taśmy, czyta lub zapisuje w segmencie nowy symbol. Wynikowy obraz taśmy, czyli efekt działania maszyny Turinga, zależy od zbioru jej instrukcji i początkowego układu symboli na taśmie.

Model danych Sposób rozumienia organizacji danych i ideologiczne lub techniczne ograniczenia w zakresie, organizacji i dostępu do danych; zespół zasad definiowania danych i operowania danymi oraz zasad integralności danych czyli reguł określających, które stany bazy danych są dozwolone.

Polimorfizm Możliwość umieszczenia w kodzie różnych zachowań w zależności od tego, jaki obiekt został wybrany.

Programowanie dynamiczne Jest to metoda będąca rozszerzeniem metody *"dziel i zwyciężaj"*, która dzieli problem na zależne od siebie podproblemy. W różnych podproblemach wykonywane są wiele razy te same obliczenia. Wyniki obliczeń są zapamiętywane w tablicy pomocniczej, która jest wykorzystywana w kolejnych krokach algorytmu, co eliminuje potrzebę wielokrotnego wykonywania tych samych obliczeń. Programowanie dynamiczne polega więc na wykonaniu obliczeń każdego podproblemu tylko raz i zapamiętaniu jego wyniku w tabeli.

Przetwarzanie danych Wszelkie operacje dokonywane na danych wejściowych powodujące wygenerowanie nowych danych niosących ze sobą żądaną, nową informację (np. wyniki analizy korelacji pozwalające na prognozowanie) lub powodujących zmianę sposobu ich prezentacji (np. przekształcenie tabeli w wykres).

Relacja Relacją R na zbiorach $D_1, D_2, \ldots, D_n$ jest dowolny podzbiór iloczynu kartezjańskiego $D_1 \times D_2 \times \ldots \times D_n$.

System otwarty Zbudowany zgodnie z normą ISO 7498 i zdolny do wymiany informacji z innymi systemami otwartymi.

B.2 Akronimy i skróty

COM Component Object Model jest specyfikacją opisującą współpracę obiektów i ich klientów poprzez interfejsy w binarnym standardzie. Jest też implementacją zwaną biblioteką COM. Implementacja ta jest dostarczona przez biblioteki DLL w systemie Windows.

COM+ COM+ realizuje wiele zadań zarządzania zasobami w szczególności komponentami COM. Powstała w wyniku ewolucji technologii Component Object Model *(COM)*, Microsoft Transaction Server *(MTS)* i Microsoft Message Queue *(MSMQ)*.

CORBA Common Object Request Broker Architecture jest specyfikacją, która definiuje model rozproszonych obiektów.

DCOM Distributed Component Object Model jest rozszerzeniem architektury COM. Pozwala komponentom programowym na komunikację bezpośrednio przez sieć w sposób bezpieczny, niezawodny i sprawnie działający.

EJB Architektura Enterprise Java Beans jest architekturą przeznaczoną do tworzenia aplikacji bazujących na rozproszonych komponentach, wspierających transakcje, zarządzanych przez środowisko wykonawcze w postaci serwera aplikacji.

ISO Międzynarodowa Organizacja Normalizacyjna *(ang. International Organization for Standardization)* założona w 1946 roku, która jest odpowiedzialna za tworzenie międzynarodowych standardów w wielu dziedzinach, łącznie z komputerami i komunikacją.

J2EE Java 2 Platform Enterprise Edition definiuje standard tworzenia aplikacji opartych na architekturze wielowarstwowej. J2EE wykorzystuje język Java jako podstawę programowania logiki aplikacji oraz definiuje środowisko wykonania i model aplikacji.

JMS Java Message Service jest interfejsem programistycznym języka Java, który pozwala aplikacjom tworzyć, wysyłać, odbierać i odczytywać komunikaty. Zapewnia poprawne dostarczenie komunikatu do procesu, który w danej chwili nie jest dostępny. Kolejkuje komunikaty do momentu, aż może je dostarczyć w późniejszym czasie.

MSMQ Microsoft Message Queue jest protokołem pozwalającym aplikacjom na wzajemne wysyłanie komunikatów. Zapewnia poprawne dostarczenie komunikatu do procesu, który w danej chwili nie jest dostępny. Kolejkuje komunikaty do momentu, aż może je dostarczyć w późniejszym czasie.

XPath Język do adresowania fragmentów dokumentu XML [52].

XUpdate Język aktualizacji danych XML; opisujący w składni XML wybór, uaktualnianie i przetwarzanie warunkowe elementów dokumentu XML [53].

Spis rysunków

Bibliografia

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Kompilatory. Metody, reguły i narzędzia. Wydawnictwa Naukowo - Techniczne. Warszawa 2002.

[2] Apache Xindice. http://xml.apache.org/xindice/

[3] Bajer R., Hellen M., Reiser A.: Parralelism and Recovery in Database Systems. ACM TODS 5, No. 2 (June 1980).

[4] Ben-Ari M.: Podstawy programowania współbieżnego i rozproszonego. Wydawnictwa Naukowo - Techniczne. Warszawa 1996.

[5] Beynon-Davies Paul: Systemy baz danych. Wydawnictwa Naukowo - Techniczne. Warszawa 1998, 2000.

[6] Bielecki Jan: Java 3RMI. Podstawy programowania rozproszonego. Wydawnictwo Helion. Gliwice 1999.

[7] Bison 1.35. http://www.gnu.org/manual/bison-1.35/bison.html

[8] Booch Grady, Rumbaugh James, Jacobson Ivar: UML przewodnik użytkownika. Wydawnictwa Naukowo - Techniczne. Warszawa 2002.

[9] Budzyński Robert: Wprowadzenie do technologii baz danych. http://bobo.fuw.edu.pl/DB/

[10] Comparison of ADO.NET and ADO. MSDN for Visual Studio .NET. Microsoft Corporation 2001.

[11] Coulouris G., Dollimore J., Kindberg T.: Systemy rozproszone podstawy i projektowanie. Wydawnictwa Naukowo - Techniczne. Warszawa 1998.

[12] Date C. J.: Wprowadzenie do systemów baz danych. Wydawnictwa Naukowo - Techniczne. Warszawa 1981, 2000.

[13] Eddon Guy, Eddon Henry: COM+ Programowanie. Wydawnictwo RM. Warszawa 2001.

[14] Ellis Jon, Ho Linda, Fisher Linda: JDBC 3.0 Specification. Sun Microsystems 2001.

[15] Robert A. van Engelen: The gSOAP Stub and Skeleton Compiler for C and C++ 2.1.10. Department of Computer Science Florida State University. http://www.cs.fsu.edu/ engelen/soapdoc2.html

[16] eXist. Open Source XML Database. http://exist-db.org/

[17] Flex - a scanner generator. http://www.gnu.org/manual/flex-2.5.4/flex.html

[18] Gabassi Michel, Dupouy Bertrand: Przetwarzanie rozproszone w systemie UNIX. Wydawnictwo Lupus. Warszawa 1995.

[19] Galvin Peter B., Silberschatz Abraham: Podstawy systemów operacyjnych. Wydawnictwa Naukowo - Techniczne. Warszawa 1993, 2000.

[20] Gopalan Suresh Raj: A Detailed Comparison of CORBA, DCOM and Java/RMI. http://www.execpc.com/ gopalan/misc/compare.html

[21] Gruźlewski Tadeusz, Weiss Zbigniew: Programowanie współbieżne i rozproszone. Wydawnictwa Naukowo - Techniczne. Warszawa 1993.

[22] Haas Laure, Lin Eileen: IBM Federated Database Technology. International Business Machines Corporation 2002. http://www7b.software.ibm.com/dmdd/library/techarticle/0203haas/ 0203haas.html

[23] Haase Kim: Java Message Service Tutorial. Sun Microsystems 2002. http://java.sun.com/products/jms/tutorial/index.html.

[24] Jaszkiewicz Andrzej: Inżynieria Oprogramowania. Wydawnictwo Helion. Gliwice 1997.

[25] Jones Anthony, Ohlund Jim: Microsoft Windows - programowanie sieciowe. Warszawa 2000.

[26] Praca zbiorowa pod redakcją A. Karbowskiego i E. Niewiadomskiej - Szynkiewicz: Obliczenia równoległe i rozproszone. Oficyna Wydawnicza Politechniki Warszawskiej. Warszawa 2001.

[27] Kasza Rafał: Obiektowy model baz danych. http://casha.civ.pl/bd/objbd.php

[28] Kasza Rafał: Porównanie Relacyjnych, Obiektowych i Obiektowo-Relacyjnych Baz Danych. http://casha.civ.pl/bd/porownanie.php

[29] Kossmann Donald, Stocker Konrad: Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. http://www.db.fmi.uni-passau.de/˜kossmann

[30] Kossmann Donald: The State of the Art in Distributed Query Processing. http://www.db.fmi.uni-passau.de/~kossmann

[31] Kozielski Stanisław, Szczerbiński Zdzisław: Komputery równoległe. Wydawnictwa Naukowo - Techniczne. Warszawa 1993.

[32] Informix: Informix Extended Parallel Server 8.3. Informix Corporation 1999.

[33] Lausen Georg, Vossen Gottfried: Obiektowe bazy danych. Modele danych i języki. Wydawnictwa Naukowo - Techniczne. Warszawa 2000.

[34] Loney Kevin, Theriault Marlene: Oracle8i. Podręcznik administratora baz danych. Wydawnictwo Helion. Gliwice 2002.

[35] Mazur Zygmunt: Bazy danych. Prace Naukowe Wydziałowego Zakładu Informatyki Politechniki Wrocławskiej z.2. Oficyna Wydawnicza Politechniki Wrocławskiej. Wrocław 2001.

[36] MySQL Reference Manual for version 4.0.6-gamma. http://www.mysql.com/documentation/mysql/

[37] Özsu M. Tamer, Valduriez Patrick: Principles of distributed database systems. Englewood Cliffs, N.J.: Prentice Hall, c1991.

[38] Płoski Zdzisław: Informatyka. Słownik encyklopedyczny. Wydawnictwo Europa. Wrocław 1999.

[39] PostgreSQL 7.2.1 Documentation. http://www.postgresql.org/idocs/

[40] Replikacja danych w SQL Serwerze 7.0. Software 2.0. 2/99 str. 97 i nast.

[41] Stevens W. Richard: Advanced Programming in the UNIX Environment. Addison-Wesley, 1992.

[42] Stevens W. Richard: UNIX. Programowanie usług sieciowych. Tom 1 - API: gniazda i XTI. Wydawnictwa Naukowo - Techniczne. Warszawa 2002.

[43] Stevens W. Richard: Programowanie w środowisku systemu UNIX. Wydawnictwa Naukowo - Techniczne. Warszawa 2002.

[44] Szyperski Clemens: Oprogramowanie komponentowe obiekty to za mało. Wydawnictwa Naukowo - Techniczne. Warszawa 2001.

[45] Tannenbaum Andrew S.: Rozproszone systemy operacyjne. Wydawnictwo Naukowe PWN. Warszawa 1997.

[46] Tari Zahir, Bukhres Omran: Fundamentals of Distributed Object Systems: The CORBA Perspective. John Wiley & Sons 2001.

[47] Ullman Jeffrey D., Widom Jennifer: Podstawowy wykład z systemów baz danych. Wydawnictwa Naukowo - Techniczne. Warszawa 2000.

[48] Vaskevitch David: Strategie klient - serwer. Wydawnictwo IDG Poland S.A. Warszawa 1995.

[49] Williams All: Programowanie Windows 2000 Czarna księga. Wydawnictwo Helion. Gliwice 2000.

[50] W3C: Web Services Activity. http://www.w3c.org/2002/ws/Activity

[51] X-Hive/DB. http://www.x-hive.com/

[52] XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath

[53] XUpdate - XML Update Language. http://www.xmldb.org/xupdate/

Skorowidz