Silesian
University
of Technology

POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

KIERUNEK: INFORMATYKA

Praca dyplomowa magisterska

Analiza cech obiektowych w relacyjnych bazach danych

Autor: inż. Krzysztof Sobocik

Promotor: dr inż. Ewa Płuciennik

Gliwice, wrzesień 2021

Spis treści

Streszczenie

Celem pracy jest analiza cech obiektowych występujących w dostępnych na rynku bazach danych. Na początku pracy przybliżono tematykę relacyjnych, obiektowych i obiektowo-relacyjnych systemów zarządzania bazami danych. Omówiono także paradygmat programowania obiektowego i jego wpływ na rozwój baz danych. Następnie przedstawiono dostępne na rynku systemy obiektowo-relacyjne i szczegółowo przeanalizowano mechanizmy trzech systemów: Oracle Database, IBM Db2 i PostgreSQL. Przedstawiono wszystkie istotne cechy obiektowe dostępne w analizowanych systemach i przeprowadzono ich porównanie z rozwiązaniami opisanymi w standardzie SQL.

W ramach badań, w każdym systemie zaprojektowano obiektowo-relacyjną strukturę bazy danych dla dziedziny hurtowni sprzętów elektronicznych. Projektując model bazy wykorzystano jak najwięcej spośród dostępnych mechanizmów obiektowych. W każdym systemie stworzono także standardową strukturę relacyjną dla tej samej dziedziny. Korzystając ze stworzonej aplikacji, wygenerowano zbiór danych i skrypty wstawiające dane do każdej bazy. Dla wszystkich struktur przygotowano i wykonano zapytania realizujące przykładowe wymogi biznesowe. Porównano treści zapytań dla różnych struktur i czasy ich wykonania. Pracę zamykają wnioski z badań i syntetyczne podsumowanie analizy.

Slowa kluczowe: bazy danych, relacyjne bazy danych, obiektowo-relacyjne bazy danych, paradygmat obiektowy, programowanie obiektowe, ORDBMS, SQL, Oracle, Db2, Postgres

Rozdział 1

Wstęp

Postępujący rozwój technik analizy danych i szeroko pojętego *data science* powoduje, że bazy danych stają się coraz bardziej istotne w systemach informatycznych. Powstaje wiele nowych modeli baz danych, a istniejące systemy poszerzają swoje możliwości by nadążyć za potrzebami rynku. Jednym z najpopularniejszych trendów w informatyce jest podejście obiektowe, czego efekty spotykane są także w systemach bazodanowych. Do dziś, najczęściej stosowanym modelem bazy jest model relacyjny, jednak wiele z najpopularniejszych systemów tego typu wprowadziło do swoich rozwiązań dodatkowe cechy obiektowe, rozszerzające standardowe możliwości bazy. Systemy wspierające mechanizmy obiektowe realizują tak zwany model obiektowo-relacyjny.

Celem niniejszej pracy magisterskiej jest przeanalizowanie cech obiektowych występujących w dostępnych na rynku relacyjnych bazach danych. Omówiony został paradygmat programowania obiektowego oraz jego wpływ na systemy bazodanowe. Przedstawiono charakterystykę relacyjnych, obiektowych i obiektowo-relacyjnych systemów zarządzania bazami danych. Szczegółowo rozpatrywane są mechanizmy obiektowe dostępne w systemach Oracle Database, IBM Db2 oraz PostgreSQL, a także rozwiązania opisane w ramach standardu SQL.

W ramach badań, w każdej bazie stworzono porównywalne struktury obiektowo-relacyjne, opierając się na przykładowej fikcyjnej dziedzinie biznesowej. Dla stworzonych struktur przygotowano zbiór danych i przetestowano działanie baz za pomocą zróżnicowanych zapytań. Poszczególne systemy porównywane są pomiędzy

sobą, a także względem standardowego rozwiązania relacyjnego, stworzonego w tej samej technologii. Badania mają na celu sprawdzenie przydatności dostępnych mechanizmów obiektowych i problemów spotykanych przy ich stosowaniu.

Praca składa się z 6 rozdziałów. Rozdział drugi stanowi wprowadzenie do analizowanego tematu oraz nakreślenie historycznej i obecnej pozycji podejścia obiektowego w bazach danych. Rozdział trzeci przybliża rozpatrywane w pracy systemy zarządzania bazą danych. Rozdział czwarty rozpatruje poszczególne mechanizmy obiektowe, ich opis w standardzie SQL i sposób ich implementacji w analizowanych systemach. W rozdziale piątym opisane są badania i testy rozwiązań obiektowych, które przeprowadzono z wykorzystaniem wybranych baz danych. Pracę zamyka podsumowanie zawierające wnioski końcowe z przeprowadzonej analizy.

Rozdział 2

Analiza tematu

Bazy danych są podstawą wielu systemów informatycznych, od najprostszych aplikacji mobilnych, przez sieciowe portale społecznościowe, aż po ogromne hurtownie danych nastawione na badania analityczne. Są dziś domyślnym rozwiązaniem dla przechowywania danych, naturalną ewolucją utrwalania danych w prostych plikach tekstowych.

Można wyróżnić wiele rodzajów baz danych. Standardowym rozwiązaniem są bazy relacyjne, używane z sukcesem od wielu lat. W ostatnich czasach popularnością cieszą się również szeroko pojęte bazy NoSQL, do których zalicza się większość systemów nierelacyjnych, takich jak bazy dokumentowe, grafowe czy kolumnowe. Dostępność wielu alternatywnych rozwiązań pozwala twórcom systemów informatycznych na dobór modelu bazy najbardziej dopasowanego do rozwiązywanego problemu domenowego i wymogów systemu [27].

Warto zaznaczyć różnicę między bazą danych, a SZBD, czyli systemem zarządzania bazą danych. Precyzyjnie mówiąc, baza danych to zbiór powiązanych ze sobą informacji, przechowywanych w uporządkowanej strukturze, zwykle w formie elektronicznej, a SZBD to oprogramowanie umożliwiające pracę z tą bazą. Zapewnia komunikację z bazą danych i pełni rolę interfejsu, przez który użytkownik wykonuje wszelkie akcje na zbiorze danych. Najczęściej określenie “baza danych” odnosi się do kompletu bazy jako zbioru informacji i towarzyszącego mu systemu zarządzania - takie znaczenie tego pojęcia będzie najczęściej spotykane w ramach niniejszej pracy [17].

W tym rozdziale omówione zostaną trzy typy baz danych istotne dla tematu pracy, czyli bazy relacyjne, obiektowe i ostatecznie obiektowo-relacyjne, łączące cechy obydwu tych podejść. Zostanie przedstawione ich pochodzenie oraz różnice w ich działaniu. Jako wprowadzenie do baz obiektowych i obiektowo-relacyjnych zostanie przybliżony paradygmat programowania obiektowego i jego założenia.

2.1 Relacyjne bazy danych

Relacyjne bazy danych są najczęściej stosowanym rodzajem bazy danych. Według rankingu DB-Engines na rok 2021, cztery najpopularniejsze bazy na świecie wykorzystują model relacyjny, a w pierwszej dziesiątce są tylko trzy bazy o innym modelu [14]. Popularność baz relacyjnych wynika w dużej mierze z ich silnych podstaw teoretycznych i przyzwyczajenia wykształconego w środowiskach programistycznych - przez wiele lat były dominującym rozwiązaniem, gdy alternatywy nie oferowały wystarczających zalet.

Autorem teoretycznych podstaw relacyjnych baz danych jest Edgar F. Codd, który w 1970 po raz pierwszy użył tego określenia w swojej pracy "*A Relational Model of Data for Large Shared Data Banks*". Określił w niej tzw. "Dwanaście Reguł Codd'a", które definiują wymagania stawiane wobec Relacyjnych Systemów Zarządzania Bazą Danych (RSZBD). Dzisiejsze systemy relacyjne na ogół realizują tylko część spośród wszystkich reguł Codd'a [22].

Model relacyjny realizuje intuicyjny sposób przechowywania danych, poprzez grupowanie danych w relacje reprezentowane przez tabele. Każda relacja, a więc także tabela, posiada unikatową nazwę, zbiór atrybutów oraz zawartość w postaci zbioru krotek. Atrybuty reprezentowane są przez kolumny tabeli, a krotki to poszczególne wiersze. Unikalnym identyfikatorem krotki jest tzw. klucz główny, czyli atrybut lub zbiór atrybutów o niepowtarzalnej wartości w danej tabeli. Zależności pomiędzy danymi przechowywanymi w różnych tabelach reprezentowane są za pomocą kluczy obcych, czyli atrybutów przechowujących wartości kluczy innych krotek.

Ogromną zaletą modelu relacyjnego są jego korzenie oparte o matematyczną teorię zbiorów i wynikającą z niej algebrę relacji. Silne i dobrze zbadane podstawy teoretyczne zapewniają niezawodność i wysoką wydajność działania baz relacyj-

nych. Wszystkie zapytania realizowane w relacyjnym modelu sprowadzają się do operacji algebry relacji, z których podstawowe to selekcja, rzutowanie, iloczyn kartezjański, suma, czy różnica. Bardzo istotnym zadaniem algorytmów wchodzących w skład RSZBD jest rozkładanie zapytań na operacje algebry relacji i ich optymalizacja, tak by uzyskać możliwie najszybszy schemat realizacji zapytania.

Bazy relacyjne są ściśle powiązane ze strukturalnym językiem zapytań SQL (ang. *Structured Query Language*). Opracowany przez firmę IBM w latach siedemdziesiątych, a pierwszy raz wykorzystany komercyjnie w bazie Oracle, szybko stał się standardem używanym przez wszystkie relacyjne serwery baz danych. Jest językiem deklaratywnym, dzięki czemu zapytania w SQL skupiają się na oczekiwanych wynikach, a nie na sposobie ich pozyskania - określenie szczegółów realizacji (np. wykorzystanie indeksów) pozostaje rolą SZBD.

Standard SQL przeszedł już wiele modyfikacji. Od pierwszej wersji w 1986 roku co kilka lat pojawiają się rewizje, które wprowadzają nowe obowiązkowe lub opcjonalne mechanizmy i usuwają nieaktualne fragmenty. W ramach standardu opisywane są nie tylko sposoby tworzenia zapytań pobierających dane ale także wiele mechanizmów pobocznych i integracja SZBD z zewnętrznymi językami lub aplikacjami. Poszczególne bazy danych wykorzystują zróżnicowane implementacje języka SQL, które realizują mniejszą lub większą część standardu.

Relacyjne bazy danych przez wiele lat dominowały jako rozwiązanie przechowywania danych, choć w ostatnich kilkunastu latach wzrasta zainteresowanie bazami NoSQL, które realizują potrzeby nowych typów aplikacji. Przy wyborze odpowiedniego systemu bazodanowego należy rozpatrzyć wiele potrzeb systemu z perspektywy biznesowej i technologicznej. Główne zalety, które przemawiają za relacyjną bazą danych to:

- Intuicyjny, a jednocześnie wydajny model relacyjny o silnych podstawach teoretycznych.
- Wysoka elastyczność w tworzeniu zapytań pozwalająca na wyspecjalizowane operacje.
- Duża liczba dostępnych na rynku rozwiązań o wysokim stopniu dopracowania i wsparcia technicznego.

- Transakcyjność pozwalająca zachować spójność przy współbieżnym dostępie do bazy przez wielu użytkowników.
- Język SQL - standard w rozwiązaniach relacyjnych, jego dialekty są podobne w różnych systemach i możliwe jest przenoszenie rozwiązań i zgromadzonej wiedzy.

2.2 Programowanie obiektowe

Programowanie obiektowe to jeden z najpopularniejszych trendów w rozwoju technologii informatycznych. Jest to paradygmat programowania, w którym kluczową rolę pełnią obiekty - niezależne elementy, które zawierają w sobie pewien stan i zachowanie. Początki historii programowania obiektowego sięgają lat 50'tych i 60'tych. Za pierwsze języki o cechach obiektowych uznaje się języki SIMULA i Smalltalk. Od tego czasu powstało wiele języków programowania, które opierają się o obiektowość, wśród nich popularne dzisiaj C#, Ruby, Java i Kotlin [26].

Kluczowymi pojęciami stosowanymi w podejściu obiektowym są klasa i obiekt. Klasa określa definicję obiektu, jego szablon, a obiekty są konkretnymi instancjami klasy. Istotne jest także pojęcie metod, czyli procedur przynależących do konkretnej klasy. To nazewnictwo jest powszechnie stosowane we wszystkich językach obiektowych i będzie wykorzystywane w dalszej części pracy podczas opisu mechanizmów obiektowych w bazach danych.

Choć termin "zorientowane obiektowo" najczęściej stosuje się w odniesieniu do języków programowania, spotyka się jego użycia w kontekście systemów lub narzędzi. Trzymając się idei obiektowości można stworzyć system obiektowy nawet w językach, które oryginalnie nie były do tego przeznaczone. Z programowania obiektowego wynika wiele pochodnych podejść, stosowanych także na wyższym poziomie niż pojedyncza aplikacja. Przykładem może być architektura systemu oparta o zdarzenia, które wysyłane są pomiędzy serwisami wchodzącymi w skład systemu - w takim podejściu każde zdarzenie przesyłane między aplikacjami również jest obiektem i przekazuje informacje o innych obiektach [20].

Najważniejsze cechy spotykane w językach i systemach obiektowych [31]:

- Abstrakcja - wykorzystanie modeli, które nie reprezentują żadnych istnie-

jących obiektów, a jedynie ich kategorie. System może operować na abstrakcjach i ich cechach, bez wiedzy o dokładnym sposobie implementacji. Przykładem abstrakcji jest określenie "roślina" i realizujące tą abstrakcję konkretne typy "dąb" lub "dynia".

- Dziedziczenie - możliwość tworzenia nowych klas poprzez rozszerzenie już istniejących. Tworzone w ten sposób klasy dziedziczą zachowanie rodzica, dzięki czemu nie trzeba ponownie definiować całej logiki działania, a jedynie sprecyzować różnice.

- Polimorfizm - różne obiekty o wspólnej abstrakcji można poddawać tym samym akcjom, bez znajomości ich konkretnych typów. Osiągnięty rezultat operacji jest zależny od wewnętrznej implementacji obiektów.

- Enkapsulacja - zamknięcie stanu wewnętrznego obiektu przed zewnętrznymi ingerencjami. Tylko własne metody obiektu mogą dokonywać zmiany jego stanu, a interakcja z innymi obiektami odbywa się poprzez ograniczony publiczny interfejs.

Podejście obiektowe niesie ze sobą wiele zalet. Dla wielu jest najbardziej intuicyjnym sposobem tworzenia programów, ponieważ obiekty starają się odwzorować prawdziwe byty występujące w rzeczywistości. Podział programu na klasy o ograniczonej odpowiedzialności wprowadza modularyzację, co pozwala rozwiązywać mniejsze problemy i łatwiej testować tworzone rozwiązania. Mechanizmy dziedziczenia i polimorfizm pomagają w wielokrotnym wykorzystaniu istniejącego kodu, jednocześnie zapewniając dużą elastyczność implementacji.

Idee obiektowe są stosowane w dziedzinach oprogramowania, baz i hurtowni danych, sztucznej inteligencji i wielu innych. Cieszą się dużą popularnością szczególnie na etapie analizy i projektowania, gdzie zapewniają stosunkowo proste podejście niewymagające dużej wiedzy. Rozwiązania projektowane od podstaw zgodnie z metodologią obiektową są przejrzyste i łatwe w rozwoju [31].

2.3 Obiektowe bazy danych

Podejście obiektowe naturalnie znalazło swoich zwolenników także w kręgach bazodanowych. Aplikacje tworzone w obiektowych językach programowania korzystające z bazy relacyjnej posiadały wyraźny rozdział w strukturze danych używanej w aplikacji, a przechowywanej w bazie - aplikacja korzysta z obiektów, klas i metod, a relacyjna baza operuje na tabelach, wierszach i zapytaniach SQL. Powstał ruch mający na celu stworzenie bazy, która będzie przechowywać obiekty w takiej postaci, jak rozumiane są w programowaniu obiektowym.

W wyniku działań tego ruchu, w 1989 roku na potrzeby konferencji w Kyoto powstał "*Manifest Zorientowanych Obiektowo Systemów Baz Danych*", w którym autorzy starają się zdefiniować obiektową bazę danych. Wszystkie cechy bazy dzielą na konieczne, opcjonalne i otwarte, czyli pozostawione decyzji użytkownika. Pośród koniecznych cech obiektowego systemu wyróżniono dwie najważniejsze:

- System powinien być Systemem Zarządzania Bazą Danych i zapewniać jego podstawowe funkcje.

- System powinien być zorientowany obiektowo i spójny z działaniem współczesnych języków programowania obiektowego.

Każdą z tych cech rozbito na mniejsze podpunkty i opisano ich znaczenie. Pierwsza składa się z pięciu wymagań: zarządzanie warstwą fizyczną, trwałość, współbieżność, odzyskiwanie i zapytania *ad hoc*. Na drugą cechę składa się aż osiem wymagań: złożone obiekty, tożsamość obiektów, enkapsulacja, typy lub klasy, dziedziczenie, przeładowywanie i późne wiązanie, rozszerzalność i kompletność według definicji Turinga [18].

Dziś przyjęło się stosować mniej precyzyjną definicję, gdzie obiektową bazą danych określa się dowolny zbiór informacji trwale przechowywanych w postaci obiektów. Powstało wiele eksperymentalnych prototypów i komercyjnych baz zorientowanych obiektowo ale żadne nie przyjęły się do powszechnego użytku. Przykłady współcześnie wykorzystywanych systemów tego typu to ObjectDB dostępny w Javie, Realm dostępny na systemy mobilne i stworzony przez twórców MongoDB, czy GemStone Object Server stworzony przez GemStone Systems.

Jednym z głównych celów stosowania obiektowej bazy danych, jest trwałe zapisywanie (ang. *persistence*) obiektów obsługiwanych w aplikacjach, dokładnie w takiej formie w jakiej są przechowywane w pamięci programu. W obiektowych językach programowania instancje obiektów są ulotne i znikają wraz z końcem działania programu - obiektowa baza danych może wydłużyć czas istnienia takich obiektów, tak by można je było odzyskać po zamknięciu i ponownym uruchomieniu aplikacji [32]. By umożliwić takie powiązanie, obiektowe bazy danych muszą zapewnić mechanizmy odwzorowania pomiędzy obiektami przechowywanymi w pamięci ulotnej, a obiektami bazodanowymi. Prowadzi to do silnego związania logiki działania bazy ze wspieranymi językami programowania.

Dużym wyzwaniem w bazach obiektowych jest implementacja relacji między obiektami. Standardowym rozwiązaniem jest przydzielanie obiektom identyfikatorów OID (ang. Object ID), które są unikatowe w całym systemie i mogą służyć jako wskaźniki na obiekty, podobnie do mechanizmu kluczy głównych i obcych. W takim podejściu konieczne jest zapewnienie szybkiego odszukiwania powiązanych obiektów, na przykład wprowadzając mechanizmy indeksów, podobnie jak w bazach relacyjnych, co może być trudniejsze gdy operuje się na obiektach różnej postaci, nieprzechowywanych jako tabele. Podobnie trudna jest wizualizacja powiązań pomiędzy obiektami - unikalne identyfikatory są mniej czytelne niż klucze wskazujące na konkretną tabelę.

Obiektowe systemy przechowywania wychodzą na przeciw potrzeb złożonych aplikacji, pozwalając użytkownikom określać dowolną strukturę obiektów oraz operacje, które mogą być bezpośrednio wykonywane na tych obiektach. Realizuje to ideę enkapsulacji, czyli jedną z podstawowych cech obiektowych języków programowania ale jednocześnie utrudnia tworzenie zapytań - by w pełni korzystać z zalet tego podejścia użytkownik musi wiedzieć jakie metody są przygotowane dla typów obiektów.

Mimo dużego zainteresowania tematem, do dziś nie udało się stworzyć powszechnie stosowanego Obiektowego SZBD. Główne wady utrudniające rozwój obiektowych systemów:

- Niestandardowy model danych - w relacyjnych bazach model danych jest stały i reprezentowany w postaci tabel. Obiektowe bazy zezwalają na więk-

szą swobodę modelowania danych w postaci dowolonych typów, co utrudnia optymalizację zapytań i rozwój wydajnego sposobu przechowywania danych.

- Brak standardowego języka zapytań - w obiektowych bazach danych brakuje odpowiednika języka SQL. Istnieje ustandaryzowany język OQL (ang. *Object Query Language*), jednak nie jest on powszechnie stosowany.

- Brak matematycznych podstaw - obiektowe bazy nie mogą korzystać z szeregu korzyści, które przynosi podstawa teoretyczna w postaci algebry relacji [32].

- Ograniczenia zapytań - obiektowe bazy zapewniają mniejszą swobodę w tworzeniu złożonych zapytań, których budowa silnie zależy od projektu bazy. Relacyjne bazy pozwalają na swobodne łączenie tabel i zagnieżdżanie zapytań, co jest znacznie ograniczone w bazach obiektowych [17].

Przez lata pojawiały się także alternatywne podejścia do przechowywania obiektów. W pracy "Storing and using objects in a relational database" opisano rozszerzenie do relacyjnej bazy danych IBM Db2 nazwane Shared Memory-Resident Cache (SMRC), które pozwala na składowanie obiektów w bazie relacyjnej jako tablice bajtów, dokładnie w takim formacie jaki wykorzystywany jest przez aplikację. Rozwiązanie wspiera język C++ i pozwala na utrwalenie obiektów wraz z relacjami, poprzez odpowiednie pakowanie i odpakowywanie wskaźników. Propozycja ma wiele wad, na przykład brak możliwości podglądu danych, które przechowywane są jako bajty niezrozumiałe dla człowieka, pozostaje jednak sugestią, że podejście obiektowe można połączyć z funkcjonującymi rozwiązaniami relacyjnymi [33].

Dziś obiektowe bazy danych pozostają złożonym, lecz niszowym działem technologii. Prace badawcze prowadzone są sporadycznie, a dużo większą popularnością cieszą się inne podejścia NoSQL, takie jak bazy dokumentowe czy grafowe.

2.4 Obiektowo-relacyjne bazy danych

Mimo braku powszechnego sukcesu obiektowych baz danych, idea wprowadzenia mechanizmów obiektowych do warstwy danych wciąż cieszyła się dużą popu-

larnością. Wiele aplikacji z sukcesem wykorzystywało relacyjne bazy danych ale z biegiem czasu pojawiało się w nich coraz więcej potrzeb realizowanych przez podejście obiektowe. Szczególnie wpływowa okazała popularyzacja nowych rodzajów danych obsługiwanych przez aplikacje - powszechne stały się złożone struktury, dane multimedialne oraz informacje przestrzenne i geograficzne [28]. Wynikiem tych zjawisk było wprowadzenie ograniczonej ilości cech obiektowych do standardu SQL i istniejących rozwiązań relacyjnych.

W standardzie SQL mechanizmy obiektowe wprowadzono już w wersji SQL:1999 i stale aktualizowano w późniejszych wydaniach. Duża część rozszerzeń obiektowych została dołączona do najbardziej centralnej drugiej części standardu zwanej *SQL/Foundation*, a cała część dziesiąta *SQL/OLB Object Language Bindings* poświęcona jest osadzaniu SQL w obiektowym języku Java. Niemal wszystkie opisane w standardzie mechanizmy obiektowe określane są jako opcjonalne, by zapewnić systemom możliwość realizacji tylko podstaw relacyjnych.

Systemy, które implementują opcjonalne rozszerzenia obiektowe opisane w standardzie SQL lub inne funkcje obiektowe nazywa się Obiektowo-Relacyjnymi Systemami Zarządzania Bazą Danych (ORSZBD). Dzisiaj wiele spośród najpopularniejszych komercyjnych systemów bazodanowych reklamuje się jako obiektowo-relacyjne, m.in. Oracle i PostgreSQL. Rozwiązanie to stanowi kompromis pomiędzy podejściem relacyjnym, a czysto obiektowym.

O sukcesie ORSZBD mówiono już w pierwszych latach ich stosowania. W swojej książce "*Object-Relational DBMSs: The Next Great Wave*" z 1996 roku Michael Stonebraker przewidywał, że to podejście stanie się nowym standardem i historia potwierdziła wiele z jego tez. Podejście obiektowo-relacyjne realizuje niemal wszystkie potrzeby użytkowników, pozwalając na jednoczesne wykorzystanie zalet relacyjnej bazy oraz wprowadzanie zaawansowanych mechanizmów obiektowych [34, 35].

Systemy obiektowo-relacyjne posiadają bardzo zróżnicowany zakres cech obiektowych. Często spotyka się mechanizmy unikatowe dla danego systemu, nie występujące w innych. Najczęściej spotykane mechanizmy obiektowe to:

- Typy użytkownika - tworzenie własnych typów danych o skomplikowanych lub zagnieżdżonych strukturach. Stworzone typy można wykorzystać jako

podstawę tabel lub jako typ danych w kolumnie.

- Dziedziczenie - wprowadzane jest na poziomie typów danych lub na poziomie tabel. Pozwala na ponowne wykorzystanie istniejących definicji i wspiera realizację polimorfizmu.

- Enkapsulacja - implementowana za pomocą metod i funkcji operujących na konkretnych typach danych. Ściśle wiąże operacje biznesowe z danymi. Wprowadzenie funkcji o tych samych nazwach operujących na różnych typach danych pozwala na ich polimorficzne wywoływanie.

- Integracja z językami obiektowymi - niektóre systemy pozwalają na bezpośrednie mapowanie typów, metod lub tabel na struktury w pamięci programu, ułatwiając komunikację aplikacji z bazą. Często spotykane są też generatory modeli, mapujące pomiędzy strukturą w bazie a kodem aplikacji.

Najbardziej podstawowym zadaniem ORSZBD jest zmniejszenie różnicy pomiędzy modelem obiektowym spotykanym w aplikacjach a modelem relacyjnym. W tym celu częściej niż bazy obiektowo-relacyjne stosuje się narzędzia mapowania obiektowo-relacyjnego (ang. ORM - *Object-Relational Mapping*). Zapewniają automatycznie generowaną warstwę mapowania pomiędzy klasami używanymi w aplikacji a strukturami relacyjnymi. Najczęściej stosowane narzędzia tego typu to Hibernate popularny w Javie i Entity Framework stosowany w języku C# i środowiskach .NET. Mechanizmy ORM są powszechnie znane i stosowane w środowiskach programistycznych i znacznie ułatwiają pracę z relacyjnymi bazami danych.

Dla baz obiektowo-relacyjnych nie ma dostępnych dobrze rozwiniętych odpowiedników ORM, które realizowałyby mapowanie pomiędzy klasami w językach programowania a strukturami obiektowymi tworzonymi w bazie danych. Cechy obiektowe występujące wśród dostępnych na rynku baz są zbyt zróżnicowane, by powstało uniwersalne narzędzie do automatycznego tworzenia schematu bazy - na ten moment takie rozwiązania pozostają w sferze badań [29].

Poprawne projektowanie struktury bazy relacyjnej jest bardzo istotne by zapewnić wydajność działania systemu. Jako wsparcie procesu analizy i projektowania stosuje się wiele narzędzi CASE (ang. *Computer-Aided Software Engineering*), które umożliwiają wizualizację tworzonych modeli i strukur, generację diagramów,

czy szybką i wygodną refaktoryzację. Podobne wsparcie nie jest dostępne dla większości mechanizmów obiektowych dostępnych w ORSZBD. Brakuje standardowej metodologii projektowania, która zapewniałaby szybką pracę i zadowalający efekt końcowy [25].

Mimo stosunkowych trudności w ich wprowadzeniu, zastosowanie rozwiązań obiektowych może rozwiązać problemy spotykane w czysto relacyjnych systemach. ORSZBD mogą być szczególnie przydatne dla systemów obsługujących bardzo zróżnicowane dane o niestandardowych strukturach, na przykład w domenie architektury lub projektowania przestrzennego [30]. Dobre efekty przynoszą także w systemach decyzyjnych, które oprócz standardowych typów danych obsługują dane multimedialne i tekstowe [28]. Wykorzystanie rozszerzeń obiektowych może pomóc także w procesie edukacji, gdzie rozwiązania obiektowe mogą stanowić intuicyjny wstęp do zapoznania się z działaniem baz danych [36].

Rozdział 3

Omówienie wybranych baz danych

W ramach pracy szczegółowo przeanalizowano możliwości udostępniane przez następujące systemy zarządzania relacyjnymi bazami danych:

- Oracle Database
- PostgreSQL
- IBM Db2

Wszystkie wybrane systemy określają się mianem obiektowo-relacyjnej bazy danych. Podczas wyboru sugerowano się wysoką popularnością systemów, możliwością darmowej instalacji i wykorzystania w celach naukowych oraz dostępnością dokumentacji. W kolejnych podrozdziałach zostaną przybliżone wybrane systemy.

3.1 Oracle Database

Baza Oracle Database, najczęściej nazywana po prostu Oracle, to najstarszy z analizowanych systemów. Pierwsza komercyjna wersja bazy określona Oracle v2 została wydana już w 1979 roku i była pierwszym powszechnie dostępnym relacyjnym systemem zarządzania bazą danych opartym o język SQL [23].

Współcześnie baza Oracle to system wielomodelowy, który w jednym narzędziu łączy wiele modeli bazodanowych [9]. Wspierane modele to:

- Relacyjny, z dodatkowymi mechanizmami obiektowymi, rozbudowanym shardingiem oraz możliwością uruchomienia bazy w trybie pamięci ulotnej (ang. *in-memory*).
- Dokumentowy, ze wsparciem dla formatów JSON, XML i tekstów.
- Baza Danych Przestrzennych, przeznaczona dla danych geometrycznych.
- Grafowy, z dodatkowym wsparciem dla danych w formacie RDF.

Baza Oracle jako jedna z pierwszych zaadoptowała mechanizmy obiektowe i wprowadziła model obiektowo-relacyjny już w wersji 8 w 1997 roku. Posiada najbardziej kompleksowy wachlarz możliwości obiektowych. Wszystkie dostępne możliwości są wyczerpująco i przystępnie opisane w dokumentacji systemu. Do dzisiaj jest najczęściej wybieraną bazą w rozwiązaniach skupiających się na mechanizmach obiektowych [8].

Ciekawą cechą wprowadzoną w wersji Oracle 12 jest tzw. Multitenant Architecture, dzięki której jedna instalacja systemu może funkcjonować jako baza kontenerowa (ang. CBD - *Container Database*), w ramach której można tworzyć wiele baz podłączanych (ang. PDB - *Pluggable Database*). Każda baza PDB jest niezależną i przenośną kolekcją schematów, tabel, widoków i innych elementów. Pozwala to na jednoczesne wykorzystanie bazy danych przez wiele niezależnych aplikacji, znacznie ułatwia przenoszenie bazy i pozwala optymalnie wykorzystać pojedynczą zakupioną licencję.

Oracle Database jest dostępna w wielu edycjach, dostosowanych do potrzeb różnych użytkowników. Oprócz najczęściej używanych edycji Standard i Enterprise, dostępna jest darmowa edycja Oracle Express Edition, która zapewnia komplet funkcji ale ogranicza zasoby sprzętowe do dwóch wątków procesora, 2 GB pamięci RAM i 12 GB danych. Ta edycja została wykorzystana podczas badań przeprowadzanych na potrzeby niniejszej pracy.

Oracle Corporation udostępnia również zintegrowane środowisko pracy dla bazy Oracle pod nazwą Oracle SQL Developer. Zapewnia interfejs graficzny i dostęp do pełnego kompletu funkcji bazy danych, od administracji, przez konfigurację przechowywania danych, aż po wykonywanie zapytań i przeglądanie zawartości bazy.

3.2 IBM Db2

IBM Db2 to rodzina produktów przeznaczonych do zarządzania danymi. Historycznie uwzględniała szereg produktów dostosowanych do wielu systemów wspieranych przez IBM, takich jak *z/OS* czy *IBM i*. W 1990 roku IBM wprowadził model wspólnego produktu dla wielu systemów nazwanego Db2 LUW (Linux-Unix-Windows), który dziś jest najpopularniejszym narzędziem z rodziny Db2. Oprócz niego dostępne są chmurowe wersje Db2 dla IBM Cloud i Amazon Web Services (AWS), o bardzo zbliżonym interfejsie i zakresie możliwości do rozwiązania lokalnego. Rozwiązania IBM nastawione są w pierwszej kolejności na korporacje i klientów dużej skali, intensywnie wykorzystujących system bazodanowy.

Dostępna dziś wersja IBM Db2 w głównej mierze skupia się na modelu relacyjnym, który rozbudowuje pewna liczba zintegrowanych mechanizmów obiektowych. Oprócz tego baza zapewnia wsparcie dla nierelacyjnych danych dokumentowych w formacie JSON i XML [21]. Db2 udostępnia obszerną dokumentację możliwości systemu, zawierającą wiele przykładów.

Ciekawą i często kontrowersyjną cechą Db2 jest sposób obsługi użytkowników. Db2 nie pozwala na tworzenie użytkowników na poziomie samej bazy danych, a zamiast tego wymaga tworzenia użytkowników systemowych, na poziomie systemu Windows czy Linux. Autentykacja jest w dużej mierze zintegrowana z autentykacją systemową i domyślnie po zalogowaniu do systemu uzyskuje się również dostęp do bazy danych - ograniczany jest jedynie zakres tego dostępu, poprzez odpowiednie opcje autoryzacyjne konfigurowane przez administratora.

IBM udostępnia darmową wersję bazy Db2 pod postacią IBM Db2 Community Edition, dawniej zwaną wersją Express-C. Przeznaczona jest do nauki i poznania narzędzia przed zakupem, a także celów badawczych. Ta wersja bazy została wykorzystana na potrzeby niniejszej pracy.

Podobnie jak Oracle, IBM zapewnia zintegrowane środowisko do pracy ze swoją bazą pod nazwą IBM Data Studio. Jest to bardzo rozbudowane środowisko programistyczne stworzone w oparciu o Eclipse. Oprócz komunikacji z bazą danych, funkcji administracyjnych i przeglądania danych, zapewnia także wsparcie dla tworzenia aplikacji wykorzystujących rozwiązania bazodanowe IBM oraz pomaga w ich wdrażaniu i testowaniu.

3.3 PostgreSQL

Baza PostgreSQL, najczęściej nazywana po prostu Postgres, w przeciwieństwie do dwóch poprzednich systemów jest systemem w pełni otwartym (ang. *open-source*). Głównym celem tego systemu jest zapewnienie wysokiej rozszerzalności i jak najwierniejsze spełnienie standardu SQL. Oryginalna nazwa POSTGRES odnosi się do projektu Ingres, który był pierwowzorem stosowanej dziś bazy. Rozwój projektu Ingres i później Postgres był silnie związany z Uniwersytetem Kalifornijskim w Berkeley.

Dzisiaj baza Postgres implementuje większość standardu SQL 2011 i posiada bardzo mało elementów sprzecznych ze standardem. Jednym z niewielu aspektów, w których Postgres odbiega od standardu SQL jest implementacja cech obiektowych. Zarówno zakres dostępnych mechanizmów, jak i składnia ich wykorzystania, znacznie odbiegają od propozycji rozszerzeń obiektowych opisanych w standardzie, co może utrudniać ich zrozumienie i zastosowanie w praktyce. Dostępna jest jednak przystępna dokumentacja systemu, która opisuje wszystkie dostępne możliwości.

Postgres jest bardzo lekkim narzędziem - pełna instalacja zajmuje tylko 60 MB, a każdy pusty klaster bazodanowy 40 MB. Dostępne są oficjalne obrazy pozwalające na łatwe uruchomienie systemu w środowisku skonteneryzowanym za pomocą narzędzia Docker. Z tego względu jest bardzo często wybieranym rozwiązaniem dla niewielkich aplikacji sieciowych lub systemów zbudowanych w architekturze mikroserwisów, gdzie uruchamianych jest wiele serwerów baz danych, przechowujących ograniczony zakres danych systemu [12].

Postgres jest systemem całkowicie darmowym i nie posiada żadnych płatnych wersji. Dla ułatwienia pracy z bazą dostępne jest narzędzie pgAdmin, zapewniające przeglądarkowy interfejs do komunikacji z bazą. To narzędzie również udostępnia kod źródłowy na licencji open-source i jego rozwój opiera się na współpracy ze strony społeczności użytkowników.

3.4 Inne systemy

Wymienione systemy to nie wszystkie SZBD, które realizują model obiektowo-relacyjny - są to jedynie najpopularniejsze spośród dostępnych systemów. Podczas

wyboru baz do analizy, oprócz popularności sugerowano się także dostępnością i kompletnością dokumentacji systemów - Oracle, IBM Db2 i Postgres posiadają bardzo skrupulatnie prowadzoną dokumentację techniczną. Systemy, które posiadają wsparcie dla mechanizmów obiektowych, a nie są omawiane w ramach tej pracy z racji małej popularności, płatnego licencjonowania lub innych problemów to m.in. CUBRID, OpenLink Virtuoso, czy WakandaDB.

Bardzo popularna baza Microsoft SQL Server (MSSQL) nie jest analizowana w pracy, ponieważ posiada bardzo ubogi wachlarz mechanizmów obiektowych. Wspierane są w niej typy tworzone przez użytkownika ale ich tworzenie wymaga uprzedniego napisania i skompilowania biblioteki DLL, na przykład za pomocą środowiska .NET. Ogranicza to znacznie swobodę korzystania z typów użytkownika. MSSQL nie wspiera dziedziczenia, ani dla tabel ani dla typów i nie ma żadnych mechanizmów polimorficznych. Dodatkowo, Microsoft aktywnie zabrania publikacji wyników testów wydajnościowych wykonywanych w oparciu o swój system bez uzyskania odpowiedniej zgody [7], co utrudnia wykorzystanie systemu w badaniach.

Choć cechy obiektowe stają się coraz częstsze w relacyjnych bazach danych, wciąż obecnych jest wiele systemów prezentujących całkowicie relacyjne rozwiązania. Bardzo popularna baza MySQL dostępna na całkowicie darmowej licencji nie posiada żadnych mechanizmów obiektowych. Baza SQLite, ogromnie popularna w rozwiązaniach mobilnych podobnie nie ma wsparcia dla cech obiektowych. Dostępnych jest także wiele baz dokumentowych, kolumnowych i innych, w których można odnaleźć cechy obiektowe, nie są one jednak przedmiotem tej pracy, która skupia się na systemach relacyjnych.

Rozdział 4

Mechanizmy obiektowe

Mimo wyboru tylko trzech Obiektowo-Relacyjnych SZBD, już wśród nich można zaobserwować ogromne różnice w zakresie i sposobie implementacji cech obiektowych. Dodatkowe trudności wynikają ze swobodnej interpretacji standardu SQL - wszystkie analizowane systemy w wielu aspektach wprowadzają zupełnie nowatorskie rozwiązania, często sprzeczne ze standardem.

By uporządkować opis mechanizmów obiektowych, typowe cechy obiektowe zostały podzielone na dziedziny i każda z nich omawiana jest niezależnie. W kolejnych sekcjach przybliżone są rozwiązania obiektowe przedstawione w standardzie SQL oraz implementacja powiązanych rozwiązań dostępna w analizowanych bazach danych. Na koniec rozdziału przedstawiono krótkie podsumowanie analizy.

Źródłem wszystkich informacji zawartych w tym rozdziale jest treść standardu SQL [15, 16] oraz aktualna dokumentacja każdego z systemów [8, 5, 12]. Wiedzę teoretyczną potwierdzono praktycznymi testami przeprowadzonymi na uruchomionych systemach.

4.1 Typy użytkownika

Podstawą wszystkich systemów obiektowych jest mechanizm klas i obiektów. Klasy pełnią rolę schematów, określających strukturę i zachowanie pewnego rodzaju bytów, a obiekty to konkretne instancje danych, spełniające założenia wybranej klasy. Podobnie jest w ORSZBD, gdzie podstawą dla wszystkich rozwiązań

obiektowych jest mechanizm definiowania typów użytkownika i tworzenia instancji obiektów.

W standardzie SQL typy użytkownika były pierwszą wprowadzoną i opisaną cechą obiektową. Określane jako UDT (ang. *User-Defined Type*), pozwalają na tworzenie obiektów o dowolnych złożonych strukturach. Stosując podejście obiektowe, proces definiowania typów powinien być podstawową częścią kreowania struktury bazy danych i poprzedzać etap tworzenia tabel.

Przykład polecenia tworzącego typ zgodnie ze składnią standardu SQL przedstawiono na fragmencie kodu 4.1. Tworząc typ należy podać jego nazwę, atrybuty i ich typ, pożądane modyfikatory oraz nagłówki wszystkich metod, określające nazwę oraz typy parametrów i zwracanej wartości. Ciała metod definiuje się w osobnych poleceniach `CREATE METHOD`, co opisano w sekcji 4.2 Enkapsulacja operacji. Podając typy atrybutów można wskazać również inny typ użytkownika, co pozwala na tworzenie struktur z zagnieżdżonymi obiektami. Niedozwolone jest natomiast podawanie ograniczeń atrybutów, takich jak `NOT NULL`.

```
1 CREATE TYPE Book_Type AS
2   title    VARCHAR(60),
3   author   VARCHAR(100),
4   price    DECIMAL(9,2)
5 INSTANTIABLE
6 NOT FINAL
7 REF IS SYSTEM GENERATED
8 METHOD details( ) RETURNS VARCHAR(200);
```

Kod 4.1: Tworzenie typu użytkownika według standardu SQL.

Możliwe jest wprowadzanie zmian do zdefiniowanych typów użytkownika, za pomocą komendy `ALTER TYPE`. Pozwala ona dodać lub usunąć atrybuty, a także dodać, usunąć lub przesłonić nagłówek metody.

W programowaniu obiektowym oprócz zwykłych klas stosuje się klasy abstrakcyjne, które reprezentują byty abstrakcyjne. Z tego typu klas nigdy nie są tworzone instancje obiektów i zwykle służą jako podstawa do dziedziczenia. Przykładem klasy abstrakcyjnej może być “Figura Geometryczna”, której potomkami są zwy-

kłe klasy “Koło” i “Trójkąt”. Standard SQL umożliwa tworzenie typów abstrakcyjnych poprzez użycie frazy `NON INSTANTIABLE`, która uniemożliwia tworzenie instancji obiektów danego typu. Domyślnie wszystkie tworzone typy traktowane są jako `INSTANTIABLE`.

Według standardu SQL, dla każdego utworzonego typu automatycznie powstaje odpowiadający mu typ referencyjny. Jest to odpowiednik wskaźnika z języków obiektowych - wartość skalarna, która wskazuje i identyfikuje obiekt konkretnego typu użytkownika. Typ referencyjny dla pewnego typu użytkownika `X` oznacza się jako `REF(X)`. Za pomocą typów referencyjnych możliwa jest realizacja powiązań pomiędzy obiektami. Rodzaj typu referencyjnego i sposób przechowywania identyfikatorów obiektu można sprecyzować słowem kluczowym `REF` podczas tworzenia UDT. Dostępne są trzy opcje:

- `REF IS SYSTEM GENERATED` - opcja domyślna, identyfikator jest generowany automatycznie przez system.
- `REF USING` - pozwala wskazać predefiniowany typ danych, który będzie użyty jako identyfikator. Musi być to skończony typ numeryczny lub znakowy.
- `REF FROM` - pozwala wskazać atrybut lub kilka atrybutów typu, które mają stanowić unikalny identyfikator i razem pełnić rolę typu referencyjnego.

Po utworzeniu typu użytkownika, standard SQL pozwala wykorzystać go jako pole innego typu, kolumnę tabeli, podstawę do stworzenia tabeli typowanej, lub jako typ przyjmowany lub zwracany przez dowolną operację.

Fragment kodu 4.2 pokazuje tworzenie typowanej tabeli. Tworzenie tabeli w ten sposób wymaga tych samych uprawnień co tworzenie zwykłej relacyjnej tabeli oraz dodatkowo uprawnienia `USAGE` na wykorzystanym UDT. Fraza `REF IS` służy podaniu nazwy dodatkowej kolumny, która powstanie w tabeli by przechowywać identyfikator obiektu, czyli unikalną wartość wyróżniającą każdy obiekt.

```
1 CREATE TABLE Book
2    OF Book_Type
3    REF IS id_column;
```

Kod 4.2: Tworzenie typowanej tabeli według standardu SQL.

Z typowanej tabeli można korzystać niemal identycznie jak ze standardowej tabeli relacyjnej. Fragment kodu 4.3 przedstawia wstawianie, aktualizowanie i odczytywanie danych.

```
1 INSERT INTO Book (title, author, price)
2 VALUES ('Some Title', 'Some Author', 20.00);
3
4 UPDATE Book SET title = 'New Title';
5
6 SELECT title, author, price
7 FROM   Book
8 WHERE  title LIKE '%Title%';
```

Kod 4.3: Operacje na typowanej tabeli według standardu SQL.

W przypadku tabeli relacyjnej przechowującej typ użytkownika w jednej z kolumn, wstawianie danych jest bardziej złożone i wymaga uprzedniego stworzenia instancji obiektu. Fragment kodu 4.4 przedstawia kolejno tworzenie tabeli zawierającej UDT jako kolumnę (linie 1-3), złożony blok SQL pozwalający wstawić dane do tabeli (linie 5-12) oraz zapytanie odczytujące dane z tabeli (linie 14-15). Przy wstawianiu danych wykorzystano automatycznie tworzony konstruktor bezargumentowy (linia 7). Zapytanie `SELECT` pokazuje wykorzystanie operatora kropki (linia 14), który pozwala odnieść się do atrybutów obiektu.

```
1  CREATE TABLE Book_Rel (
2     stock     INTEGER,
3     book_column Book_Type);
4
5  BEGIN
6    DECLARE b Book_Type;
7    SET b = Book_Type();
8    SET b = b.title('Some Title');
9    SET b = b.author('Some Author');
10   SET b = b.price(20.00);
```

```
11    INSERT INTO Book_Rel VALUES (5, b);
12  END;
13
14  SELECT book_column.title(), book_column.author()
15  FROM   Book_Rel;
```

Kod 4.4: Wykorzystanie UDT jako kolumny według standardu SQL.

Wszystkie analizowane bazy stosują bardzo podobną składnię tworzenia typów. W systemie Oracle rolę typów użytkownika pełnią typy abstrakcyjne (ang. ADT - *Abstract Data Type*). Przykład polecenia `CREATE TYPE` tworzącego typ abstrakcyjny przedstawiono na fragmencie kodu 4.5. Podobnie jak w standardzie SQL, oprócz atrybutów typu podaje się nagłówki metod. Ciała metod definiuje się w osobnej komendzie `CREATE TYPE BODY`, omawianej w sekcji 4.2 Enkapsulacja operacji. W poleceniu tworzącym typ nie stosuje się frazy `REF IS` do zdefiniowania rodzaju typu referencyjnego - w Oracle z góry narzucony jest typ identyfikatora. Fraza `INSTANTIABLE` i jej alternatywa `NON INSTANTIABLE` mają znaczenie zgodne ze standardem SQL.

```
1  CREATE TYPE Book_Type AS OBJECT (
2    title    VARCHAR(60),
3    author   VARCHAR(100),
4    price    DECIMAL(9,2),
5    MEMBER FUNCTION details RETURN VARCHAR(200))
6  INSTANTIABLE
7  NOT FINAL;
```

Kod 4.5: Tworzenie typu w Oracle Database.

Zależnie od potrzeb, obiekty w Oracle mogą być przechowywane w tabelach obiektowych lub relacyjnych. Obiektowe tabele tworzone są na bazie pojedynczego typu i każdy wiersz tabeli reprezentuje instancję obiektu, nazywaną wtedy obiektem wierszowym (ang. *row object*). Obiekty przechowywane w kolumnach tabeli relacyjnej lub wchodzące w skład struktury innego obiektu nazywa się obiektami kolumnowymi (ang. *column object*). Dodatkową opcją udostępnianą przez Oracle

przy tworzeniu typu jest możliwość podania frazy `NON PERSISTABLE`, która powoduje, że instancje tego typu nie mogą być utrwalane i przechowywane w tabelach, a jedynie wykorzystywane jako tymczasowe obiekty w procedurach.

Tworzenie tabeli obiektowej jest bardzo zbliżone do standardu SQL. Przykład przedstawiono na fragmencie kodu 4.6. Fraza `OBJECT IDENTIFIER IS` pozwala wybrać, czy identyfikatory obiektów mają być generowane automatycznie i wstawiane do dodatkowej kolumny `object_id` (opcja `SYSTEM GENERATED`), czy jako identyfikator obiektu należy wykorzystać klucz główny tabeli (opcja `PRIMARY KEY`). Druga opcja wymaga, by przy tworzeniu tabeli utworzyć klucz główny, wskazując któryś z atrybutów wykorzystanego typu.

```
1 CREATE TABLE Book
2     OF Book_Type
3     OBJECT IDENTIFIER IS SYSTEM GENERATED;
```

Kod 4.6: Tworzenie tabeli obiektowej w Oracle Database.

Wstawianie i odczytywanie danych z tabeli obiektowej przedstawiono na fragmencie kodu 4.7. W pierwszym zapytaniu do stworzenia instancji obiektu wykorzystano domyślny konstruktor typu obejmujący wszystkie atrybuty. Wstawianie obiektów kolumnowych zrealizować można w taki sam sposób. Funkcja `VALUE` wykorzystana w drugim zapytaniu pozwala zwracać wiersze tabeli jako instancje obiektów.

```
1 INSERT INTO Book VALUES (
2   Book_Type('Some Title', 'Some Author', 20.0));
3
4 SELECT VALUE(b)
5 FROM Book b
6 WHERE b.title LIKE '%Title%';
```

Kod 4.7: Korzystanie z tabeli obiektowej w Oracle Database.

W IBM Db2 typy tworzone przez użytkownika nazywa się typami strukturalnymi (ang. *structured type*). Przykład komendy tworzącej taki typ przedstawia fragment kodu 4.8. Fraza `INSTANTIABLE` i jej alternatywa `NON INSTANTIABLE` mają

znaczenie zgodne ze standardem SQL. Fraza `REF USING` ma znaczenie podobne do standardu SQL i pozwala określić typ używany jako typ referencyjny. Domyślną wartością jest `VARCHAR(16)FOR BIT DATA`. Fraza `MODE DB2SQL` jest zawsze wymagana i obecnie nie posiada alternatyw. Nagłówki metod podaje się w ostatniej części komendy.

```
1 CREATE TYPE Book_Type AS (
2   title   VARCHAR(60),
3   author  VARCHAR(100),
4   price   DECIMAL(9,2))
5 INSTANTIABLE
6 NOT FINAL
7 REF USING VARCHAR(13) FOR BIT DATA
8 MODE DB2SQL
9 METHOD details() RETURNS VARCHAR(200);
```

Kod 4.8: Tworzenie typu w IBM Db2.

Db2 pozwala na przechowywanie obiektów w kolumnach tabel relacyjnych lub jako wiersze w tabelach typowanych. Tworzenie typowanej tabeli przedstawiono na fragmencie kodu 4.9. Fraza `REF IS` pozwala wskazać dodatkową kolumnę, która powstanie by przechowywać unikatowy identyfikator obiektu. Db2 wymaga podania opcji `USER GENERATED`, wymuszając na użytkowniku podanie wartości identyfikatora przy wstawianiu danych do tabeli.

```
1 CREATE TABLE Book OF Book_Type
2   (REF IS id_column USER GENERATED)
```

Kod 4.9: Tworzenie typowanej tabeli w IBM Db2.

Przykłady wstawiania, aktualizowania i odczytywania danych w typowanej tabeli przedstawiono na fragmencie kodu 4.10. W przeciwieństwie do rozwiązania Oracle, przy wstawianiu danych do typowanej tabeli w Db2 konieczne jest podanie wartości kolumny identyfikatora. Wykorzystuje się w tym celu funkcję tworzenia typu referencyjnego, generowaną automatycznie podczas definiowania typu, z domyślną nazwą odpowiadającą nazwie typu.

```
1 INSERT INTO Book (id_column, title, author, price)
2 VALUES(Book_Type('aaa'), 'Some Title', 'Some Author',
     20.00);
3
4 UPDATE Book SET title='New Title';
5
6 SELECT title, author, price
7 FROM   Book
8 WHERE  title LIKE '%Title%';
```

Kod 4.10: Korzystanie z typowanej tabeli w IBM Db2.

System PostgreSQL do określania typów użytkownika stosuje nazwę typu złożonego (ang. *composite type*) lub typu wierszowego (ang. *row type*). Druga nazwa wynika z faktu, że w PostgreSQL każde stworzenie tabeli relacyjnej powoduje automatyczne powstanie towarzyszącego jej typu złożonego, zawierającego jako atrybuty kolumny tej tabeli. Oprócz tego możliwe jest stworzenie niezależnych typów złożonych za pomocą komendy `CREATE TYPE`, podobnie jak w pozostałych analizowanych systemach.

Przykład komendy tworzącej typ złożony przedstawia fragment kodu 4.11. Postgres posiada najprostszą składnię tej komendy spośród omawianych rozwiązań - oprócz nazwy typu podaje sie jedynie atrybuty i ich typ, bez możliwości podania jakichkolwiek dodatkowych fraz. Nie podaje się również nagłówków metod, ponieważ Postgres w ogóle nie umożliwia tworzenia metod, a jedynie udostępnia rozbudowany mechanizm funkcji, co dokładniej omówiono w sekcji 4.2 Enkapsulacja operacji.

```
1 CREATE TYPE Book_Type AS (
2   title  VARCHAR(60),
3   author VARCHAR(100),
4   price  DECIMAL(9,2)
5 );
```

Kod 4.11: Tworzenie typu w PostgreSQL.

Tworzenie tabeli na podstawie typu w Postgres przedstawia fragment kodu 4.12. Tak stworzona tabela nie różni się niczym od tabeli utworzonej przez jawne podanie kolumn o nazwach i typach odpowiadającym atrybutom wykorzystanego typu złożonego. Wstawianie danych do tabeli przebiega identycznie jak dla standardowej tabeli relacyjnej.

```
1 CREATE TABLE Book OF Book_Type;
```

Kod 4.12: Tworzenie tabeli na podstawie typu w PostgreSQL.

Fragment kodu 4.13 przedstawia wykorzystanie typu złożonego jako kolumny tabeli oraz wstawianie i odczytywanie danych w takiej tabeli. Przy wstawianiu danych wykorzystuje się konstruktor wiersza `ROW`, który pozwala tworzyć instancje obiektów. Przy odczytywaniu danych wykorzystuje się operator kropki, by uzyskać dostęp do atrybutów obiektu.

```
1  CREATE TABLE Book_Rel (
2    stock         INTEGER,
3    book_column   Book_Type
4  );
5
6  INSERT INTO Book_Rel
7  VALUES (5, ROW('Some Title', 'Some Author', 20.00));
8
9  SELECT book_column.title, book_column.author
10 FROM   Book_Rel;
```

Kod 4.13: Wykorzystanie typu złożonego w PostgreSQL.

Wszystkie analizowane systemy posiadają podobne mechanizmy tworzenia typów i instancjonowania obiektów. Dostępne rozwiązania są zbliżone do rozwiązań proponowanych przez standard SQL, jednak wystepują zauważalne różnice w składni i szczegółach implementacji. Od reszty systemów najbardziej odbiega Postgres, w którym typy złożone tworzone są automatycznie wraz z każdą tabelą relacyjną i nie ma tabel typowanych.

4.2 Enkapsulacja operacji

Idea enkapsulacji jest jedną z głównych cech języków i systemów obiektowych. Jest powiązana z ideą abstrakcyjnych typów danych i ukrywaniem informacji w językach programowania. W tradycyjnych SZBD nie stosuje się enkapsulacji - jawnie pokazuje się strukturę obiektów bazodanowych użytkownikom i zewnętrznym aplikacjom. Udostępnia się szereg generycznych operacji, które można stosować w celu wstawiania, modyfikacji, usuwania i odczytywania krotek w dowolnych relacjach w bazie. Zarówno relacja, jak i jej atrybuty są widoczne dla wszystkich użytkowników.

Podejście obiektowe, zamiast na strukturze obiektu, skupia się na definicji jego zachowania, poprzez wskazanie operacji, które można wykonać na obiektach danej klasy. Pewne operacje pozwalają tworzyć obiekty (konstruktory), inne zmieniać stan (mutatory), a inne pobierać informacje o obiekcie lub wykonywać obliczenia (akcesory). Mogą występować także bardziej rozbudowane operacje, które łączą kilka z wymienionych aspektów. Zewnętrzni użytkownicy otrzymują informacje o dostępnych operacjach poprzez tzw. interfejs, który określa nazwy i argumenty możliwych działań. Implementacja operacji i wewnętrza struktura typu jest ukryta, realizując założenia enkapsulacji.

Całkowita enkapsulacja, w której ukryte są wszystkie składowe klasy, jest bardzo rygorystyczna i często stosuje się podział atrybutów i operacji na jawne (publiczne) i ukryte (prywatne). Standard SQL ani żaden z analizowanych systemów nie uwzględnia takich modyfikatorów dostępu. Jedynym sposobem na ograniczenie dostępu do pewnych składowych typów użytkownika jest odpowiednie zarządzanie uprawnieniami do poszczególnych elementów typu.

By realizować enkapsulację, standard SQL udostępnia mechanizm metod, które służą do interakcji z obiektami. Każde wywołanie komendy `CREATE TYPE` w celu stworzenia typu użytkownika powoduje automatyczne stworzenie następujących metod związanych z typem:

- Konstruktor - funkcja o nazwie odpowiadającej nazwie typu, pozwala stworzyć nową instancję obiektu. Domyślnie tworzony jest konstruktor bezargumentowy, którego wywołanie wypełnia wszystkie atrybuty wartością domyślną.

- Akcesory atrybutów - dla każdego atrybutu powstaje bezargumentowa funkcja o nazwie odpowiadającej nazwie atrybutu, która zwraca wartość atrybutu.

- Mutatory atrybutów - dla każdego atrybutu powstaje jednoargumentowa funkcja o nazwie odpowiadającej nazwie atrybutu, która pozwala ustawić wartość atrybutu.

Możliwe jest stworzenie dowolnej liczby dodatkowych metod wykorzystując komendę `CREATE METHOD`. Fragment kodu 4.1 pokazuje stworzenie typu `Book_Type`, w którym zadeklarowano jedną metodę `details()`. Ta deklaracja informuje jedynie o interfejsie danego typu i pozwala korzystać z nazwy metody w definicjach metod lub zapytaniach, jednak wywołanie tej metody przed sprecyzowaniem jej ciała spowoduje wystąpienie błędu. Stworzenie ciała metody przedstawia fragment kodu 4.14. Fraza `FOR` (linia 3) wskazuje typ, którego dotyczy metoda. Fraza `DETERMINISTIC` (linia 4) informuje, że wartość zwracana przez metodę jest deterministyczna i stała dla tych samych warunków wejściowych. W ciele metody można odwoływać się do pól obiektu za pomocą parametru `SELF` (linia 6).

```
1 CREATE METHOD details (SELF Book_Type)
2   RETURNS VARCHAR(200)
3   FOR Book_Type
4   DETERMINISTIC
5   BEGIN
6     RETURN 'Book ' + SELF.title + ' is written by ' +
        SELF.author;
7   END
```

Kod 4.14: Definicja ciała metody według standardu SQL.

W praktyce interfejs typu można swobodnie rozszerzać, dodając nowe metody, których nie uwzględniono przy wywołaniu komendy `CREATE TYPE`. Fragment kodu 4.15 przedstawia komendę tworzącą nowy konstruktor, który przyjmuje trzy argumenty. W konstruktorze istotne jest podanie frazy `SELF AS RESULT` by wskazać, że zwracany jest instancjonowany obiekt.

```
1  CREATE CONSTRUCTOR METHOD Book_Type (SELF Book_Type
       RESULT,
2    t VARCHAR(60), a VARCHAR(200), p DECIMAL(9,2))
3  RETURNS Book_Type
4  FOR Book_Type
5  SELF AS RESULT
6  DETERMINISTIC
7  NO SQL
8  BEGIN
9    SET SELF.title  = 'Some Title';
10   SET SELF.author = 'Some Author';
11   SET SELF.price  = 20.00;
12   RETURN SELF;
13 END
```

Kod 4.15: Stworzenie nowego konstruktora według standardu SQL.

Myląc może być rozróżnienie pomiędzy funkcją a metodą w standardzie SQL. Wszystkie metody definiowane w SQL są funkcjami, których pierwszym parametrem jest powiązany typ użytkownika. Komenda `CREATE METHOD` jest jednym z podtypów komendy `CREATE FUNCTION` i ich składnia jest bardzo zbliżona.

W niektórych językach obiektowych spotyka się z ideą metod statycznych. Tego typu metody można wywoływać bez instancji obiektu, ponieważ związane są z samą klasą i nie wykorzystują żadnych instancjowanych atrybutów. Ten mechanizm opisany jest również w standardzie SQL i komenda `CREATE STATIC METHOD` pozwala tworzyć tego typu metody, z zachowaniem ograniczenia, by w ich ciele nie korzystać z żadnych atrybutów składowych typu.

Metody obiektów wywołuje się za pomocą operatora kropki. Przykład przedstawiono na fragmencie kodu 4.16. Warto zauważyć, że do metod nie przekazuje się parametru `SELF`, automatycznie wynika on z obiektu, na którym wywołano metodę. Fragment pokazuje też wykorzystanie metody statycznej, którą wywołuje się bezpośrednio z nazwy typu.

```
1 SELECT b.details(), Book_Type.static_method('param')
2 FROM Book b;
```

Kod 4.16: Wywołanie metody obiektu i metody statycznej według standardu SQL.

Wszystkie spośród analizowanych ORSZBD wspierają pewne możliwości enkapsulacji. Bazy Oracle i Db2 posiadają rozbudowane mechanizmy metod, zbliżone do rozwiązań opisywanych przez standard SQL. Postgres nie umożliwia tworzenia typowych metod, jednak zamiast tego udostępnia kompleksowe mechanizmy funkcji, które mogą operować na typach użytkownika i w wielu przypadkach skutecznie zastąpić metody.

W Oracle tworzenie typu dzieli się na dwa wywołania. Komenda `CREATE TYPE` tworzy podstawową definicję typu, która zawiera nazwę, atrybuty oraz nagłówki metod, a komenda `CREATE TYPE BODY` pozwala zdefiniować ciała metod. Podział ten może przypominać podział na pliki nagłówkowe i pliki źródłowe stosowany w języku C++. Obie komendy można rozpocząć frazą `CREATE OR REPLACE` zamiast `CREATE` by zastąpić typ lub jego ciało, jeśli już istnieją w systemie. Przykład definicji ciała typu przedstawia fragment kodu 4.17. Oracle, w przeciwieństwie do standardu SQL, wymusza wcześniejsze zadeklarowanie wszystkich metod w definicji typu, nie pozwalając na swobodne dodawanie metod do typu.

```
1 CREATE OR REPLACE TYPE BODY Book_Type AS
2   MEMBER FUNCTION details RETURN VARCHAR(200) IS
3   BEGIN
4     RETURN 'Book ' + SELF.title + ' is written by ' +
        SELF.author;
5   END
6 END
```

Kod 4.17: Definicja ciała typu w Oracle Database.

Podczas tworzenia typu domyślnie powstaje jeden konstruktor, który w parametrach przyjmuje wartości dla wszystkich atrybutów typu. Jest to inne podejście niż w standardzie SQL, gdzie domyślny konstruktor jest bezargumentowy. Oracle nie generuje również metod akcesorów i mutatorów, stawiając zamiast tego na bez-

pośredni dostęp do atrybutów typu. Możliwe jest zdefiniowanie dodatkowych konstruktorów użytkownika, czego przykład przedstawia fragment kodu 4.18. Oprócz tego, możliwe jest definiowanie metod statycznych określanych frazą `STATIC`. Metody wywołuje się identycznie jak opisano w standardzie SQL.

```
1  CREATE OR REPLACE TYPE Book_Type AS OBJECT (
2    ...
3    CONSTRUCTOR FUNCTION Book_Type(
4      SELF IN OUT NOCOPY Book_Type,
5      title VARCHAR(60), price DECIMAL(9,2))
6    RETURN SELF AS RESULT
7    ...
8  );
9
10 CREATE OR REPLACE TYPE BODY Book_Type AS
11   CONSTRUCTOR FUNCTION Book_Type(
12     SELF IN OUT NOCOPY Book_Type,
13     title VARCHAR(60), price DECIMAL(9,2))
14   RETURN SELF AS RESULT IS
15   BEGIN
16     SELF.title  := title;
17     SELF.author := 'Unknown';
18     SELF.price  := price;
19     RETURN;
20   END
21 END
```

Kod 4.18: Definicja konstruktora typu w Oracle Database.

Wszystkie operacje definiowane w bazie Oracle, w tym również metody, dzieli się na funkcje i procedury tworzone za pomocą słów kluczowych `FUNCTION` oraz `PROCEDURE`. Funkcje zwracają zawsze jedną wartość i mogą być wykorzystywane dynamicznie w dowolnych zapytaniach SQL. Procedury nie zwracają wartości i tworzone są w języku proceduralnym PL/SQL, który rozszerza możliwości stan-

dardowego języka SQL.

Specjalnym typem metody stosowanym w Oracle są tzw. metody mapujące (ang. *Map Method*) i metody porządkujące (ang. *Order Method*). Przy tworzeniu typu można zdefiniować jedną z tych metod (lecz nie obie), by umożliwić porównywanie i porządkowanie obiektów danego typu. Metoda mapująca nie przyjmuje żadnych argumentów i ma na celu sprowadzić instancję typu użytkownika do typu podstawowego, który system może wykorzystać w operacjach porównania. Metoda porządkująca ma na celu porównanie dwóch obiektów danego typu. Jako parametr przyjmuje instancję obiektu i porównuje ją z wartością `SELF`, zwracając -1, 1 lub 0, jeśli porównywany obiekt jest uznany za mniejszy, większy lub równy obecnemu.

W systemie Db2 do tworzenia metod służy komenda `CREATE METHOD`, której składnia jest zbliżona do rozwiązania opisanego w standardzie SQL. Komenda pozwala na swobodne i atomowe definiowanie metod, w przeciwieństwie do Oracle, gdzie w jednej komendzie tworzenia ciała typu trzeba jednocześnie zawrzeć ciała wszystkich metod. Tworzenie metod w Db2 wymaga ich wcześniejszej deklaracji w definicji typu. Przykład tworzenia metody przedstawia fragment kodu 4.19. Na przykładzie widać niestandardowe podejście do wywoływania metod - w Db2 stosuje się operator podwójnej kropki ".." by uzyskać dostęp do metod obiektu.

```
1 CREATE METHOD details ()
2   RETURNS VARCHAR(200)
3   FOR Book_Type
4   RETURN 'Book ' + SELF..title + ' is written by ' +
      SELF..author;
```

Kod 4.19: Tworzenie metody w IBM Db2.

W Db2, tak jak w Oracle, również możliwe jest tworzenie metod mapujących i porządkujących. Ich znaczenie i cel są dokładnie takie same - pozwalają porównywać i sortować obiekty danego typu. Db2 nie posiada możliwości tworzenia metod statycznych.

Przy tworzeniu typu w Db2 domyślnie tworzone są takie same metody, jak opisano w standardzie SQL - bezargumentowy konstruktor oraz akcesory i mutatory dla wszystkich atrybutów. Korzystając z instancji obiektów zwykle konieczne jest

wykorzystanie stworzonych metod do operacji na atrybutach, choć zdarzają się odstępstwa od tej reguły - na przykład z typowanych tabel i ich pól można korzystać w standardowy sposób, mimo że przechowują instancje obiektów. W Db2 również można zdefiniować dodatkowe konstruktory, w ciele konstruktora wywołując konstruktor domyślny i ustawiając wartości pól przed zwróceniem obiektu.

System Postgres nie udostępnia mechanizmu metod, definiowanych razem z typem. Spośród trzech analizowanych baz danych najmniej skupia się na zapewnieniu enkapsulacji i powiązaniu operacji z instancjami obiektów. Zamiast metod, Postgres udostępnia jedynie rozbudowane narzędzia do tworzenia i wykorzystywania funkcji. Funkcje mogą operować na dowolnych typach, w tym również typach złożonych definiowanych przez użytkownika. Treść funkcji tworzona jest w języku proceduralnym, który rozszerza możliwości standardowego języka SQL.

Z racji braku metod, w Postgres nie są generowane akcesory, mutatory ani konstruktor przy tworzeniu typu. Dostępny jest uniwersalny konstruktor wiersza wywoływany słowem kluczowym `ROW`, który nie jest powiązany z żadnym konkretnym typem. Wywołując konstruktor wiersza z dowolnymi parametrami można tworzyć instancje tymczasowych typów anonimowych. Przykłady użycia konstruktora przedstawia fragment kodu 4.20. Ostatnie wywołanie w przykładzie pokazuje jak można wykonać jawne rzutowanie na wskazany typ złożony.

```
1 SELECT ROW(1, 50.0 ,'This is an Anonymous Type');
2 SELECT ROW('Title','Author', 20.0);
3 SELECT ROW('Title','Author', 20.0)::Book_Type;
```

Kod 4.20: Wykorzystanie konstruktora wiersza w PostgreSQL.

Ciekawą cechą w Postgres jest wysoka swoboda notacji przy obsłudze atrybutów typów złożonych oraz funkcji, które korzystają z typów. Notacja funkcyjna może być zastosowana do uzyskania pól obiektu i co ważniejsze, funkcje przyjmujące jako parametr obiekt danego typu można wywołać w notacji atrybutowej. Przykłady tego zachowania przedstawia fragment kodu 4.21, gdzie oba zapytania pokazują równoważne sposoby dostępu do odpowiednio atrybutu typu i funkcji korzystającej z typu. Możliwość stosowania notacji atrybutowej przy korzystaniu z funkcji jednoargumentowych pozwala w ograniczony sposób symulować mecha-

nizm metod.

```
1 SELECT b.title, title(b)
2 FROM Book_Rel b;
3
4 SELECT details_func(b), b.details_func
5 FROM Book_Rel b;
```

Kod 4.21: Przykłady równoważnych notacji w PostgreSQL.

4.3 Dziedziczenie

Dziedziczenie polega na budowaniu nowych klas w oparciu o istniejące i pozwala tworzyć hierarchiczne rodziny klas. Klasy potomne otrzymują po swoich rodzicach atrybuty oraz zachowanie, jednocześnie pozwalając na zmianę lub rozbudowanie szczegółów implementacji. Dziedziczenie umożliwia ponowne wykorzystanie kodu i pozwala na polimorficzne traktowanie obiektów.

W obiektowo-relacyjnych systemach zarządzania bazą danych można spotkać dwie wersje dziedziczenia: dziedziczenie typów i dziedziczenie tabel. Dziedziczenie typów odpowiada mechanizmowi dziedziczenia klas znanemu z języków obiektowych. Dziedziczenie tabel pozwala na odwzorowanie hierarchii typów w tabelach przechowujących dane.

Standard SQL opisuje dziedziczenie typów, realizowane za pomocą frazy `UNDER` wskazującej rodzica przy tworzeniu typu użytkownika. Przykład tworzenia typu dziedziczącego przedstawiono na fragmencie kodu 4.22. Nowo stworzony UDT posiada wszystkie atrybuty rodzica oraz jeden dodany atrybut. Podczas dziedziczenia można nadpisywać metody typu bazowego za pomocą frazy `OVERRIDING`. Po nadpisaniu metody należy stworzyć jej ciało za pomocą `CREATE METHOD`, podobnie jak dla metody bazowej.

Podobnie do dziedziczenia typów można stosować dziedziczenie tabel. Wymagane jest by relacja dziedziczenia w tabelach odpowiadała takiej samej relacji dziedziczenia w typach. Druga część fragmentu kodu 4.22 pokazuje wykorzystanie komendy `CREATE TABLE` z frazą `UNDER` do stworzenia podtabeli.

```
1 CREATE TYPE Novel_Type UNDER Book_Type AS
2   genre VARCHAR(20)
3 NOT FINAL
4 OVERRIDING METHOD details( ) RETURN VARCHAR(200);
5
6 CREATE TABLE Novel UNDER Book;
```

Kod 4.22: Dziedziczenie typu i tabeli według standardu SQL.

By typ użytkownika mógł być użyty w hierarchii dziedziczenia jako rodzic, konieczne jest użycie frazy `NON FINAL` podczas tworzenia tego typu. Domyślnie nadawana jest wartość `FINAL`, która uniemożliwia tworzenie podtypów. W SQL dziedziczenie jest zawsze jednobazowe, czyli typ może posiadać tylko jedną klasę bazową (rodzicielską).

Oracle udostępnia bardzo podobne mechanizmy dziedziczenia typów co standard SQL, nie umożliwia jednak realizacji dziedziczenia tabel. Dziedziczenie typów wykonuje się poleceniem `CREATE TYPE` z frazą `UNDER`, co przedstawiono na fragmencie kodu 4.23. Możliwe jest nadpisywanie metod za pomocą frazy `OVERRIDING`. Po nadpisaniu metody, jej ciało tworzy się w komendzie `CREATE TYPE BODY`.

Oracle nie pozwala na tworzenie tabel dziedziczących po innych tabelach ale nic nie stoi na przeszkodzie by stworzyć niepowiązaną tabelę obiektową na bazie podtypu. Ciekawym zachowaniem dostępnym w Oracle, jest możliwość wstawiania instancji podtypów do tabeli obiektowej zbudowanej na bazie rodzica. Wszystkie dane, wraz z dodatkowymi atrybutami typów potomnych, zostaną poprawnie przechowane i możliwe jest korzystanie z metod obiektów.

```
1 CREATE TYPE Novel_Type UNDER Book_Type (
2   genre VARCHAR(20)
3   OVERRIDING MEMBER FUNCTION details RETURN VARCHAR(200)
      )
4 NOT FINAL;
5
6 CREATE TYPE BODY Novel_Type AS
```

```
7    OVERRIDING MEMBER FUNCTION details RETURN VARCHAR(200)
         IS
8    BEGIN
9      RETURN  'Novel ' + SELF.title + ' in genre ' + SELF.
           genre + ' is written by ' + SELF.author;
10   END
11 END
```

Kod 4.23: Dziedziczenie typu w Oracle Database.

IBM Db2 udostępnia najbardziej rozbudowane mechanizmy dziedziczenia spośród omawianych systemów. Dziedziczenie typów realizuje się podobnie jak w standardzie SQL, za pomocą słowa kluczowego `UNDER` w komendzie `CREATE TYPE`. Tutaj również możliwe jest nadpisanie metody poprzez frazę `OVERRIDING` i określenie ciała metody w komendzie `CREATE METHOD`. Dostępne jest także dziedziczenie tabel w komendzie `CREATE TABLE`. Przykłady obu typów dziedziczenia przedstawiono na fragmencie kodu 4.24. Przy tworzeniu podtabeli wymagane jest podanie frazy `INHERIT SELECT PRIVILIGES`, która sprawia, że wynikowa podtabela jest dostępna dla wszystkich użytkowników posiadających uprawnienia do tabeli bazowej.

```
1 CREATE TYPE Novel_Type UNDER Book_Type AS (
2   genre VARCHAR(20))
3 MODE DB2SQL
4 OVERRIDING METHOD details() RETURNS VARCHAR(200);
5
6 CREATE TABLE Novel OF Novel_Type UNDER Book
7 INHERIT SELECT PRIVILEGES;
```

Kod 4.24: Dziedziczenie typu i tabeli w IBM Db2.

Tabele stworzone w hierarchii dziedziczenia są ze sobą powiązane. Obiekty wstawione do podtabeli są dostępne z poziomu rodzica, ale wstawianie obiektów do tabeli rodzica nie wprowadza danych do podtabeli. Na fragmencie kodu 4.25 przedstawiono przykładowe zapytania korzystające z hierarchii tabel. Wyniki za-

pytań SELECT prezentują opisaną zależność pomiędzy tabelami - podtabela zawiera wpisy tylko dla obiektów podtypu, lecz tabela rodzica zawiera wpisy dla wszystkich typów w hierarchii. Przy wykonywaniu zapytania SELECT możliwe jest ograniczenie wyników do jednej konkretnej tabeli za pomocą słowa kluczowego ONLY.

```
1  INSERT INTO Book (id_column, title, author, price)
2  VALUES(Book_Type('A'), 'Book A', 'Author A', 20.00)
3
4  SELECT COUNT(*) FROM Book;  % Returns: 1
5
6  INSERT INTO Novel (id_column, title, author, price,
       genre)
7  VALUES(Novel_Type('B'), 'Book B', 'Author B', 50.00, '
       science-fiction')
8
9  SELECT COUNT(*) FROM Novel; % Returns: 1
10 SELECT COUNT(*) FROM Book;  % Returns: 2
```

Kod 4.25: Zależności w hierarchii tabel w IBM Db2.

Postgres w kwestii dziedziczenia posiada najprostsze mechanizmy spośród omawianych systemów. W tym systemie nie ma możliwości dziedziczenia typów. Dostępne jest jedynie dziedziczenie na poziomie tabel, realizowane za pomocą słowa kluczowego INHERITS w komendzie CREATE TABLE. Przykład tworzenia podtabeli przedstawia fragment kodu 4.26. Dokumentacja Postgres odnosi się do dziedziczenia typów opisanego w standardzie SQL i wyraźnie zaznacza, że wprowadzone rozwiązanie w wielu aspektach odbiega od standardu.

```
1  CREATE TABLE Novel_Rel (
2    genre  VARCHAR(20),
3  ) INHERITS (Book_Rel);
```

Kod 4.26: Dziedziczenie tabeli w PostgreSQL.

Korzystanie z hierarchii tabel funkcjonuje podobnie jak w Db2. Wstawienie danych do podtabeli powoduje, że są one dostępne w tabeli rodzica, lecz nie od-

wrotnie. Tutaj również możliwe jest ograniczenie zakresu zapytań do jednej tabeli z hierarchii dziedziczenia za pomocą słowa kluczowego `ONLY`. Przykład takiego zapytania przedstawiono na fragmencie kodu 4.27.

```
1 SELECT title, author
2 FROM Book_Rel
3 % Returns objects from Book_Rel and Novel_Rel tables.
4
5 SELECT title, author
6 FROM ONLY Book_Rel;
7 % Returns objects only from Book_Rel table.
```

Kod 4.27: Korzystanie z hierarchii tabel w PostgreSQL.

Mechanizmy dziedziczenia w Postgres zawierają szereg wad. Dużym problemem jest brak przenoszenia kluczy obcych, ograniczeń pól oraz indeksów przy tworzeniu podtabeli. Wymóg unikalności pola w tabeli rodzica może być swobodnie ignorowany w podtabeli. Takiej sytuacji nie rozwiązuje nawet ręczne nałożenie ograniczenia unikalności na kolumnę w podtabeli, bo ta sama wartość może zostać wprowadzona w tabeli rodzica i potomka bez naruszenia żadnego z ograniczeń. Oprócz tego, nie wszystkie komendy systemu wspierają operacje na hierarchii tabel. W szczególności problematyczne są komendy związane z utrzymywaniem i optymalizacją działania bazy, na przykład polecenie `REINDEX`.

Jako dodatkowa opcja, dziedziczenie tabel w bazie Postgres może być wykorzystane do realizacji partycjonowania. Partycjonowanie tabel ma na celu podzielenie dużego zakresu danych na mniejsze, by zwiększyć wydajność przeszukiwania. Zwykle taki podział wykonuje się korzystając z funkcji hashującej wartość atrybutu lub określając zakresy danych przydzielane do konkretnych partycji. Alternatywą może byc wykorzystanie dziedziczenia tabel, gdy możliwe jest dokładne określenie różnic pomiędzy typami danych. Takie rozwiązanie ma pewne zalety, jak możliwość dodania nowych kolumn do podtabeli (odpowiednika partycji) czy wykorzystanie wielu potomków. Mimo to, wciąż obecne są omówione wcześniej wady dziedziczenia tabel w Postgres, które mogą okazać się zbyt krytyczne dla wielu użytkowników.

4.4 Polimorfizm

Polimorfizm to zasada, według której wiele typów obiektów może charakteryzować się tym samym interfejsem. Najprostszym i najczęściej występującym przejawem poliformizmu jest możliwość operowania na pewnym podtypie tak, jakby był instancją typu rodzica. Ignorowane są wtedy dodatkowe informacje określone w definicji podtypu. Bardziej zaawansowanym mechanizmem jest polimorficzny wybór zachowania obiektu, zależnie od szczegółów jego implementacji, mimo operowania na wskaźnikach typu bazowego. Obydwa te zjawiska spotykane są w ORSZBD.

Standard SQL przedstawia ogólną zasadę, według której instancja podtypu może być wykorzystana w każdym kontekście wymagającym typu rodzica. Każda operacja wymagająca pewnego typu użytkownika może operować na typie dziedziczącym po tym typie. Możliwe jest także jawne rzutowanie na typ rodzica za pomocą operatora `AS` lub specjalnie przygotowanych funkcji rzutowania `CAST`.

Według standardu SQL, wiele metod w rodzinie typów może posiadać taką samą nazwę. Zadaniem SZBD jest dobór właściwej metody bazując na sygnaturze metody i typie obiektu, na którym wywołano metodę. Ten sam mechanizm działa w przypadku wywołania funkcji lub procedury - jeśli dostępne są dwie definicje funkcji, z których jedna przyjmuje jako parametr pewien typ użytkownika, a druga typ pochodny od tego typu, to zawsze wybrana zostanie precyzyjniej dopasowana wersja funkcji.

Wszystkie analizowane systemy wspierają podstawowe zastosowania polimorfizmu. Bazy Oracle i Db2 posiadają mechanizm przesłaniania metod w podtypach i w obydwu możliwe jest polimorficznie korzystanie z tych metod.

W Oracle do tabeli obiektowej można wprowadzać dane typu bazowego tabeli lub dowolnego podtypu. Zapytania korzystające z metod obiektów przechowywanych w tabeli będą automatycznie polimorficzne. Przykład takiego wykorzystania tabeli obiektowej przedstawia fragment kodu 4.28.

```
1 INSERT INTO Book VALUES (Book_Type('A', 'Author AA',
    20.0));
2 INSERT INTO Book VALUES (Novel_Type('B', 'Author BB',
    50.0, 'thriller'));
```

```
3
4 SELECT b.details() FROM Book b;
5 % Returns:
6 % 1. "Book A is written by Author AA"
7 % 2. "Novel B in genre thriller is written by Author BB"
```

Kod 4.28: Przykład polimorfizmu w Oracle Database.

Db2 pozwala na bardzo podobne wykorzystanie przesłanianych metod. Zbudowanie hierarchii tabel i wypełnienie jej obiektami różnych typów, pozwala operować na tabeli będącej korzeniem hierachii i korzystać z wszystkich danych dostępnych w podtabelach. Jeśli w podtypach nadpisano jakąś metodę, wywołanie tej metody na tabeli rodzica spowoduje zwrócenie polimorficznych wyników odpowiednich dla typów przechowywanych obiektów. By ograniczyć zasięg operacji do jednej tabeli w hierarchii, a jednocześnie ograniczyć działanie polimorfizmu, można skorzystać z frazy `ONLY`.

Możliwości dostępne w Postgres są bardziej ograniczone. Podobnie jak w Db2, hierarchia tabel pozwala na automatyczne operowanie na obiektach i typach przechowywanych w podtabelach. Ograniczenie zasięgu operacji możliwe jest za pomocą frazy `ONLY`. Postgres nie posiada jednak metod i jego funkcje nie są polimorficzne. Możliwe jest stworzenie wielu funkcji o tej samej nazwie przyjmujących parametry różnych typów, jednak proces dobrania wersji funkcji nie sprawdza faktycznego typu przekazanych obiektów, lecz dobiera funkcję w uproszczony sposób. W hierarchii tabel, wywołanie funkcji na tabeli rodzica nie zapewni wywołania różnych funkcji dla obiektów w podtabelach. Operując na typie rodzica konieczne jest jawne rzutowanie na podtyp by wykorzystać metodę podtypu.

Na uwagę zasługuje jednak dodatkowy mechanizm dostępny w Postgres w postaci typów polimorficznych (ang. *polymorphic type*). Są to dostępne z góry typy służące do tworzenia tzw. funkcji polimorficznych, które jako parametry mogą przyjmować szeroki zakres typów i odpowiednio dostosować swoje działanie. Typy polimorficzne posiadają nazwy rozpoczynające się od “any” i są wśród nich m.in. typy `anyelement`, `anyarray`, `anynonarray` i `anyenum`. Można je wykorzystywać tylko podczas definiowania funkcji.

4.5 Relacje i struktury zagnieżdżone

Bazy relacyjne pozwalają modelować zależności pomiędzy obiektami za pomocą kluczy obcych. W podejściu obiektowo-relacyjnym również istotne jest tworzenie struktur składających się z wielu obiektów. Standard SQL opisuje dwa mechanizmy modelowania złożnych struktur - zagnieżdżanie oraz referencję.

Zagnieżdżanie obiektów polega na wykorzystaniu typu użytkownika jako atrybutu innego typu. Typ zagnieżdżony staje się częścią zawierającego go obiektu. Przykład zagnieżdżenia przedstawia fragment kodu 4.29. Utworzona tabela przechowuje produkty, a w każdym produkcie zawarty jest obiekt z informacjami o wymiarach.

```
1 CREATE TYPE Dimensions_Type AS
2   x  INTEGER
3   y  INTEGER;
4
5 CREATE TYPE Product_Type AS
6   product_id INTEGER
7   dims       Dimensions_Type;
8
9 CREATE TABLE Product OF Product_Type;
```

Kod 4.29: Zagnieżdżanie typu według standardu SQL.

Drugim i bardziej zaawansowanym sposobem modelowania złożonych struktur jest wykorzystanie referencji. Według standardu SQL, atrybut UDT może być referencją do obiektu, przechowywanego w tej samej lub innej tabeli. Wartość referencyjna `REF` to wartość, która wskazuje na wiersz w typowanej tabeli. Typ takiej wartości określa się jako `REF(typ_wskazywany)`. Każda wartość referencyjna ma przydzielony zakres (ang. *scope*), czyli nazwę tabeli, w której mogą znajdować się wskazywane obiekty. Referencje zapewniają łatwy mechanizm do nawigacji pomiędzy obiektami i umożliwiają modelowanie zależności, w szczególności relacji wiele-do-jednego.

Przykład tworzenia UDT, gdzie jeden z atrybutów to typ referencyjny przedsta-

wia fragment kodu 4.30. Na tym samym przykładzie przedstawiono też wstawianie danych z referencją (linie 14-15). Do określenia zakresu referencji służy fraza `WITH OPTIONS SCOPE` (linia 10).

```
1  CREATE TYPE Company_Type AS
2    company_id  INTEGER;
3
4  CREATE TABLE Company OF Company_Type;
5
6  INSERT INTO Company (company_id) VALUES (123);
7
8  CREATE TYPE Employee_Type AS
9    employee_id  INTEGER,
10   company      REF(Company_Type) WITH OPTIONS SCOPE
       Company;
11
12 CREATE TABLE Employee OF Employee_Type;
13
14 INSERT INTO Employee (employee_id, company)
15 VALUES (400, (SELECT REF(c) FROM Company c WHERE c.
     company_id = 123));
```

Kod 4.30: Tworzenie typu i tabeli z referencją według standardu SQL.

By uzyskać dostęp do obiektów powiązanych przez referencję, w zapytaniach wykorzystuje się operator dereferencji "->". Przykład zapytania wykorzystującego dereferencję przedstawia fragment kodu 4.31. Przedstawione zapytanie pozwala uzyskać pracowników firmy o identyfikatorze 123.

```
1 SELECT  e.employee_id
2 FROM    Employee e
3 WHERE   e.company->company_id = 123;
```

Kod 4.31: Zapytanie wykorzystujące dereferencję według standardu SQL.

Każdy z trzech analizowanych systemów pozwala tworzyć struktury z wyko-

rzystaniem zagnieżdżania, które realizowane jest w bardzo prosty sposób, zgodnie ze standardem SQL. Tworzenie referencji do obiektów możliwe jest tylko w Oracle i Db2. W Postgres relacje między obiektami (tabelami) realizować można tylko w sposób relacyjny, korzystając z kluczy obcych.

Oracle umożliwia zagnieżdżanie obiektów tak jak opisano w standardzie SQL. Dowolny atrybut obiektu może być instancją typu użytkownika. Korzystanie z referencji jest również zbliżone do rozwiązania ze standardu. Przykład przedstawiono na fragmencie kodu 4.32. Przy tworzeniu typu nie wskazano zakresu referencji - Oracle pozwala na ominięcie definicji zakresu i mieszanie referencji do różnych tabel, kosztem gorszej wydajności operacji dereferencji.

```
1  CREATE TYPE Employee_Type AS OBJECT (
2    employee_id   INTEGER,
3    manager       REF Employee_Type
4  );
5
6  CREATE TABLE Employee OF Employee_Type;
7
8  INSERT INTO Employee VALUES (Employee_Type(123, NULL));
9
10 INSERT INTO Employee
11   SELECT Employee_Type (400, REF(e))
12   FROM Employee e
13   WHERE e.employee_id = 123;
```

Kod 4.32: Korzystanie z referencji w Oracle Database.

W celu modelowania relacji jeden-do-wielu, Oracle udostępnia także mechanizm zagnieżdżonych tabel (ang. *Nested Table*), który opisany jest w sekcji 4.6 Typy kolekcji.

Dereferencja w Oracle następuje niejawnie i domyślnie. Zamiast operatora dereferencji zawsze wykorzystuje się zwykły operator kropki. Przykład zapytania SELECT z dereferencją przedstawia fragment kodu 4.33. Dereferencja w Oracle działa w języku SQL, ale nie w proceduralnym języku PL/SQL, który wykorzysty-

wany jest do tworzenia funkcji, procedur oraz metod, co ogranicza ich możliwości w podejściu obiektowo-relacyjnym.

```
1 SELECT  e.employee_id
2 FROM    Employee e
3 WHERE   e.manager.employee_id = 123;
```

Kod 4.33: Zapytanie z dereferencją w Oracle Database.

W systemie Db2 typ referencyjny określa się podczas definicji typu użytkownika, ale zakres referencji można sprecyzować dopiero przy tworzeniu typowanej tabeli. Jest to bardziej elastyczne podejście niż proponowane w standardzie SQL. Przykład tworzenia typów i tabel z wykorzystaniem referencji przedstawiono na fragmencie kodu 4.34. Dereferencja w Db2 wymaga użycia specjalnego operatora dereferencji "->", zgodnie z rozwiązaniem opisanym w standardzie SQL.

```
1  CREATE TYPE Company_Type AS (company_id INTEGER)
2  MODE DB2SQL;
3
4  CREATE TYPE Employee_Type AS (
5    employee_id INTEGER,
6    company REF(Company_Type))
7  MODE DB2SQL;
8
9  CREATE TABLE Company OF Company_Type (
10   REF IS Oid USER GENERATED);
11
12 CREATE TABLE Employee OF Employee_Type (
13   company WITH OPTIONS SCOPE Company,
14   REF IS Oid USER GENERATED);
```

Kod 4.34: Korzystanie z referencji w IBM Db2.

Zarówno standard SQL, jak i implementacje poszczególnych systemów, nie pozwalają tworzyć ograniczeń dla wartości referencyjnych. Nie ma również sposobu na wymuszenie więzów integralności przy korzystaniu z referencji - każda wartość

`REF` może wskazywać na nieistniejący obiekt. Może to prowadzić do większego ryzyka nieprawidłowości danych, niż przy zastosowaniu kluczy głównych i obcych jak w podejściu czysto relacyjnym.

4.6 Typy kolekcji

Złożone typy kolekcji nie są mechanizmem typowo obiektowym, lecz bardziej uniwersalnym, spotykanym w odmiennych formach w wielu różnych systemach. Mimo to, często uznawane są za element konieczny w językach obiektowych, bo umożliwiają tworzenie obiektów o skomplikowanych strukturach w relacjach jeden-do-wielu, wiele-do-wielu lub innych. Standard SQL opisuje dwa rodzaje kolekcji:

- `ARRAY` - tablica, uporządkowana kolekcja o określonym rozmiarze, w której każdy element ma przyporządkowaną pozycję;
- `MULTISET` - wielozbiór, nieuporządkowana i nieograniczona kolekcja, która może zawierać duplikaty elementów.

W językach obiektowych spotyka się również wiele innych kolekcji, nie uwzględnianych obecnie przez standard SQL, jednak wiele z nich można zasymulować za pomocą opisanych typów. Bardzo popularne kolekcje kolejki FIFO (ang. *First-In, First-Out*) oraz stosu LIFO (ang. *Last-In, First-Out*) można zasymulować poprzez ograniczenie działania `ARRAY`, tak by elementy mogły być dodawane i pobierane jedynie na początku lub końcu tablicy. Wielozbiór może służyć jako zwykły zbiór, o ile użytkownik zadba o nieumieszczanie w nim duplikatów. Popularnym typem niewspieranym przez standard SQL i trudnym do zasymulowania jest słownik (ang. *Dictionary*), w którym przechowywane są pary klucz-wartość.

System Oracle wspiera dwa typy kolekcji:

- *Varray* - uporządkowana tablica o określonym rozmiarze. Zalecana, gdy potrzebne jest przechowanie określonej liczby elementów, uporządkowane iterowanie po elementach lub częste manipulowanie całą kolekcją jako pojedynczą wartością;
- *Nested table* - zagnieżdżona tabela, nieuporządkowana kolekcja o nieograniczonym rozmiarze. Zalecana, gdy potrzebne jest wykonywanie wydajnych

zapytań na kolekcji, obsługa nieznanej liczby elementów lub przeprowadzanie masowej manipulacji danych.

Fragment kodu 4.35 przedstawia tworzenie typów obu kolekcji, wykorzystanie ich w tabeli oraz wstawianie do nich danych. Tworzenie typów kolekcji w Oracle jest bardzo podobne do tworzenia typu użytkownika. Tworząc *Varray* podaje się maksymalny rozmiar tablicy, choć możliwa jest późniejsza modyfikacja tego rozmiaru. Oba rodzaje kolekcji można wykorzystać tworząc tabelę. Używając zagnieżdżonej tabeli jak atrybutu, trzeba wskazać nazwę dodatkowej tabeli, która powstanie by przechowywać elementy kolekcji. Dodatkową zaletą kolekcji w Oracle jest możliwość przechowywania typów pochodnych. Możliwe jest także tworzenie kolekcji wielopoziomowych, przez umieszczanie w kolekcjach innych typów kolekcji.

```
1  CREATE TYPE Book_Array AS VARRAY(10) OF Book_Type;
2
3  CREATE TYPE Book_NT AS TABLE OF Book_Type;
4
5  CREATE TABLE Collections (
6    col_1 Book_Array,
7    col_2 Book_NT
8  ) NESTED TABLE col_2 STORE AS books_nt;
9
10 INSERT INTO Collections VALUES (
11   Book_Array(
12     Book_Type('Title A', 'Author A', 20.00),
13     Novel_Type('Title B', 'Author B', 30.00, 'sci-fi')),
14   Book_NT(
15     Book_Type('Title C', 'Author C', 40.00),
16     Book_Type('Title D', 'Author D', 50.00))
17 );
```

Kod 4.35: Typy kolekcji w Oracle Database.

Db2 wspiera dwa rodzaje kolekcji typu Array:

- *Ordinary array* - uporządkowana kolekcja o określonym rozmiarze. Elementy

indeksowane są za pomocą kolejnych liczb całkowitych oznaczających pozycję w tablicy.

- *Associative array* - nieuporządkowana kolekcja o nieograniczonym rozmiarze. Elementy indeksowane są za pomocą wartości dowolnego typu liczbowego lub znakowego.

Przykłady tworzenia obu typów tabel przedstawia fragment kodu 4.36. Tworząc zwykłą tablicę podaje się jej docelowy rozmiar, a tworząc tablicę asocjacyjną podaje się typ będący indeksem w tablicy. Bardzo dużym ograniczeniem kolekcji `array` w Db2 jest możliwość przechowywania w nich tylko predefiniowanych typów prostych. W tym systemie nie ma możliwości tworzenia kolekcji przechowujących obiekty typu użytkownika.

```
1 CREATE TYPE simple_arr AS INTEGER ARRAY[100];
2
3 CREATE TYPE assoc_arr AS INTEGER ARRAY[INTEGER];
```

Kod 4.36: Typy kolekcji w IBM Db2.

PostgreSQL posiada wsparcie dla typu kolekcji `Array`. W implementacji Postgres są to kolekcje zmiennej długości o dowolnej liczbie wymiarów. Tablice `Array` mogą przechowywać dowolne typy obiektów, w tym również typy złożone stworzone przez użytkownika. Przykład tworzenia tabeli z wykorzystaniem tablicy oraz wstawianie danych do kolekcji przedstawia fragment kodu 4.37. Wartość przypisywaną do tablicy `Array` podaje się jako łańcuch znakowy zawierający nawiasy klamrowe, w których po przecinku rozpisane są elementy kolekcji (linie 7-9).

```
1 CREATE TABLE Sales (
2   payments      INTEGER[],
3   schedule      TEXT[][]
4 );
5
6 INSERT INTO Sales VALUES (
7   '{10000, 10000, 10000, 10000}',
8   '{{"June", "first"}, {"June", "fifth"},
```

```
9       {"July", "first"}, {"July", "tenth"}}'
10 );
```

Kod 4.37: Typy kolekcji w PostgreSQL.

4.7 Współpraca z językami obiektowymi

Jednym z głównych powodów wprowadzenia mechanizmów obiektowych do standardu SQL i RSZBD był zamiar zmniejszenia różnicy pomiędzy modelem danych używanym w aplikacjach i zapisywanym w bazie danych. Od czasu pierwszych prac nad rozszerzeniami obiektowymi minęło wiele lat i dziś dostępnych jest wiele narzędzi ułatwiających integrację aplikacji z bazą danych. Przodują wśród nich narzędzia ORM zapewniające mechanizmy mapowania pomiędzy klasami w językach obiektowych, a strukturą relacyjnej bazy danych. Same RSZBD także zapewniają pewne sposoby integracji z zewnętrznymi środowiskami obiektowymi.

Dziesiąta część standardu SQL o tytule "*Object Language Binding (SQL/OLB)*" w całości poświęcona jest integracji z językiem obiektowym Java. Wzorując się na rozwiązaniach opisanych w tej części standardu powstał ogromnie popularny standard ODBC (ang. *Open Database Connectivity*) i stworzony przez Sun Microsystems interfejs JDBC (ang. *Java Database Connectivity*). Wszystkie powszechnie stosowane relacyjne bazy danych, wliczając analizowane bazy Oracle, Db2 i Postgres, wspierają co najmniej kilka protokołów komunikacji zgodnych ze standardem ODBC.

Oracle zapewnia kompleksowe wsparcie dla języka Java. Dostępne są rozszerzenia obiektowe dla JDBC, które pozwalają korzystać ze wszystkich mechanizmów obiektowych bazy Oracle. Możliwe jest implementowanie w Javie logiki i operacji na obiektach, które wykonywane są bezpośrednio w bazie danych. Typy użytkownika tworzone w Oracle można mapować do klas w Javie i zapewniać trwałe przechowywanie obiektów używanych w aplikacji, w sposób podobny do baz obiektowych.

By umożliwić szybki dostęp do obiektów przechowywanych w bazie, Oracle udostępnia obszerną pamięć podręczną. Składowane w niej kopie obiektów są dostępne dla aplikacji w pamięci dynamicznej, a wprowadzone zmiany można utrwalić w ba-

zie odpowiednim poleceniem. Inne rozwiązania udostępniane przez Oracle, które pomagają w pracy z obiektami:

- Oracle Call Interface i Oracle C++ Call Interface - kompleksowy interfejs i środowisko komunikacji z bazą, zapewnia kontrolę transakcji przy pracy na obiektach i pozwala manipulować obiekty i ich atrybuty w pamięci podręcznej.

- Pro*C/C++ Object Extensions - prekompilator i rozszerzenia dla języków C i C++, zapewniające wysoki poziom abstrakcji przy pracy na obiektach w pamięci podręcznej i bazie danych.

- .NET Object Extensions - rozszerzenia obsługujące mapowanie obiektów ze środowiska .NET na typy, kolekcje i referencje tworzone w bazie danych.

Db2 pozwala tworzyć tzw. funkcje transformacji (ang. *Transform functions*), które określają jak SZBD ma przeprowadzać konwersję pomiędzy obiektem przechowywanym w bazie i wykorzystywanym w poleceniach SQL, a obiektem wykorzystywanym w aplikacjach. Odpowiednia implementacja tych funkcji pozwala rozszerzać możliwości bezpośredniej integracji Db2 z językami programowania.

Dla bazy Postgres dostępna jest bardzo obszerna lista bibliotek stworzonych przez społeczność użytkowników, które pozwalają na integrację bazy z wieloma językami. Wśród nich popularne są biblioteki m.in. dla języków C, Java, Go, Javascript, Php, czy Python. Interfejs tworzenia tego typu bibliotek jest aktywnie wspierany i rozwijany przez twórców systemu.

Wszystkie trzy omawiane systemy pozwalają na implementację funkcji, procedur oraz metod w zewnętrznych językach programowania, poza dialektem SQL. Db2 wspiera w tym celu środowiska C, Java, COBOL, CLR (wykorzystywane w .NET), OLE i wszystkie inne rozwiązania o równoznacznych mechanizmach wywoływania metod. Oracle wspiera podobną listę środowisk: C i C++, Java, COBOL, Visual Basic. Postgres wspiera wszystkie wymienione środowiska i dotakowo dzięki dostępności na warunkach open-source pozwala samodzielnie dodawać wsparcie dla nowych języków.

4.8 Podsumowanie

Wszystkie analizowane systemy charakteryzują się znaczącą liczbą cech obiektowych. We wszystkich dostępne są podstawowe fundamenty podejścia obiektowego, takie jak tworzenie typów użytkownika oraz dziedziczenie. Praca z wykorzystaniem podejścia obiektowego jest jak najbardziej możliwa w tych systemach, choć może wymagać częstej konsultacji z dokumentacją.

Występują jednak zauważalne różnice w zakresie spotykanych mechanizmów oraz w szczegółach ich implementacji. Przykładowo, Postgres w ogóle nie wspiera metod, a Oracle nie posiada dziedziczenia tabel, jedynie dziedziczenie typów. Wszystkie tego typu różnice wpływają na sposób pracy z systemem i możliwości projektowania bazy danych. Z powodu różnic, praktycznie niemożliwe jest przenoszenie rozwiązań opartych o typy użytkownika i inne cechy obiektowe pomiędzy systemami, co zwykle nie jest dużym problemem przy podejściu relacyjnym.

We wszystkich trzech systemach występują znaczne różnice w implementacji cech obiektowych w stosunku do rozwiązań opisanych w standardzie SQL. Najbardziej zbliżony do standardu wydaje się być zestaw mechanizmów systemu IBM Db2, choć tam także występują różnice, np. w składni tworzenia metod lub dziedziczenia tabel. Zaskakująca jest implementacja cech obiektowych w systemie PostgreSQL. Jako jedną z wartości przewodnich utrzymuje on zgodność ze standardem SQL, a mimo to wiele spośród dostępnych w nim mechanizmów jest otwarcie sprzecznych ze standardem i ich dostosowanie do niego może wymagać ogromnych zmian i spowodować brak kompatybilności wstecznej.

Opisując cechy obiektowe w tym rozdziale naturalnie nie przedstawiono wszystkich niuansów i szczegółów dotyczących dostępnych rozwiązań. Najbardziej kompletną bazą wiedzy o mechanizmach obiektowych pozostaje dokumentacja systemów. O ile standard SQL jest bardzo techniczny i trudny w zrozumieniu, dokumentacje analizowanych systemów są napisane przystępnym językiem i mogą skutecznie wspomagać użytkowników.

Rozdział 5

Badania

Dotychczas w pracy zostały przybliżone różnorodne cechy obiektowe opisane w standardzie SQL i spotykane w trzech obiektowo-relacyjnych systemach zarządzania bazą danych: Oracle Database, PostgreSQL i IBM Db2. Badania przedstawione w tym rozdziale mają na celu przetestowanie omówionych mechanizmów w praktyce oraz ocenę ich przydatności podczas projektowania i późniejszego wykorzystywania bazy danych.

W ramach badań rozpatrywana jest dziedzina fikcyjnej hurtowni sprzętów elektronicznych. W pierwszych sekcjach rozdziału przedstawiony jest opis dziedziny oparty o diagram klas oraz struktura bazy danych zaprojektowanej w podejściu relacyjnym. Następnie przedstawione są struktury obiektowo-relacyjne stworzone w każdym z analizowanych systemów, wykorzystujące jak najwięcej spośród dostępnych mechanizmów obiektowych. Omówione są różnice w strukturach i problemy napotkane podczas projektowania, wynikające ze zróżnicowanych możliwości systemów.

Dla wszystkich struktur bazy danych przygotowano szereg zapytań realizujących przykładowe potrzeby biznesowe. Omówione są różnice pomiędzy zapytaniami dla poszczególnych systemów i porównane są czasy realizacji zapytań. Opisany jest również sposób generacji danych wykorzystanych do wypełnienia baz na potrzeby testów. Pod koniec rozdziału opisane są wnioski z przeprowadzonych badań.

5.1 Środowisko badań

Do stworzenia środowiska badań wykorzystano narzędzie Docker, które umożliwia wdrażanie konfigurowalnych i przenośnych kontenerów, na bazie wcześniej przygotowanych obrazów. Każdy uruchomiony kontener zawiera instancję systemu operacyjnego, zwykle niewielkiego systemu z rodziny Unix, na którym zainstalowane i uruchomione są wymagane aplikacje i narzędzia.

Wszystkie trzy omawiane SZBD wspierają uruchamianie w środowisku Docker. PostgreSQL i IBM Db2 udostępniają gotowe obrazy kompletnego serwera bazy danych (dla Db2 tylko darmowa edycja Community) w repozytorium Docker Hub [4, 13]. Oracle zamiast gotowego obrazu udostępnia repozytorium GitHub zawierające skrypty i materiały pomocnicze do samodzielnego zbudowania obrazu dowolnej wersji i edycji systemu [10].

Na potrzeby badań wykorzystano darmowe wersje każdej bazy. Postgres domyślnie jest całkowicie darmowy i dostępny na licencji *open-source*. Db2 udostępnia obraz bazy w edycji Community, która ogranicza zasoby sprzętowe wykorzystywane przez bazę do 16 GB pamięci stałej i 4 rdzeni procesora, ale daje dostęp do wszystkich funkcji bazy. Dla bazy Oracle zbudowano obraz oparty o edycję Express Edition, która podobnie jak w Db2 ogranicza zasoby sprzętowe do 12 GB pamięci stałej, 2 GB pamięci RAM i 2 wątków procesora.

Dla każdej bazy przygotowano plik konfiguracyjny narzędzia Docker Compose, które umożliwia wygodne definiowanie i uruchamianie wielu kontenerów. Przykład zawartości pliku Docker Compose dla bazy Oracle przedstawia fragment kodu 5.1. Oprócz wskazania obrazu, z którego ma powstać kontener, definiowane są zmienne środowiskowe, mapowanie portów z wnętrza kontenera na system zewnętrzny oraz konfiguracja woluminów. Skorzystanie z woluminów pozwala przechować fragment systemu plików z wnętrza kontenera w systemie zewnętrznym. Zapewnia to trwałe przechowywanie danych pomiedzy uruchomieniami kontenera oraz wygodny wgląd w strukturę plików z danymi zapisaną na dysku.

```
1 services:
2   oracle:
3     image: oracle/database:18.4.0-xe
```

```
4     environment:
5       ORACLE_PWD: mysecurepassword
6       ORACLE_CHARACTERSET: AL32UTF8
7       POSTGRES_PASSWORD: postgres
8       POSTGRES_USERNAME: postgres
9       POSTGRES_DB: postgres
10    volumes:
11      - ./oracle_volume:/opt/oracle/oradata
12    ports:
13      - 51521:1521
14      - 55500:5500
```

Kod 5.1: Zawartość pliku Docker Compose dla Oracle Database.

By przeprowadzić porównywalne testy działania wielu systemów bazodanowych, konieczne jest zapewnienie możliwie jak najbardziej zbliżonych do siebie warunków dla każdego systemu. Należy zadbać o jednolite zasoby sprzętowe, takie jak moc procesora czy ilość i szybkość pamięci dynamicznej i statycznej. Wykorzystano w tym celu dostepną w narzędziu Docker możliwość ograniczania zasobów dostępnych dla kontenerów. W plikach Docker Compose dla każdej bazy dopisano fragment konfiguracji 5.2, który pozwala kontenerom baz danych wykorzystać maksymalnie 2 GB pamięci RAM i jeden rdzeń procesora.

```
1     deploy:
2       resources:
3         limits:
4           cpus: 1
5           memory: 2048M
```

Kod 5.2: Konfiguracja Docker Compose dla ograniczenia zasobów sprzętowych.

Po przygotowaniu plików konfiguracyjnych, uruchomienie każdego kontenera wymaga wykonania polecenia `docker-compose -f plik-compose.yml up`. Na potrzeby wszystkich testów kontenery uruchamiane były pojedynczo, tak by zminimalizować obciążenie sprzętowe.

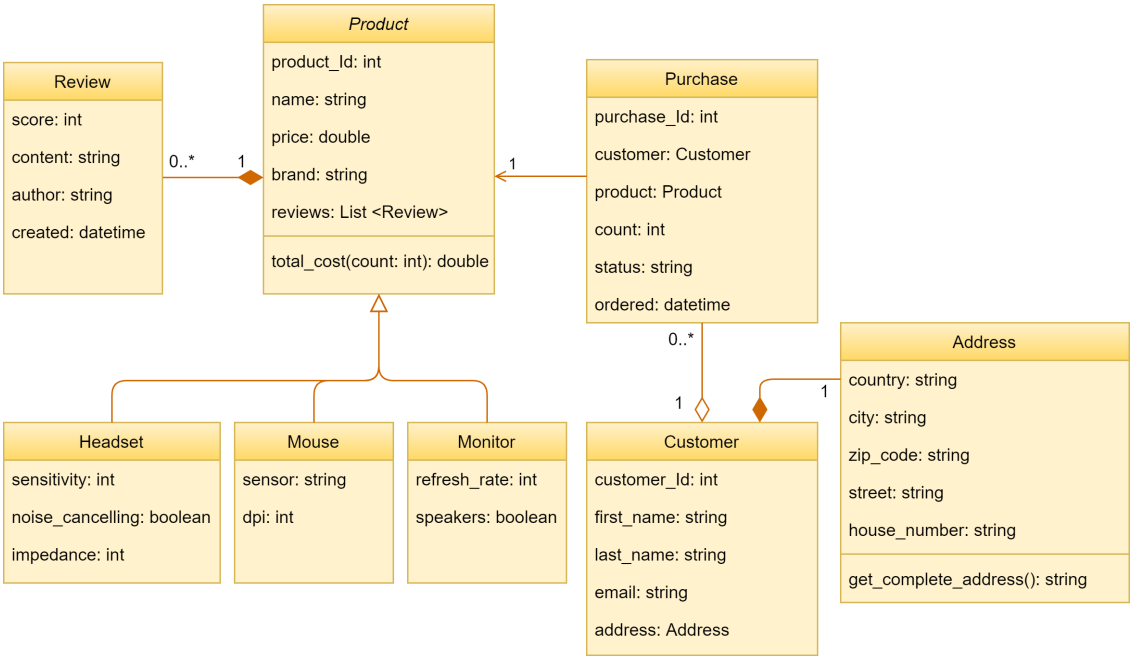
Większość pracy z uruchomionymi systemami wykonywano za pomocą zintegrowanych interfejsów wspomagających komunikację z bazą. Producent każdego systemu wspiera konkretne narzędzie do pracy ze swoim rozwiązaniem - dla bazy Postgres jest to narzędzie pgAdmin 4, dla Oracle środowisko Oracle SQL Developer, a dla Db2 program IBM Data Studio. Każde z narzędzi zapewnia interfejs do administracji bazą, wykonywania zapytań i przeglądania wszystkich obiektów stworzonych w bazie - tabel, typów, funkcji itp. Narzędzie pgAdmin 4 uruchamiane jest jako osobny kontener w narzędziu Docker, korzystając z obrazu dostępnego na Docker Hub [11], a pozostałe dwa narzędzia zostały zainstalowane lokalnie.

5.2 Modelowana dziedzina

Podstawą dla przeprowadzanych badań jest model dziedziny fikcyjnej hurtowni sprzętów elektronicznych. Podczas testów relacyjnych baz danych, często korzysta się z dostępnych publicznie struktur i zbiorów danych, takich jak baza filmów IMDb [6], czy zbiory udostępniane przez Amazon [1]. Niestety, przy gotowym rozwiązaniu relacyjnym, bardzo trudno jest przeprowadzić konwersję struktury oraz danych do modelu obiektowo-relacyjnego, który wykorzystywałby możliwości obiektowe ORSZBD [25]. Z tego względu, postanowiono przygotować autorski model i zbiór danych, tak by móc dostosować go do potrzeb badań.

Schemat klas dla rozpatrywanej domeny biznesowej pokazany jest na rysunku 5.1. Przedstawia on model hurtowni sprzętów elektronicznych, w której sprzedawane są monitory, słuchawki i myszy komputerowe. Klienci mogą dokonywać zakupów wybranych produktów w dowolnej ilości, ale zakłada się, że pojedynczy zakup dotyczy tylko jednego rodzaju produktu. Dla klientów przechowywane są ich dane osobowe oraz adres. Wszystkie produkty posiadają producenta, nazwę, cenę i listę recenzji, które tworzyć mogą użytkownicy. Konkretne rodzaje produktów posiadają dodatkowe pola unikatowe dla typu produktu. Recenzje składają się z oceny w skali od 1 do 5, treści recenzji, daty i nazwy autora. Autorzy recenzji nie mają żadnego powiązania z klientami zapisywanymi w systemie.

Głównym celem przyświecającym projektowaniu modelu było wykorzystanie różnych mechanizmów obiektowych, które będzie można odwzorować w ORSZBD. W zrealizowanym modelu występują następujące zjawiska obiektowe:



Rysunek 5.1: Diagram klas modelowanej aplikacji.

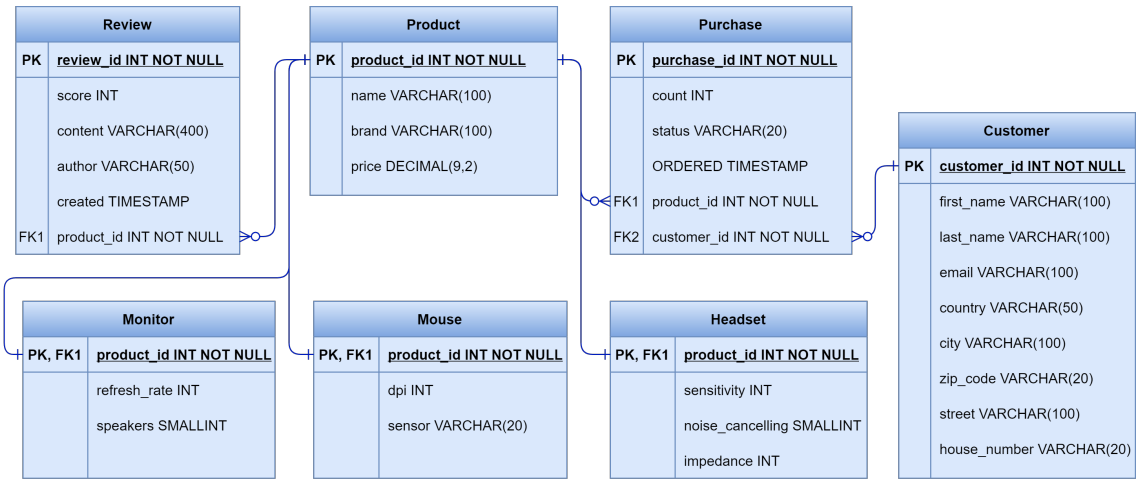
- Dziedziczenie - klasa *Product* posiada trzy klasy potomne *Monitor*, *Headset* i *Mouse*, z których każda posiada swoje własne unikalne pola.
- Zagnieżdżenie - klasa *Address* jest zagnieżdżona w klasie *Customer* i adres nigdy nie występuje jako niezależny obiekt.
- Referencje - klasa *Purchase* zawiera referencje (wskaźniki) do obiektów klas *Customer* i *Product*.
- Typ kolekcji - klasa *Product* zawiera w sobie listę obiektów klasy *Review*.
- Metody - klasa *Address* zawiera metodę przygotowującą pełny adres w postaci jednego łańcucha tekstowego, a klasa *Product* zawiera metodę obliczania kosztu wskazanej liczby produktów.

Można zauważyć, że metody klas zaproponowane w modelu są bardzo proste. Podczas projektowania modelu sugerowano się ograniczeniami analizowanych systemów, tak by nie wprowadzać metod, które są niemożliwe do zrealizowania. Żaden z systemów nie pozwala na trwałą modyfikację stanu obiektu podczas wywołania

metody, ani na odwołanie do obiektów powiązanych przez referencję. Te ograniczenia uniemożliwiają tworzenie zaawansowanych metod, które bez problemu możnaby zrealizować w obiektowym języku programowania. Metody w ORSZBD mogą służyć w praktyce tylko do realizacji obliczeń lub wypisania informacji dla użytkownika, na podstawie pól obiektu i przekazanych parametrów.

5.3 Podejście relacyjne

Zasady projektowania struktury relacyjnej bazy danych zostały dobrze dopracowane przez wiele lat popularności tego rodzaju systemów. Schemat bazy został ręcznie zaprojektowany na podstawie rozpatrywanego diagramu klas z rysunku 5.1. Uzyskana struktura relacyjna przedstawiona jest na rysunku 5.2. Powstałe tabele w dużym stopniu odpowiadają modelowanym klasom, choć występuje kilka znaczących różnic.



Rysunek 5.2: Diagram relacyjnej bazy danych.

Związki pomiędzy danymi w różnych tabelach w świecie relacyjnym realizowane są za pomocą kluczy obcych. Warto zwrócić uwagę, że klucze obce modelujące powiązania przechowywane są zawsze w tabeli odpowiadającej większej krotności relacji (związek wiele-do-wielu wymaga utworzenia dodatkowej tabeli powiązań). Na przykładzie rozpatrywanej dziedziny, w tabeli recenzji w każdym wierszu przechowywana jest informacja, do którego produktu przynależy dana recenzja. Struktury

obiektowe pozwalają na odwrócenie tej zależności i przechowanie listy obiektów wewnątrz innego obiektu. W takiej sytuacji, instancja produktu przechowywałaby informację o przynależących do niego recenzjach.

W przedstawionym rozwiązaniu nie ma odwzorowania osobnej tabeli dla adresu - pola klasy *Address* zostały uwzględnione bezpośrednio w tabeli klientów. Alternatywą jest przechowanie adresów w osobnej tabeli i zapewnienie relacji z klientem za pomocą klucza obcego - to rozwiązanie byłoby bardziej odpowiednie, gdyby adresy powtarzające się pomiędzy klientami traktowane były jako te same instancje danych.

Model relacyjny nie posiada mechanizmów, które pozwoliłyby na dokładne odwzorowanie relacji dziedziczenia występującej w środowisku obiektowym. By umożliwić przechowywanie obiektów tworzących hierarchię dziedziczenia, powszechnie stosuje się trzy wzorce projektowe [24]:

- *Single Table Inheritance* - spłaszczenie całej hierarchii dziedziczenia do jednej tabeli o wielu polach, które są wybiórczo wypełnione w różnych wierszach, zależnie od typu obiektu przechowywanego w danym wierszu.

- *Concrete Table Inheritance* - stworzenie po jednej tabeli dla każdej nieabstrakcyjnej klasy i uwzględnienie w nich pól abstrakcyjnych klas rodziców. To rozwiązanie uniemożliwia tworzenie relacji do abstrakcyjnego typu - klucze obce innych tabel mogą wskazywać jedynie na konkretne podtypy.

- *Class Table Inheritance* - stworzenie tabeli dla każdej klasy w hierarchii dziedziczenia i powiązanie ich wszystkich za pomocą jednego klucza.

W przedstawionym przykładzie zastosowano trzecią z opcji. Pole *product_id* występuje we wszystkich tabelach hierarchii produktów i pełni w nich rolę klucza głównego, a dodatkowo, w tabelach Monitor, Mouse i Headset pełni rolę klucza obcego, wskazującego na tabelę Product.

W modelu relacyjnym naturalnie nie uwzględniono metod, które występują wyłącznie w podejściu obiektowym. Zamiast nich użytkownik powinien stworzyć odpowiednie procedury albo powtórzyć logikę metod bezpośrednio w poleceniach SQL.

Jako że wiele SZBD nie wspiera typu *Boolean*, we wszystkich omawianych rozwiązaniach wszystkie wystąpienia typu *Boolean* reprezentowane są przez typ *Smallint*, gdzie wartości 1 i 0 odpowiadają wartościom *true* i *false*.

Bazując na diagramie bazy 5.2 przygotowano skrypt SQL tworzący odpowiednie tabele. Skrypt jest uniwersalny i bez żadnych modyfikacji może być wykonany we wszystkich analizowanych systemach. Skrypt można znaleźć pod nazwą *Relational-Structure.sql* w katalogu "Scripts\Structure" na płycie dołączonej do pracy.

5.4 Podejście obiektowo-relacyjne

Tworząc strukturę bazy w systemach obiektowo-relacyjnych, starano się wiernie odwzorować strukturę klas z diagramu 5.1, uwzględniając wszystkie relacje dziedziczenia, referencji i agregacji. Mimo to, mechanizmy udostępniane przez poszczególne systemy są na tyle zróżnicowane, że występują znaczące różnice w powstałych strukturach, które zostaną omówione na podstawie komend tworzących poszczególne typy i tabele.

Kolejne podsekcje omawiają zaprojektowane struktury w oparciu o polecenia wykonywane w poszczególnych systemach. Skrypty zawierające komplet poleceń do utworzenia struktury obiektowo-relacyjnej w każdym systemie można znaleźć w katalogu "Scripts\Structure" na płycie dołączonej do pracy.

5.4.1 Oracle Database

Odwzorowanie typów adresu i klienta jest dosyć proste i zrealizowane bardzo podobnie we wszystkich systemach. Fragment kodu 5.3 przedstawia komendy tworzące odpowiednie typy w bazie Oracle. Bez problemu można zawrzeć atrybut typu adresu wewnątrz typu klienta.

```
1 CREATE TYPE Address_Type AS OBJECT (
2   country        VARCHAR(50),
3   city           VARCHAR(100),
4   zip_code       VARCHAR(20),
```

```
5    street          VARCHAR(100),
6    house_number    VARCHAR(20),
7    MEMBER FUNCTION get_complete_address RETURN VARCHAR
8  );
9  CREATE TYPE Customer_Type AS OBJECT (
10   customer_id     INTEGER,
11   first_name      VARCHAR(100),
12   last_name       VARCHAR(100),
13   email           VARCHAR(100),
14   address         Address_Type
15 );
```

Kod 5.3: Definicja typów adresu i klienta w Oracle.

Korzystając ze stworzonego typu klienta można utworzyć tabelę klientów, korzystając z komendy 5.4. Przy tworzeniu tabeli, atrybut *customer_id* wskazany jest jako klucz główny tabeli i jednocześnie wykorzystany jest jako identyfikator obiektu. Dzięki temu system nie będzie generował dodatkowych identyfikatorów, operując wartościami w kolumnie *customer_id* na potrzeby referencji i identyfikacji obiektów.

```
1  CREATE TABLE Customer OF Customer_Type (
2          PRIMARY KEY (customer_id)
3  ) OBJECT IDENTIFIER IS PRIMARY KEY;
```

Kod 5.4: Tworzenie tabeli klientów w Oracle.

W celu stworzenia typu produktu, konieczne jest wcześniejsze utworzenie typu recenzji, które przechowywane są jako lista wewnątrz każdego produktu. Oracle udostępnia świetnie przystosowany do tego mechanizm tabel zagnieżdżonych (ang. *Nested Table*). Na fragmencie kodu 5.5 przedstawiono tworzenie typu recenzji oraz typu tabeli zagnieżdżonej, przechowującej recenzje. Alternatywą dla tego rozwiązania jest wykorzystanie kolekcji `VARRAY`, która byłaby wskazana, gdyby maksymalna liczba recenzji była znana z góry. W przypadku nieograniczonej liczby elementów kolekcji bardziej uniwersalna jest tabela zagnieżdżona.

```
1 CREATE TYPE Review_Type AS OBJECT (
2   score          INTEGER,
3   content        VARCHAR(400),
4   author         VARCHAR(50),
5   created        TIMESTAMP
6 );
7 CREATE TYPE Reviews_Type AS TABLE OF Review_Type;
```

Kod 5.5: Definicja typu recenzji i tabeli zagnieżdżonej recenzji w Oracle.

Fragment kodu 5.6 przedstawia tworzenie typu produktu i tabeli produktów. Jako jeden z atrybutów wykorzystano typ tabeli zagnieżdżonej recenzji. W komendzie `CREATE TABLE` wskazano nazwę tabeli, w której przechowywane mają być elementy zagnieżdżonej kolekcji. Podobnie jak dla tabeli klientów, tworząc tabelę produktów wskazano atrybut klucza głównego i nadano mu rolę identyfikatora obiektów. Definiując typ produktu wykorzystano frazę `NOT FINAL`, by umożliwić tworzenie typów pochodnych od produktu.

```
1  CREATE OR REPLACE TYPE Product_Type AS OBJECT (
2    product_id     INTEGER,
3    brand          VARCHAR(100),
4    name           VARCHAR(100),
5    price          DECIMAL(9,2),
6    reviews        Reviews_Type,
7    MEMBER FUNCTION total_cost (how_many IN INTEGER)
       RETURN DECIMAL
8  ) NOT FINAL;
9
10 CREATE TABLE Product OF Product_Type (
11         PRIMARY KEY (product_id)
12 ) OBJECT IDENTIFIER IS PRIMARY KEY
13 NESTED TABLE reviews STORE AS Reviews;
```

Kod 5.6: Definicja typu produktu i tworzenie tabeli produktów w Oracle.

Bazując na typie produktu zdefiniowano dla niego trzy typy pochodne, jak przedstawiono na fragmencie kodu 5.7. W rozwiązaniu Oracle, w kwestii dziedziczenia określa się tylko hierarchię typów i nie występuje dziedziczenie tabel, w takim znaczeniu jak w innych omawianych systemach. Stworzenie hierarchii typów automatycznie umożliwia wstawianie dowolnego podtypu do tabeli zbudowanej w oparciu o typ rodzica. W związku z tym nie ma potrzeby tworzenia osobnych tabel dla każdego podtypu.

```
1  CREATE TYPE Monitor_Type UNDER Product_Type (
2    refresh_rate  INTEGER,
3    speakers      SMALLINT
4  );
5  CREATE TYPE Headset_Type UNDER Product_Type (
6    sensitivity       INTEGER,
7    impedance         INTEGER,
8    noise_cancelling  SMALLINT
9  );
10 CREATE TYPE Mouse_Type UNDER Product_Type (
11   dpi           INTEGER,
12   sensor        VARCHAR(20)
13 );
```

Kod 5.7: Definicja podtypów produktu w Oracle.

Tworzenie typu reprezentującego zakup wymaga wykorzystania referencji, by wskazać powiązany produkt i klienta dokonującego zakupu. Odpowiednia komenda tworząca typ i tabelę zakupów przedstawiona jest na fragmencie kodu 5.8. Podczas tworzenia tabeli określono zakres (ang. *scope*) referencji, wskazując tabelę, w której przechowywane będą powiązane obiekty. Wskazanie zakresu jest opcjonalne, ale zwiększa wydajność dereferencji w zapytaniach. W przypadku tej tabeli również wskazano klucz główny i wykorzystano go jako identyfikator obiektów.

```
1  CREATE TYPE Purchase_Type AS OBJECT (
2    purchase_id   INTEGER,
```

```
3    customer        REF Customer_Type,
4    product         REF Product_Type,
5    count           INTEGER,
6    status          VARCHAR(20),
7    ordered         TIMESTAMP
8  );
9  CREATE TABLE Purchase OF Purchase_Type (
10   product    SCOPE IS Product,
11   customer   SCOPE IS Customer,
12   PRIMARY    KEY (purchase_id)
13 ) OBJECT IDENTIFIER IS PRIMARY KEY;
```

Kod 5.8: Definicja typu zakupu i tworzenie tabeli zakupów w Oracle.

Na fragmentach kodu 5.3 i 5.6 w definicjach typów zawarte są specyfikacje metod. Dla każdego z tych typów konieczne jest określenie ciała typu, zawierającego treści metod, co przedstawia fragment kodu 5.9. Warto zwrócić uwagę, że określając typ zwracany przez metodę nie trzeba precyzować maksymalnego rozmiaru łańcucha znakowego `VARCHAR` (linia 3) ani precyzji i skali typu `DECIMAL` (linia 11).

```
1  CREATE OR REPLACE TYPE BODY Address_Type AS
2    MEMBER FUNCTION get_complete_address
3    RETURN VARCHAR IS
4    BEGIN
5      RETURN street || ' ' || house_number || '\n' ||
           zip_code || ' ' || city || ', ' || country;
6    END;
7  END;
8
9  CREATE OR REPLACE TYPE BODY Product_Type AS
10   MEMBER FUNCTION total_cost (how_many IN INTEGER)
11   RETURN DECIMAL IS
12   BEGIN
13     RETURN how_many * price;
```

```
14    END;
15  END;
```

Kod 5.9: Definicja ciał typów w Oracle.

Przykłady wstawiania danych do utworzonej struktury przedstawia fragment kodu 5.10. Wstawianie obiektów różnych podtypów produktu wykonywane jest bezpośrednio na tabeli produktów. Oracle pilnuje unikalności klucza głównego również przy wstawianiu podtypów. Dodawanie wpisów do tabeli zakupów wymaga wykonania zagnieżdżonego zapytania `SELECT` (linie 13-15), w celu uzyskania referencji do obiektów klienta i produktu.

```
1  INSERT INTO Customer VALUES (1, 'Abel', 'Doe',
2    'abeldoes@mail.com', Address_Type('PL', 'Warsaw',
3    '44-300', 'Lubuska', '5/2'));
4
5  INSERT INTO Product VALUES (Monitor_Type(1, 'Aker',
6    'A1', 1200, Reviews_Type(), 60, 0));
7  INSERT INTO Product VALUES (Headset_Type(2, 'Fraction',
8    'F1', 1200, Reviews_Type(), 80, 32, 0));
9  INSERT INTO Product VALUES (Mouse_Type(3, 'Roar',
10   'R1', 500, Reviews_Type(), 1600, 'optical'));
11
12 INSERT INTO Purchase (
13  SELECT Purchase_Type(1, REF(c), REF(p), 1, 'complete',
       '2021-08-27 15:23:54')
14  FROM Customer c, Product p
15  WHERE c.customer_id = 1 AND p.product_id = 5);
```

Kod 5.10: Wstawianie danych do bazy Oracle.

Dodawanie recenzji dla istniejącego produktu przedstawia fragment kodu 5.11. Wykorzystano w tym celu operator `MULTISET UNION ALL`, który łączy zawartość dwóch zbiorów - istniejącej listy recenzji i listy zawierającej jedną nową recenzję. Jest to najprostszy sposób na rozszerzenie zawartości zagnieżdżonej tabeli.

```
1 UPDATE Product SET reviews = reviews MULTISET UNION ALL
2   Reviews_Type(Review_Type(5, 'Very good product!', '
      anon01', '2021-08-27 18:20:15'))
3 WHERE product_id = 5;
```

Kod 5.11: Dodawanie nowej recenzji produktu w Oracle.

5.4.2 IBM Db2

Tworzenie typów adresu i klienta w Db2 przedstawia fragment kodu 5.12. Podobnie jak w Oracle, w Db2 nie ma problemu z wykorzystaniem typu adresu jako jednego z atrybutów typu klienta. Wszystkie definicje typów w Db2 zakończone są wymaganą frazą `MODE DB2SQL`.

```
1  CREATE TYPE Address_Type AS (
2    country        VARCHAR(50),
3    city           VARCHAR(100),
4    zip_code       VARCHAR(20),
5    street         VARCHAR(100),
6    house_number   VARCHAR(20)
7  ) MODE DB2SQL
8  METHOD get_complete_address () RETURNS VARCHAR(300)
9  LANGUAGE SQL DETERMINISTIC;
10
11 CREATE TYPE Customer_Type AS (
12   first_name     VARCHAR(100),
13   last_name      VARCHAR(100),
14   email          VARCHAR(100),
15   address        Address_Type
16 ) REF USING INTEGER MODE DB2SQL;
```

Kod 5.12: Definicja typów adresu i klienta w Db2.

Na szczególną uwagę zasługuje brak atrybutu `customer_id` w definicji typu

klienta. W bazie Db2 tworzenie typowanej tabeli wymaga utworzenia kolumny identyfikatora obiektów, wykorzystywanej na potrzeby referencji. Kolumną identyfikatora nie może być żaden z atrybutów wchodzących w skład definicji typu. W rozpatrywanym modelu, typ klienta posiada identyfikator w postaci atrybutu `customer_id`. By uniknąć przechowywania dwóch unikalnych identyfikatorów, usunięto wspomniany atrybut z definicji typu, ale dodano go jako kolumnę identyfikatora przy tworzeniu tabeli klientów, co widać na fragmencie kodu 5.13. Fraza `REF USING` na fragmencie 5.12 pozwala określić typ identyfikatora - wybrano typ `INTEGER` by zachować spójność z pierwotnym modelem. To samo rozwiązanie wykorzystano przy tworzeniu pozostałych typów i tabel typowanych.

```
1 CREATE TABLE Customer OF Customer_Type
2 (REF IS customer_id USER GENERATED);
```

Kod 5.13: Tworzenie tabeli klientów w Db2.

System Db2 rozróżnia mechanizm dziedziczenia typów od dziedziczenia tabel. By odwzorować hierarchię produktów z rozpatrywanego modelu, należy najpierw utworzyć hierarchię typów, a następnie w analogiczny sposób stworzyć hierarchię tabel. Fragment kodu 5.14 przedstawia tworzenie typu produktu i jego trzech podtypów, a fragment 5.15 tworzenie hierarchii tabel.

```
1  CREATE TYPE Product_Type AS (
2    brand      VARCHAR(100),
3    name       VARCHAR(100),
4    price      DECIMAL(9,2)
5  ) NOT FINAL REF USING INTEGER MODE DB2SQL
6  METHOD total_cost(how_many INTEGER)
7  RETURNS Decimal(12,2) LANGUAGE SQL DETERMINISTIC;
8
9  CREATE TYPE Monitor_Type UNDER Product_Type AS (
10   refresh_rate  INTEGER,
11   speakers      SMALLINT
12 ) MODE DB2SQL;
```

```
13
14 CREATE TYPE Headset_Type UNDER Product_Type AS (
15   sensitivity        INTEGER,
16   impedance          INTEGER,
17   noise_cancelling   SMALLINT
18 ) MODE DB2SQL;
19
20 CREATE TYPE Mouse_Type UNDER Product_Type AS (
21   dpi             INTEGER,
22   sensor          VARCHAR(20)
23 ) MODE DB2SQL;
```

Kod 5.14: Tworzenie typu produktu i jego podtypów w Db2.

```
1 CREATE TABLE Product OF Product_Type
2   (REF IS product_id USER GENERATED);
3 CREATE TABLE Monitor OF Monitor_Type UNDER Product
4   INHERIT SELECT PRIVILEGES;
5 CREATE TABLE Headset OF Headset_Type UNDER Product
6   INHERIT SELECT PRIVILEGES;
7 CREATE TABLE Mouse OF Mouse_Type UNDER Product
8   INHERIT SELECT PRIVILEGES;
```

Kod 5.15: Tworzenie hierarchii tabel produktów w Db2.

Niestety w systemie Db2 nie ma możliwości wiernego zamodelowania relacji produktu i jego recenzji, tak by każdy produkt zawierał w ramach swojej struktury informację o liście recenzji. Choć dostępny jest typ kolekcji `ARRAY`, ograniczony jest do przechowywania jedynie typów podstawowych - nie można wykorzystać go do przechowania obiektów typu użytkownika, ani typów referencji.

W wyniku tego, jedyną pozostałą opcją jest odwrócenie relacji, tak by każda recenzja posiadała informację, do którego produktu jest przypisana. Rozwiązanie to jest bardzo zbliżone do rozwiązania relacyjnego - jedyną różnicą jest wykorzy-

stanie referencji zamiast klucza obcego do wskazania produktu. Fragment kodu 5.16 przedstawia tworzenie typu i tabeli recenzji.

```
1  CREATE TYPE Review_Type AS (
2          score           INTEGER,
3          content         VARCHAR(400),
4          author          VARCHAR(50),
5          created         TIMESTAMP,
6          product         REF(Product_Type)
7  ) REF USING INTEGER MODE DB2SQL;
8
9  CREATE TABLE Review OF Review_Type (
10         REF IS review_id USER GENERATED,
11         product WITH OPTIONS SCOPE Product
12 );
```

Kod 5.16: Tworzenie typu i tabeli recenzji w Db2.

Ostatnim tworzonym typem jest typ zakupu. Definicję typu i tworzenie typowanej tabeli przedstawia fragment kodu 5.17. Do wskazania klienta i produktu wykorzystano typ referencyjny (linie 2-3). Podczas tworzenia tabeli określono zakres referencji wskazując tabelę przechowującą referowane obiekty (linie 11-12).

```
1  CREATE TYPE Purchase_Type AS (
2    customer  REF(Customer_Type),
3    product   REF(Product_Type),
4    count     INTEGER,
5    status    VARCHAR(20),
6    ordered   TIMESTAMP
7  ) REF USING INTEGER MODE DB2SQL;
8
9  CREATE TABLE Purchase OF Purchase_Type (
10   REF IS purchase_id USER GENERATED,
11   customer WITH OPTIONS SCOPE Customer,
```

```
12   product  WITH OPTIONS SCOPE Product
13 );
```

Kod 5.17: Tworzenie typu i tabeli zakupów w Db2.

Definicje typów na fragmentach kodu 5.12 i 5.14 zawierają specyfikacje metod. Ciała metod definiuje się za pomocą komendy `CREATE METHOD`, co przedstawia fragment 5.18. Dostęp do pól instancji obiektu wewnątrz metod wymaga wykorzystania zmiennej `SELF` i operatora dwóch kropek "..". W odróżnieniu od rozwiązania Oracle, w przypadku Db2 w nagłówku metody konieczne jest dokładne określenie typu zwracanego, poprzez podanie maksymalnego rozmiaru łańcucha znakowego `VARCHAR` oraz precyzji i skali typu `DECIMAL`.

```
1 CREATE METHOD get_complete_address () RETURNS VARCHAR
     (300) FOR Address_Type
2 RETURN SELF..street || ' ' || SELF..house_number || '\n'
      || SELF..zip_code || ' ' || SELF..city || ', ' ||
     SELF..country;
3
4 CREATE METHOD total_cost(how_many INTEGER) RETURNS
     DECIMAL(12,2) FOR Product_Type
5 RETURN how_many * SELF..price;
```

Kod 5.18: Tworzenie ciał metod w Db2.

Wstawianie danych klientów i produktów do utworzonej struktury bazy przedstawia fragment kodu 5.19. Jako pierwszy element we wszystkich poleceniach wstawiany jest typ referencyjny tworzony poprzez wywołanie odpowiedniej funkcji o nazwie odpowiadającej typowi użytkownika.

W pierwszej komendzie widać charakterystyczne wykorzystanie bezargumentowego konstruktora typu adresu i łańcuchowe wywołanie automatycznie generowanych metod mutatorów w celu ustawienia wartości pól nowo utworzonego obiektu. Ponieważ adres przechowywany jest jako obiekt zagnieżdżony w typie klienta, dla tego obiektu nie ma potrzeby tworzenia typu referencyjnego.

Inaczej niż w bazie Oracle, obiekty podtypów produktu wstawiane są do od-

powiadających im tabel uczestniczących w hierarchii dziedziczenia. Db2 pilnuje unikalności identyfikatora obiektów w ramach całej hierarchii tabel.

```
INSERT INTO Customer VALUES (Customer_Type(1), 'Abel',
  'Doe', 'abeldoe@mail.com', Address_Type()
  ..country('PL')..city('Warsaw')..zip_code('44-300')
  ..street('Lubuska')..house_number('5/2'));

INSERT INTO Monitor VALUES (Monitor_Type(1), 'Aker',
  'A1', 1200, 60, 0);
INSERT INTO Headset VALUES (Headset_Type(2), 'Fraction',
  'F1', 1200, 80, 32, 0);
INSERT INTO Mouse VALUES (Mouse_Type(3), 'Roar',
  'R1', 500, 1600, 'optical');
```

Kod 5.19: Wstawianie danych klientów i produktów do bazy Db2.

Fragment kodu 5.20 pokazuje dwa sposoby wstawiania danych zakupów. Pierwsze polecenie jest bardzo podobne do swojego odpowiednika z bazy Oracle i zawiera zagnieżdżone zapytanie `SELECT` pozwalające wyszukać identyfikatory (referencje) klienta i produktu. Druga wersja jest krótsza i może być wykorzystana gdy znane są wartości identyfikatorów. W przeciwieństwie do Oracle, Db2 pozwala na tworzenie pełnoprawnych referencji bez odnośników do obiektu, bazując na samej wartości identyfikatora.

```
INSERT INTO Purchase
  SELECT Purchase_Type(1), c.customer_id, p.product_id,
    1, 'complete', '2021-08-27 15:23:54'
  FROM Customer c, Product p
  WHERE c.customer_id = Customer_Type(1) AND
    p.product_id = Product_Type(5);

INSERT INTO Purchase VALUES (Purchase_Type(1),
    Customer_Type(1), Product_Type(5),
```

```
10      1, 'complete', '2021-08-27 15:23:54');
```

Kod 5.20: Wstawianie danych zakupów do bazy Db2.

Wstawianie recenzji dla istniejącego produktu przedstawiono na fragmencie kodu 5.21. Ta operacja jest prostsza niż w bazie Oracle, ponieważ recenzje nie są zagnieżdżone w produkcie, tylko przechowywane w osobnej tabeli.

```
1 INSERT INTO Review VALUES (
2   Review_Type(1), 5, 'Very good product!', 'anon01',
3   '2021-08-27 18:20:15', Product_Type(5)
4 );
```

Kod 5.21: Wstawianie recenzji do bazy Db2.

5.4.3 PostgreSQL

Baza Postgres ma dosyć proste i ograniczone mechanizmy obiektowe, więc wiele spośród komend do generacji struktury bazy jest prostszych niż w pozostałych systemach. Komendy tworzące typy adresu są bardzo podobne do odpowiedników przedstawionych wcześniej. Przedstawia je fragment kodu 5.22. W tym systemie również bardzo łatwo można zrealizować zagnieżdżenie adresu.

```
1  CREATE TYPE Address_Type AS (
2    country        VARCHAR(50),
3    city           VARCHAR(100),
4    zip_code       VARCHAR(20),
5    street         VARCHAR(100),
6    house_number   VARCHAR(20)
7  );
8  CREATE TYPE Customer_Type AS (
9    customer_id    INTEGER,
10   first_name     VARCHAR(100),
11   last_name      VARCHAR(100),
12   email          VARCHAR(100),
```

```
13   address        Address_Type
14 );
```

Kod 5.22: Tworzenie typów adresu i klienta w Postgres.

Tworzenie tabeli klientów przedstawia fragment kodu 5.23. Definiując tabelę wskazany został klucz główny. Tabele tworzone na podstawie typów nie posiadają specjalnego znaczenia w systemie Postgres, jak tabele typowane czy obiektowe w Db2 i Oracle. Równoważny efekt do przedstawionego polecenia możnaby uzyskać wywołując komendę `CREATE TABLE` bez podania typu użytkownika, a zamiast tego podając nazwy i typy wszystkich kolumn.

```
1 CREATE TABLE Customer OF Customer_Type (
2   PRIMARY KEY (customer_id)
3 );
```

Kod 5.23: Tworzenie tabeli klientów w Postgres.

Postgres umożliwia tworzenie kolekcji za pomocą typów tablic `array`. W tablicach można przechowywać dowolne typy wartości, w tym również obiekty typów użytkownika, co nie jest możliwe w systemie Db2. Przy korzystaniu z `array` możliwe jest nawet tworzenie tablic wielowymiarowych.

Fragment kodu 5.24 przedstawia tworzenie typu recenzji oraz typu produktu, w którym jednym z atrybutów jest tablica recenzji. Wykorzystanie tablicy nie wymaga definiowania dodatkowego typu, jak w bazie Oracle. Nie ma również potrzeby określenia maksymalnego rozmiaru tablicy - można ją swobodnie rozszerzać i dodawać elementy.

```
1 CREATE TYPE Review_Type AS (
2   score          INTEGER,
3   content        VARCHAR(400),
4   author         VARCHAR(50),
5   created        TIMESTAMP
6 );
7 CREATE TYPE Product_Type AS (
8   product_id     INTEGER,
```

```
9    brand          VARCHAR(100),
10   name           VARCHAR(100),
11   price          DECIMAL(9,2),
12   reviews        Review_Type[]
13 );
```

Kod 5.24: Tworzenie typu recenzji i produktu w Postgres.

Mechanizmy dziedziczenia w Postgres są stosunkowo ubogie w porównaniu do Oracle i Db2. Dostępne jest jedynie dziedziczenie tabel, podczas którego nie ma żadnej możliwości wskazania typu będącego wzorem dla struktury podtabeli. Tworzenie hierarchii tabel dla produktu i jego podtypów przedstawia fragment kodu 5.25. W każdej definicji tabeli konieczne jest ponowne wskazanie klucza głównego - indeksy, klucze główne i obce oraz ograniczenia kolumn nie są dziedziczone w Postgres.

```
1  CREATE TABLE Product OF Product_Type (
2    reviews   DEFAULT '{}',
3    PRIMARY KEY (product_id)
4  );
5
6  CREATE TABLE Monitor (
7    refresh_rate  INTEGER,
8    speakers      SMALLINT,
9    PRIMARY KEY (product_id)
10 ) INHERITS (Product);
11
12 CREATE TABLE Headset (
13   sensitivity       INTEGER,
14   impedance         INTEGER,
15   noise_cancelling  SMALLINT,
16   PRIMARY KEY (product_id)
17 ) INHERITS (Product);
18
```

```
19 CREATE TABLE Mouse (
20   dpi        INTEGER,
21   sensor     VARCHAR(20),
22   PRIMARY KEY (product_id)
23 ) INHERITS (Product);
```

Kod 5.25: Tworzenie tabeli produktów i jej podtabel w Postgres.

Przy tworzeniu tabeli produktów dodatkowo wskazano wartość domyślną pustej tablicy dla atrybutu `reviews` (linia 2), tak by tablica recenzji była zawsze zainicjalizowana, nawet jeśli recenzje nie zostaną wstawione wraz z produktem. Pozwala to na wygodne dodawanie elementów do tablicy bez konieczności sprawdzania czy tablica już istnieje.

Fragment kodu 5.26 przedstawia tworzenie typu i tabeli zakupów. W poleceniu tworzenia tabeli widać wadę Postgres w kwestii podejścia obiektowego - Postgres nie posiada żadnego mechanizmu referencji ani typów referencyjnych. Zamiast tego konieczne jest wykorzystanie kluczy obcych, dokładnie jak realizowane jest to w rozwiązaniu relacyjnym.

```
1  CREATE TYPE Purchase_Type AS (
2    purchase_id    INTEGER,
3    customer_id    INTEGER,
4    product_id     INTEGER,
5    count          INTEGER,
6    status         VARCHAR(20),
7    ordered        TIMESTAMP
8  );
9
10 CREATE TABLE Purchase OF Purchase_Type (
11   PRIMARY KEY (purchase_id),
12   FOREIGN KEY (customer_id) REFERENCES Customer
13 );
```

Kod 5.26: Tworzenie typu i tabeli zakupów w Postgres.

Podczas tworzenia tabeli zakupów celowo nie założono klucza obcego wskazującego na tabelę produktów. Wynika to z ogromnego problemu związanego ze stosowaniem dziedziczenia tabel - Postgres nie obsługuje propagacji i śledzenia klucza głównego pomiędzy tabelami w hierarchii dziedziczenia. Założenie klucza obcego na tabelę produktów pozwala tworzyć związki tylko do wartości przechowywanych bezpośrednio w tej tabeli, a nie w podtabelach. Jako że wszystkie instancje produktów należą do któregoś z podtypów i wstawiane są do odpowiedniej podtabeli, tworzenie takiego związku jest całkowicie nieprzydatne.

Niestety obecnie nie ma rozwiązania dla tego problemu i bardzo ogranicza on przydatność dziedziczenia tabel w Postgres. Na potrzeby badań założono, że utrzymanie więzów integralności pomiędzy tabelą zakupów a tabelą produktów oraz zapewnienie unikalności kluczy głównych w całej hierarchii produktów pozostaje w gestii użytkownika bazy.

Postgres nie posiada mechanizmu metod powiązanych z typami. Zamiast tego możliwe jest tworzenie funkcji operujących na dowolnych typach, które w praktyce mogą skutecznie zastąpić metody. Definicje funkcji, odpowiadających metodom przygotowanym w Oracle i Db2, przedstawiono na fragmencie kodu 5.27. Pierwszy parametr obu funkcji jest odpowiednikiem instancji obiektu, na której wywoływane są metody w pozostałych systemach. Podając parametry nie nadano im nazw - zamiast tego w treści metody odwołuje się do nich za pomocą numeru parametru. Symbol "$$" określa początek i koniec ciała metody.

```
1  CREATE OR REPLACE FUNCTION get_complete_address(
2    Address_Type) RETURNS VARCHAR(300) AS
3    $$ SELECT ($1).street || ' ' || ($1).house_number ||
4    '\n' || ($1).zip_code || ' ' || ($1).city || ', ' ||
5    ($1).country $$
6  LANGUAGE sql;
7
8  CREATE OR REPLACE FUNCTION total_cost(
9    Product_Type, integer) RETURNS DECIMAL AS
10   $$ SELECT ($1).price * $2 $$
```

```
11 LANGUAGE sql;
```

Kod 5.27: Definicje funkcji w Postgres.

Wstawianie danych do tabel stworzonych w Postgres w dużej mierze realizowane jest tak samo jak dla standardowej struktury relacyjnej. Przykłady poleceń wstawiania danych przedstawia fragment kodu 5.28. Wstawiając dane do tabeli klientów wykorzystano konstruktor wiersza `ROW` do stworzenia obiektu adresu, którego typ jawnie wskazano za pomocą operatora "`::`".

Dane produktów wstawiane są do odpowiednich podtabel w hierarchii dziedziczenia. Jak już wspomniano, Postgres nie posiada mechanizmów propagacji klucza głównego i nie pilnuje unikalności wartości klucza pomiędzy tabelami w hierarchii. Nic nie stoi na przeszkodzie by do każdej podtabeli wstawić wiersz o takiej samej wartości `product_id`.

```
1  INSERT INTO Customer VALUES (1, 'Abel', 'Doe',
2    'abeldoes@mail.com', ROW('PL', 'Warsaw', '44-300',
3    'Lubuska', '5/2')::Address_Type);
4
5  INSERT INTO Monitor VALUES
6    (1, 'Aker', 'A1', 1200, '{}', 60, 0);
7  INSERT INTO Headset VALUES
8    (2, 'Fraction', 'F1', 1200, '{}', 80, 32, 0);
9  INSERT INTO Mouse VALUES
10   (3, 'Roar', 'R1', 500, '{}', 1600, 'optical');
11
12 INSERT INTO Purchase VALUES
13   (1, 1, 5, 1, 'complete', '2021-08-27 15:23:54');
```

Kod 5.28: Wstawianie danych do bazy Postgres.

Dodawanie recenzji dla istniejącego produktu przedstawia fragment kodu 5.29. W poleceniu wykorzystano wbudowaną funkcję `array_append` (linia 2), która dodaje nowy element na ostatnią pozycję tablicy.

```
1 UPDATE Product SET reviews =
2   array_append(reviews, ROW(5, 'Very good product!',
3   'anon01', '2021-08-27 18:20:15')::Review_Type)
4 WHERE product_id = 5;
```

Kod 5.29: Tworzenie typu i tabeli zakupów w Postgres.

5.5 Generacja danych

Do przeprowadzenia testów działania zaprojektowanych struktur, konieczne jest wprowadzenie do nich znaczącej liczby danych. W celu generacji danych stworzono aplikację w języku Java, z wykorzystaniem szkieletu aplikacyjnego Spring Boot, której zadaniem jest wygenerowanie skryptów SQL zawierających instrukcje wstawiające dane do wszystkich przygotowanych baz danych.

W aplikacji odwzorowano wszystkie klasy występujące w modelowanej domenie biznesowej, przedstawione na diagramie 5.1. Zachowano hierarchię dziedziczenia produktów, przynależność kolekcji recenzji do produktu oraz zagnieżdżenie adresu w klasie klienta. Dla każdej klasy modelu stworzono odpowiadającą klasę generatora, która tworzy instancje obiektów danego typu. Wartości pól obiektów dobierane są w sposób losowy według z góry określonych zasad.

Aplikacja posługuje się biblioteką *lorem* [2], która pozwala generować dowolną liczbę słów lub akapitów tekstu w stylu *Lorem Ipsum*, co wykorzystano do tworzenia treści recenzji i nazw ulic. Oprócz tego *lorem* pozwala generować losowe kody pocztowe, nazwy miast oraz państw. Miasta i państwa wybierane są ze skończonej listy predefiniowanych treści, dzięki czemu adresy w zbiorze danych zawierają powtarzające się wartości dla tych atrybutów.

Imiona i nazwiska klientów dobierane są losowo z listy wartości uzyskanej z repozytorium *NameDatabases* [3], które zawiera zbiory imion i nazwisk dla różnych grup kulturowych. Email klienta tworzony jest poprzez złączenie jego imienia, nazwiska oraz końcówki “mail.com”. Autorzy recenzji generowani są na podstawie tego samego zbioru, poprzez złączenie kilku pierwszych liter losowego imienia i nazwiska.

Producenci produktów dobierani są z autorskiej listy trzydziestu fikcyjnych nazw. Przygotowano także autorską listę trzydziestu słów składowych, z których budowane są nazwy produktów, poprzez połączenie od 1 do 3 losowo wybranych słów. Dla pewnej liczby produktów dodatkowo do nazwy dodawany jest dopisek "Pro" lub numer z zakresu od 2 do 6.

Unikalne identyfikatory wszystkich obiektów tworzone są poprzez inkrementację licznika obecnego w klasie generatora każdego typu. Wszystkie wartości liczbowe generowane są losowo w ramach określonych przedziałów. Daty i godziny stworzenia recenzji oraz dokonania zakupu losowane są z zakresu od 01.01.2015 do 05.09.2021. Wartości tekstowe dla sensora myszy oraz statusu zakupu wybierane są ze zbioru kilku predefiniowanych wartości.

Efektem działania aplikacji jest powstanie czterech plików tekstowych zawierających polecenia SQL. Trzy z nich zawierają komendy dostosowane do struktur obiektowo-relacyjnych w systemach Oracle, Db2 i Postgres. Czwarty plik zawiera uniwersalne polecenia dla struktury relacyjnej, które są poprawne dla każdego systemu. Większość poleceń w pliku to polecenia `INSERT`, jedynie dla struktur obiektowo-relacyjnych w Oracle i Postgres występują polecenia `UPDATE` dla wstawiania recenzji do zagnieżdżonych kolekcji. Recenzje nie są wstawiane razem z produktem, tylko w osobnych, niezależnych poleceniach, by symulować zachowanie prawdziwego systemu, w którym recenzje dodawane byłyby stopniowo dla istniejących produktów. Wszystkie wygenerowane skrypty można znaleźć na płycie dołączonej do pracy w katalogu "Scripts\Data".

Zbiór danych wygenerowany na potrzeby pracy zawiera 100 000 klientów, 10 000 każdego typu produktów, 300 000 zakupów i 200 000 recenzji. Wstawianie tak dużej ilości danych do bazy jest stosunkowo czasochłonne. Czasy wykonania skryptów wstawiających dane dla każdego systemu w obu strukturach bazy przedstawia tabela 5.1. Najdłuższe czasy zaobserwowano dla systemu Db2, a najkrótsze dla PostgreSQL. We wszystkich systemach wstawianie danych dla struktury obiektowo-relacyjnej trwało dłużej niż dla struktury relacyjnej.

Tabela 5.1: Czasy wstawiania danych do stworzonych struktur.

Struktura	**Oracle**	**Db2**	**Postgres**
Obiektowo-relacyjna	72 min	130 min	54 min
Relacyjna	60 min	122 min	49 min

5.6 Zapytania testowe

Dla wszystkich utworzonych struktur obiektowo-relacyjnych przygotowano szereg zapytań realizujących przykładowe operacje biznesowe. Przygotowano również zapytania dla struktur relacyjnych, utworzonych w każdym systemie zgodnie z modelem przedstawionym w sekcji 5.3 Podejście relacyjne. Głównym celem przedstawionych zapytań jest prezentacja możliwości oraz ograniczeń występujących podczas pracy ze strukturami obiektowo-relacyjnymi w analizowanych systemach. Celowo wybrano zróżnicowane zagadnienia, tak aby pokazać szeroki zakres problemów i sposobów ich rozwiązania.

Wszystkie przygotowane zapytania wykonano pięciokrotnie na tym samym zbiorze danych i zmierzono czas ich realizacji. Dla każdego zapytania przedstawiono najdłuższy i najkrótszy czas wykonania, wartość średnią oraz odchylenie standardowe. Porównując uzyskane wyniki, należy jednak pamiętać o dużych różnicach w strukturach zaimplementowanych w poszczególnych systemach. W rozwiązaniach nie wprowadzono także żadnych indeksów, które mogłyby znacząco wpłynąć na prędkość realizacji zapytań.

W procesie testowania, w każdym systemie stworzono strukturę obiektowo-relacyjną, wstawiono do niej zbiór danych i wykonano wszystkie zaplanowane zapytania. Następnie całkowicie wyczyszczono bazę i powtórzono proces dla struktury relacyjnej, tak by w bazie przechowywany był zawsze tylko jeden zestaw danych.

5.6.1 Zapytanie 1 - Średnia ocena wybranego produktu

Cel zapytania: *Obliczyć średnią ocenę wybranego produktu na podstawie wszystkich jego recenzji.*

Na potrzeby tego zagadnienia zakłada się, że znany jest identyfikator produktu, który został wyszukany wcześniej, w oparciu o nazwę, producenta lub inne cechy.

Ta sama zasada dotyczy wszystkich zapytań, które odnoszą się do wybranego produktu lub klienta.

Fragment kodu 5.30 przedstawia realizację operacji w podejściu relacyjnym. Recenzje przechowywane są w osobnej tabeli i posiadają klucz obcy wskazujący na produkt, który wykorzystano do wyboru recenzji związanych z konkretnym produktem.

```
1 SELECT  AVG(score)
2 FROM    Review
3 WHERE   product_id = 15000;
```

Kod 5.30: Zapytanie pierwsze w podejściu relacyjnym.

Treść zapytania dla bazy Oracle przedstawiona jest na fragmencie 5.31. W tej strukturze lista recenzji przechowywana jest jako tabela zagnieżdżona wewnątrz typu produktu. By uzyskać dostęp do kolekcji wykorzystano operator `TABLE`, który wydobywa zagnieżdżone dane do zmiennej i pozwala z nich korzystać jak ze standardowej tabeli.

```
1 SELECT AVG(r.score)
2 FROM   product p, TABLE(p.reviews) r
3 WHERE  product_id = 15000;
```

Kod 5.31: Zapytanie pierwsze w Oracle.

Realizacja zapytania w Db2 przedstawiona jest na fragmencie 5.32. Ponieważ recenzje w Db2 przechowywane są w osobnej tabeli, treść zapytania jest bardzo podobna do rozwiązania relacyjnego. Zamiast klucza głównego wybór recenzji wykonuje się przyrównując kolumnę referencji do instancji typu referencyjnego. By zwrócić wynik w postaci ułamka, w pierwszej linii zapytania dodatkowo wykonano rzutowanie oceny na typ `DECIMAL` - bez tego zwrócony byłby typ całkowity, bo operacja `AVG` w Db2 domyślnie zwraca taki sam typ jak typ uśrednianych elementów.

```
1 SELECT  AVG(CAST(score AS DECIMAL(4,2)))
2 FROM    Review
```

```
3 WHERE   product = Product_Type(15000);
```

Kod 5.32: Zapytanie pierwsze w Db2.

Treść zapytania w Postgres przedstawia fragment 5.33. W tej strukturze recenzje przechowywane są jako tablica w obiekcie produktu. By wydobyć elementy listy recenzji wykorzystano operator `UNNEST`, który tworzy zbiór wierszy na podstawie kolekcji. Wyliczenie średniej odbywa się w zagnieżdżonym zapytaniu `SELECT`.

```
1 SELECT (SELECT AVG(r.score) FROM UNNEST(reviews) r)
2 FROM  product
3 WHERE product_id = 15000;
```

Kod 5.33: Zapytanie pierwsze w Postgres.

Czasy wykonania zapytań we wszystkich systemach przedstawia tabela 5.2. Na uwagę zasługuje szczególnie niski czas uzyskany w Oracle w strukturze obiektowo-relacyjnej, gdzie średni czas to 7,6 ms. Tak dobre wyniki są prawdopodobnie efektem wykorzystania tabeli zagnieżdżonej do przechowywania recenzji. Dla Db2 czasy realizacji są bardzo zbliżone w obu strukturach. W Postgres uzyskano około 25% lepsze wyniki dla struktury obiektowo-relacyjnej.

Tabela 5.2: Czasy wykonania pierwszego zapytania.

Struktura relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	26,8 ms	21 ms	46 ms	10,76 ms
Db2	35,6 ms	25 ms	74 ms	21,49 ms
Postgres	61,8 ms	58 ms	68 ms	4,49 ms
Struktura obiektowo-relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	7,6 ms	5 ms	12 ms	2,70 ms
Db2	36 ms	30 ms	55 ms	10,70 ms
Postgres	45,8 ms	42 ms	52 ms	4,15 ms

5.6.2 Zapytanie 2 - Negatywne recenzje produktów producenta

Cel zapytania: *Wypisać szczegóły wszystkich recenzji z oceną 1 utworzonych dla produktów wybranego producenta oraz nazwę i cenę powiązanych produktów.*

Realizacja zagadnienia w podejściu obiektowym przedstawiona jest na fragmencie kodu 5.34. Za pomocą identyfikatora `product_id` łączone są tabele recenzji i produktów. Wykonane jest filtrowanie wyników po nazwie producenta oraz wyniku recenzji i zwrócone są wymagane atrybuty.

```
1 SELECT p.name, p.price, r.*
2 FROM   Review r JOIN Product p
3 ON     p.product_id = r.product_id
4 WHERE p.brand = 'Vester' AND r.score = 1;
```

Kod 5.34: Zapytanie drugie w podejściu relacyjnym.

Zapytanie dla Oracle przedstawia fragment kodu 5.35. Dostęp do elementów zagnieżdżonej kolekcji recenzji uzyskuje się za pomocą funkcji `TABLE`, po czym kolekcja wykorzystana jest tak jak zwykła tabela i reszta zapytania wygląda prawie identycznie do wersji relacyjnej.

```
1 SELECT p.name, p.price, r.*
2 FROM   product p, TABLE(p.reviews) r
3 WHERE  brand = 'Vester' AND r.score = 1;
```

Kod 5.35: Zapytanie drugie w Oracle.

Wersję zapytania dla Db2 przedstawia fragment 5.36. W tej strukturze recenzje przechowywane są w osobnej tabeli, a relację recenzji i produktu realizuje kolumna przechowująca referencję. Za pomocą operatora dereferencji "->" możliwy jest dostęp do atrybutów powiązanego produktu.

```
1 SELECT  product->name, product->price, r.*
2 FROM    Review r
```

```
3 WHERE   product->brand = 'Vester' AND r.score = 1;
```

Kod 5.36: Zapytanie drugie w Db2.

Realizację zapytania w Postgres przedstawia fragment kodu 5.37. Treść zapytania jest bardzo podobna do rozwiązania w Oracle, jednak tutaj zamiast funkcji `TABLE` wykorzystano funkcję `UNNEST` do wydobycia zagnieżdżonej tablicy recencji. Pozostała część zapytania jest identyczna jak w Oracle.

```
1 SELECT p.name, p.price, r.*
2 FROM   product p, UNNEST(p.reviews) r
3 WHERE  brand = 'Vester' AND r.score = 1;
```

Kod 5.37: Zapytanie drugie w Postgres.

Czasy wykonania zapytań we wszystkich systemach przedstawia tabela 5.3. Podobnie jak dla pierwszego zapytania, widoczny jest szybki czas wykonania dla struktury obiektowo-relacyjnej w Oracle, jednak w tym przypadku równie dobre rezultaty widać dla struktury relacyjnej w Oracle i Db2. Czas realizacji dla struktury obiektowo-relacyjnej w Db2 jest około dwa razy dłuższy niż dla struktury relacyjnej. Można po tym wnioskować, wykorzystanie dereferencji skutkuje gorszą wydajnością niż przy relacyjnym łączeniu tabel. Czasy wykonania w Postgres dla obu struktur są znacznie wyższe niż w pozostałych systemach.

Tabela 5.3: Czasy wykonania drugiego zapytania.

Struktura relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	6 ms	5 ms	7 ms	0,71 ms
Db2	8,2 ms	6 ms	14 ms	3,27 ms
Postgres	98,6 ms	94 ms	105 ms	5,13 ms
Struktura obiektowo-relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	7,2 ms	6 ms	9 ms	1,10 ms
Db2	16,6 ms	15 ms	21 ms	2,51 ms
Postgres	84 ms	78 ms	93 ms	5,66 ms

5.6.3 Zapytanie 3 - Suma wydatków klienta w wybranym okresie czasu

Cel zapytania: *Obliczyć sumę kosztów wszystkich zakupów wykonanych przez wybranego klienta w okresie czasu pomiędzy dwoma datami.*

Wersja zapytania w podejściu relacyjnym przedstawiona jest na fragmencie kodu 5.38. Za pomocą wartości identyfikatorów łączone są trzy tabele: produktów, klientów oraz zakupów. We frazie `WHERE` wskazany jest klient oraz zakres czasu zakupów. Ceny produktów mnożone są przez liczbę produktów zamówionych w ramach pojedynczego zakupu, a suma uzyskanych wartości stanowi wynik zapytania.

```
1 SELECT SUM(price * count)
2 FROM  Customer, Purchase, Product
3 WHERE Customer.customer_id = 25000
4 AND ordered >= '2018-01-01' AND ordered <= '2021-08-30'
5 AND Customer.customer_id = Purchase.customer_id
6 AND Product.product_id = Purchase.product_id;
```

Kod 5.38: Zapytanie trzecie w podejściu relacyjnym.

Rozwiązanie zagadnienia w Oracle przedstawia fragment 5.39. Tutaj możliwe jest wykorzystanie referencji, co upraszcza składnię zapytania. Z tabeli zakupów za pomocą referencji można odnieść się do obiektów klienta i produktu, dzięki czemu nie ma potrzeby wskazywania połączeń pomiędzy tabelami. W celu obliczenia kosztów poszczególnych zakupów wykorzystano metodę `total_cost`, która wykonuje mnożenie kosztu produktu i liczby zakupionych elementów.

```
1 SELECT SUM(p.product.total_cost(count))
2 FROM  Purchase p
3 WHERE p.customer.customer_id = 25000
4 AND ordered >= '2018-01-01' AND ordered <= '2021-08-30';
```

Kod 5.39: Zapytanie trzecie w Oracle.

Zapytanie w Db2 przedstawia fragment 5.40. Format zapytania jest bardzo po-

dobny do wersji Oracle. Tutaj również wykorzystano referencję, by odnieść się do klienta i produktu. W celu wyboru klienta nałożono warunek porównujący referencję ze skonstruowanym typem referencyjnym. Podobnie jak w rozwiązaniu dla Oracle wykorzystano metodę `total_cost`, wywołując ją za pomocą charakterystycznego operatora dereferencji "->".

```
1 SELECT SUM(p.product->total_cost(p.count))
2 FROM   Purchase p
3 WHERE  p.customer = Customer_Type(25000)
4 AND ordered >= '2018-01-01' AND ordered <= '2021-08-30';
```

Kod 5.40: Zapytanie trzecie w Db2.

Treść zapytania w systemie Postgres przedstawia fragment 5.41. W tym systemie nie występują referencje, więc zapytanie jest bardzo zbliżone do rozwiązania relacyjnego i jedyną różnicą jest wykorzystanie funkcji `total_cost`. Ponieważ Postgres nie obsługuje metod, operacja ta stworzona jest jako funkcja, do której przekazuje się produkt i liczbę zakupionych elementów.

```
1 SELECT SUM(total_cost(Product, count))
2 FROM Customer, Purchase, Product
3 WHERE Customer.customer_id = 25000
4 AND ordered >= '2018-01-01' AND ordered <= '2021-08-30'
5 AND Customer.customer_id = Purchase.customer_id
6 AND Product.product_id = Purchase.product_id;
```

Kod 5.41: Zapytanie trzecie w Postgres.

Czasy wykonania zapytań we wszystkich systemach przedstawia tabela 5.4. Wyniki w Postgres dla obu struktur są niemal identyczne, ponieważ w tym systemie fragment bazy istotny dla realizowanego zagadnienia jest taki sam w obu podejściach. Wykorzystanie funkcji zamiast bezpośredniego łączenia pól w zapytaniu nie ma znaczącego wpływu na czas realizacji zapytania. Dla struktury obiektowo-relacyjnej Db2 uzyskało bardzo podobny wynik co Postgres, a Oracle uzyskało czasy około 55% krótsze. Różnica w wynikach jest zaskakująca, ponieważ treść zapytania dla tej struktury jest bardzo podobna w Db2 i Oracle. Najlepsze wy-

niki uzyskano w strukturze relacyjnej w Db2 i Oracle - w obu systemach średni czas wykonania zapytania wyniósł około 17 ms. W przypadku Oracle można zaobserwować wysokie odchylenie standardowe i dużą różnicę pomiędzy minimalnym i maksymalnym czasem. Najdłuższy czas 56 ms uzyskano przy pierwszym wykonaniu zapytania; kolejne czasy wykonania były dużo krótsze, ponieważ system skutecznie wykorzystał mechanizm pamięci podręcznej (ang. *cache*) w celu szybszej realizacji zapytania.

Tabela 5.4: Czasy wykonania trzeciego zapytania.

Struktura relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	17,4 ms	7 ms	56 ms	21,59 ms
Db2	16,4 ms	16 ms	17 ms	0,55 ms
Postgres	69,2 ms	57 ms	103 ms	19,29 ms
Struktura obiektowo-relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	30,8 ms	28 ms	37 ms	3,63 ms
Db2	69,6 ms	61 ms	76 ms	6,15 ms
Postgres	68,8 ms	62 ms	88 ms	10,80 ms

5.6.4 Zapytanie 4 - Największa liczba recenzji dla pojedynczego produktu

Cel zapytania: *Znaleźć maksymalną liczbę recenzji utworzonych dla pojedynczego produktu.*

Relacyjna wersja zapytania przedstawiona jest na fragmencie kodu 5.42. Jednocześnie jest to także rozwiązanie dla struktury obiektowo-relacyjnej w Db2, gdzie recenzje nie są przechowywane jako zagnieżdżona kolekcja, tylko osobna tabela. Recenzje są grupowane i zliczane za pomocą podzapytania, a następnie spośród otrzymanych rezultatów wybierana jest maksymalna wartość.

```
1 SELECT MAX(how_many) FROM (
```

```
2   SELECT count(*) how_many
3   FROM Review
4   GROUP BY product_id) r;
```

Kod 5.42: Zapytanie czwarte w podejściu relacyjnym i strukturze Db2.

Dla systemów Oracle i Postgres również można wykorzystać identyczną tresć zapytania, którą przedstawiono na fragmencie kodu 5.43. W obu tych systemach recenzje są zagnieżdżone w tabeli produktów (w Postgres wykorzystując tablicę, a w Oracle tabelę zagnieżdżoną). Do odczytania aktualnego rozmiaru kolekcji w obu systemach wykorzystano wbudowaną funkcję `cardinality`. Zwraca ona rozmiar pierwszego wymiaru kolekcji, co jest wystarczające w przypadku jednowymiarowej listy.

```
1 SELECT MAX(cardinality(reviews))
2 FROM Product;
```

Kod 5.43: Zapytanie czwarte w Oracle i Postgres.

Czasy wykonania zapytań we wszystkich systemach przedstawia tabela 5.5. Dla systemów Oracle i Postgres czas realizacji zapytania dla struktury relacyjnej jest dużo krótszy niż dla struktury obiektowo-relacyjnej. Sugeruje to, że wbudowana funkcja `cardinality` jest mało wydajna i dużo szybsze jest obliczanie liczby wierszy zwykłej tabeli relacyjnej. Czas wykonania zapytania w Db2 jest bardzo podobny dla obu struktur, ponieważ obie bardzo podobnie realizują przechowywanie recenzji. Podobnie jak w poprzednim zagadnieniu, w wynikach systemu Oracle dla struktury relacyjnej widać dużą różnicę pomiędzy maksymalnym i minimalnym czasem realizacji, związaną z wykorzystaniem pamięci podręcznej w kolejnych wykonaniach zapytania.

5.6.5 Zapytanie 5 - Pełny adres klientów, którzy kupili wybrany produkt

Cel zapytania: *Wypisać w postaci pojedynczego tesktu pełne adresy wszystkich klientów, którzy kiedykolwiek dokonali zakupu wybranego produktu.*

Tabela 5.5: Czasy wykonania czwartego zapytania.

Struktura relacyjna				
System	**Średnia**	**Min**	**Max**	σ
Oracle	50,4 ms	31 ms	84 ms	25,19 ms
Db2	47,6 ms	47 ms	50 ms	1,34 ms
Postgres	92,2 ms	90 ms	95 ms	1,92 ms
Struktura obiektowo-relacyjna				
System	**Średnia**	**Min**	**Max**	σ
Oracle	198,2 ms	195 ms	201 ms	2,17 ms
Db2	52,8 ms	51 ms	58 ms	2,95 ms
Postgres	172,2 ms	165 ms	188 ms	9,68 ms

Treść zapytania w podejściu relacyjnym przedstawia fragment kodu 5.44. Wykonano złączenie tabel zakupów i klientów oraz filtrowanie danych na podstawie identyfikatora produktu. Rezultatem jest pojedynczy łańcuch znakowy stworzony z wykorzystaniem wszystkich elementów adresu. Fraza `DISTINCT` wykorzystana w tym zapytaniu i jego wersjach dla innych systemów zapewnia, że na liście adresów nie występują duplikaty.

```
1 SELECT c.street || ' ' || c.house_number || '\n' ||
2        c.zip_code || ' ' || c.city || ', ' ||
3        c.country
4 FROM   Purchase p JOIN Customer c
5 ON     p.customer_id = c.customer_id
6 WHERE  p.product_id = 100;
```

Kod 5.44: Zapytanie piąte w podejściu relacyjnym.

Treść zapytania w Oracle przedstawia fragment 5.45. Dzięki referencji można w łatwy sposób odnieść się do powiązanych obiektów klienta i produktu, w celu pozyskania adresu i stworzenia warunku filtrującego produkty. W celu wypisania adresów, skorzystano z metody `get_complete_address`, która zawiera logikę składania łańcucha tekstowego ze wszystkich elementów adresu.

```
1 SELECT DISTINCT
2   p.customer.address.get_complete_address()
3 FROM  Purchase p
4 WHERE p.product.product_id = 100;
```

Kod 5.45: Zapytanie piąte w Oracle.

Rozwiązanie dla Db2 przedstawia fragment kodu 5.46. Treść zapytania jest bardzo podobna do wersji Oracle. Referencje umożliwiają łatwy dostęp do obiektów klienta i produktu. W celu wybrania produktu wykorzystano porównanie ze stworzonym typem referencyjnym. Do uzyskania pełnego adresu wykorzystano metodę `get_complete_address`, której wywołanie na zagnieżdżonym obiekcie adresu wymaga użycia specjalnego operatora "..".

```
1 SELECT DISTINCT
2   p.customer->address..get_complete_address()
3 FROM  Purchase p
4 WHERE p.product = Product_Type(100);
```

Kod 5.46: Zapytanie piąte w Db2.

Treść zapytania dla Postgres przedstawia fragment 5.47. Ponieważ Postgres nie posiada referencji, rozwiązanie jest niemal identyczne z zapytaniem dla podejścia relacyjnego. Jedyną różnicą występującą w zapytaniu jest wykorzystanie funkcji `get_complete_address` do stworzenia pojedynczego łańcucha tekstowego z pełnym adresem klienta.

```
1 SELECT get_complete_address(c.address)
2 FROM  Purchase p JOIN Customer c
3 ON    p.customer_id = c.customer_id
4 WHERE p.product_id = 100
```

Kod 5.47: Zapytanie piąte w Postgres.

Czasy wykonania zapytań we wszystkich systemach przedstawia tabela 5.6. Dla struktury obiektowo-relacyjnej systemy Db2 i Oracle osiągnęły bardzo podobne

wyniki o średnim czasie wykonania około 52 ms. Czasy realizacji w Postgres są około 15% gorsze. Dla struktury relacyjnej systemy Db2 i Oracle uzyskały dużo lepsze rezultaty, mimo że wykonywane zapytanie zawiera złączenie dwóch tabel, które nie występuje w podejściu obiektowym. Można wnioskować, że operacja dereferencji zastosowana dla struktury obiektowo-relacyjnej jest mniej wydajna niż relacyjne złączenie tabel. System Postgres osiągnął niemal identyczne wyniki dla obu struktur, ponieważ w tym systemie istotny fragment modelu bazy jest taki sam w obu podejściach. Zastosowanie funkcji zamiast zdefiniowania operacji w treści zapytania nie ma znaczącego wpływu na czas realizacji zapytania.

Tabela 5.6: Czasy wykonania piątego zapytania.

Struktura relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	10,6 ms	7 ms	21 ms	5,86 ms
Db2	15,6 ms	13 ms	23 ms	4,16 ms
Postgres	55,6 ms	53 ms	58 ms	1,82 ms
Struktura obiektowo-relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	52 ms	51 ms	55 ms	1,73 ms
Db2	51,6 ms	49 ms	56 ms	2,88 ms
Postgres	58,6 ms	56 ms	64 ms	3,21 ms

5.6.6 Zapytanie 6 - Szczegóły wszystkich myszy kupionych przez klienta

Cel zapytania: *Wypisać producenta, nazwę, precyzję oraz rodzaj czujnika wszystkich myszy zakupionych przez wybranego klienta.*

Głównym zamiarem tego zapytania jest sprawdzenie możliwości poszczególnych rozwiązań w kwestii ograniczenia zwracanych rezultatów do konkretnego rodzaju produktów. Oprócz filtrowania rezultatów, należy jednocześnie wyświetlić informacje przechowywane w polach podtypu oraz jego rodzica.

Relacyjną wersję zapytania przedstawia fragment kodu 5.48. W rozwiązaniu relacyjnym tabela produktów przechowuje podstawowe informacje wszystkich rodzajów produktów, a dane specyficzne dla podtypów przechowywane są w osobnych tabelach powiązanych kluczem obcym. W związku z tym konieczne jest złączenie aż trzech tabel, przechowujących zakupy, produkty oraz detale myszy. Z tabeli produktów odczytano producenta i nazwę produktu, a z tabeli myszy rodzaj czujnika i precyzję.

```
1 SELECT m.product_id, pr.brand, pr.name, m.dpi, m.sensor
2 FROM   purchase p
3 JOIN   mouse m ON p.product_id = m.product_id
4 JOIN   product pr ON pr.product_id = m.product_id
5 WHERE  p.customer_id = 2000;
```

Kod 5.48: Zapytanie szóste w podejściu relacyjnym.

Rozwiązanie tego zagadnienia w systemie Oracle jest stosunkowo skomplikowane. Treść zapytania, które pozwala wybrać podtyp i uzyskać jego atrybuty przedstawia fragment kodu 5.49. Konieczne jest wykorzystanie podzapytania, które zwraca wartość referowanego obiektu, uzyskaną za pomocą funkcji `DEREF`. Tą wartość można następnie rzutować na wymagany typ za pomocą funkcji `TREAT`, co zapewnia dostęp do pól podtypu. Filtrowanie rezultatów według typu możliwe jest za pomocą frazy `IS OF`. Złożoność zapytania wynika z wbudowanych ograniczeń systemu. Oracle nie pozwala w jednym zapytaniu wykonać dereferencji, a następnie wykorzystać rezultat w celu rzutowania lub porównania typu, na przykład wykonując `DEREF(pp.product)IS OF (Mouse_Type)`. W takiej sytuacji konieczne jest wykorzystanie podzapytania, które zwraca wynik dereferencji.

```
1 SELECT product_id, brand, name,
2  TREAT(p as Mouse_Type).dpi,
3  TREAT(p as Mouse_Type).sensor
4 FROM (SELECT DEREF(pp.product) as p,
5         pp.product.product_id as product_id,
6         pp.product.brand as brand,
```

```
7           pp.product.name as name
8         FROM Purchase pp
9         WHERE pp.customer.customer_id = 2000)
10 WHERE p IS OF (Mouse_Type);
```

Kod 5.49: Zapytanie szóste w Oracle.

Zapytanie dla Db2 pokazuje fragment kodu 5.50. W tym rozwiązaniu także użyto rzutowania produktu w celu uzyskania atrybutów konkretnego podtypu. Funkcja rzutująca `TREAT` operuje na wartości obiektu uzyskanej za pomocą funkcji `DEREF`. Filtrowanie produktów według typu także wymaga dereferencji oraz użycia frazy `IS OF`. W przeciwieństwie do Oracle, wszystkie działania wykonać można w jednym zapytaniu, bez tworzenia podzapytań. Ciekawą cechą widoczną w zapytaniu jest wykorzystanie różnych operatorów w celu dostępu do atrybutów. Dla typu referencji wykorzystuje się operator "->", a dla całych obiektów operator "..".

```
1 SELECT p.product, p.product->brand, p.product->name,
2 TREAT(DEREF(p.product) AS Mouse_Type)..dpi,
3 TREAT(DEREF(p.product) AS Mouse_Type)..sensor
4 FROM   Purchase p
5 WHERE  p.customer = Customer_Type(2000)
6 AND DEREF(p.product) IS OF (Mouse_Type);
```

Kod 5.50: Zapytanie szóste w Db2.

Rozwiązanie dla bazy Postgres przedstawia fragment 5.51. Ta wersja zapytania zbliżona jest do rozwiązania relacyjnego, ale dzięki dziedziczeniu tabel możliwe jest uzyskanie wszystkich potrzebnych atrybutów z tabeli myszy, bez konieczności złączenia z tabelą produktów. Dzięki temu treść zapytania w strukturze obiektowo-relacyjnej w Postgres jest prostsza niż pozostałe przedstawione wersje.

```
1 SELECT m.product_id, m.brand, m.name, m.dpi, m.sensor
2 FROM   Purchase p JOIN Mouse m
3 ON     p.product_id = m.product_id
```

```
4 WHERE  p.customer_id = 2000;
```

Kod 5.51: Zapytanie szóste w Postgres.

Czasy wykonania zapytań we wszystkich systemach przedstawia tabela 5.7. We wszystkich systemach uzyskano gorsze wyniki dla struktury obiektowo-relacyjnej. Największą różnicę widać w systemie Db2, gdzie dla podejścia obiektowego średni czas wykonania był ponad cztery razy dłuższy niż dla struktury relacyjnej. Różnica w systemie Oracle również jest znacząca i średni czas wykonania jest ponad trzy razy dłuższy dla struktury obiektowo-relacyjnej. Gorszy czas realizacji zapytania w obu systemach prawdopobodnie wynika z konieczności rzutowania obiektów na właściwy podtyp produktu. W Postgres nie ma potrzeby rzutowania danych i czasy wykonania są bardzo zbliżone dla obu struktur. Wprowadzenie dziedziczenia tabel w Postgres pozwoliło uprościć treść zapytania, jednocześnie nie pogarszając czasu jego realizacji.

Tabela 5.7: Czasy wykonania szóstego zapytania.

Struktura relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	8,6 ms	7 ms	14 ms	3,05 ms
Db2	15,8 ms	13 ms	24 ms	4,66 ms
Postgres	55,4 ms	55 ms	56 ms	0,55 ms
Struktura obiektowo-relacyjna				
System	**Średnia**	**Min**	**Max**	$\boldsymbol{\sigma}$
Oracle	31 ms	28 ms	36 ms	3,32 ms
Db2	75,4 ms	73 ms	79 ms	2,19 ms
Postgres	61,4 ms	56 ms	66 ms	3,58 ms

5.7 Wnioski

Porównując czasy wykonania zapytań we wszystkich strukturach i systemach, można zauważyć, że dla większości operacji lepsze wyniki uzyskano w strukturach relacyjnych, szczególnie w systemach Oracle i Db2. Jest to zrozumiały rezultat,

ponieważ dzisiejsze systemy bazodanowe są bardzo dobrze dostoswane do modelu relacyjnego i posiadają zaawansowane mechanizmy optymalizacji zapytań. Jeszcze lepsze rezultaty można by uzyskać poprzez przemyślane zaaplikowanie indeksów do odpowiednich tabel.

Obiektywna ocena struktury bazy wykorzystującej mechanizmy obiektowe jest bardzo skomplikowana, głównie przez ogromne zróżnicowanie implementacji cech obiektowych w poszczególnych ORSZBD. Większość technik oceny architektury i wydajności baz dostosowana jest tylko do rozwiązań relacyjnych, które są stosunkowo porównywalne pomiędzy systemami. Cechy obiektowe traktowane są jako opcjonalne i ich implementacja jest niespójna ze standardem SQL, więc nie istnieją uniwersalne metryki dla systemów obiektowo-relacyjnych [19].

Porównanie stworzonych struktur obiektowo-relacyjnych oraz przygotowanych dla nich zapytań pokazuje ogromne różnice w korzystaniu z cech obiektowych w poszczególnych systemach. Niektóre operacje są bardzo proste w jednym systemie, a wyjątkowo trudne do realizacji w innym. Problematyczna jest także trudność w przenoszeniu rozwiązań - każdy system obsługuje inne mechanizmy, wykorzystując niezgodne pomiędzy systemami składnie poleceń.

Mechanizmy metod w systemach, które je posiadają są bardzo ograniczone. Dużo bardziej praktyczne jest korzystanie ze zwykłych procedur lub funkcji tworzonych bez powiązania z konkretnymi typami. Oprócz większej wygody użytkowania zapewniają znacznie szersze możliwości, poprzez wykonywanie dodatkowych zapytań SQL, czy bezpośrednie edytowanie danych w tabelach.

Dużą zaletą stosowania podejścia obiektowo-relacyjnego, zaobserwowaną podczas tworzenia zapytań, jest wygoda odnoszenia się do powiązanych obiektów za pomocą referencji, co możliwe jest w systemach Db2 i Oracle. Dzięki dereferencji można łatwo odnosić się do pól i metod referowanych obiektów bez konieczności wykonywania złączeń.

Zaskakująco wygodne w wykorzystaniu okazały się kolekcje dostępne w systemach Postgres i Oracle. Szczególnie na uwagę zasługuje kolekcja zagnieżdżonej tabeli systemu Oracle, której wykorzystanie pozwoliło uzyskać dobre czasy realizacji zapytań w stosownych operacjach.

Rozdział 6

Podsumowanie

Celem pracy było przeanalizowanie cech obiektowych występujących w dostępnych na rynku bazach danych. Wykonano kompleksowy przegląd mechanizmów dostępnych w trzech obiektowo-relacyjnych systemach zarządzania bazą danych: Oracle Database, IBM Db2 i PostgreSQL. Wszystkie istotne mechanizmy zostały przetestowane i opisane. Porównano je również z obiektowymi rozwiązaniami standardu SQL. Jako uzupełnienie analizy teoretycznej, wykonano i zaprezentowano praktyczne eksperymenty, które dobrze obrazują wyzwania spotykane przy pracy z wszystkimi analizowanymi systemami.

Przeanalizowane systemy obiektowo-relacyjne udostępniają szeroką gamę cech i mechanizmów obiektowych. Ich wykorzystanie może być przydatne podczas projektowania struktury bazy danych, należy jednak ostrożnie rozważyć plusy i minusy wprowadzenia rozszerzeń obiektowych w tworzonych rozwiązaniach. Niezaprzeczalne są ogromne zalety modelu relacyjnego i wykorzystane mechanizmy obiektowe powinny stanowić uzupełnienie standardowej struktury relacyjnej, a nie zastąpić ją w całości. Szczególnie w systemach dużej skali, oparcie struktury bazy na rozwiązaniach obiektowych może prowadzić do problemów z wydajnością i realizacją zaawansowanych operacji.

Występują duże różnice wśród cech obiektowych dostępnych w poszczególnych ORSZBD i żadna z implementacji nie zachowuje zgodności ze standardem SQL. Powoduje to, że rozwiązania stworzone w oparciu o mechanizmy obiektowe są praktycznie niemożliwe do przeniesienia pomiędzy systemami. Oprócz migracji, brak

spójności rozwiązań utrudnia także naukę i popularyzację dostępnych mechanizmów. Dopóki rozwiązania dostępne na rynku nie wprowadzą wspólnego standardu implementacji, pozostanie to zasadniczą wadą stosowania rozszerzeń obiektowych.

Podczas realizacji pracy przygotowano aplikację do generacji danych, stworzoną w języku Java, z wykorzystaniem szkieletu aplikacyjnego Spring Boot. Aplikacja generatora danych jest bardzo uniwersalna i ma duży potencjał rozwoju. Dzięki odpowiednim modyfikacjom, może być dostosowana do tworzenia wsadowych skryptów SQL dla dowolnego SZBD i struktury bazy danych, co może być przydane przy różnego rodzaju badaniach i testach.

Analiza zrealizowana w ramach pracy może stanowić podstawę do dalszych badań. W celu dokładniejszej oceny mechanizmów obiektowych dostępnych w różnych systemach, wskazane jest przeprowadzenie bardziej wyczerpujących testów, np. z wykorzystaniem bardziej zróżnicowanych struktur lub większej ilości danych. Bardzo wartościowe może być zbadanie wpływu włączenia indeksowania na wydajność operacji w bazach obiektowo-relacyjnych. Można także rozszerzyć rozważania o dodatkowe systemy, które nie zostały szczegółowo omówione w tej pracy.

Bibliografia

[1] Amazon datasets. https://registry.opendata.aws/. [data dostępu: 2021-08-26].

[2] Github repository for lorem ipsum generator. https://github.com/mdeanda/lorem. [data dostępu: 2021-08-28].

[3] Github repository for names dataset. https://github.com/smashew/NameDatabases. [data dostępu: 2021-08-28].

[4] Ibm db2 community edition docker image on docker hub. https://hub.docker.com/r/ibmcom/db2. [data dostępu: 2021-08-26].

[5] Ibm db2 version 11.5 documentation. https://www.ibm.com/docs/en/db2/11.5. [data dostępu: 2021-08-21].

[6] Imdb datasets. https://www.imdb.com/interfaces/. [data dostępu: 2021-08-26].

[7] Microsoft sql server - end-user license agreement. https://documentation.commvault.com/11.21/essential/50060_microsoft_sql_server_eula.html. [data dostępu: 2021-08-21].

[8] Oracle database 19 - database concepts. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt. [data dostępu: 2021-08-21].

[9] Oracle database 19 - multimodel database. https://www.oracle.com/a/tech/docs/multimodel19c-wp.pdf. [data dostępu: 2021-08-21].

[10] Oracle database docker images repository on github. `https://github.com/oracle/docker-images`. [data dostępu: 2021-08-26].

[11] pgadmin 4 docker image on docker hub. `https://hub.docker.com/r/dpage/pgadmin4/`. [data dostępu: 2021-08-26].

[12] Postgresql 13.4 documentation. `https://www.postgresql.org/docs/13/index.html`. [data dostępu: 2021-08-21].

[13] Postgresql docker image on docker hub. `https://hub.docker.com/_/postgres`. [data dostępu: 2021-08-26].

[14] Ranking baz danych DB-Engines. `https://db-engines.com/en/ranking`. [data dostępu: 2021-08-10].

[15] Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation). Standard ISO/IEC 9075-2:2003, International Organization for Standardization, Stany Zjednoczone, 2003.

[16] Information technology – Database languages – SQL – Part 2: Object Language Bindings (SQL/OLB). Standard ISO/IEC 9075-10:2003, International Organization for Standardization, Stany Zjednoczone, 2003.

[17] Hibatullah Alzahrani. Evolution of Object-Oriented Database Systems. *Global Journal of Computer Science and Technology: Software & Data Engineering*, 16(3), 2016.

[18] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, S. Zdonik. The object-oriented database system manifesto. *First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989.

[19] Aline Lúcia Baroni, Coral Calero, Mario Piattini, Fernando Brito e Abreu. A formal definition for object-relational database metrics. *Seventh International Conference on Enterprise Information Systems*, 25-28 Maja 2005.

[20] Adam Bellemare. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media, 2020.

[21] Whei-Jen Chen, Angus Beaton, David Kline, Glen Johnson. *DB2 UDB Evaluation Guide for Linux and Windows*. Redbooks, 2003.

[22] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[23] Lex de Haan, Tim Gorman, Inger Jørgensen, Melanie Caffrey. *Beginning Oracle SQL for Oracle Database 12c*. Aspress, New York, 2014.

[24] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003.

[25] Maria Fernanda Golobisky, Aldo Vecchietti. Fundamentals for the Automation of Object-Relational Database Design. *International Journal of Computer Science Issues*, 8-3(2):9–22, 2011.

[26] Ian Graham. *Metody obiektowe w teorii i w praktyce*. WNT, Warszawa, 2004.

[27] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, Dishant Gosain. A survey and comparison of relational and non-relational database. *International journal of engineering research and technology*, 1(6), 2012.

[28] Ticiano Jordão, Iuliana Botha, Bâra Adela, Simona Oprea. Integrating xml technology with object-relational databases into decision support systems. *Database Systems Journal*, III(1), 2012.

[29] Carlos Alberto Rombaldo Jr, Solange N. Alves Souza. O-ODM Framework for Object-Relational Databases. *International Journal of Interactive Multimedia and AI*, 1(6):29–35, 2012.

[30] Hang Li, Hua Liu, Yong Liu, Yuan Wang. An Object-Relational IFC Storage Model based on Oracle Database. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLI-B2:625–631, 2016.

[31] Bertrand Meyer. *Programowanie zorientowane obiektowo*. Helion, Warszawa, 2004.

[32] Sam Ogunlere, Sunday Idowu. Comparison analysis of object-based databases, object-oriented databases, and object relational databases. *Asian Journal of Computer and Information Systems*, 3(2), 2015.

[33] B. Reinwald, T. J. Lehman, H. Pirahesh, V. Gottemukkala. Storing and using objects in a relational database. *Ibm Systems Journal*, 35(2):172–191, 1996.

[34] Michael Stonebraker, Paul Brown, Dorothy Moore. *Object-relational DBMSs: Tracking the Next Great Wave*. Morgan Kaufmann Publishers, San Francisco, 1999.

[35] Michael Stonebraker, Dorothy Moore. *Object-relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers, San Francisco, 1996.

[36] Ming Wang. Solving Relational Database Problems with ORDBMS in an Advanced Database Course. *Information Systems Education Journal*, 9(4):80–90, 2011.

Dodatki

Spis skrótów i symboli

ADT	Abstract Data Type - typ abstrakcyjny w systemie Oracle
AWS	Amazon Web Services
CASE	Computer-Aided Software Engineering - klasa narzędzi wspierających tworzenie oprogramowania
CDB	Container Database - baza kontenerowa w systemie Oracle
FIFO	First-In, First-Out
JDBC	Java Database Connectivity
LIFO	Last-In, First-Out
LUW	Linux-Unix-Windows - wersja bazy Db2 dla systemów z rodziny Unix i Windows
MSSQL	Microsoft SQL Server
ODBC	Open Database Connectivity
OQL	Object Query Language - obiektowy język zapytań
ORM	Object-Relational Mapping - mapowanie obiektowo-relacyjne
ORSZBD	Obiektowo-Relacyjny System Zarządzania Bazą Danych (ang. *ORDBMS - Object-Relational Database Management System*)
PDB	Pluggable Database - baza wtykowa w systemie Oracle

RSZBD Relacyjny System Zarządzania Bazą Danych (ang. *RDBMS - Relational Database Management System*)

SQL Structured Query Language - strukturalny język zapytań

SZBD System Zarządzania Bazą Danych (ang. *DBMS - Database Management System*)

UDT User-Defined Type - typ użytkownika w ORSZBD

Zawartość dołączonej płyty

Do pracy dołączona jest płyta CD z następującą zawartością:

- praca magisterska w formacie `pdf` (katalog główny),
- źródła LaTeXowe pracy (katalog "Praca magisterska - źródła LaTeX"),
- skrypty SQL tworzące strukturę relacyjną oraz struktury obiektowo-relacyjne (katalog "Scripts\Structure"),
- skrypty SQL wstawiające do każdej struktury bazy zbiór danych wykorzystany podczas badań (katalog "Scripts\Data"),
- arkusz zawierający czasy wykonania zapytań i obliczenia przedstawianych wartości (katalog główny),
- źródła aplikacji generatora danych (katalog "Generator danych"),
- pliki konfiguracyjne Docker Compose dla systemów Oracle, IBM Db2 i PostgreSQL wykorzystane do uruchomienia środowiska (katalog "Docker") .

Zawartość płyty jest również dostępna w repozytorium GitHub pod adresem `https://github.com/kaes1/master-thesis`.

Spis rysunków

Spis tablic

Spis fragmentów kodu