BAZY
DANYCH
TEORIA I PRAKTYKA

Katolicki Uniwersytet Lubelski Jana Pawła II
Wydział Matematyczno-Przyrodniczy
Instytut Matematyki i Informatyki

PAWEŁ URBANOWICZ
MARCIN PŁONKOWSKI
DMITRY URBANOWICZ

BAZY DANYCH

TEORIA I PRAKTYKA

PODRĘCZNIK DLA STUDENTÓW
UCZELNI WYŻSZYCH

Wydawnictwo KUL
Lublin 2010

Recenzenci
Dr hab. Vladimir Kudriavtsev
(Białoruski Państwowy Uniwersytet Technologiczny)
Prof. dr hab. Volodymyr Harbarchuk
(Politechnika Lubelska)

Opracowanie redakcyjne
Elżbieta Struś

Opracowanie komputerowe
Robert Kryński

Projekt okładki i stron tytułowych
Patrycja Czerniak

© Copyright by Wydawnictwo KUL, Lublin 2010

ISBN 978-83-7702-148-4

Wydawnictwo KUL
ul. Zbożowa 61, 20-827 Lublin
tel. 81 740-93-40, fax 81 740-93-50
e-mail: wydawnictwo@kul.lublin.pl
http://wydawnictwo.kul.lublin.pl

Druk i oprawa
elpil
ul. Artyleryjska 11
08-110 Siedlce
e-mail: info@elpil.com.pl

Spis treści

Wstęp ... 9

Część I. Podstawy teoretyczne baz danych

1. Struktura i modele baz danych ... 13
1.1. Struktura i podstawowe pojęcia baz danych ... 13
1.1.1. Wprowadzenie do baz danych ... 13
1.1.2. Typy baz danych ... 15
1.1.3. Składniki systemu bazy danych ... 17
1.1.4. Architektura SZBD ... 22
1.2. Wprowadzenie do projektowania BD ... 25
1.2.1. Modelowanie koncepcyjne i diagramy związków encji ... 25
1.2.2. Typy połączeń binarnych ... 27
1.3. Modele baz danych ... 31
1.3.1. Hierarchiczny model BD ... 32
1.3.2. Sieciowy model BD ... 36
1.3.3. Wprowadzenie do relacyjnego modelu BD ... 39
1.3.4. Obiektowy model BD ... 46
1.3.5. Obiektowo-relacyjny model BD ... 53

2. Składniki modelu relacyjnego ... 57
2.1. Integralność danych ... 57
2.1.1. Klucze relacyjne ... 57
2.1.2 Definicja i typy logicznego połączenia relacji (tabel) ... 60
2.1.3. Typy uczestnictwa tabel w relacji ... 66
2.1.4. Typy integralności w relacyjnych BD ... 69
2.2. Manipulowanie danymi ... 70
2.2.1. Podstawy algebry relacyjnej ... 70
2.2.2. Operatory relacyjne E. Codda ... 71

3. Koncepcje i etapy projektowania baz danych ... 85
3.1. Planowanie tworzenia i projektowanie baz danych ... 85
3.1.1. Cykl życia BD ... 85
3.1.2. Podstawowe etapy projektowania BD ... 88
3.2. Normalizacja BD ... 96
3.2.1. Cele i podstawowe pojęcia normalizacji ... 96
3.2.2. Zależności funkcyjne w BD ... 100
3.2.3. Proces normalizacji BD ... 104

Część II. Definiowanie oraz manipulowanie danymi

4. Manipulowanie danymi w SQL ... 129
4.1. Typy danych, frazy i polecenia w SQL ... 129
4.1.1. Podstawowe typy danych ... 129
4.1.2. Struktura poleceń w SQL ... 132
4.2. Zdanie SELECT ... 134
4.2.1. Struktura i składnia polecenia ... 134
4.2.2. Operatory SQL ... 139
4.2.3. Funkcje w języku SQL ... 155
4.2.4. Pomocnicze frazy zdania SELECT ... 164
4.3. Inne operacje manipulowania danymi ... 171
4.3.1. Zapytania do kilku tabel ... 171
4.3.2. Podzapytania w SQL ... 173
4.3.3. Złączenie tabel w SQL ... 177
4.4. Modyfikacja danych w SQL ... 183
4.4.1. Zdanie `INSERT` ... 183
4.4.2. Zdanie `UPDATE` ... 185
4.4.3. Zdanie `DELETE` ... 186

5. Definiowanie danych ... 189
5.1. Tworzenie obiektów BD w Database Desktop ... 189
5.1.1. Tworzenie struktury tabel BD ... 189
5.1.2. Kontrola poprawności danych ... 194
5.1.3. Maski poprawności danych ... 196
5.1.4. Odnośnik tabeli ... 198
5.1.5. Definiowanie indeksów (dodatkowych) ... 199
5.1.6. Integralność referencyjna ... 202
5.1.7. Język tabeli ... 204
5.2. Język definiowania danych ... 205
5.2.1. Zdanie CREATE ... 206
5.2.2. Zdanie ALTER ... 219
5.2.3. Zdanie DROP ... 223
5.2.4. Operacje nad BD w IBConsole. ... 224
5.2.5. Typy danych serwera InterBase ... 226

6. Kontrola dostępu do danych ... 233
6.1. Mechanizmy zabezpieczeń ... 233
6.1.1. Podstawowe pojęcia ... 233
6.1.2. Uwierzytelnianie użytkowników ... 236
6.1.3. Role w bazach danych ... 239
6.1.4. Przywileje w bazach danych ... 241
6.2. Perspektywy ... 245
6.2.1. Typy perspektyw ... 245
6.2.2. Tworzenie perspektyw ... 246
6.3. Wyzwalacze ... 250
6.3.1. Podstawowe informacje ... 250
6.3.2. Definiowanie wyzwalacza ... 252

Część III. Organizacja dostępu do danych

7. Aplikacje korzystające z baz danych 259
7.1. Organizacja połączenia z bazami danych w C++ Builder 259
7.1.1. Ogólne sposoby dostępu do baz danych 259
7.1.2. Dostęp do baz danych przez Borland Database Engine 260
7.2. Dostęp do baz danych na podstawie komponentów środowiska Borland C++ Builder 262
7.2.1. Komponenty BDE 262
7.2.2. Klasa TDataSet 264
7.2.3. Klasa TTable 277
7.2.4. Klasa TField 288
7.2.5. Komponent Query 301
7.2.6. Komponent StoredProc 307
7.3. Dostęp do baz danych InterBase 308
7.3.1. Ogólna charakterystyka komponentów InterBase Express 308
7.3.2. Komponent IBDatabase 310
7.3.3. Komponet IBTransaction 313
7.3.4. Komponet IBTable 315
7.3.5. Komponet IBQuery 316
7.3.6. Komponet IBDataSet 317
7.4. Dostęp do BD przez Microsoft ActiveX Data Objects 318
7.4.1. Ogólna charakterystyka ADO 318
7.4.2. Komponenty BDE a komponenty ADO 320
7.5. Dostęp do BD przez komponenty dbExpress 323

8. Przykładowe aplikacje BD 327
8.1. Aplikacja wykorzystująca komponenty InterBase Express 327
8.1.1. Tworzenie bazy danych 327
8.1.2. Tworzenie aplikacji 333
8.1.3. Pierwsze poprawki aplikacji 337
8.1.5. Drugie poprawki aplikacji 342
8.1.6. Oprogramowywanie tabel łączących 352
8.1.7. Tworzenie raportów 360
8.2. Aplikacja bazodanowa wypożyczalni filmów ‘Matrix’ 370
8.2.1. Formatka tytułowa 371
8.2.2. Formatka *Filmy* 371
8.2.3. Formatka *Klienci* 373
8.2.4. Formatka *Pracownicy* 374
8.2.5. Formatka *Wypożyczenia* 374
8.2.6. Formatka *Raporty* 376

Literatura 379

Wstęp

W dzisiejszych czasach trudno wyobrazić sobie dziedzinę działalności człowieka, w której nie istniałaby możliwość wykorzystania systemów informatycznych. Można nawet zaryzykować stwierdzenie, że wiele dziedzin zarówno informatyki, jak i życia codziennego nie rozwijałoby się lub ich rozwój byłby ograniczony, gdyby nie istniały systemy informatyczne, a w szczególności systemy baz danych (BD). W zasadzie każdy użytkownik komputera, a nawet osoba, która z niego nie korzysta, spotyka się dość często z bazami danych. Stanowią one niezbędne narzędzie wspomagające działalność przedsiębiorstw i instytucji, dzięki któremu mogą funkcjonować w sposób efektywny, bardziej wydajny i konkurencyjny wobec innych podmiotów.

Bazy danych są podstawowym elementem współczesnych systemów komputerowych. Dlatego też bez odpowiedniego zapoznania się z podstawami przedmiotu „Bazy danych" nie jest możliwe zostanie nie tylko dobrym programistą, ale i nawet doświadczonym użytkownikiem komputera.

Obecnie zgodnie ze standardami edukacyjnymi przedmiot „Bazy danych" wpisany jest do programów studiów uczelni wyższych jako przedmiot samodzielny (na kierunkach informatycznych) albo jako część przedmiotu podstaw informatyki (na innych kierunkach innych specjalności).

Głównym celem niniejszej książki jest zapoznanie czytelnika z podstawami projektowania i implementacji współczesnych (głównie relacyjnych) systemów bazodanowych. Model relacyjny stał się bardzo popularny z powodu swej prostoty, ale przede wszystkich dzięki zapewnieniu niezależności danych od szczegółów fizycznej implementacji bazy, dzięki czemu aplikacje, jak i użytkownicy z nich korzystający nie muszą się troszczyć o zawiłe detale implementacyjne. Większość nowoczesnych systemów zarządzania bazami opiera się właśnie na modelu relacyjnym sformułowanym w 1969 r. przez E.F. Codda. Znajomość teorii relacji okazuje się przydatna nie tylko twórcom takich systemów. Programiści korzystający z baz danych i administratorzy takich baz również powinni posiadać tego typu wiedzę, aby w pełni wykorzystać możliwości, jakie oferuje im model relacyjny.

Książka jest opracowana na podstawie treści wykładów i ćwiczeń z baz danych przeprowadzonych przez autorów na Katolickim Uniwersytecie Lubelskim Jana Pawła II. Jednym z podstawowych celów autorów było przedstawienie materiałów teoretycznych oraz praktycznych w takiej formie i w takiej kolejności, aby umożliwić czytelnikom samodzielne zaprojektowanie efektywnej bazy danych, a także zaimplementowanie jej za pomocą nowoczesnych systemów zarządzania bazami danych (SZBD) oraz nowoczesnych narzędzi programistycznych. Odpowiednia kolejność pojawiających się zagadnień jest niezbędna, gdyż, jak wiadomo, baza obsługiwana nawet przez najbardziej wydajny system zarządzania nie będzie spełniać swoich funkcji, jeśli nie będzie zaprojektowana w sposób zgodny z teoretycznymi podstawami.

Poniżej wymienione zostały przedmioty, których podstawy teoretyczne zostały wykorzystane w niniejszej książce:

- struktury danych;
- algorytmy i podstawy matematyki dyskretnej;

- badania operacyjne;
- systemy i języki programowania;
- projektowanie systemów informatycznych.

Książka wymaga od czytelników podstawowej wiedzy w zakresie takich dziedzin, jak: teoria mnogości, algebra, logika, znajomość podstawowych struktur danych oraz koncepcji programowania obiektowego i strukturalnego.

Struktura podręcznika składa się z trzech części:

- Część 1. Podstawy teoretyczne komputerowych baz danych.
- Część 2. Definiowanie oraz manipulowanie danymi.
- Część 3. Organizacja dostępu do danych.

Każda z wymienionych części składa się z kilku rozdziałów, posiadających dość dużą liczbę przykładów, jak również pytania do samodzielnej kontroli.

Pierwsza część obejmuje trzy rozdziały, w których zostały przeanalizowane podstawowe pojęcia, struktury i modele baz danych, podstawy modelu relacyjnego, zmienne relacyjne i predykaty, podstawy algebry relacyjnej, a także typy integralności danych.

W części tej (rozdział trzeci) zostały przeanalizowane nowoczesne metody projektowania baz danych, opierające się na koncepcji cyklu życia bazy danych. Wiele miejsca i uwagi autorzy poświęcili analizie kolejnych etapów projektowania baz danych. Bardzo ważną częścią tego rozdziału jest też analiza zależności funkcyjnych i związanej z nimi procedury normalizacji tabel w relacyjnych bazach danych. Oprócz przedstawienia teoretycznych podstaw normalizacji autorzy chcieli przybliżyć czytelnikowi praktyczne techniki przekształcania schematu bazy do stanu znormalizowanego.

W części drugiej (rozdziały czwarty, piąty i szósty) szczegółowo opisane zostały ważniejsze aspekty języka SQL (*Struktured Query Language*), mianowicie: semantyka i składnia podstawowych poleceń SQL, przeznaczonych do manipulowania danymi (*Data Manipulation Language*) oraz pozwalających na definiowanie danych (*Data Definition Language*). Zwrócono uwagę czytelnika na istniejące różnice w strukturze tych samych zdań języka, napisanych w różnych dialektach (działających na podstawie różnych SZBD). Zostały także przedstawione techniki i możliwości tworzenia oraz modyfikacji obiektów baz danych w środowisku C++ Builder. W tej części również zostały omówione ważniejsze aspekty bezpieczeństwa danych i metody ochrony przed nieautoryzowanym dostępem.

Ostatnia część obejmuje rozdział siódmy i ósmy, w których zostały opisane mechanizmy dostępu do baz danych oraz technologie tworzenia aplikacji. Wykorzystano proste i zarazem efektywne, a także często używane w praktyce platformy *Interbase* i *FireBird*. Wszystkie etapy tworzenia baz danych zostały zilustrowane konkretnymi przykładami.

Ważniejsze pojęcia zostały przedstawione w formie definicji. Natomiast ważne wnioski i uwagi przedstawiono w ramkach.

Materiał książki obejmuje około 40-60 godzin wykładów oraz tyle samo godzin zajęć laboratoryjnych (ćwiczeń).

Książka będzie pomocna również dla studiów podyplomowych kierunków ekonomiczno-finansowych bądź biznesowych, posiadających wstępną wiedzę o roli i zastosowaniu baz danych w wymienionych dziedzinach.

Część I
Podstawy teoretyczne baz danych

1. Struktura i modele baz danych

1.1. Struktura i podstawowe pojęcia baz danych

1.1.1. Wprowadzenie do baz danych

Nowoczesne technologie przechowywania i wykorzystywania informacji są jednymi z najważniejszych elementów, pozwalających osiągnąć sukces w prawie każdej dziedzinie biznesu. Możliwe jest to dzięki wykorzystaniu baz danych, pozwalających na: przechowywanie informacji korporacyjnych, przedstawianie danych użytkownikom i klientom oraz analizę danych. Oprócz tego bazy danych (BD) są ważnymi obiektami wielu projektów i badań naukowych. Według niektórych źródeł komputerowe bazy danych przechowują ponad 85% wszystkich informacji.

Pojęcie *baza danych* (ang. *DataBase*, DB) pojawiło się w latach 1962-1963 jako odpowiedź na wymagania użytkowników komputerów, związane ze sposobem przechowywania danych w pamięciach zewnętrznych komputera, jak i sposobem dostępu do tych danych. Warto tutaj podkreślić, że do tego czasu (od pojawienia się pierwszego komputera – ENIAC, Electronic Numerical Integrator And Calculator – pod koniec lat 40.) narzędzia elektroniczne były wykorzystywane wyłącznie do wykonywania obliczeń i sterowania obiektami (procesami).

Jeszcze 20 lat temu badania nad bazami danych miały głównie charakter naukowy. Dzisiaj na badaniach BD opiera się cały przemysł usług informacyjnych z corocznym budżetem rzędu kilkudziesięciu miliardów dolarów. Historia badań systemów BD jest w rzeczywistości historią rozwoju aplikacji, charakteryzujących się obecnie bardzo wysokim poziomem wydajności i znacznym wpływem na ekonomię.

Definicja 1-1

Terminem ***baza danych*** **(BD)** określamy zbiór powiązanych ze sobą danych o określonej strukturze, zapisany na zewnętrznym nośniku pamięci komputera, który może zaspokoić wymagania wielu użytkowników, korzystających z niego w sposób selektywny w dogodnym dla siebie czasie. Innymi słowy, BD umożliwia centralizację, koordynację i integrację informacji oraz przesyłanie ich wielu użytkownikom.

Najbardziej istotnymi elementami tej definicji są:

- **zbiór** – BD jest zbiorem danych, przy czym najczęściej jest to zbiór bardzo duży, który zawiera informacje ważne oraz użyteczne dla użytkowników BD;
- **struktura** – BD jest zbiorem, który musi być zorganizowany w jakąś wewnętrzną strukturę. Jest ona ściśle zależna od przyjętego modelu bazy danych (o czym będzie mowa w dalszej części tego rozdziału);
- **związki** – struktura danych musi zapewniać reprezentację związków między danymi oraz ich elementami, stąd bazę stanowią logicznie powiązane ze sobą dane;
- **użyteczność** – BD musi zawierać informacje użyteczne, które znajdują zastosowanie w praktyce, przy czym dane te mogą być wykorzystywane przez wielu użytkowni-

ków, zaspokajając ich wymagania dotyczące różnych informacji niezbędnych do funkcjonowania instytucji.

Warto podkreślić, że idee nowoczesnych technologii informatycznych opierają się na koncepcji danych, które organizowane są w struktury bazy danych w celu adekwatnego odwzorowania świata rzeczywistego, realizując jednocześnie wymagania użytkowników.

Definicja 1-2

Dane to informacja przedstawiona w formie umożliwiającej jej automatyczne pobieranie, przechowywanie i dalsze opracowanie przez człowieka (użytkownika) lub przez narzędzia informatyczne.

Definicja 1-3

Pod pojęciem ***informacja*** w ogólnym znaczeniu rozumiemy dowolne wiadomości, dotyczące jakiegoś zdarzenia, procesu, obiektu.

W BD powinny być przechowywane dane logicznie ze sobą powiązane. Aby umożliwić powiązanie tych danych w sposób odpowiadający rzeczywistości, korzystamy z takich pojęć, jak: *encje* lub *obiekty*.

Definicja 1-4

Encjami (ang. *entity*) lub ***obiektami*** (ang. *object*) są wszystkie elementy, o których chcemy przechowywać informacje w BD. Obiekt (*encja*) jest przedmiotem (materialnym lub abstrakcyjnym), który może być wyróżniony i określony w świecie rzeczywistym, a także o którym chcemy przechowywać informacje.

Przykład 1-1

Przykładami *encji* mogą być *Pracownicy, Studenci, Wydziały, Katedry, Grupy_Studentów, Zajęcia, Sale* i in. w BD *Uczelni.* Encje określane są przez cechy lub właściwości nazywane *atrybutami.*

Zwróćmy uwagę na to, że nazwy encji powinny być pisane jako jedno słowo. Jest to zalecenie techniczne wymagane na etapie implementacji.

Definicja 1-5

Atrybutem (ang. *attribute*) obiektu nazywamy cechy, za pomocą których określamy obiekty. Z drugiej strony możemy powiedzieć, że atrybuty to informacje o obiektach, które chcemy przechowywać w BD. Atrybut jest bytem konceptualnym (pojęciowym).

Przykładowo: obiekt *Pracownik* (Prz. 1-1) określany jest przez *Identyfikator_Pracownika*, *Nazwisko_Pracownika*, *Imię_Pracownika* itd. Oznacza to, że *Identyfikator_Pracownika*, *Nazwisko_Pracownika*, *Imię_Pracownika* są atrybutami obiektu *Pracownik.*

Wymienione informacje o obiektach mają określone wartości. W*artości* te są *danymi*, które przechowuje BD.

Schemat powiązań pomiędzy *obiektami*, *atrybutami* i *wartościami* atrybutów przedstawiony jest na rys. 1.1.

Z rys. 1.1 widać, że BD przechowuje m.in. następujące informacje o pracowniku: *Id_Pracownika*=3, *Nazwisko_Pracownika*='Lipska', *Imię_Pracownika*='Ewa'. Element, który stoi przed znakiem równości, jest atrybutem (zmienną), po znaku równości – wartością tego atrybutu.

Wartości tekstowe wpisujemy pomiędzy znakami apostrofu lub cudzysłowu.

Jak już podkreśliliśmy, wszystkie encje powinny być ze sobą powiązane. Połączenia te są częścią danych i powinny być przechowywane w BD.

Definicja 1-6
Związkiem nazywamy połączenie dwóch lub większej liczby obiektów.

W Prz. 1-1 widzimy, że *Studenci* powiązani są z *Grupami_Studentów*, z *Wydziałami* i *Zajęciami*, *Pracownicy* – z *Wydziałami* itd.

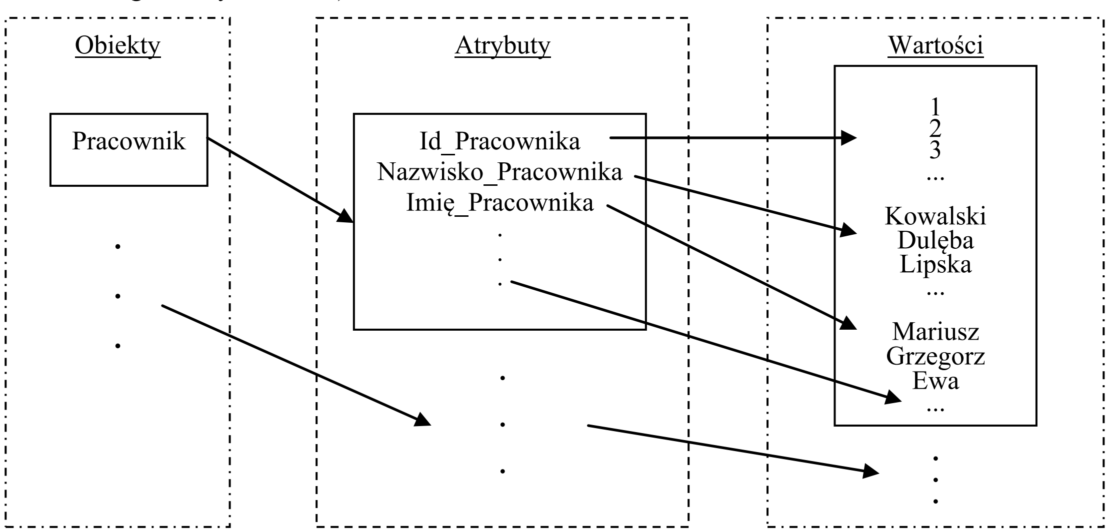
Komputerowa BD składa się z dwóch podstawowych części: samych danych oraz narzędzi, pozwalających na manipulowanie danymi (czyli centralizację, koordynację, integrację oraz przesyłanie). Takim narzędziem jest *system zarządzania bazą danych,* SZBD (ang. *DataBase Management System,* DBMS).

Definicja 1-7
SZBD to najważniejszy programistyczny element systemu bazodanowego (innymi komponentami są menedżer transakcji, pakiety użytkowe i in.). SZBD jest niejako pośrednikiem pomiędzy danymi a użytkownikami danych, gdyż w rzeczywistości jest to oprogramowanie łączące aplikację użytkownika z bazą.

1.1.2. Typy baz danych

Z punktu widzenia architektury BD można podzielić na następujące grupy:

- autonomiczne – baza i SZBD są rozmieszczone na jednym komputerze. Z punktu widzenia ochrony danych jest to najbezpieczniejsza klasa BD. Użytkownikiem może być jedna lub kilka osób, ale jednocześnie dostęp do danych może mieć tylko jeden użytkownik. Dlatego takie systemy mają nazwę jednoużytkownikowych (ang. *Singleuser*);
- BD w architekturze *klient/serwer* (ang. *client/server*, C/S). Można je podzielić na:
  - systemy *wieloużytkownikowe* (ang. *multiusers*) z rozmieszczeniem danych na serwerze (patrz rys. 1.2a); w takich sytuacjach połączenie użytkownika (klienta C) z serwerem (BD) najczęściej realizuje się za pomocą sieci (lokalnych LAN lub globalnych WAN)[1];



Rys. 1.1. Schemat powiązań elementów bazy danych

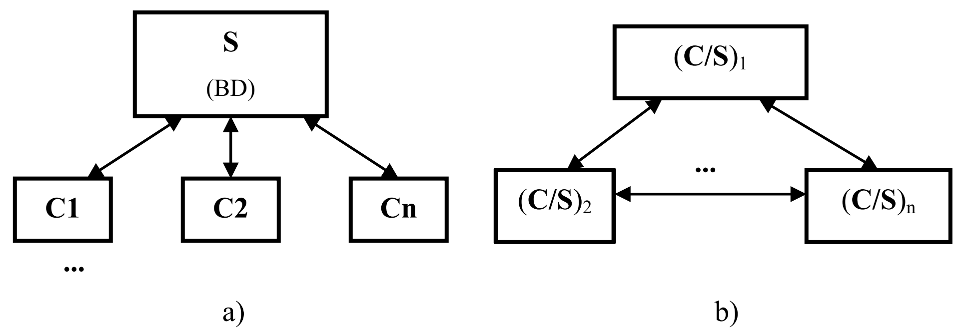
[1] Tutaj nie wyróżniamy sieci korporacyjnych, miejskich i innych.

– systemy rozproszone charakteryzują się tym, że dane częściowo rozmieszczone są na wszystkich komputerach (lub na większości) w ten sposób, że jeżeli użytkownik (1) pobiera dane z innego komputera (2), to użytkownik (bądź jego komputer 1) występuje jako klient, a komputer po drugiej stronie (2) jako serwer (rys. 1.2b). Taki typ BD określamy terminem *rozproszone BD* (ang. *distributed DB*) lub *korporacyjne BD*. Połączenie pomiędzy serwerem S i klientem C realizuje się również za pomocą sieci.

Technologię klient/serwer wspiera większość nowoczesnych SZBD, np.: Informix, Ingres, Sybase, Oracle, MS SQL *Server*. Jednym z ważniejszych aspektów tej technologii jest mechanizm *procedur*. Pozwala on na tworzenie podprogramów, które mogą być wywoływane przez aplikację, co daje możliwość zmniejszenia ruchu sieciowego pomiędzy serwerem a klientem.

Z punktu widzenia przeznaczenia BD można podzielić na następujące grupy:

- *analityczne* – służą do przechowywania danych historycznych i informacji związanych z pewnymi wydarzeniami.



Rys. 1.2. Struktura systemu BD oparta na architekturze klient/serwer: a – scentralizowana BD, b – rozproszona BD

Sięganie do tego typu danych najczęściej następuje wtedy, kiedy chcemy przeanalizować jakieś zjawisko na przestrzeni danego okresu, np. chcemy zbadać tendencje rynkowe, poczynić prognozy na przyszłość. Baza analityczna przechowuje dane statyczne, a więc takie, które nie ulegają zmianom w ogóle albo ulegają zmianom bardzo rzadko. Dane te odzwierciedlają stan obiektu z pewnego ustalonego momentu, a nie stan obecny, jak to ma miejsce w przypadku baz operacyjnych. Przykładem takich baz są np. bazy testów chemicznych czy próbek geologicznych.

W klasie analitycznych BD dużą rolę pełnią hurtownie danych – korporacyjne bazy firm, które produkują bądź sprzedają swoje produkty oraz mogą mieć wiele oddziałów (nawet w wielu różnych państwach). Hurtownie danych pomagają im rozwiązywać marketingowe problemy;

- *operacyjne* – znajdują głównie zastosowanie we wszelkiego rodzaju firmach, organizacjach, instytucjach, gdzie istnieje potrzeba gromadzenia, przechowywania oraz modyfikacji danych. Cechą charakterystyczną ich jest to, że zawierają dane dynamiczne, a więc takie, które ciągle ulegają zmianom i odzwierciedlają stan jakiegoś obiektu. Przykładami takich baz mogą być np. bazy obsługi dostaw, wypożyczeń kaset wideo czy rezerwacji biletów. Zmiany w bazie tego typu są bardzo częste, dane ulegają modyfikacjom dynamicznie.

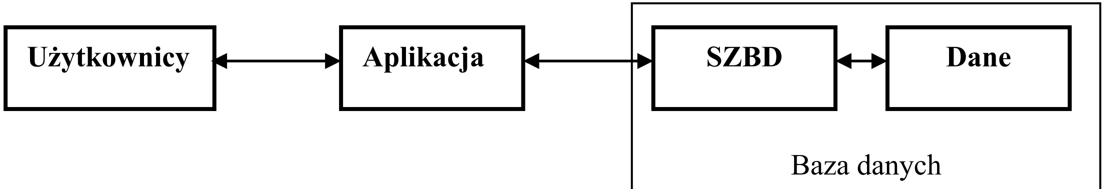
1.1.3. Składniki systemu bazy danych

Schematycznie cały system bazodanowy (bez kanałów komunikacyjnych), który często określa się terminem *środowisko SZBD*[2], można przedstawić w formie zaprezentowanej na rys. 1.3.

Użytkownicy. Wszystkich użytkowników tradycyjnie dzieli się na trzy kategorie:

- *zwykły (końcowy, prosty) użytkownik* – jest osobą, dla której projektuje się BD. Często takich użytkowników nazywa się klientami, wśród których zazwyczaj są osoby mało doświadczone. Dla takich użytkowników tworzy się specjalne aplikacje i narzędzia użytkowe, umożliwiające w znacznym stopniu uproszczenie wykonywanych przez nich zadań;
- *programiści i twórcy BD* – projektują BD na podstawie celów i przeznaczenia BD, biorąc pod uwagę logiczne połączenia między obiektami, jak również funkcjonalne możliwości wybranego SZBD; za pomocą języków programowania (C++, Java i in.) tworzą kody aplikacji i narzędzi użytkowych (wspomnianych powyżej);
- *administrator bazy danych*, ABD (ang. *DataBase Administrator*, DBA) – specjalna osoba, która jest odpowiedzialna za poprawność funkcjonowania SZBD, integralność i ochronę informacji przechowywanych w bazie danych oraz za współdziałanie z innymi użytkownikami.

Aplikacja. O jej przeznaczeniu mówiliśmy już wcześniej. Technologia tworzenia i zastosowania aplikacji będzie opisana w trzeciej części książki.



Rys. 1.3. Schematyczne połączenie elementów systemu bazy danych

System zarządzania bazą danych (**SZBD**, patrz: Def. 1-7). Przede wszystkim należy pamiętać, że SZBD jest narzędziem programistycznym. W procesie realizacji swoich funkcji SZBD współdziała z BD, a także z innym oprogramowaniem wykorzystywanym przez użytkownika.

System zarządzania bazą danych można traktować jako pośrednika pomiędzy bazą danych a jej użytkownikami. Wszystkie operacje na danych użytkownik realizuje za pomocą specjalnego *procesora języka zapytań,* który umożliwia użytkownikom wprowadzanie zapytań do BD w trybie konsolowym. Najbardziej znanym językiem jest SQL (ang. *Structured Query Language*), strukturalny język zapytań. Właśnie za pomocą poleceń (zapytań, kwerend) tego języka użytkownicy realizują podstawowe operacje na danych:

- wstawianie (*insert*)[3];
- modyfikację (*update*);

[2] W literaturze pojęcia 'baza danych' i 'system zarządzania bazą danych' często nie są rozróżniane; można spotkać u tego samego autora frazy '... BD Oracle' i '… SZBD Oracle' lub '…BD MS Access' i '…SZBD MS Access'.

[3] W nawiasach podane są odpowiednie zdania języka SQL.

- usuwanie (*delete*);
- pobieranie (*select*).

Z tego powodu język SQL często nazywamy językiem manipulowania danymi, JMD (ang. *Data Manipulation Language,* DML*)*.

Drugą funkcją SZBD jest definiowanie baz danych, głównie za pomocą *języka definicji danych* (ang. *Data Definition Language,* DDL)[4]. Dzięki niemu możliwe staje się określenie typów, struktury i warunków poprawności danych przechowywanych w bazie.

W najbardziej ogólnym znaczeniu funkcje SZBD można podzielić na cztery kategorie:

- sterowanie procesami przechowywania danych (w pamięci komputera lub na nośnikach zewnętrznych);
- przetwarzanie zapytań użytkowników;
- sterowanie transakcjami;
- ochrona danych (chociaż każda z poprzednich funkcji może być związana z ochroną danych).

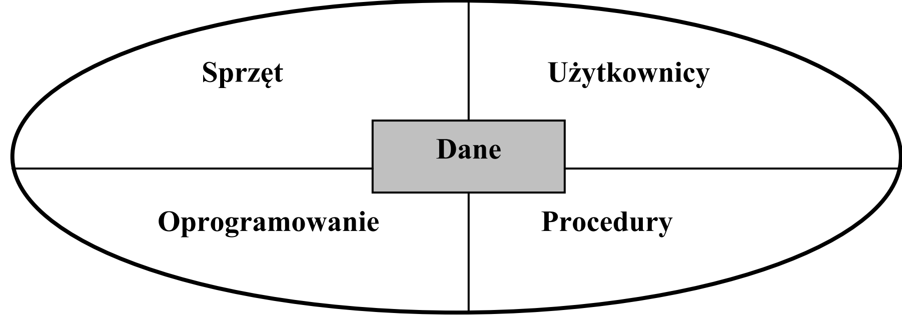
Ważnymi komponentami wspomnianego *środowiska SZBD* są *procedury*. Najczęściej są to operacje umożliwiające: logowanie do SZBD, uruchamianie i zamknięcie SZBD, tworzenie kopii bazy danych, obsługę awarii sprzętu bądź oprogramowania, zmianę struktury tabel, przeniesienie bazy danych na wiele dysków itp.

Sprzęt *(*ang. *hardware)* jest niezbędnym elementem każdego SZBD. Sprzętem może być jeden, jak i wiele komputerów połączonych w sieć. Dobór sprzętu jest uzależniony od wymagań i możliwości konkretnej instytucji, a także od wybranego SZBD.

Występuje wiele różnych SZBD. Jedne mogą być uruchamiane tylko na konkretnym sprzęcie i pod konkretnym systemem operacyjnym, inne zaś charakteryzują się dość dużą elastycznością. Mimo iż większość SZBD może być poprawnie uruchomiona z minimalną (podawaną w wymaganiach) ilością pamięci, to dla optymalnego działania systemu należy rozważyć zaopatrzenie się w dodatkową jej ilość.

Oprogramowanie (ang. *software*) obejmuje zarówno program SZBD, programy narzędziowe, aplikacje korzystające z bazy, system operacyjny, jak i oprogramowanie sieciowe. Aplikacje są najczęściej pisane w językach programowania trzeciej generacji (3GL), do których należą m.in. C, C++, *Java*, *Visual Basic*, COBOL czy *Pascal*, lub czwartej generacji (4GL), jak np. osadzony SQL.

W omawianym środowisku SZBD najważniejszym składnikiem są oczywiście dane, i to one stanowią pomost pomiędzy poszczególnymi komponentami środowiska. Schematycznie przedstawione zostało to na rys. 1.4.



Rys. 1.4. Komponenty środowiska SZBD

[4] Dalej przeanalizujemy zastosowania SQL w celu definicji danych (zdania *create*).

Baza danych oprócz samych danych zawiera również ich opis nazywany *katalogiem systemowym* (także *metadanymi* lub *słownikiem danych*). Taki opis struktury danych zapewnia wzajemną niezależność aplikacji obsługujących bazę danych od samej bazy. Niezależność ta polega na tym, że dane mają swoją definicję (strukturę wewnętrzną), która niekoniecznie musi być znana użytkownikowi, a także definicję zewnętrzną, a więc taką, którą widzi użytkownik korzystający z bazy danych, a ściślej aplikacje, z których on korzysta.

Oczywiście, każdy SZBD musi udostępniać użytkownikowi katalog systemowy. W takim katalogu można znaleźć informacje, takie jak: nazwy związków, nazwy i typy elementów danych, więzy integralności czy nazwy użytkowników, którzy mają dostęp do bazy. Dzięki zastosowaniu takiego katalogu możliwe jest centralne przechowywanie informacji o danych. Poza tym opis znaczenia danych zawarty w katalogu pozwala użytkownikom na lepsze zrozumienie ich przeznaczenia.

Główną zaletą takiej metody, zwanej *abstrakcją danych*, jest to, że można zmienić strukturę wewnętrzną danych bez konieczności zmiany aplikacji z niej korzystającej, o ile nie ulegnie zmianie definicja zewnętrzna. Taka zmiana może polegać np. na dodaniu nowego pola do rekordu lub dodaniu do bazy danych nowego pliku. Po takiej zmianie działanie aplikacji nie będzie zakłócone, a więc nie trzeba jej będzie zmieniać w celu przystosowania do nowej struktury bazy.

Ponieważ wciąż pojawiają się nowe wymagania użytkowników baz danych, przed SZBD stawiane są coraz to nowe zadania. Dlatego też ulegają one ciągłemu rozwojowi, co wpływa na pojawienie się wielu zalet, ale także i wad. Do podstawowych zalet SZBD należą:

- **kontrola redundancji danych** – w BD stosuje się wiele procedur, pozwalających wyeliminować powtarzanie się tych samych danych, chociaż całkowicie nie zapobiegają one temu zjawisku. Czasem staje się konieczne powtórzenie pewnego zbioru danych, chociażby po to, aby zachować związek. Dlatego ważne jest, by kontrolować rozmiar *redundancji* (nadmiarowości) i ograniczać ją do sytuacji, w których jest ona niezbędna;
- **więcej informacji na podstawie tych samych danych** – dzięki logicznemu powiązaniu między sobą różnych danych możemy wyciągnąć z nich dodatkowe wnioski, w przeciwieństwie do sytuacji, w której takiego powiązania by nie było;
- **wspólny dostęp do danych** – baza danych jest własnością całej instytucji (patrz: schematy na rys. 1.2), stąd z informacji zgromadzonych w bazie mogą korzystać wszystkie uprawnione osoby. W ten sposób do danych ma dostęp większa liczba użytkowników. Poza tym aplikacje mogą dodawać do bazy te dane, których w niej jeszcze nie ma, natomiast nowe aplikacje mogą korzystać z danych i mechanizmów już dostępnych w SZBD bez potrzeby realizowania ich we własnym zakresie;
- **poprawa integralności danych**[5] – administrator bazy danych może zdefiniować więzy integralności, czyli reguły spójności (poprawności i niesprzeczności), które nie mogą być w niej naruszone, a SZBD będzie wymuszał ich przestrzeganie. Takie więzy mogą dotyczyć zarówno pojedynczego rekordu, jak i związku pomiędzy rekordami[6];
- **poprawa bezpieczeństwa** – administrator BD może i powinien zdefiniować mechanizmy bezpieczeństwa, które będą realizowane przez SZBD[7];

---

[5] Integralność jest jedną z najważniejszych cech BD, co będzie dokładniej omówione w dalszej części książki.

[6] Chodzi o relacyjne BD. Zob. pkt 1.3.3. Wprowadzenie do relacyjnego modelu BD.

[7] Problem bezpieczeństwa BD odnosi się przede wszystkim do uniemożliwienia dostępu do bazy osobom nieuprawnionym. Dlatego jednym z takich mechanizmów może być stosowanie identyfikatorów i haseł użytkow-

- **przestrzeganie standardów** – administrator BD może zdefiniować standardy dotyczące np. formatu zapisu danych, zasad nazewnictwa, dokumentacji, aktualizacji i wymusić ich przestrzeganie. Standardy te mogą wynikać m.in. z wewnętrznych przepisów danej firmy oraz z regulacji ogólnopaństwowych albo międzynarodowych;
- **oszczędność** – dzięki połączeniu danych operacyjnych całej instytucji w jedną bazę oraz stworzeniu zestawu aplikacji ją obsługujących nie ma potrzeby tworzenia odrębnych systemów dla poszczególnych działów, jak to miało miejsce we wcześniejszych systemach przetwarzania plików. Zaoszczędzone w ten sposób środki można przeznaczyć na udoskonalenie istniejącego systemu, np. zakup komputera dużej mocy;
- **godzenie sprzecznych wymagań** – każdy użytkownik BD może mieć swoje potrzeby, które jednak mogą kolidować z potrzebami innych. Administrator BD powinien pogodzić ze sobą te wymagania poprzez wybranie rozwiązania optymalnego i najlepszego z punktu widzenia całej instytucji. Dzięki temu zasoby mogą być wykorzystywane optymalnie, choć niektóre rozwiązania mogą kolidować z chęciami i oczekiwaniami poszczególnych użytkowników;
- **poprawa zakresu i czasu dostępu do danych** – dzięki integracji bazy BD możliwy jest bezpośredni dostęp do danych dotyczących działalności poszczególnych działów instytucji. Dzięki temu osiągnięta jest większa funkcjonalność systemu, który może sprawniej obsługiwać użytkowników i klientów instytucji. Języki zapytań (np. SQL) pozwalają użytkownikom szybciej pobierać dane bez konieczności zatrudniania programisty w celu pisania programu wyszukującego żądane informacje;
- **wzrost wydajności** – każdy SZBD dostarcza wiele procedur służących do obsługi plików bazodanowych. Dzięki temu programista może skierować cały swój wysiłek na tworzenie funkcjonalnego oprogramowania, spełniającego wymagania użytkowników, bez potrzeby martwienia się o szczegóły implementacyjne na niskim poziomie. Wiele SZBD zawiera również gotowe narzędzia, ułatwiające i przyspieszające tworzenie aplikacji bazodanowych. To wpływa w dużej mierze na zwiększenie wydajności i efektywności pracy programistów;
- **łatwość utrzymania wynikająca z niezależności danych** – dzięki niezależności danych, rozumianej jako niezależność opisu danych od aplikacji, aplikacje są uniezależnione od zmian w opisie danych, co pozwala w łatwiejszy sposób je modernizować;
- **zwiększenie możliwości wielodostępu** – większość SZBD dba o integralność danych, w przypadku gdy są one modyfikowane przez wielu użytkowników, co pozwala zapobiec ich utracie lub naruszeniu spójności;
- **rozszerzona obsługa składowania i odtwarzania BD po awarii** – SZBD zawiera mechanizmy, które minimalizują utratę efektów pracy użytkowników bazodanowych w wyniku wystąpienia awarii sprzętu bądź oprogramowania. Mechanizmy te pozwalają zapamiętywać stan bazy, a także odtworzyć go tak szybko, jak to jest tylko możliwe po zaistniałej awarii.

Systemy oparte na bazach danych zawierają również pewne wady. Najważniejsze z nich to:

- **złożoność** – ponieważ od SZBD oczekuje się coraz to nowych funkcji, stają się one coraz bardziej złożone, rozbudowane, a przez to i skomplikowane. Aby w pełni wykorzystać system, jego użytkownicy muszą go doskonale znać, gdyż nieznajomość

ników w celu weryfikacji, czy dany użytkownik może korzystać z bazy i w jakim zakresie (np. czy może tylko pobierać dane, czy również je modyfikować).

może doprowadzić do popełnienia poważnych błędów, szczególnie w fazie projektowania, co może później mieć poważne konsekwencje dla całej instytucji;

- **rozmiar** – SZBD ze względu na swoją złożoność są najczęściej dużymi programami, stąd wymagają dużo miejsca w pamięci dyskowej i operacyjnej komputera, aby mogły działać na takim poziomie funkcjonalności, do jakiego zostały stworzone;
- **koszt SZBD** – systemy wielodostępowe obsługujące tysiące użytkowników są bardzo drogie, do tego dochodzą jeszcze stałe wydatki związane z konserwacją takiego systemu;
- **dodatkowe koszty sprzętu** – ponieważ zapotrzebowanie SZBD na pamięć wzrasta wraz z rozmiarem danych, może stać się konieczne dokupienie większej liczby dysków. Poza tym, aby zwiększyć efektywność działania systemu, należy rozważyć zakup mocniejszej maszyny, a czasem i maszyny specjalnie przeznaczonej dla SZBD. To oczywiście wiąże się z koniecznością poniesienia większych kosztów;
- **koszty przeniesienia** – jeżeli instytucja chce przenieść obecne aplikacje pod nowy SZBD i nowy sprzęt, musi ponieść duże koszty związane przede wszystkim z przeszkoleniem pracowników i zatrudnieniem specjalistów, których zadaniem będzie przeniesienie i uruchomienie nowego systemu. Dlatego wielu firmom trudno jest przejść na nowsze i wydajniejsze systemy;
- **sprawność** – SZBD jest tworzony do celów bardzo ogólnych, do obsługi wielu bardzo różnych aplikacji, dlatego też niektóre z nich mogą działać wolniej niż systemy tworzone do obsługi jednej konkretnej aplikacji;
- **większy zasięg awarii** – ponieważ w SZBD mamy do czynienia z centralizacją zasobów danych, awaria może spowodować zatrzymanie pracy wszystkich aplikacji i użytkowników uzależnionych od SZBD.

Systemy zarządzania bazą danych są niezwykle skomplikowanymi systemami komputerowymi. Ich złożoność wynika głównie z tego, że mają do wykonania wiele zadań, których liczba wciąż rośnie. Jednak do funkcji, które musi pełnić każdy z SZBD, można zaliczyć:

- **zapis, odczyt i aktualizację danych** – jak podkreśliliśmy wyżej, każdy SZBD powinien umożliwiać te operacje użytkownikowi, ale w taki sposób, aby ukryć przed nim szczegóły implementacji wewnętrznej (np. struktury plików);
- **obsługę transakcji**[8] – SZBD czuwa nad tym, aby w danej transakcji wykonały się wszystkie aktualizacje stanu bazy albo nie wykonała się żadna;
- **sterowanie współbieżnością** – współbieżność polega na jednoczesnym dostępie do danych wielu użytkowników (rys. 1.1). Gdy grupa użytkowników tylko odczytuje dane z bazy, obsługa współbieżności jest łatwa, operacje odczytu bowiem nie zmieniają stanu bazy, stąd nie kolidują ze sobą. Jednak gdy kilku użytkowników jednocześnie próbuje aktualizować dane, może dojść do sprzeczności i przekłamania. SZBD musi więc dostarczyć mechanizmy przeciwdziałające takim sytuacjom;
- **obsługę odtwarzania danych** – SZBD musi dostarczać mechanizmy odtworzenia danych, w przypadku gdy zostaną one z jakichś powodów uszkodzone lub utracone.

[8] Pod pojęciem transakcji kryje się ciąg operacji wykonywanych przez użytkownika lub aplikację, który aktualizuje bazę lub używa informacji w niej zawartych; innymi słowy, wynikiem transakcji jest zapis lub odczyt danych. Dzięki zastosowaniu transakcji możliwe jest utrzymanie niesprzecznego stanu bazy danych poprzez powrót do stanu poprzedniego w razie jakiejkolwiek awarii.

Takimi powodami mogą być awarie sprzętu, uszkodzenie nośnika czy awaria oprogramowania;

- **obsługę autoryzacji**[9] – SZBD musi czuwać nad tym, aby do danych mieli dostęp tylko uprawnieni użytkownicy, jak również nad tym, aby ci użytkownicy mogli korzystać tylko z tych danych, które mogą być im udostępnione. Na przykład, kierownictwo instytucji może mieć dostęp do wszystkich danych, a pracownicy tylko do danych niezbędnych dla ich efektywnej pracy;
- **obsługę transmisji danych** – ponieważ użytkownicy bazodanowi najczęściej nie pracują bezpośrednio na komputerze obsługującym SZBD, ale na własnych stacjach roboczych (klienckich), które łączą się z bazą poprzez sieć, każdy liczący się SZBD musi współpracować z programami zarządzającymi komunikacją, tzw. DCM *(ang. Data Communication Manager).* SZBD otrzymuje dzięki DCM żądania w postaci przesyłanych komunikatów, na które może odpowiedzieć również w taki sposób;
- **obsługę integralności danych** – integralność bazy danych jest rozumiana jako niesprzeczność, spójność i poprawność danych, które w niej występują. Integralność jest określana zazwyczaj za pomocą więzów, czyli nienaruszalnych warunków nałożonych na dane. SZBD musi zapewniać mechanizmy, które czuwają nad tym, aby zarówno dane, jak i operacje na nich dokonywane spełniały wcześniej określone warunki;
- **usługi wspierające niezależność danych** – SZBD musi zapewniać niezależność programów od rzeczywistej struktury bazy zarówno na poziomie fizycznym, jak i logicznym. Fizyczna niezależność jest dość łatwa do osiągnięcia, natomiast niezależność logiczna może przynieść wiele problemów. Może być np. możliwe dodanie jakiegoś atrybutu bez konieczności zmiany programu korzystającego z bazy, ale usunięcie atrybutu może już uniemożliwić jego działanie;
- **programy narzędziowe** – SZBD musi zawierać pewien zestaw programów narzędziowych wspomagających pracę administratora bazy danych. Do takich programów można zaliczyć m.in.: programy importujące zawartość bazy z plików czy eksportujące dane do nich, programy diagnostyczne, pozwalające kontrolować działanie i obciążenie bazy, programy odśmiecające, które zwalniają pamięć po usuniętych rekordach i wiele innych.

Pojawienie się *systemów zarządzania bazą danych* wpłynęło w znacznym stopniu na poprawę efektywności działania wielu instytucji, a systemy te stanowią obecnie podstawę wielu systemów informatycznych. Cały czas prowadzone są bardzo intensywne badania, mające na celu zwiększenie ich efektywności oraz możliwości. Niestety, istnieje cały czas wiele problemów, które nie doczekały się satysfakcjonującego rozwiązania.

1.1.4. Architektura SZBD

Mówiąc o tym, czym powinien być tak skomplikowany produkt programistyczny jak SZBD, należy przede wszystkim wyjaśnić koncepcję działania tego systemu. Podstawami tej koncepcji są:

- architektura SZBD powinna umożliwiać rozgraniczenie poziomów użytkownika i systemu;

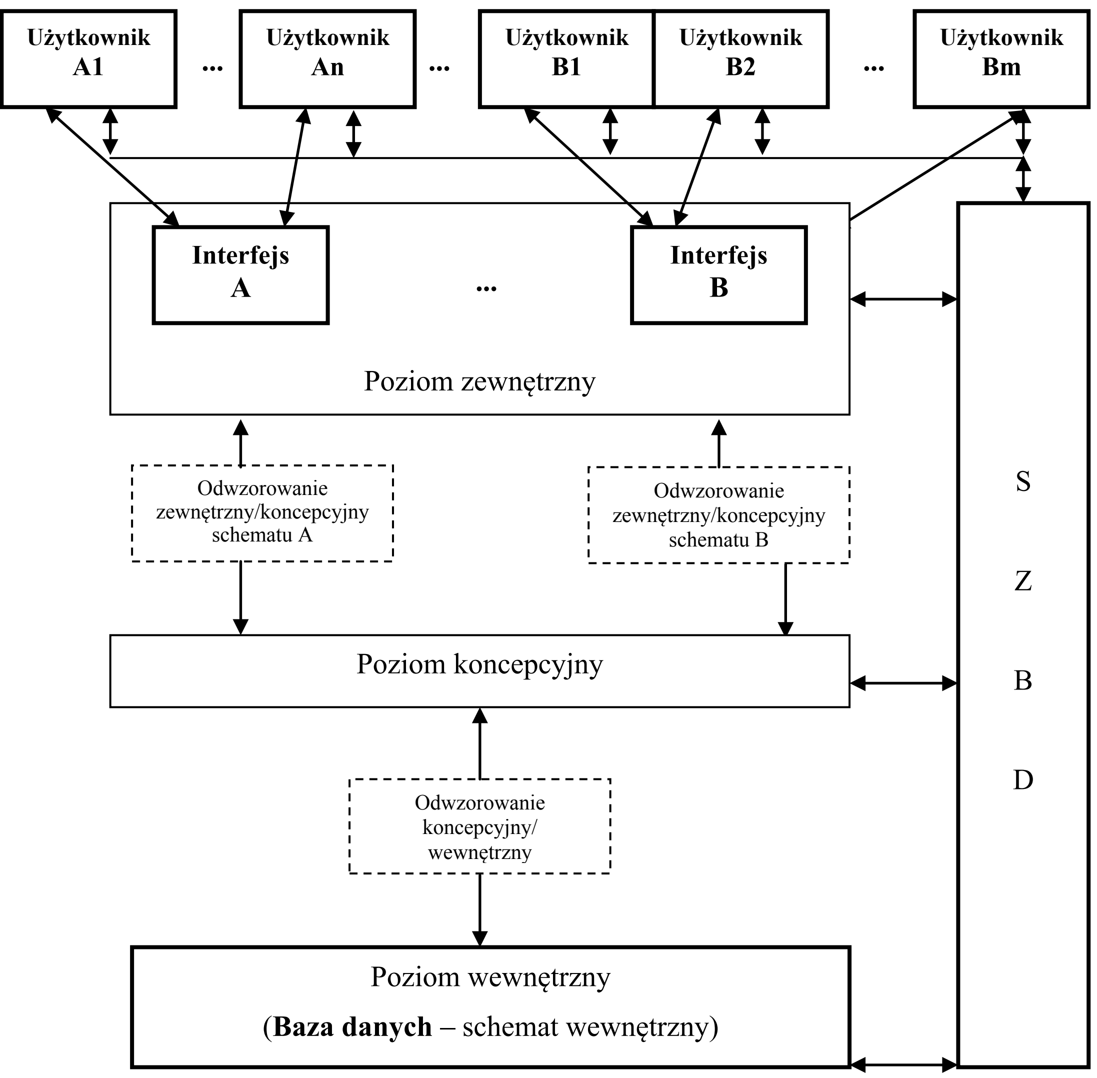
---

[9] Autoryzacja dotyczy problemu bezpieczeństwa BD.

- każdemu użytkownikowi powinna być dana możliwość posiadania własnych, różniących się od innych, reprezentacji właściwości przechowywanych danych.

Pierwsza i ważniejsza z wymienionych części koncepcji polega na idei oddzielenia logicznej struktury BD i manipulowania danymi (poziom użytkownika) od fizycznego przedstawienia danych (poziom systemowy). Większość nowoczesnych SZBD może być opisana przez połączenie tych poziomów. Ponadto ta sama BD w zależności od punktu widzenia może mieć różne poziomy opisu.

Dwupoziomowa architektura SZBD (czasem używany jest termin *architektura systemu BD*) została opracowana przez grupę roboczą CODASYL (Conference on Data System and Languages) w 1971 r. W 1975 r. Komitet Planowania Standardów i Normatywów SPARC (Standards Planning and Requirements Committee) Amerykańskiego Państwowego Instytutu Standardów ANSI (American National Standards Institute) stworzył trójwarstwową architekturę, która stała się oficjalnym standardem w 1978 r.



Rys. 1.5. Trzypoziomowa architektura SZBD

- *zewnętrzny* (poziom *logiczny użytkowników*). Użytkownikiem może być programista, użytkownik końcowy lub ABD. Przedstawienie BD z punktu widzenia użytkownika lub grupy użytkowników nazywa się *zewnętrznym*. Każdy użytkownik (grupa)

analizuje encje i ich związki odpowiednio pod kątem własnych potrzeb i wymagań. Przykładowo: pracownika (użytkownika) kwestury w BD *Uczelnia* interesują przede wszystkim te informacje o studentach, które dotyczą kwoty stypendium, a nie adresu zamieszkania czy daty urodzenia. Analizując ten poziom, warto podkreślić, że każdy użytkownik (*A1* lub *Bm* na rys. 1.5) lub grupa użytkowników (*A* lub *B*) mogą wykorzystywać do pracy z BD swój własny język (interfejs). Może to być język wysokiego poziomu albo specjalny język zapytań (np. SQL). W dowolnej sytuacji częścią tego języka powinien być *podjęzyk danych*, który generalnie składa się z *języka definiowania danych* (DDL) i języka *manipulowania danymi* (DML);

- *koncepcyjny* (lub *logiczny*), jest poziomem pośrednim. Daje możliwość prezentacji danych w formie abstrakcyjnej. Opis BD na tym poziomie nazywa się *schematem koncepcyjnym*. W ten sposób schemat koncepcyjny jest integralnym opisem logicznym wszystkich elementów danych i relacji między nimi – jest logiczną strukturą całej BD. Dla każdej BD istnieje tylko jeden schemat koncepcyjny, który powinien się składać z:
  - obiektów (encji) i ich atrybutów,
  - *związków* między obiektami,
  - *ograniczeń* dotyczących danych,
  - *informacji semantycznych* o danych,
  - *realizacji polityki bezpieczeństwa* danych.

Poziom ten jednak nie posiada żadnych informacji o metodach przechowywania danych;

- *wewnętrzny* (poziom *fizyczny*) jest najbardziej bliski sposobom przechowywania danych na nośnikach fizycznych. Wewnętrzna reprezentacja danych jest niskopoziomową reprezentacją całej BD, opierającą się na opisie wewnętrznego modelu danych, zwanego też *schematem wewnętrznym*. Dla każdej BD istnieje tylko jeden schemat wewnętrzny. Na tym poziomie przechowuje się następujące informacje:
  - rozdzielenie obszaru dyskowego do przechowywania danych i *indeksów* (indeksy są elementami BD, zwiększającymi szybkość wyszukiwania danych; szczegółowe omówienie indeksów nastąpi w dalszej części książki),
  - opis szczegółów rozmieszczenia i przechowywania poszczególnych danych (w tym ich rozmiarów),
  - informacje o kompresji i szyfrowaniu[10] danych.

SZBD odpowiada za dokładność odwzorowań (na rys. 1.5 oznaczone prostokątami obramowanymi przerywanymi liniami) między trzema typami schematów. Warto podkreślić, że odwzorowanie koncepcyjny/wewnętrzny jest kluczem do fizycznej niezależności danych, natomiast odwzorowanie zewnętrzny/koncepcyjny jest kluczem do logicznej niezależności danych.

Zadaniem administratora BD jest definicja każdego z przeanalizowanych schematów, jak i współdziałanie z użytkownikiem (najczęściej końcowym) w celu udzielenia mu niezbędnej pomocy.

[10] Zwiększa poziom poufności (tajności) danych.

Zadania do samokontroli

1. Podać definicje terminów i pojęć:
   - baza danych;
   - dane;
   - informacja;
   - system zarządzania BD;
   - obiekt;
   - atrybut;
   - połączenie.
2. Scharakteryzować podstawowe funkcje SZBD.
3. Podać klasyfikację BD.
4. Scharakteryzować architekturę BD.

1.2. Wprowadzenie do projektowania BD

1.2.1. Modelowanie koncepcyjne i diagramy związków encji

Proces projektowania BD zaczyna się od analizy tego, jakie informacje powinna przechowywać BD oraz jakie połączenia występują pomiędzy tymi informacjami. Fundamentalnymi elementami w modelowaniu koncepcyjnym są:

- dane;
- właściwości danych;
- połączenia danych.

Istnieje kilka systemów (metod) opisu projektów BD. Od początku lat 70. opracowano *semantyczne (koncepcyjne) metody* projektowania, dające możliwość odwzorowywania rzeczywistości w konstrukcje modelu BD. Najbardziej popularnym semantycznym modelem jest *model* '*encja – połączenie (relacja)*' (ang. *entity – relationship model, ER model*). Autorem tego modelu jest P. Chen (1976 r.)[11].

Podstawowymi elementami modelu są:

- *obiekty* (typy obiektów) lub *encje*;
- *atrybuty*;
- *połączenia* (relacje) obiektów.

---

[11] Alternatywami są model *definicji danych*, opierający się na ideach obiektowo-orientowanych BD, i model *danych półstrukturalnych* (ang. *semistructured data*), który daje nieograniczony wybór struktury danych.

Jednorodne *encje* tworzą *zbiór encji* (ang. *entity set*)[12]. Przykładowym zbiorem mogą być *Grupy_Studentów*, *Wydziały*, *Zajęcia*, *Pracownicy*. Model ER dotyczy wyłącznie obiektów statycznych, gdyż odnosi się do struktur danych, nie zaś do operacji na tych danych.

W większości implementacji tego modelu atrybuty mogą być przypisane do następujących typów:

- *atomowych* (ang. *atomic types*), składających się z liczb całkowitych (ang. *integer*), rzeczywistych (ang. *float*), łańcuchów znakowych (ang. *string*) i innych;
- *strukturalnych* ('*struct*', jak w języku C).

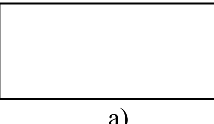
Połączenie (ang. *relationship*) w modelu ER może istnieć między dwoma lub większą ilością zbiorów encji. Jak podkreślaliśmy wyżej, w Prz. 1-1 zbiór encji *Studenci* powiązany jest (przynajmniej) ze zbiorami encji *Grupy_Studentów*, *Wydziały* i *Zajęcia*, *Pracownicy* – z *Wydziałami* itd.

Istotnym elementem modelu ER jest jego reprezentacja graficzna.

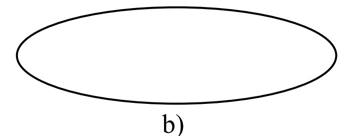
Definicja 1-8

Graficzny schemat wszystkich elementów modelu ER (elementów struktury BD: zbiorów encji, atrybutów i połączeń) nazywamy ***diagramem związków encji*** (ang. *entity-relationship diagram*, ER-D).

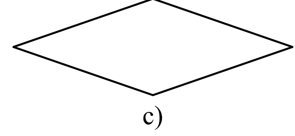
Na rys. 1.9 pokazane są podstawowe elementy ER-D: prostokąty (a) odpowiadają elementom danych (obiektom, czyli encjom), elipsy (b) – atrybutom, romby (c) – połączeniom.



a)



b)



c)

Rys. 1.9. Podstawowe elementy graficzne ER-D

Na diagramie ER-D atrybuty obiektu łączą się z obiektami za pomocą linii.

Przykład 1-2

Dla BD *Uczelni* z Prz. 1-1 i odpowiadającego mu rys. 1.1 diagram prezentujący obiekt *Pracownik* i jego atrybuty może wyglądać tak, jak pokazano na rys. 1.10.

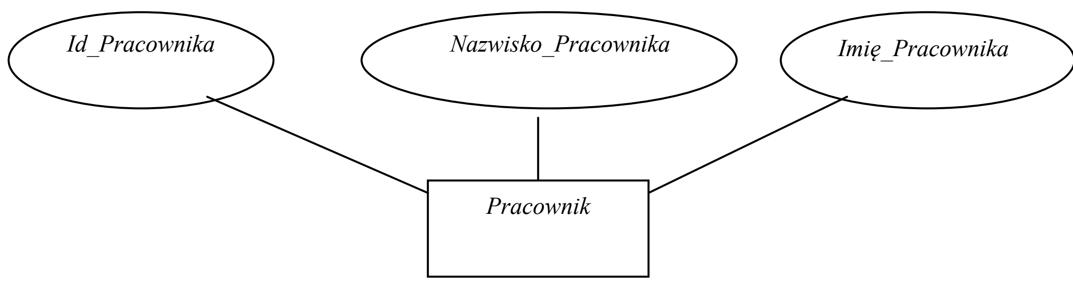
Połączenie obiektów na diagramach ER ilustruje rys. 1.11.

Jeżeli liczba uczestników połączenia wynosi dwa (jak na rys.1.11), to takie połączenie nazywamy *binarnym*. Liczba uczestników połączenia może być również większa niż dwa.

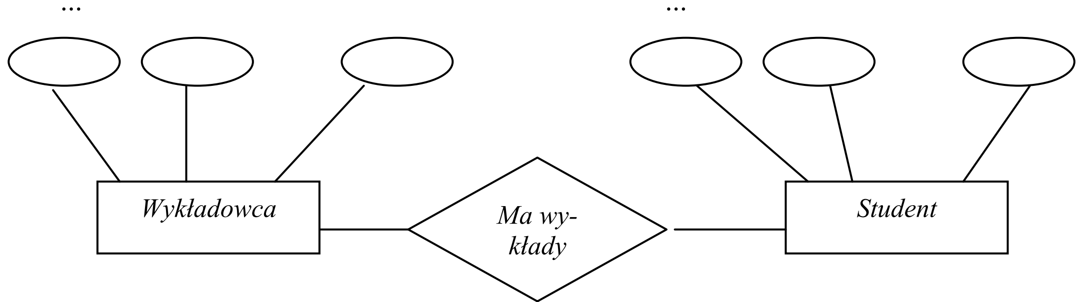
W rzeczywistych sytuacjach połączenia między trzema zbiorami encji (*połączenia ternarne*, ang. *ternary relationship*) lub połączenia między większą liczbą zbiorów występują dość rzadko. *Wielostronne połączenia* (ang. *multiway relationship*) odwzorujemy na diagramach poprzez linie łączące romb ze wszystkimi prostokątami odpowiadającymi wszystkim zbiorom.

---

[12] Mówiąc dokładniej, pojęcie 'encja' i pojęcie 'obiekt' czasem są rozróżniane, jeżeli traktujemy obiekt tak, jak jest to przyjęte w projektowaniu obiektowo-orientowanym; w ten sam sposób można traktować podobieństwo pomiędzy 'zbiorem encji' a 'klasą obiektów'.



Rys. 1.10. Diagram ER reprezentujący obiekt *Pracownik* i jego atrybuty

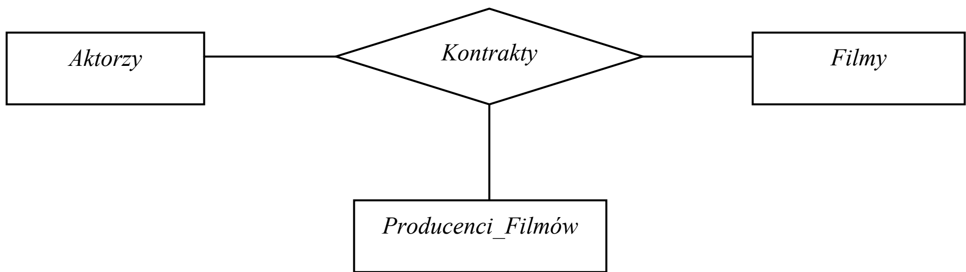


Rys. 1.11. Schemat diagramu ER z typem połączenia obiektów 'prowadzi wykłady' od strony *Wykładowca* do strony *Student*

Przykład 1-3
Połączenie *Kontrakty* między zbiorami *Producenci_Filmów*, *Aktorzy* i *Filmy* (*połączenia ternarne*) dotyczy podpisania kontraktu pomiędzy *Producentem Filmów* a jakimś *Aktorem* zobowiązującym się wziąć udział w konkretnym *Filmie*.

Przykładowi temu odpowiada rys. 1.12.

W procesie projektowania BD *połączenia wielostronne* najczęściej przekształcane są w *połączenia binarne*. W ten sposób ostatnie *połączenie ternarne* może być zastąpione przez dwa lub trzy *połączenia binarne*.



Rys. 1.12. Przykład połączenia ternarnego

1.2.2. Typy połączeń binarnych

Generalnie połączenie binarne może łączyć dowolny element jednego zbioru encji z dowolnym elementem drugiego zbioru encji. Jednakże najczęściej występują sytuacje, gdy połączenie binarne ma pewne ograniczenia.

Zakładamy, że istnieje połączenie ***P*** między zbiorami ***A*** i ***B***.

Definicja 1-9

Jeśli każdy element zbioru ***A*** za pośrednictwem ***P*** może być połączony nie więcej niż z jednym elementem zbioru ***B***, to mówimy, że ***P*** jest połączeniem ***wiele-do-jednego*** (ang. *many-one relationship*) w kierunku od ***A*** do ***B***.

W tej sytuacji każda encja zbioru ***B*** umożliwia połączenie z wieloma encjami zbioru ***A***. Z tego widać, że w odwrotnym kierunku (od ***B*** do ***A***) może istnieć inny typ połączenia.

Definicja 1-10

Jeśli każdy element zbioru ***B*** za pośrednictwem ***P*** może być połączony nie więcej niż z jednym elementem zbioru ***A***, to mówimy, że ***P*** jest połączeniem ***wiele-do-jednego*** (ang. *many-one relationship*) w kierunku od ***B*** do ***A***.

W tej sytuacji każda encja zbioru ***A*** umożliwia połączenie z wieloma encjami zbioru ***B***.

Definicje 1-9 i 1-10 mogą być zapisane w innej formie. Mamy odpowiednio:

Definicja 1-11

Jeśli dowolny element zbioru ***B*** za pośrednictwem ***P*** może być połączony nie więcej niż z jednym elementem zbioru ***A***, to mówimy, że ***P*** jest połączeniem ***jeden-do-wielu*** (ang. *one-many relationship*) w kierunku od ***A*** do ***B***.

Definicja 1-12

Jeśli dowolny element zbioru ***A*** za pośrednictwem ***P*** może być połączony z nie więcej niż z jednym elementem zbioru ***B***, to mówimy, że ***P*** jest połączeniem ***jeden-do-wielu*** (ang. *one-many relationship*) w kierunku od ***B*** do ***A***.

Na diagramach ER połączenie *jeden-do-wielu* lub *wiele-do-jednego* najczęściej oznacza się symbolami **1:n** (**1:∞**, **1:w**) lub n:1 (**∞:1**, **w:1**).

Przykład 1-4

BD *Szkoła* posiada m.in. informacje o encjach *Uczniowie* i *Klasy*. W najprostszej formie (bez analizy atrybutów) diagram ER-D dla tego przykładu może być przedstawiony tak jak na rys. 1.13.

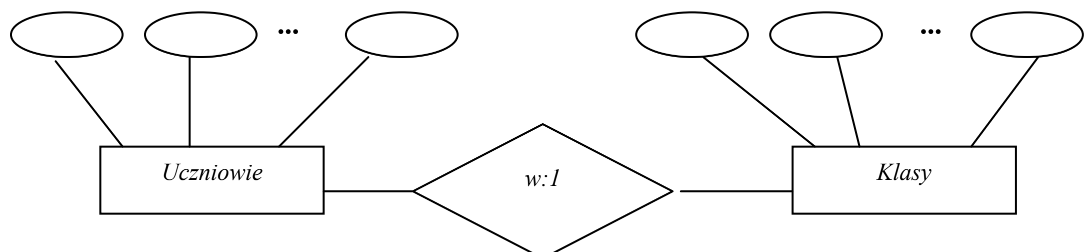
Z diagramu tego widać, że pomiędzy zbiorami encji *Uczniowie* i *Klasy* połączenie ***P*** jest połączeniem *jeden-do-wielu* w kierunku od zbioru *Klasy* do zbioru *Uczniowie*. Z drugiej strony połączenie pomiędzy oznaczonymi zbiorami encji można scharakteryzować jako *wiele-do-jednego* w kierunku odwrotnym. Praktycznie oznacza to, że klasa może składać się z dowolnej liczby *Uczniów*, ale dowolny *Uczeń* może być przypisany tylko do jednej *Klasy*.

Przykład 1-5

Podobny typ połączeń (jak w Prz. 1-4) ma miejsce w BD *Uniwersytet* pomiędzy zbiorami encji *Grupy_Studentów* a *Studenci* (patrz rys. 1.14).

Definicja 1-13

Jeśli dowolny element zbioru ***B*** za pośrednictwem ***P*** może być połączony z jednym elementem zbioru ***A***, to mówimy, że ***P*** jest połączeniem ***jeden-do-jednego*** (ang. *one-one relationship*) w kierunku od ***B*** do ***A***.



Rys. 1.13. Połączenie wiele-do-jednego (jeden-do-wielu) pomiędzy zbiorami *Uczniowie* i *Klasy* (*Klasy* i *Uczniowie*) w BD *Szkoła*[13]

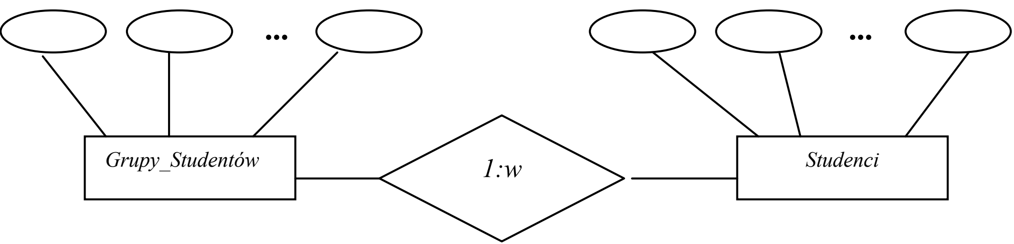
Definicja 1-14

Jeśli dowolny element zbioru ***B*** za pośrednictwem ***P*** może być połączony z jednym elementem zbioru ***A*** i dowolny element zbioru ***A*** za pośrednictwem ***P*** może być połączony z jednym elementem zbioru ***B***, to mówimy, że **P** jest połączeniem ***jeden-do-jednego*** (ang. *one-one relationship*) w obu kierunkach. Innymi słowy, pomiędzy ***A*** i ***B*** mamy połączenie ***jeden-do-jednego***.

Inna definicja tego typu połączenia może być następująca:

Definicja 1-15

Jeśli połączenie ***P*** w obu kierunkach (od ***A*** do ***B*** i od ***B*** do ***A***) jest połączeniem typu *wiele-do-jednego*, to mówimy, że ***P*** jest połączeniem ***jeden-do-jednego***.



Rys. 1.14. Połączenie *jeden-do-wielu* (*wiele-do-jednego*) pomiędzy zbiorami *Grupy_Studentów* a *Studenci* (*Studenci* a *Grupy_Studentów*) w BD *Uniwersytet*

Przykład 1-6

Zakładamy, że w BD *Ministerstwo Edukacji* występują informacje o dwóch encjach: *Uczelnie* i *Rektorzy*. Biorąc pod uwagę, że dowolna *Uczelnia* może mieć tylko jednego *Rektora*, a jeden *Rektor* może sprawować swą funkcję tylko jednej *Uczelni*, wnioskujemy, że między dwoma wymienionymi encjami może istnieć tylko połączenie *jeden-do-jednego*.

Przykładowi 1-6 odpowiada diagram na rys. 1.15.

Definicja 1-16

Jeżeli połączenie ***P*** w żadnym z kierunków nie jest połączeniem *wiele-do-jednego*, to ma miejsce połączenie typu ***wiele-do-wielu***.

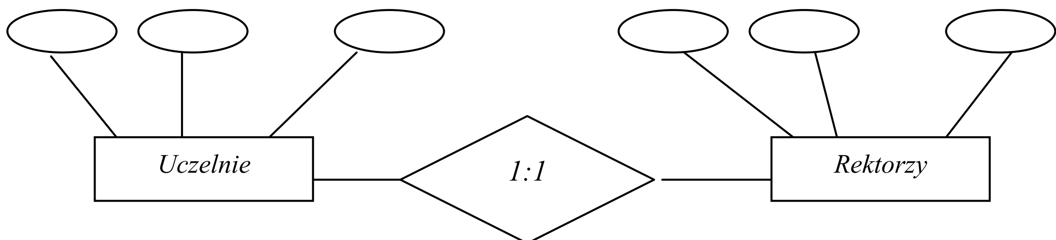
Inna definicja tego połączenia:

---

[13] W tym i w innych przykładach na diagramie podajemy (w rombie) nie nazwę połączenia, lecz jego typ.

Definicja 1-17

Jeżeli każdy element zbioru ***A*** może być połączony z dowolną liczbą elementów zbioru ***B*** i na odwrót: każdy element zbioru ***B*** może być połączony z dowolną liczbą elementów zbioru ***A***, to między tymi zbiorami ma miejsce połączenie typu ***wiele-do-wielu***.



Rys. 1.15. Połączenie *jeden-do-jednego* pomiędzy zbiorami encji *Uczelnie* i *Rektorzy* w BD *Ministerstwo Edukacji*

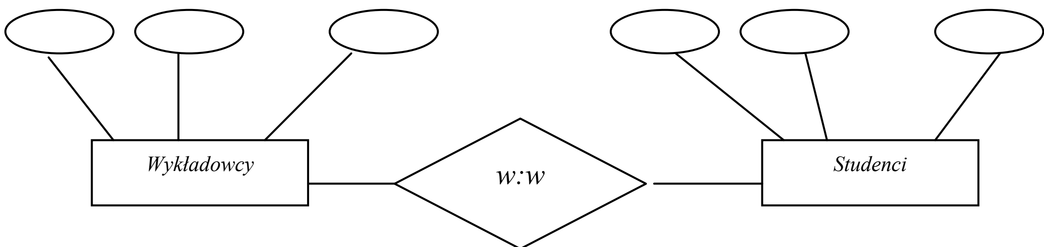
Przykład 1-7

BD *Uniwersytet* posiada informacje o zbiorach encji *Wykładowcy* i *Studenci*. Biorąc pod uwagę, że dowolny *Wykładowca* prowadzi wykłady dla wielu *Studentów* i dowolny *Student* słucha wykładów kilku *Wykładowców*, definiujemy połączenie między tymi dwoma zbiorami jako *wiele-do-wielu*.

Ostatniej sytuacji odpowiada rys. 1.16.

Bardzo ważne jest, aby podkreślić, że połączenie *wiele-do-jednego* jest szczególnym przypadkiem połączenia typu *wiele-do-wielu*, a połączenie *jeden-do-jednego* jest szczególnym przypadkiem połączenia typu *wiele-do-jednego*. Innymi słowy, dowolna cecha połączeń *wiele-do-wielu* jest właściwa połączeniom *wiele-do-jednego* i *jeden-do-wiele*, natomiast tylko niektóre cechy połączeń *wiele-do-jednego* są właściwe połączeniom *wiele-do-wielu*.

W dalszej części książki termin *połączenie zbiorów encji* będziemy czasem zastępować terminem *relacja pomiędzy zbiorami*.



Rys. 1.16. Przykład połączenia *wiele-do-wielu* pomiędzy zbiorami encji *Wykładowcy* i *Studenci* w BD *Uniwersytet*

Zadania do samokontroli

1. Przeznaczenie i budowa diagramów ER.
2. Stworzyć koncepcyjny model danych:

   a) *Uniwersytetu*, który posiada następujące informacje:

   - o wykładowcach;
   - o przedmiotach;
   - o studentach;

b) *Wypożyczalni samochodów*, który (model) zawiera następujące informacje:

- o pracownikach;
- o klientach;
- o samochodach;

c) *Stacji paliw*, który zawiera następujące informacje:

- o pracownikach;
- o klientach;
- o typach i producentach paliw.

3. Stworzyć diagramy ER odpowiadające modelom z zadania 2.
4. Przeanalizować projekt BD *Bank*, posiadający informacje o *Klientach* i stanie ich *Kont*. Dane o *Klientach* opisane będą za pomocą atrybutów: *Nazwisko*, *Imię*, *Adres*, *Numer_telefonu*. *Konto* będzie opisywane przez *Numer_Konta*, *Typ* i *Stan*.

   Należy stworzyć ER-D, który powinien odwzorowywać fakt, iż konkretne *Konto* należy do konkretnego *Klienta*.
5. Podać w postaci ER-D strukturę bazy danych ‘związku piłki nożnej’, obejmującej informacje o *Zespołach (*zawierającą atrybuty: *Nazwę*, *Miasto*, *Kolor_Koszulek*), o *Trenerach* (*Nazwisko*, *Imię*, *Wiek*), o *Piłkarzach* (*Nazwisko*, *Imię*, *Wiek*, *Pozycja*).
6. Zmodyfikować diagram z zadania 5 w taki sposób, aby przechowywał on informacje o tym, w jakich zespołach grał poprzednio piłkarz.

1.3. Modele baz danych

Dowolny SZBD opiera się na jakimś modelu BD. Warto podkreślić, że pojęcie *modelu BD* pojawiło się wraz z powstaniem modelu relacyjnego.

Definicja 1-18

Terminem ***model BD*** określamy sposób organizacji danych, sposób patrzenia na dane, sposób logicznego powiązania danych i sposób dostępu do danych, które są ściśle związane z typem i parametrami komputera, na którym umieszczona jest BD.

W modelu danych możemy wyróżnić trzy następujące składniki:

- **definicja danych** (**część strukturalna**) – jest to zbiór zasad określających strukturę danych, zgodnie z którymi powinny być konstruowane bazy danych, innymi słowy, jest to definiowanie struktur modelu;
- **operowanie danymi** (**część wykonawcza**) – zbiór zasad, które określają dopuszczalne operacje na danych, a więc regulujące sposób dostępu do danych i ich modyfikacji, czyli sterowanie strukturami modelu;
- **integralność danych** – zbiór zasad, które określają, jakie stany bazy danych są uważane za poprawne, a co za tym idzie, które operacje modyfikujące dane są dozwolone, czyli ochrona spójności (jednolitości) danych.

Najbardziej ogólny podział (klasyfikacja) modeli wygląda następująco: *modele* baz danych bazujące na *obiektach*, *rekordowe* modele baz danych i *fizyczne* modele. Wobec trójpoziomowej architektury BD (rys. 1.5) należy podkreślić, iż pierwsze dwie wymienione klasy wykorzystujemy do analizy danych na poziomie zewnętrznym i koncepcyjnym, ostatnią na poziomie wewnętrznym (modele te opisują, w jaki sposób dane przechowywane są w komputerze, w jaki sposób są uporządkowane, jakie są możliwości dostępu do danych). W podrozdziale tym skupimy się na klasach dwóch pierwszych modeli.

W porządku chronologicznym powstały kolejno: *hierarchiczne* i *sieciowe bazy danych*, inaczej mówiąc, BD oparte na hierarchicznym i sieciowym modelu. W zasadzie modele te bazują na *teorii grafów*.

W obu wymienionych modelach dane mają strukturę, którą można przedstawić w formie odwróconego drzewa (czy – mówiąc prościej – struktury katalogów i plików). Stworzenie tych modeli bardzo istotnie zwiększyło szybkość wyszukiwania przechowywanych danych.

W 1970 r. Edgar F. Codd, pracownik firmy IBM, opublikował pracę[14], w której postulował podział bazy danych na warstwę logiczną i fizyczną, określił zasady, pozwalające unikać nadmiarowości informacji i niewymagające od użytkownika znajomości struktury danych. Dodatkowo sformułował związki pomiędzy *teorią baz danych* a *teorią mnogości i rachunku predykatów pierwszego rzędu*. Tak powstał najpopularniejszy do dziś *relacyjny model bazy*, w którym operacje na danych znajdują przełożenie na operacje na zbiorach.

Rozwój technologii oprogramowania w latach 80. charakteryzuje się powstaniem nowej metodologii: obiektowo-orientowanej. Na bazie tej metodologii rozpoczęto budowę nowych *obiektowych baz danych*.

Po kolei krótko przeanalizujemy niektóre szczegóły wymienionych modeli BD.

1.3.1. Hierarchiczny model BD

Dla *hierarchicznych modeli* BD (HMBD) i odpowiadających im *hierarchicznych* BD nie opracowano w zasadzie żadnych standardów, a zatem analiza modelu może być przeprowadzona poprzez przyjrzenie się rzeczywistym SZBD.

Najbardziej znanym SZBD, stworzonym (przez firmę IBM, International Business Machines) na podstawie modelu hierarchicznego, jest system IMS (*Information Management System*). Ten system został opracowany w USA na potrzeby projektu 'Apollo' (lądowanie na księżycu) w 1968 r. Jest on zarazem jednym z najstarszych i najszerzej stosowanych systemów bazodanowych. Programiści IMS byli pierwszymi, którzy musieli poradzić sobie z takimi problemami, jak: konkurencja, odzyskiwanie danych, integralność czy w końcu optymalizowanie zapytań. Kolejne wersje systemu IMS, których dotychczas stworzono 9, wprowadzały olbrzymią liczbę właściwości i opcji, w rezultacie czego jest on niezwykle skomplikowanym systemem.

Model hierarchiczny wprowadza dwie następujące koncepcje struktur danych: *rekord* oraz *relacja*[15] *ojciec-syn (rodzic-dziecko)*.

---

[14] E.F. Codd, *A relational model for large shared data banks*, „Comm. ACM" 13(1970), N° 6, p. 377-387.

[15] Przypomnijmy sobie, że tym terminem określamy połączenie (*relationship*) pomiędzy zbiorami danych.

Definicja 1-19

Rekord jest zbiorem atrybutów (pól), które dostarczają informacji o poszczególnych cechach obiektu opisywanego przez *rekord*.

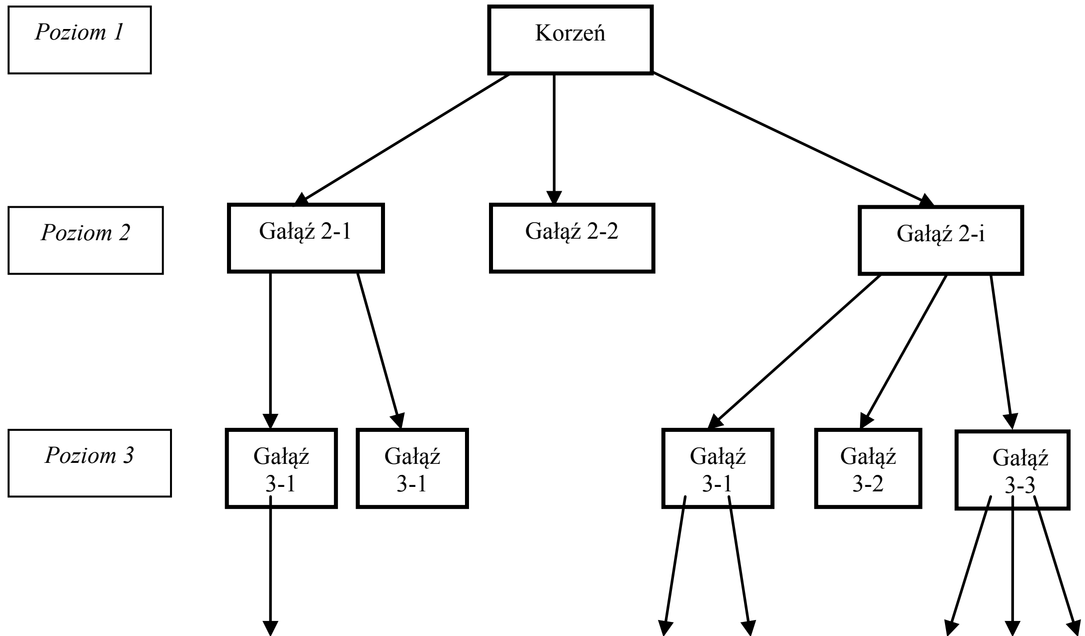
Rekordy tego samego typu są zgrupowane w *typy rekordów*. Każdy typ rekordu ma swoją nazwę, a jego struktura jest zdefiniowana przez zbiór nazwanych pól. Każde pole natomiast charakteryzuje się określonym typem danych jak liczba całkowita, rzeczywista, łańcuch znaków (*string*) czy data.

Definicja 1-20

Relacja ojciec-syn lub **rodzic-dziecko** *(*ang. *parent–child relationship type, PCR type)* jest relacją *jeden-do-wielu* (1:w) pomiędzy dwoma typami rekordów. Typ rekordu po stronie 1 jest nazywany **typem rekordu ojciec (rodzic)** *(parent)*, natomiast typ rekord po stronie *w* jest nazywany **typem rekordu syn (dziecko)** *(child*).

Każda relacja typu *PCR* zawiera jeden rekord typu *ojciec*, a także pewną liczbę (zero lub wiele) rekordów typu *syn*.

Jak już wspomniano w części wstępnej tego podrozdziału, strukturę HMBD można przedstawić w formie drzewa, ale drzewa odwróconego (rys. 1.17).



Rys.1.17. Drzewiasta struktura hierarchicznej BD

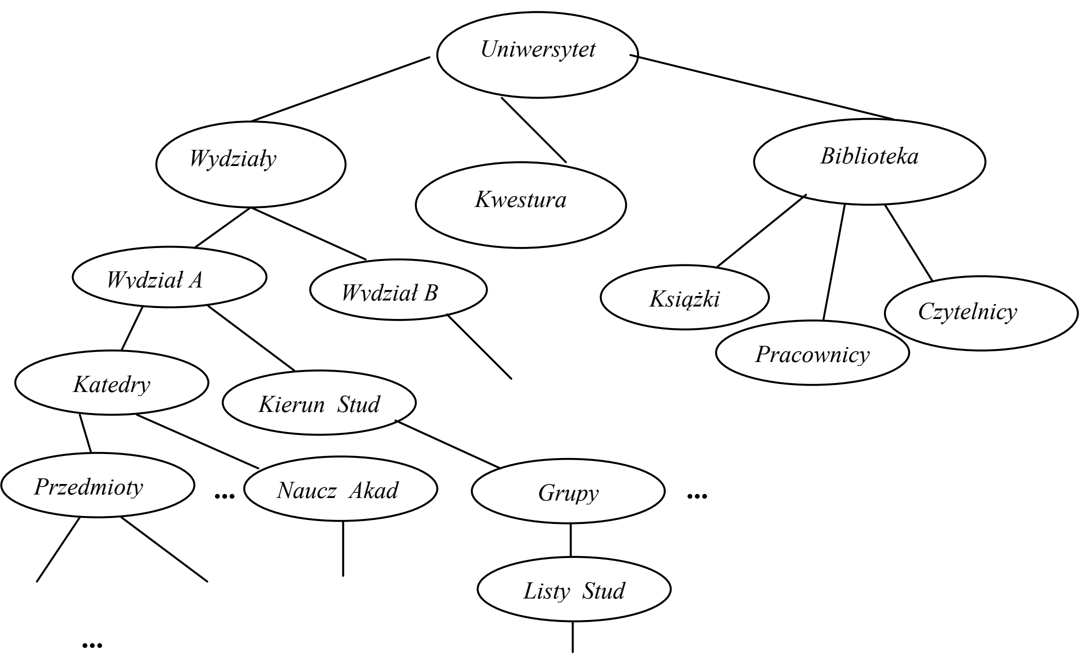
Schemat hierarchicznej bazy danych zawiera pewną liczbę schematów hierarchicznych. Każdy natomiast *schemat hierarchiczny* zawiera pewną liczbę typów rekordów, a także typów relacji PCR. Hierarchiczny schemat złożony z typów rekordów i relacji PCR musi spełniać następujące warunki:

- jeden typ rekordu, zwany *korzeniem* (ang. *root***)** w *schemacie relacyjnym* nie uczestniczy w żadnym z typów relacji PCR jako typ rekordu dziecka; każdy z typów rekordów poza korzeniem uczestniczy dokładnie w jednym typie relacji PCR w roli dziecka; gałęzie (rekordy) poziomu 2. w relacji z korzeniem (poziom 1) na rys. 1.17 pełnią rolę dziecka w relacji z *korzeniem*;

- typ rekordu może uczestniczyć w dowolnej liczbie (zero lub więcej) typów relacji PCR w roli rodzica; w relacji *Korzeń-Gałąź 1-1 Gałąź 1-1* występuje w roli dziecka, natomiast w relacji z rekordami *Gałąź 3-1* i *Gałąź 3-2* w roli rodzica;
- typ rekordu, który nie uczestniczy w żadnym typie relacji PCR w roli rodzica, jest nazywany *liściem* (ang. *leaf*) w hierarchicznym schemacie; numery gałęzi z rekordami-liśćmi są podkreślone na rys. 1.17;
- jeżeli dany typ rekordu uczestniczy w wielu typach relacji PCR w roli rodzica, to typy rekordów dzieci odpowiadające temu rekordowi w relacjach są uporządkowane. Porządek ten jest wyrażony, poczynając od lewej strony do prawej w diagramie hierarchicznym.

Korzystając z modelu hierarchicznego, możemy przedstawić *schemat pojęciowy w postaci grafu, stanowiącego strukturę drzewiastą*, którego węzły odpowiadają klasom obiektów, a linie między dwoma węzłami – powiązaniom istniejącym między tymi klasami.

Przykład 1-8
Na poniższym rys. 1.18 podany jest przykład hierarchii w BD *Uniwersytet*.

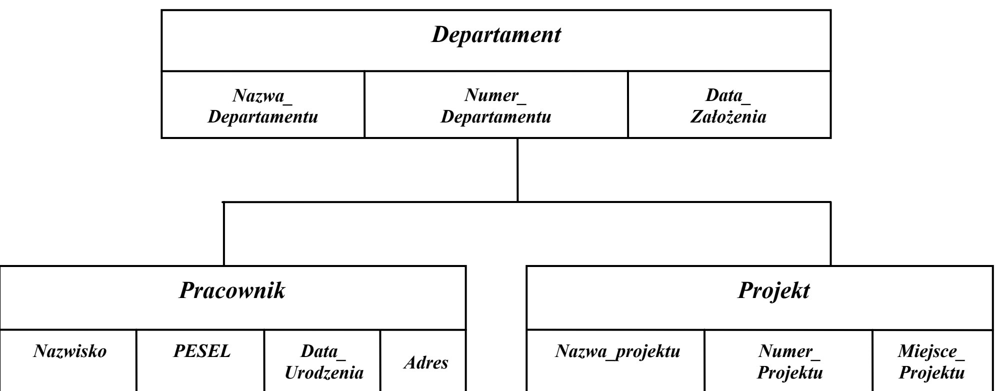


Rys. 1.18. Struktura powiązań w hierarchicznej BD *Uniwersytet*

Schemat hierarchiczny może być przedstawiony za pomocą *diagramu hierarchicznego*, w którym nazwy typów rekordów są umieszczone w prostokątach, natomiast relacje PCR są oznaczane jako linie łączące typ rekordu *ojciec* z typem rekordu *syn*. Poniższy *diagram hierarchiczny* (rys. 1.19) obrazuje schemat hierarchiczny składający się z trzech typów rekordów, a także z dwóch typów relacji PCR. Typy rekordów to *Departament*, *Pracownik* i *Projekt*. Nazwy pól mogą być umieszczone pod nazwą typów rekordów, czasami jednak dla zwiększenia czytelności w diagramach umieszcza się tylko nazwy typów rekordów.

Istnieją pewne ograniczenia relacji, które mogą być przedstawione w schemacie relacyjnym. Mianowicie, bezpośrednio nie mogą być reprezentowane relacje *wiele-do-wielu* (w:w) z tego względu, iż relacje *ojciec-syn* są relacjami typu *jeden-do-wielu* (1:w) i dany typ rekordu nie może uczestniczyć jako dziecko w dwóch oddzielnych relacjach *ojciec-*

syn. Jednakże relacje *wiele-do-wielu* można reprezentować w schemacie hierarchicznym dzięki *duplikacji instancji typów rekordów dzieci*. Jeżeli np. mielibyśmy sytuację, w której występuje relacja *wiele-do-wielu* między pracownikami i projektami, a więc sytuację, w której jeden pracownik pracuje nad kilkoma projektami i jeden projekt może być realizowany przez kilku pracowników, należałoby tę relację reprezentować jako relację (*Projekt*, *Pracownik*). Wtedy rekord opisujący poszczególnego pracownika występowałby zduplikowany pod każdym z rekordów opisujących projekt, nad którym pracuje. Alternatywnie moglibyśmy to samo osiągnąć poprzez relację (*Pracownik*, *Projekt*) i wtedy rekord reprezentujący dany projekt występowałby pod każdym z rekordów reprezentujących pracownika, który ten projekt realizuje.



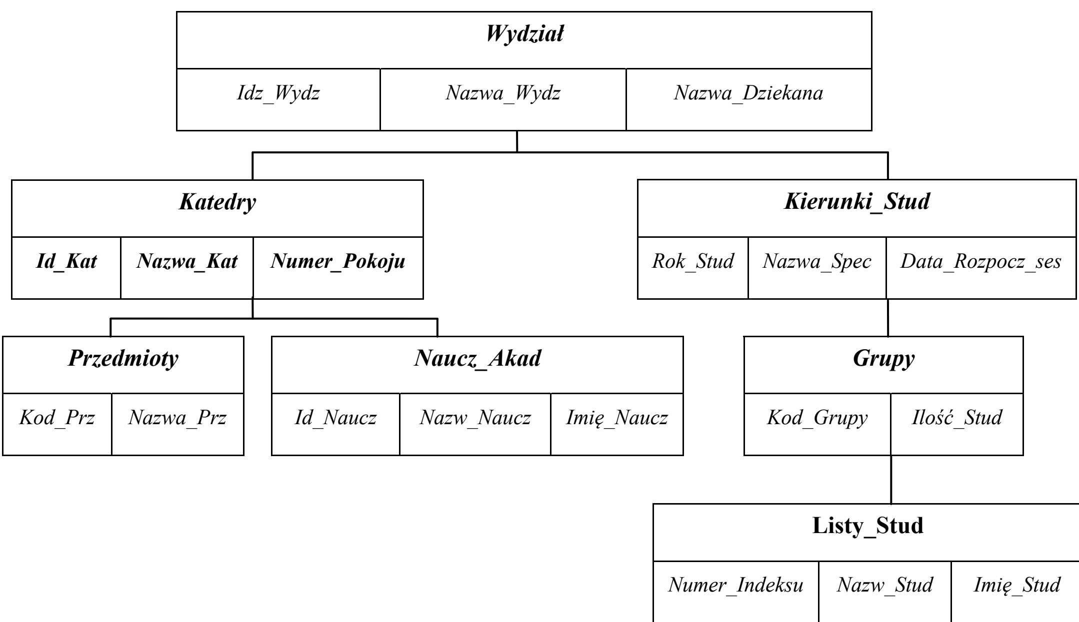
Rys. 1.19. Przykład schematu hierarchicznego

Biorąc pod uwagę te rozważania, możemy przedstawić fragment (rys. 1.20) BD *Uniwersytet*, którego struktura podana jest na rys.1.18. Niech ten fragment dotyczy danego *Wydziału*. Można również traktować ten fragment jako samodzielną bazę danych tego *Wydziału*.

Każdy model baz danych określa pewne ograniczenia integralności, które muszą być przestrzegane, aby baza pozostawała w stanie spójności i niesprzeczności. Takie ograniczenia nazywane *więzami integralności* występują również w *schemacie hierarchicznym*. Możemy do nich zaliczyć następujące ograniczenia: żaden rekord z wyjątkiem *korzenia* nie może występować, nie będąc w relacji z jakimś rekordem typu *ojciec*. Pociąga to za sobą następujące konsekwencje:

- żaden rekord będący dzieckiem nie może zostać wstawiony bez wcześniejszego połączenia go z rekordem ojcem;
- rekord będący dzieckiem może być kasowany, niezależnie od rekordu będącego jego ojcem. Jednak skasowanie rekordu ojca powoduje automatyczne skasowanie wszystkich rekordów jego dzieci i ich rekordów potomnych;
- jeżeli rekord dziecko ma dwoje albo więcej rodziców z tego samego typu rekordów, to rekord będący dzieckiem musi być duplikowany pod każdym z rekordów rodziców.

Wyszukiwanie **danych w hierarchicznych BD zaczyna się od** ***korzenia drzewa*****.**



Rys. 1.20. Hierarchiczny model BD *Wydziału*

Hierarchiczny model baz danych ma kilka wad. Jedną z nich jest to, iż każdy użytkownik takiej bazy danych **musi dobrze znać** jej strukturę. Aby uzyskać potrzebne informacje, musi przebyć drogę od rekordu *korzenia* kolejnymi gałęziami aż do poszukiwanej informacji w rekordzie *synu*.

Kolejnym problemem jest tu *nadmiarowość* danych. Hierarchiczny model baz danych jest niezdolny do reprezentowania złożonych zależności. Jak już wspomniano, wcześniej często niezbędne jest duplikowanie rekordów w celu reprezentowania relacji *wiele-do--wielu*. Taka nadmiarowość może prowadzić do niekonsekwentnego wprowadzania danych, a co za tym idzie, naruszenia integralności bazy danych.

1.3.2. Sieciowy model BD

Sieciowy model baz danych (SMBD) powstał w głównej mierze w celu wyeliminowania problemów, jakie niósł ze sobą hierarchiczny model BD. Jedną z pierwszych praktycznych realizacji tego modelu był SZBD IDS (*Integrated Data Store*), opracowany przez firmę General Electric Corp. Architektura tego systemu stała się platformą działalności grupy DBTG (DataBase Task Group), która pod koniec lat 60. rozpoczęła tworzenie standardów SZBD (było to zadanie zlecone tej grupie przez CODASYL). Innymi znanymi sieciowymi SZBD są: IDMS (*Integrated Database Management System*) firmy Computer Associates, IDS II firmy Honeywell, DMS II firmy Burroughs, DMS 1100 firmy Univac, VAX/DBMS firmy Digital Equipment, IMAGE firmy Hewlett-Packard. Większość z tych systemów została rozwinięta po 1971 r., realizując koncepcje zawarte w raporcie *CODASYL*[16].

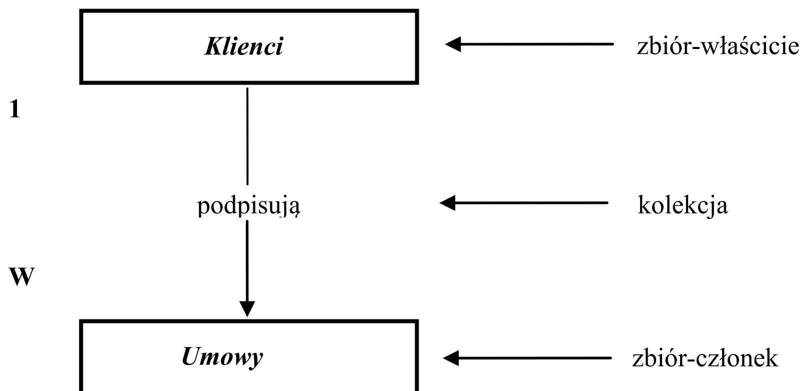
[16] Konferencja CODASYL (Conference on Data System Languages), obradująca nad rozwojem języków dla systemów danych i zrzeszająca przedstawicieli rządu USA oraz świata finansów i biznesu, powołała w 1965 r. specjalną grupę do spraw przetwarzania danych (List Processing Task Force, co można tłumaczyć jako Siły Specjalne do Zadań Przetwarzania Danych), która w 1967 r. przyjęła nazwę grupy do zadań związanych z ba-

IDMS jest szczególnie ceniony za wysoki poziom bezpieczeństwa danych i ich przetwarzania, m.in. z wykorzystaniem narzędzia IENF (*Integrated Event Notofication Facility* – Zintegrowane Śledzenie Zdarzeń). Jest to niezależny fragment oprogramowania, dzięki któremu można śledzić wszystkie sesje w systemie wielodostępowym.

W modelu sieciowym dane są umieszczone w *rekordach*, które są klasyfikowane do poszczególnych *typów rekordów*. Każdy rekord ma własne *pola* (atrybuty), każde pole natomiast musi mieć ściśle określony typ wartości. Strukturę SMBD można również przedstawić za pomocą *drzewa*, kilka takich drzew może ze sobą współdzielić gałęzie i każde z tych drzew jest częścią ogólnej struktury bazy danych.

Podstawowa różnica* w reprezentowaniu danych w sieciowej strukturze danych w stosunku do reprezentowania danych w strukturze hierarchicznej polega na tym, że w modelu sieciowym *dziecko (syn) może mieć dowolną liczbę rodziców (ojców).

W SMBD relacje są reprezentowane nieco inaczej niż w HMBD. Tutaj mamy do czynienia z tzw. *kolekcjami* (ang. *set structures*), które są niejawnymi powiązaniami między dwoma zbiorami danych. Jednemu z nich przypisuje się rolę *właściciela*, natomiast drugiemu rolę *członka*. Kolekcja reprezentuje relację typu *jeden-do-wielu* (1:w), co oznacza, że danemu rekordowi ze zbioru-właściciela może odpowiadać wiele rekordów ze zbioru-członka, natomiast każdy rekord zbioru-członka musi być powiązany z jednym rekordem ze zbioru-właściciela[17]. Odnosząc się do powyższego przykładu, każda umowa może być podpisana przez jednego klienta, każdy klient może podpisać dowolną liczbę umów, ale mogą również istnieć i tacy klienci, którzy jeszcze nie podpisali żadnej umowy. Taką typową strukturę kolekcji można przedstawić w postaci następującego rysunku (rys. 1.21).



Rys. 1.21. Struktura kolekcji

Pomiędzy dwoma zbiorami można zdefiniować dowolną liczbą kolekcji, jak również każdy zbiór może uczestniczyć w wielu różnych kolekcjach, co stanowi nowość w porównaniu z HMBD. Dla przykładu zbiór *Umowy* jest powiązany z *Klientami* poprzez kolekcję „podpisuje”. W HMBD niemożliwe było powiązanie rekordu dziecka z dwoma lub większą liczbą rekordów ojca.

zami danych (DBTG, Data Base Task Group). Polem działania DBTG były specyfikacje standardów dla środowiska umożliwiającego tworzenie bazy danych i przetwarzanie danych. Projekt raportu został sporządzony w 1969 r., a pierwszy pełny raport pojawił się 2 lata później.

[17] Praktycznie oznacza to powiązania pomiędzy encjami (obiektami), o czym mówiliśmy wyżej.

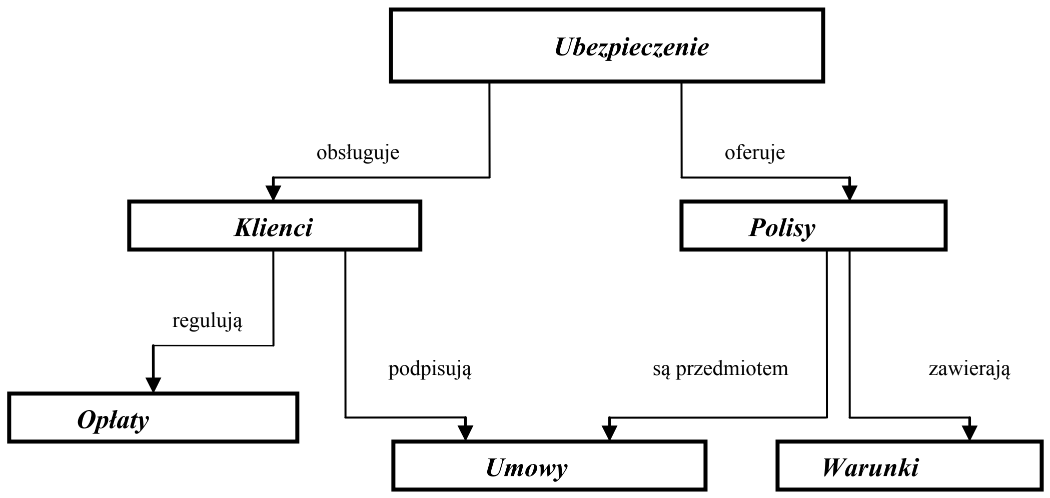
Przykładem drzewa odpowiadającego sieciowemu modelowi jest diagram przedstawiony na rys. 1.22.

HMBD i SMBD różnią się również sposobem przeszukiwania danych. Jak pamiętamy, w *modelu hierarchicznym*, aby znaleźć żądaną informację, przeszukiwanie należało rozpocząć od korzenia, schodząc w dół drzewa po gałęziach. W *sieciowym modelu* poszukiwanie można rozpocząć od dowolnej tabeli, poruszać należy się po kolekcjach, ale można kierować się zarówno w do dolnych, jak i górnych tabel powiązanych z tabelą, w której rozpoczęliśmy przeszukiwanie.

Zaletą takiego podejścia jest to, iż można znaleźć bardzo wiele informacji, ponadto poszukiwanie jest bardzo szybkie. Użytkownik może kierować do bazy o wiele bardziej skomplikowane zapytania, niż to miało miejsce w HMBD.

Wadą tego modelu, podobnie jak modelu hierarchicznego, jest to, że użytkownik musi doskonale znać całą strukturę bazy, aby mógł wyszukać informacje, które są mu potrzebne. Kolejną wadą jest to, że każda zmiana struktury bazy danych niesie za sobą konieczność zmienienia aplikacji z niej korzystających. Jeżeli np. wprowadzimy zmianę w kolekcji, musimy również zmienić wszystkie odniesienia aplikacji do tej kolekcji, ponieważ to właśnie dzięki kolekcjom poszukują one danych. Może to przysparzać wiele kłopotów i niepotrzebnie pochłaniać czas.

Korzystając z modelu sieciowego, przedstawiamy schemat pojęciowy w postaci *grafu uogólnionego*, którego węzły odpowiadają klasom obiektów, a linia między dwoma węzłami reprezentuje powiązanie. Może to być *dowolny graf*, a nie jak w przypadku modelu hierarchicznego wyłącznie struktura drzewiasta.



Rys. 1.22. Diagram modelu sieciowego

Sieciowy model BD *Wydział*, którego diagram w modelu hierarchicznym przedstawiony został na rys. 1.20, może wyglądać jak na rys. 1.23[18].

W odróżnieniu od poprzedniego diagram modelu sieciowego charakteryzuje się nowymi połączeniami (kolekcjami): pomiędzy *Katedrą* a *Grupą* (kolekcja *uczy*) oraz pomiędzy

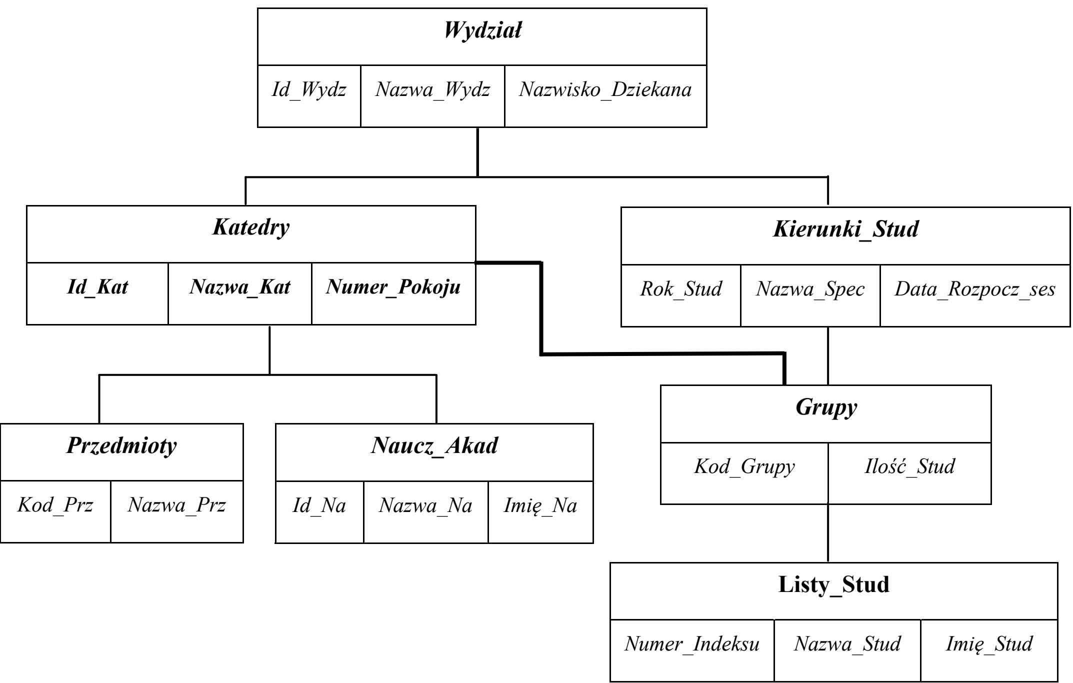
[18] Bez nazw kolekcji.

Nauczycielami_Akademickimi a *Grupą* (kolekcja *opiekują*, np. nauczyciel może być opiekunem grupy).

1.3.3. Wprowadzenie do relacyjnego modelu BD

Za twórcę *relacyjnego modelu BD* (RMBD) uważany jest pracownik firmy IBM dr Edgar F. Codd, który w 1970 r. wydał swoje największe dzieło *Relacyjny model logiczny dla dużych wielodostępowych banków danych*. Będąc z zawodu matematykiem, E. Codd zaprezentował w niej ogólne założenia modelu, które były oparte na trzech gałęziach matematyki: teorii mnogości, rachunku predykatów pierwszego rzędu oraz algebrze relacji[19].

Idea tego modelu polega na zastosowaniu struktur i procesów matematycznych w celu bardziej efektywnego zarządzania danymi, a przede wszystkim wyeliminowania głównych wad wcześniejszych modeli, takich jak: *nadmiarowość danych*, *zbyt słaba integralność danych* czy *zbyt duża zależność danych od fizycznej implementacji*. Model wprowadzony przez E. Codda pozbawiony jest tych wad.



Rys. 1.23. Diagram sieciowego modelu BD *Wydział*

W opracowywaniu RMBD E. Codd wykorzystał *teorię relacji* (*dwuwymiarowych zbiorów danych*, czyli *tabel*), na podstawie której udowodnił możliwość zastosowania operacji algebry relacyjnej i niektórych specjalnych operacji w odniesieniu do *całej mnogości rekordów relacji*, ale nie w odniesieniu do pojedynczych rekordów.

[19] W literaturze baz danych termin *relacja* często odnosi się do logicznego powiązania pomiędzy *encjami* (*zbiorami encji*) oraz do dwuwymiarowego zbioru danych.

Nazwa modelu pochodzi od nazwy zbioru danych w tym modelu, który nazywany jest *relacją*[20]. Poprzez wykorzystanie algebry abstrakcyjnej (praktycznie jest to teoria relacji razem z operacjami na relacjach) E. Codd stworzył formalną teorię relacyjnego modelu danych, która opiera się na surowych i precyzyjnych podstawach matematyki.

Do manipulowania danymi E. Codd wprowadził wspomniany wcześniej *język manipulowania danymi*, który umożliwił wykonywanie wszystkich operacji algebry relacyjnej. Za pomocą jednego *polecenia* (*kwerendy*) w tym języku (SQL) użytkownik ma możliwość pobrania danych z jednej lub kilku tabel.

***Wynikiem dowolnej operacji na danych będzie nowa tabela*, która z kolei może być rozpatrywana jako element wejściowy do wykonywania nowych operacji.**

Jedną z podstawowych zalet modelu relacyjnego jest to, że *wszystkie dane rozpatrujemy jako dane zawarte w tabelach i wyłącznie w tabelach.*

Definicja 1-21

Relacyjną bazą danych jest **skończony (ograniczony) zbiór relacji**. Relacje wykorzystujemy do reprezentacji obiektów, jak również do reprezentacji połączeń między obiektami.

Definicja 1-22

Relacja jest to ***dwuwymiarowa tabela***, która posiada ***unikatową nazwę*** i składa się z ***wierszy*** (rekordów) i ***kolumn*** (atrybutów, czyli pól).

Rozważmy relacyjny model baz danych jako zbiór tabel, z których każda skojarzona jest z *unikatową nazwą*. Wszystkie tabele mają podobną strukturę i są prezentowane w postaci odpowiednich diagramów. Wiersze w tabeli reprezentują zależności pomiędzy zbiorem wartości. Traktując tabelę jako zbiór takich powiązań, łatwo zauważyć bliższą odpowiedniość pomiędzy pojęciem tabeli a matematycznym pojęciem relacji, z którego właśnie relacyjny model danych wziął swoją nazwę.

Definicja 1-23

Relacja w sensie matematycznym jest ***dowolnym podzbiorem iloczynu kartezjańskiego zbiorów***, wiersze (rekordy) relacji zaś rozumiane są jako kolejne elementy tego podzbioru.

W tym miejscu przybliżymy związek relacji w relacyjnym modelu baz danych z relacją w sensie matematyki.

Iloczynem kartezjańskim zbiorów $A_1, A_2, A_3, \ldots, A_n$ nazywamy następujący zbiór:

$$A_1 \times A_2 \times A_3 \times \ldots \times A_n = \{(a_1, a_2, a_3, \ldots, a_n) : a_1 \in A_1, a_2 \in A_2, a_3 \in A_3, \ldots, a_n \in A_n\},$$

co można również zapisać jako $\mathop{\times}\limits_{i=1}^{n} A_i$. Z tego wynika, że iloczyn kartezjański jest zbiorem uporządkowanych n-tek.

Dla przykładu, iloczynem kartezjańskim zbiorów $A = \{2, 3, 4\}$ i $B = \{5, 6\}$ będzie zbiór

$$A \times B = \{(2, 5), (2, 6), (3, 5), (3, 6), (4, 5), (4, 6)\}.$$

Dowolny podzbiór iloczynu kartezjańskiego nazywamy *relacją*. Relacją będzie więc np. zbiór $R = \{(2, 5), (2, 6)\}$ czy $R = \{(2, 5), (3, 5), (4, 5), (4, 6)\}$. Dowolny podzbiór n-tek

[20] Niektórzy autory niepoprawnie podkreślają, że nazwa modelu ma związek z *relacjami* (powiązaniami) między zbiorami danych (tabelami).

(***n***-krotek) uporządkowanych iloczynu kartezjańskiego ***n*** zbiorów jest relacją ***n*** zbiorów (***n*** dziedzin).

Definicja 1-24

Liczbę atrybutów relacji w modelu relacyjnym nazywa się ***stopniem*** (***krotnością***) relacji.

Definicja 1-25

Liczbę rekordów (zwanych często także *krotkami*) określa się mianem ***mocy relacji***, jest to bowiem liczba elementów zbioru, którym jest relacja.

Definicja 1-26

Każdy atrybut w relacji musi mieć swoją ***dziedzinę***, a więc zbiór możliwych wartości dla tego atrybutu.

Dziedziną (ang. *domain*) jest zbiór dopuszczalnych wartości, z których pochodzą konkretne wartości określonych atrybutów danej relacji.

Na przykład dziedzina o nazwie 'Identyfikator Studenta' (*Id_Stud*) jest zbiorem wszystkich dozwolonych identyfikatorów studentów, dziedzina 'Nazwisko_Studenta' jest zbiorem wszystkich możliwych nazwisk studentów itd. A więc zbiór wartości *Id_Stud,* pojawiający się w dowolnej chwili w relacji *Studenci* lub w innej relacji musi być jego podzbiorem.

Dziedzinę definiujemy jako nazwany *zbiór wartości skalarnych*. Wszystkie muszą być tego samego typu. Inaczej mówiąc, w konkretnym polu muszą być dane tego samego typu.

Definicja 1-27

Schematem relacji nazywa się skończony zbiór nazw atrybutów wraz z ich dziedzinami.

Przez pojęcie *schematu relacji* rozumiemy relację zdefiniowaną poprzez podanie jej nazwy, a także par złożonych z nazw atrybutów oraz nazw dziedzin związanych z tymi atrybutami. Dla atrybutów $A_1, A_2, A_3, \ldots, A_n$ o dziedzinach $D_1, D_2, D_3, \ldots, D_n$ *schematem relacji* będzie zbiór $\{A_1{:}D_1, A_2{:}D_2, A_3{:}D_3, \ldots, A_n{:}D_n\}$. Relacja ***R*** o schemacie ***S*** jest zaś *zbiorem odwzorowań*, które są określone na nazwach atrybutów i o wartościach należących do odpowiadającym tym atrybutom dziedzin. Stąd relacja ***R*** jest zbiorem ***n***-krotek:

$$(A_1{:}d_1, A_2{:}d_2, A_3{:}d_3, \ldots, A_n{:}d_n), \text{ gdzie } d_1 \in D_1, d_2 \in D_2, d_3 \in D_3, \ldots, d_n \in D_n.$$

W skład ***n***-krotki wchodzą elementy złożone z atrybutu oraz jego wartości. Gdy relację zapisujemy w postaci tabeli, nazwy atrybutów umieszczamy w nagłówkach kolumn, natomiast krotki stają się wierszami postaci $(d_1, d_2, d_3, \ldots, d_n)$. Każda z tych wartości pochodzi z odpowiadającej jej dziedziny. Widać więc, że relację możemy uważać za pewien podzbiór iloczynu kartezjańskiego dziedzin tej relacji, a fizyczną reprezentacją takiej relacji będzie tabela.

Gdy już mamy dane schematy relacji ***R1, R2, R3****, ...,* ***Rn,*** możemy zdefiniować *schemat relacyjnej bazy danych* ***R*** (schemat relacyjny ***R***) jako:

$$R = \{R1, R2, R3, \ldots, Rn\},$$

a więc schematem relacyjnej bazy danych nazywamy zbiór schematów relacji o różnych nazwach.

Przykład 1-9
Niech dana będzie relacja (tabela) ***R***, składająca się z atrybutów (kolumn) $A_1, A_2, \ldots, A_n$. Schemat relacji zapisujemy w sformalizowanej formie w następujący sposób: $R\ \{A_1, A_2, \ldots, A_n\}$. Stopień tej relacji wynosi ***n***.

Przykład 1-10
Relacja (tabela) *Studenci* może być przedstawiona w postaci *Studenci {Id_Studenta, Nazwisko_Studenta, Imię_Studenta}* lub w następującej formie graficznej:

Studenci
Id_Studenta
Nazwisko_Studenta
Imię_Studenta

Definicja 1-28
Nazwy atrybutów są ***nagłówkami*** kolumn relacji (pól tabeli).

Przykład 1-11
Nagłówek kolumn relacji *Studenci* ma postać:

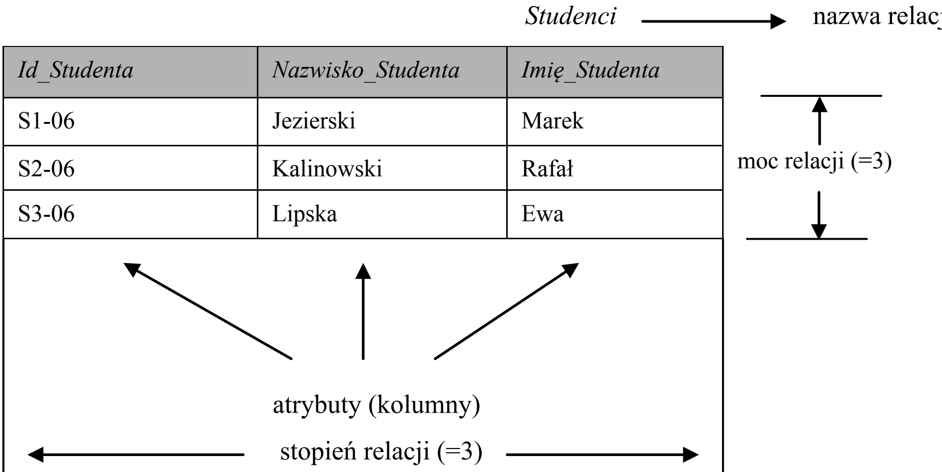
Id_Studenta	*Nazwisko_Studenta*	*Imię_Studenta*

Jeżeli wpiszemy do tej relacji (tabeli) kilka odpowiednich wartości, otrzymamy rzeczywistą relację przedstawioną na rys. 1.23.

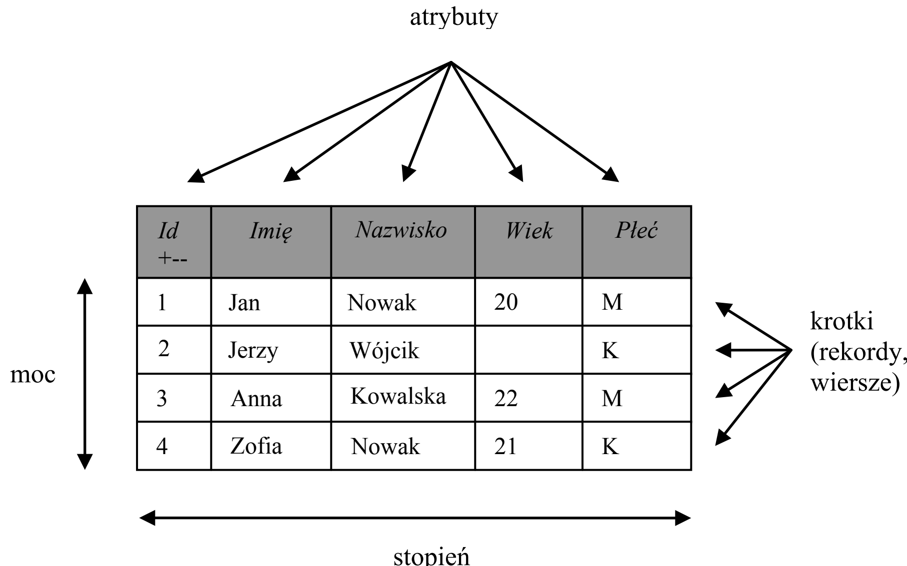
Dla atrybutu *Id_Studenta* dziedziną jest zbiór wszystkich (zastosowanych w danej uczelni) identyfikatorów studentów. Identyfikatorem może być np. numer indeksu.

Przykład 1-12
Relacja *Klienci* opisywana przez schemat *Klienci* {*Id, Imię, Nazwisko, Wiek, Płeć*} może mieć następujący wygląd i parametry (rys. 1.24).

Relacja na ostatnim rysunku ma pięć atrybutów, a więc jej stopień wynosi pięć, a także cztery krotki, a więc jej moc jest równa cztery (moc relacji jest równa 4 dla obecnego stanu relacji *Klienci*, ulegnie ona zmianie wraz z dodawaniem i usuwaniem krotek, a więc wraz ze zmianą stanu relacji).



Rys. 1.23. Relacja *Studenci* i charakteryzujące ją parametry



Rys. 1.24. Przykład relacji *Klienci* z charakteryzującymi ją parametrami

W modelu relacyjnym każda z relacji (tabel) charakteryzuje się następującymi własnościami:

- każda relacja powinna mieć nazwę będącą unikatową w całej relacyjnej bazie danych;
- każdy atrybut (kolumna) relacji powinien posiadać nazwę będącą unikatową w tej relacji;
- każda „komórka”, będąca przecięciem wiersza i kolumny, powinna zawierać wartość *atomową* (niepodzielną w sensie semantycznym), a więc nie może być to *atrybut wielowartościowy* (jest to cecha charakterystyczna tzw. pierwszej postaci normalnej, która szerzej zostanie omówiona w rozdziale poświęconym normalizacji relacyjnych baz danych);
- przykładem atrybutu wielowartościowego może być następujący wpis w komórce relacji *Pracownicy* {*Id, Imię, Nazwisko, Wiek, Płeć, Adres*}, znajdujący się w polu *Adres*: Lublin, Nadbystrzycka 23 (składa się z wartości tekstowych i liczbowych);
- wszystkie wartości danego atrybutu muszą być określonego typu – powinny należeć do jednej dziedziny. Dziedziny te – chociaż nie wszystkie – powinny również uwzględniać wartość pustą (*NULL*), stosowaną często do określenia braku danych lub po prostu wartości nieznanej;

Wartość *NULL* nie jest równa wartości „0”. Są to dwa zupełnie różne elementy. Przykładowo: wartość pola *Wiek* w drugim rekordzie relacji na rys. 1.24 jest wartością *NULL*.

- w relacji kolejność kolumn nie ma znaczenia (co różni relację w modelu relacyjnym od relacji w znaczeniu ściśle matematycznym, gdzie kolejność taka jest ważna);
- w relacji kolejność wierszy nie ma znaczenia (tak jak nie ma znaczenia kolejność elementów w zbiorze);
- każda relacja musi się składać co najmniej z jednej kolumny i może mieć zero, jeden lub większą liczbę wierszy;
- każdy wiersz w relacji jest abstrakcyjnym opisem obiektu, który ma cechy określone w atrybutach relacji;

- wiersze w relacji nie mogą się powtarzać – muszą być *unikatowe*, co jest równoznaczne z tym, że muszą się różnić co najmniej wartością jednego atrybutu relacji (jedną kolumną);
- każda relacja powinna mieć tzw. *klucz*, a więc jedną lub kilka kolumn, które w sposób jednoznaczny identyfikują każdy wiersz w relacji. W relacji *Studenci* (Prz. 1-11) oczywiście takim kluczem jest atrybut *Id_Studenta*; w relacji *Klienci* (Prz. 1-12) – atrybut *Id*.

W ten sposób przeanalizowaliśmy podstawowe pojęcia modelu relacyjnego, które w rzeczywistości są składnikami pierwszej części modelu: *obiekty*. W poniższej tabeli (tab. 1.1) podane są formalne terminy relacyjne i odpowiadające im nieformalne równoważniki.

Tabela 1.1. Terminy relacyjne i odpowiadające im nieformalne równoważniki

Formalny termin relacyjny	*Nieformalny równoważnik*
Relacja	Tabela
Nagłówek relacji	Nazwa pól tabeli
Krotka	Wiersz (rekord)
Atrybut	Kolumna (pole)
Moc relacji	Liczba wierszy
Stopień relacji	Liczba kolumn
Dziedzina (domena)	Zbiór dopuszczalnych wartości
Klucz podstawowy	Jednoznaczny identyfikator wiersza

Innym, według E. Codda, celem tworzenia RMBD było opracowanie teoretycznych podstaw organizacji zarządzania bazami danych. W czasopiśmie „Computer World" sformułował on 12 zasad[21], którym powinna odpowiadać relacyjna BD (system zarządzania relacyjną BD, SZRBD; ang. RDBMS). Wymienione zasady przedstawimy w tab.1.2.

Tabela 1.2. Zasady SZRBD E. Codda

Nr	*Nazwa zasady*	*Opis zasady*	*Komentarz*
0	Tworzenia	Każdy system zarządzania relacyjnymi bazami danych musi wykorzystywać (wyłącznie) mechanizmy relacyjne	
1	Informacji	Wszystkie dane w relacyjnej bazie przedstawiane są jako wartości w tabelach. Dane nie mogą być przechowywane w żaden inny sposób	
2	Gwarantowanego dostępu	Poprzez użycie kombinacji wartości klucza podstawowego, nazwy tabeli i nazwy kolumny musi istnieć dostęp do dowolnej partii danych	Większość RBD przestrzega tej zasady poprzez użycie kluczy głównych
3	Brakujących informacji	Program musi obsługiwać wartości `Null`. Wartości te przedstawiają brakujące bądź nieokreślone informacje	Większość RBD obsługuje wartości `Null` dla brakujących informacji i jednocześnie pozwala uniknąć wprowadzania wartości `Null` poprzez użycie innych narzędzi

[21] W niektórych źródłach podawanych jest 13 zasad E. Codda. Dodatkowa (zerowa) zasada brzmi: Każdy program typu systemu zarządzania relacyjną bazą danych musi być w stanie zarządzać bazami danych jedynie za pomocą swoich zdolności relacyjnych. Jeśli system działa na zasadach operowania danymi *rekord-po--rekordzie*, nie możemy go nazywać systemem w pełni relacyjnym.

Nr	*Nazwa zasady*	*Opis zasady*	*Komentarz*
4	Katalogu systemu	Opis bazy danych lub „katalog” na poziomie logicznym przedstawiane są jako wartości tabelaryczne. Język relacyjny powinien móc działać na projekcie bazy danych w taki sam sposób, w jaki działa na danych przechowywanych w strukturze	
5	Kompletnego języka	Program typu SZRBD musi obsługiwać jasno określony język do operowania danymi, który w pełni obsługuje definiowanie i operowanie danymi, określanie widoku[a], ograniczenia integralności, ograniczenia transakcyjne i autoryzację[b]	Takim językiem jest SQL
6	Uaktualniania widoków	Wszystkie widoki mogą być systemowo uaktualnione. W rzeczywistym programie typu SZRBD większość (ale nie wszystkie) widoków może być uaktualniana	
7	Ustawiania poziomu uaktualnień	Program typu SZRBD powinien umożliwiać nie tylko pobieranie zestawów danych. Musi także wstawiać, aktualizować i usuwać dane jako zestaw relacyjny	
8	Fizycznej niezależności danych	Dane i aplikacja muszą być od siebie fizycznie niezależne. Odpowiedni program typu SZRBD lub „optymalizator” powinny móc śledzić fizyczne zmiany w danych. Przykładowo, aplikacje SZRBD nie powinny ulegać modyfikacji na skutek dodania bądź usunięcia z tabeli indeksu	
9	Logicznej niezależności danych	Gdy to tylko możliwe, aplikacja powinna być niezależna od zmian dokonywanych w tabelach podstawowych. Przykładowo, gdy tabele są łączone w widok, nie powinny następować zmiany w kodzie	
10	Niezależności integralności	Integralność danych musi być definiowalna w języku relacyjnym i przechowywana w katalogu. Ograniczenia w integralności danych mogą być wbudowane w aplikację, jednakże podejście to jest obce dla modelu relacyjnego. W modelu relacyjnym integralność powinna być naturalną cechą projektu bazy danych	
11	Niezależności podziału	Zdolności programu typu RDBMS nie będą ograniczone dzięki umieszczeniu jego komponentów w osobnych bazach danych	
12	Braku podwersji	Jeśli program RDBMS posiada język typu „jeden rekord jednocześnie”, język ten nie może być używany do omijania zasad integralności lub ograniczeń języka relacyjnego. Dlatego też nie wystarczy, by zasady relacyjne zarządzały programem typu RDBMS, ale muszą być prawami nadrzędnymi	

a) Widokiem jest tabela tymczasowa, utworzona na potrzeby jednego lub kilku (grupy) użytkowników. Widoki (ang. *view*) szczegółowo zostaną omówione w dalszej części książki.
b) Celem autoryzacji jest potwierdzenie uprawnień użytkownika lub zasobu. Do tego aspektu wrócimy w kolejnych rozdziałach.

Podsumowując, możemy sformułować ważniejsze zalety i wady modelu relacyjnego.

Zalety:

- Oparty jest na solidnych podstawach teoretycznych.
- Tabele są niezależne, w przeciwieństwie do modeli hierarchicznego i sieciowego, gdzie występują połączenia wskaźnikowe.

- Niezależność danych, która pozwala na modyfikowanie struktury danych bez wpływu na istniejące programy, dzięki temu, że do tabel mogą być dodawane kolumny, tabele mogą być dołączane do bazy danych, a nowe relacje mogą być tworzone bez konieczności wprowadzania istotnych zmian do tabel.
- Dzięki wykorzystaniu języka SQL użytkownik określa jedynie warunki poszukiwanych danych, system natomiast zajmuje się pobieraniem danych spełniających żądanie.
- Użytkownik nie musi znać ścieżek dostępu do danych ani schematu całej bazy.

Wady:

- Niewielkie możliwości reprezentowania „rzeczywistych" obiektów. Rozbicie rzeczywistych obiektów pomiędzy wiele relacji i sposób fizycznego zapisu informacji odzwierciedlający to rozbicie wymuszają wykonywanie wielu złączeń podczas przetwarzania zapytań. Przyczynia się to również do wzrostu kosztów.
- Przeciążenie semantyczne, które jest wynikiem istnienia tylko jednej konstrukcji, służącej do przedstawiania danych i związków pomiędzy nimi (relacji). Jeśli chcielibyśmy przedstawić związek *wiele-do-wielu* pomiędzy encjami, musimy stworzyć trzy relacje. Nie można też rozróżnić encji od związków i rodzajów związków pomiędzy encjami.
- Jednorodna struktura danych, która oznacza, że krotki relacji muszą składać się z tych samych atrybutów, dane w kolumnach muszą mieć taką samą dziedzinę, a na przecięciu wierszy i kolumn musi znajdować się wartość atomowa. Prowadzi do rozbicia bardziej złożonych „rzeczywistych" obiektów i konieczności tworzenia większej liczby złączeń.
- Brak możliwości definiowania własnych operacji. W modelu relacyjnym można dokonywać jedynie ustalonych operacji na zbiorach, krotkach zdefiniowanych w specyfikacji SQL.
- Kłopotliwe zmiany schematu, które utrudniają elastyczność działania. Może je wprowadzać tylko administrator bazy. Pociągają za sobą również potrzebę modyfikacji aplikacji i zwiększenie kosztów.

1.3.4. Obiektowy model BD

Ostatnie zmiany w dziedzinie opracowywania oprogramowania są ściśle związane z wdrażaniem technologii obiektowo-orientowanych, które umożliwiają pracę na dowolnym poziomie abstrakcji.

Relacyjne bazy danych zostały powszechnie przyjęte w zastosowaniach biznesowych, takich jak: przetwarzanie zamówień, kontrola zasobów, bankowość, rezerwacje lotnicze. Jednak okazały się one niewystarczające dla aplikacji, przed którymi stoją wymagania odmienne od tradycyjnych zastosowań relacyjnych baz danych.

Przykładami takich zastosowań są:

- komputerowe wspomaganie projektowania (KWP) – przechowywanie danych związanych z projektowaniem części mechanicznych budynków, samochodów czy samolotów;
- komputerowe wspomaganie produkcji (KWP) – zawiera dane podobne jak w KWP, a ponadto informacje dotyczące produkcji etapowej lub ciągłej;

- komputerowe wspomaganie inżynierii oprogramowania (KWIO) – zawiera dane dotyczące różnych etapów rozwoju oprogramowania, od planowania aż po testowanie i konserwację;
- systemy zarządzania siecią – synchronizacja usług w sieciach komputerowych, np. zarządzanie komunikatami i planowanie sieci;
- systemy informacji biurowej (SIB) – nadzorowanie informacji biznesowych, takich jak: dokumenty, rachunki, korespondencja elektroniczna;
- systemy multimedialne – obsługa danych dowolnego formatu, np. tekstów, zdjęć, diagramów, filmów, nagrań dźwięku lub dokumentów będących połączeniem kilku tych formatów;
- publikacje elektroniczne – wymagają możliwości udostępniania, przechowywania i przetwarzania dokumentów multimedialnych;
- systemy informacji geograficznej (SIG) – zawierają informacje dotyczące przestrzeni i czasu;
- interaktywne i dynamiczne witryny WWW – np. sklepy internetowe;
- oprogramowanie medyczne i naukowe – np. informacje opisujące materiał genetyczny za pomocą modeli molekularnych.

Początek prac nad obiektowo-orientowanym modelem BD (OOMBD, ang. *Object Oriented Database Model*, OODBM) to druga połowa lat 80. Celem tych prac było stworzenie systemów komputerowego wspomagania projektowania (ang. *Computer Aided Design,* CAD) i komputerowego wspomagania produkcji (ang. *Computer Aided Manufacturing,* CAM). W r. 1991 w USA została stworzona grupa zarządzania obiektowymi BD (ang. Object Database Management Group), która skupia ponad 800 instytucji członkowskich, wśród których znajduje się m.in. Oracle i dzięki której w 1993 r. opracowany został standard OOMBD. W tym dokumencie *system zarządzania obiektowo-orientowaną* BD (SZOOBD) definiuje się jak SZBD*, który opiera się na właściwościach BD i obiektowo-orientowanych języków programowania*.

Jednym z podstawowych celów modelu obiektowego jest bezpośrednie odwzorowanie obiektów i powiązań między nimi, wchodzących w skład aplikacji na zbiór obiektów i powiązań w BD. Dzięki mechanizmom obiektowym można też zwiększyć niezależność danych od aplikacji poprzez przeniesienie procedur obsługi danych (w postaci metod) do systemu zarządzania bazą.

Podstawowe funkcje:

- definiowanie, tworzenie, modyfikowanie i udostępnianie obiektów;
- wykonywanie transakcji;
- *dziedziczenie*;
- przechowywanie.

Podstawowe założenia, jakie powinna spełniać obiektowa BD:

- system zarządzania modelem obiektowym musi mieć możliwość *definiowania* obiektów złożonych poprzez zastosowanie *konstruktorów do obiektów podstawowych*;
- konieczna jest implementacja tożsamości obiektów, czyli obiekty muszą być jednoznacznie identyfikowane niezależnie od wartości ich atrybutów;

- możliwość stosowania *hermetyzacji* poprzez zapewnienie programiście dostępu jedynie do metod;
- system musi realizować *typy* lub *klasy*;
- możliwość dziedziczenia przez *podtypy* i *podklasy* odpowiednio po *nadtypach* i *nadklasach* oraz przesłanianie i przeciążanie;
- musi być zaimplementowane *wiązanie dynamiczne*;
- *język manipulowania danymi* musi być obliczeniowo zupełny;
- użytkownik musi mieć możliwość *definiowania własnych typów* na podstawie już istniejących;
- musi być zapewniona *trwałość* danych oraz ich *odtwarzalność*;
- system musi mieć możliwość sprawnego przetwarzania dużych ilości informacji, czyli powinien istnieć *mechanizm efektywnego zastosowania pamięci dyskowej*;
- system musi obsługiwać *wielodostęp*;
- system musi posiadać *prosty mechanizm zapytań*;
- cechą opcjonalną może być *dziedziczenie wielokrotne*, *transakcje projektowe* oraz *wersje*.

System zarządzania obiektową bazą danych (SZOBD) obsługuje również zarządzanie transakcjami i kontrolę dostępu do danych przez *uprawnionych* do tego użytkowników. Powinien on także spełniać wymogi zasad *bezpieczeństwa*.

Krótko przeanalizujemy podstawowe pojęcia, na których opiera się model.

Definicja 1-29

Obiektem jest jednoznacznie identyfikowalny element, składający się zarówno z atrybutów opisujących stan „rzeczywistego obiektu", jak i związanych z nim przekształceń.

Przykład 1-13

Przykładem obiektu, tak jak w RMBD, może być *Student*, który przedstawiony jest na rys. 1.25.

Każdy obiekt może być zdefiniowany i przechowywany niezależnie od pozostałych. Jest on opisywany przez atrybuty, które dzielimy na *proste* i *złożone*. *Atrybut prosty* ma typ podstawowy, np. liczby rzeczywiste. *Atrybut złożony* może zawierać *kolekcje* lub odwołania do innych obiektów.

Atrybuty będące częścią definicji klasy poprzez przypisywane im wartości tworzą stan obiektu. Jak widać z rys. 1.25, atrybuty obiektu są analogiczne do atrybutów (kolumn) krotki relacji w relacyjnych BD. Dziedziną atrybutu może być jakakolwiek klasa, wliczając w to klasy *wartości pierwotnych* (np. *integer*, *string*).

System w momencie tworzenia obiektu generuje *unikalny identyfikator obiektu*, który jest niezmienny przez cały czas istnienia obiektu i niezależny od wartości jego atrybutów. Warto zwrócić uwagę na to, że w relacyjnej bazie danych unikalność krotki jest zapewniona poprzez *klucz*. Niestety, jest on unikalny tylko w obrębie relacji, a nie w całym systemie, co nie gwarantuje poziomu tożsamości, jaka jest wymagana w obiektowej bazie danych.

Student

Nazwisko: Nowakowska
Imię: Anna
Data-Ur: 12.03.1981r.
Pesel: 81031213226
Miasto: Lublin

Rys. 1.25. Definiowanie obiektu w OOBD

Wartość atrybutów obiektu można zmieniać bez ingerencji w jego tożsamość i wartość identyfikatora, gdyż jest on niezależny od zawartości obiektu. Jednak dwa obiekty mogą mieć różne identyfikatory, ale użytkownik może je postrzegać jako takie same za względu na jednakową wartość atrybutów.

Dwa obiekty są identyczne (równoważne) wtedy i tylko wtedy, kiedy są tym samym obiektem, czyli mają takie same identyfikatory.

Definicja 1-30

Kolekcja jest uporządkowanym ciągiem elementów tego samego typu, które mają tylko jeden wymiar i indeksowane są liczbami całkowitymi.

Każdy element kolekcji ma unikalny wskaźnik, określający jego położenie w kolekcji. Kolekcje mogą być przekazywane poprzez parametry procedur i funkcji. Przykładowo, w bazie danych *Oracle* są dwa rodzaje kolekcji:

- *tabele zagnieżdżone* – wartość typu TABLE;
- *tablice zmiennej długości* – wartości typu VARRAY.

Definicja 1-31

Klasa łączy obiekty, które posiadają takie same atrybuty i reagują na te same komunikaty.

Obiekty należące do klasy są nazywane *instancjami* tej klasy. Mają one własną wartość każdego atrybutu, tak jak to pokazano na rys. 1.26.

Definicja 1-32

Metoda opisuje zachowanie obiektu i składa się z nazwy, listy argumentów oraz implementacji wykonującej odpowiednie działania.

Służą one do zmiany stanu obiektu poprzez modyfikację wartości jego atrybutów lub do odczytania wartości wybranych atrybutów.

Metoda może być procedurą, funkcją lub operacją przypisaną do klasy obiektów i dziedziczoną przez jej podklasy.

Definicja 1-33

Komunikaty są to sposoby porozumiewania się obiektów, inaczej mówiąc, są to wiadomości wysłane od jednego obiektu do drugiego, zawierające polecenie wykonania jednej z metod.

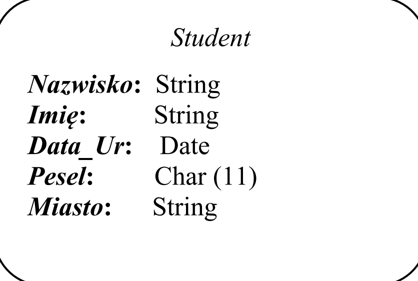
Nazwa użyta w komunikacie jest nazwą wywoływanej metody. Źródłem komunikatu jest działający aktualnie program, w szczególności może to być wykonywana aktualnie metoda. Komunikat może posiadać parametry, których może być kilka.

Obiekt otrzymujący komunikat wykonuje odpowiednią metodę, która może zmienić jego stan (patrz: rys. 1.27).

Definicja 1-34

Dziedziczenie pozwala zdefiniować klasę jako szczególny przypadek innej, bardziej ogólnej klasy, która ma podobne, ale nie identyczne, atrybuty i metody.

Klasę stanowiącą szczególny przypadek nazywamy *podklasą*, a klasę bardziej ogólną – *nadklasą*. Na rys. 1.28 podany jest przykład definicji *nadklasy* i *podklasy.*



Rys. 1.26. Przykład definiowania klasy

Student

Nazwisko: String
Imię: String
Data_Ur: Date
Pesel: Char (11)
Miasto: String

Dodaj ( );
Usuń ( );
Modyfikuj ( );

Rys. 1.27. Przykład zastosowanych metod

Proces tworzenia *nadklasy* nazywamy *generalizacją*, a *podklasy – specyfikacją*. *Podklasa* dziedziczy wszystkie własności *nadklasy*, które można przedefiniować, a dodatkowo posiada swoje własne metody i atrybuty. *Nadklasa* może być również podklasą innej klasy, w ten sposób powstaje hierarchia klas. Dziedziczenie pojedyncze oznacza, że podklasa dziedziczy tylko z jednej *nadklasy*.

Pojęcie dziedziczenia może stwarzać pewne problemy, takie jak konflikty nazw.

Definicja 1-35

Enkapsulacja lub ***zapakowanie*** (ang. *encapsulation*) oznacza zawarcie w obiekcie zarówno struktury danych, jak i operacji działających na obiekcie.

Enkapsulacja jest osiągana dzięki abstrakcyjnym typom danych, dzięki którym obiekt składa się z interfejsu, zawierającego opis operacji wykonywanych na obiekcie, i części implementacyjnej, zawierającej struktury danych oraz funkcje realizujące operacje opisane w interfejsie.

Użytkownicy OOBD mają dostęp tylko do interfejsu. Implementacja abstrakcyjnych typów danych może być zmieniana bez wpływu na aplikacje z nich korzystające, co zapewnia logiczną niezależność danych.

Kolejnym istotnym pojęciem dotyczącym analizowanego problemu jest *polimorfizm*.

Definicja 1-35

Polimorfizm w terminologii obiektowej oznacza możliwość istnienia wielu metod o takiej samej nazwie, powiązanych z możliwością wyboru konkretnej metody podczas czasu wykonania (dynamicznego wiązania).

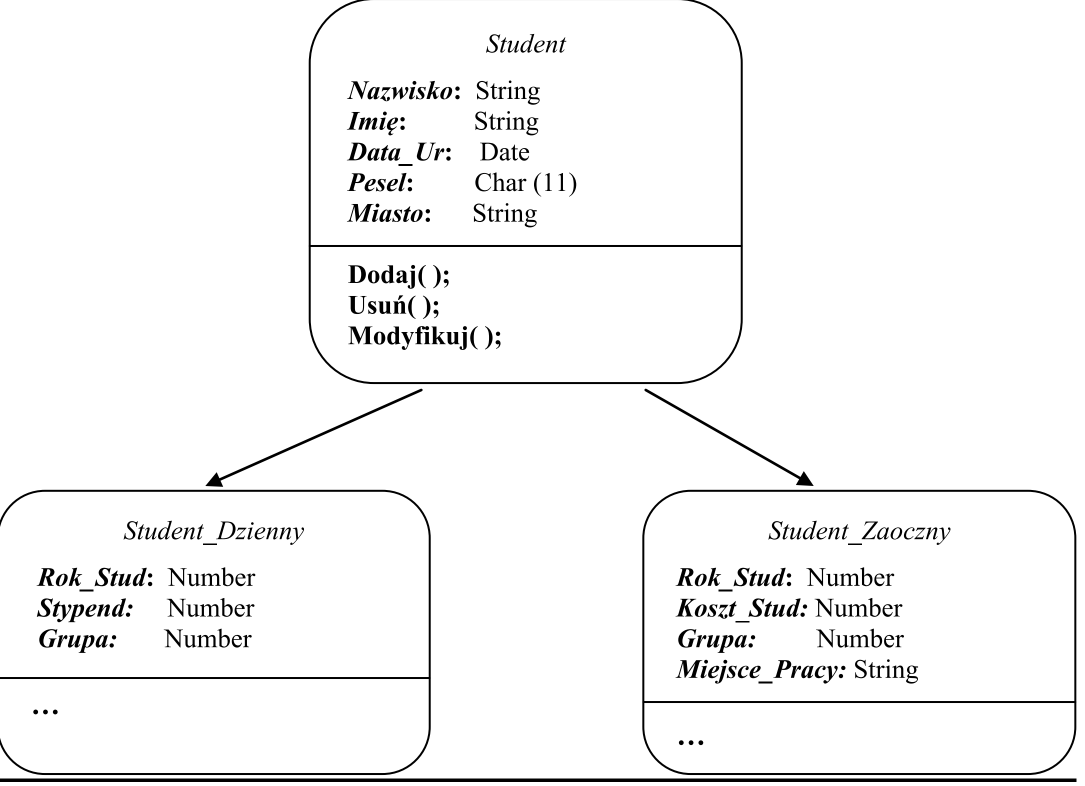
Wybór nazwy jest określany wyłącznie jej zewnętrznym, pojęciowym znaczeniem w ramach danej klasy obiektów. Wybór ten nie jest uwarunkowany własnościami lub istnieniem innych klas. Identyczny komunikat wysłany do różnych obiektów może wywołać różne metody.

SZOBD opiera się na:

- języku definiowania obiektów (JDO, ang. *Object Definition Language*, ODL);
- języku zapytań obiektowych (JZO, ang. *Object Query Language*, OQL);
- połączeniach z językami programowania.

  Podstawą JDO jest język definiowania interfejsów (ang. *Interface Definition Language*, IDL), który został krótko opisany we wspomnianym wyżej nieoficjalnym standardzie ODMG (ODMG-93).

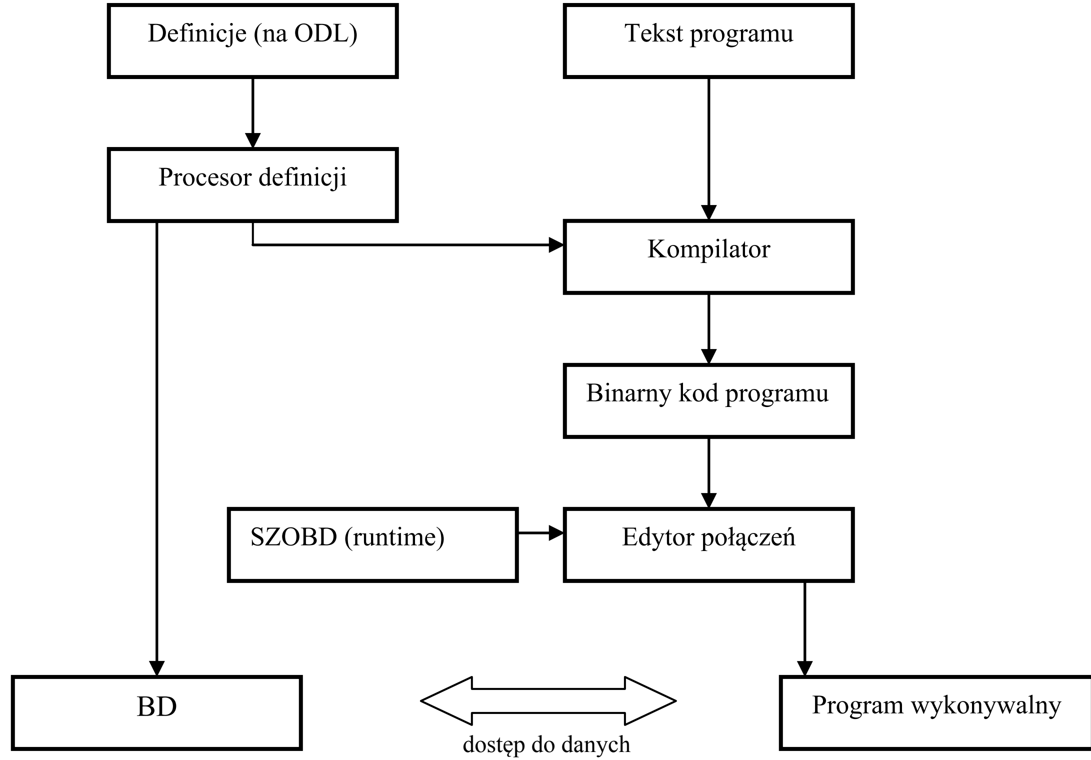
  Język zapytań obiektowych został opisany w formie deklaratywnej. Niestety, do tej pory nie opracowano standardu JZO. Obecnie wykorzystywaną semantyczną podstawą JZO jest język SQL.



Rys. 1.28. Hierarchia klas i dziedziczenie

Powiązanie z SZOBD opiera się na zdefiniowaniu wersji ODL na bazie składni języka C++.

Na rys. 1.29 zaprezentowano strukturę typowego SZOBD.



Rys. 1.29. Struktura typowego SZOBD

Programista wykorzystuje definicje schematów BD na bazie ODL lub ich odwzorowanie w niektóre języki programowania, np. w C++. Program użytkownika (aplikacja) realizowany jest w tym samym języku programowania, uzupełnionym o elementy manipulowania danymi. Definicje schematów BD i kod programu kompiluje się i opracowuje za pomocą *edytora połączeń*. W wyniku tego uzyskujemy program gotowy do współdziałania z BD.

Poza zaletami obiektowych baz danych istnieje wiele powodów powstrzymujących ich dynamiczny rozwój i powodujących, iż systemy relacyjnych baz danych wciąż posiadają znacznie większy udział w rynku.

Do powodów tych zaliczyć można:

- *brak oficjalnych standardów* – jak już podkreśliliśmy, do tej pory nie ma „właściwego” standardu na OQL;
- *niedoskonała teoretyczna baza* do formalizacji procedur i operacji na danych;
- *niedojrzałość technologii* – wiele organizacji wykorzystuje obecnie OSZBD do aplikacji o mniejszym znaczeniu z uwagi na ryzyko wiążące się z wykorzystaniem nowej technologii;
- *niestabilność producentów* – producenci OSZBD wydają się relatywnie małymi firmami, co powstrzymuje niektóre organizacje przed inwestowaniem w te systemy z obawy, że w niedalekiej przyszłości firmy te mogą zniknąć z rynku;

- *brak wykwalifikowanego personelu* – niestety, wciąż brakuje informatyków wykwalifikowanych w obsłudze obiektowych systemów zarządzania bazami danych;
- *koszty konwersji* – niebagatelną przeszkodę stanowią koszty konwersji na nowy system. Należy tu wziąć pod uwagę koszty nowego oprogramowania i jego instalacji, a także w przypadku firm już istniejących na rynku inwestycje poczynione dotychczas w systemy relacyjnych baz danych.

1.3.5. Obiektowo-relacyjny model BD

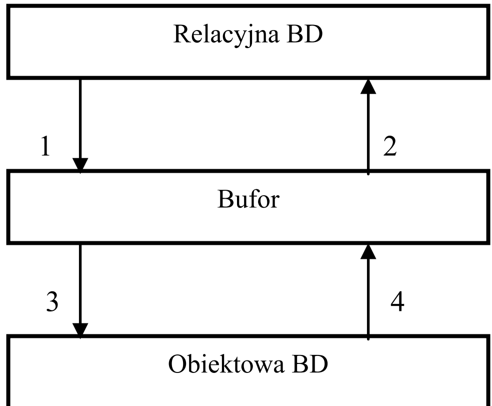
Z poprzedniej analizy i porównania RMBD z OOMBD wynika, iż technologia obiektowa i tradycyjne podejście relacyjne mają różne dziedziny zastosowań. Jeżeli dane mają formę krótkich, prostych pól o stałej długości, to optymalnym rozwiązaniem będzie zastosowanie relacyjnej BD. Natomiast jeżeli dane posiadają strukturę dynamiczną, definiowaną przez użytkownika (typu multimedia i in.), to zaprezentowanie ich w postaci tabel będzie rozwiązaniem nieefektywnym i nieoptymalnym.

Tak więc dochodzimy tu do jednego z ważniejszych problemów, dotyczącego wymiany danych pomiędzy dwoma różnymi SZBD. Około 10-13 lat temu najprostszym rozwiązaniem problemu było zastosowanie specjalnych elementów programistyczno-sprzętowych, stanowiących bufor (pomost) pomiędzy RBD a OOBD. Przykładowy schemat tego typu połączenia pokazano na rys. 1.30.

Na rysunku linie 1, 3 oznaczają przesyłanie danych z relacyjnej do obiektowej BD, linie 2 i 4 – w kierunku przeciwnym. W obu przypadkach bufor konwertuje dane na właściwy format. Najistotniejszą wadą tego podejścia jest bardzo wyraźne zmniejszenie efektywności całego systemu.

Inną metodą połączenia zalet obu modeli BD w jednym systemie było pojawienie się na rynku modelu hybrydowego – *obiektowo-relacyjnego* (ORMBD).

Według opinii jednego z najbardziej znanych specjalistów w dziedzinie teorii BD, Stonebrakera, najczęściej używane SZBD (w porównaniu z systemami plikowymi) można przedstawić tak jak zaprezentowano na rys. 1.31.



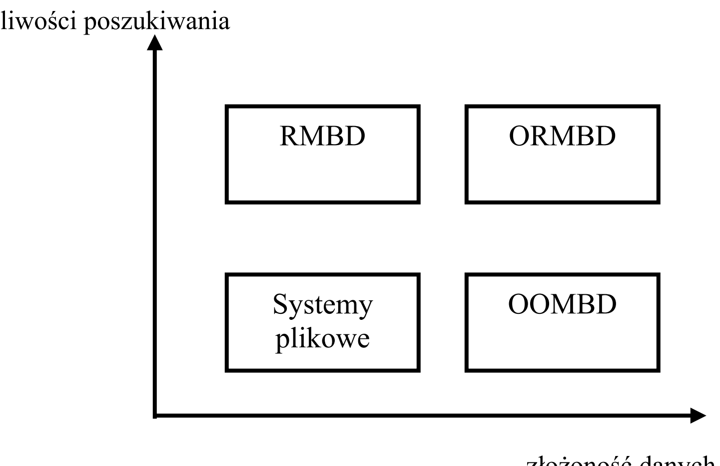
Rys. 1.30. Przykładowy schemat eksportu/importu danych pomiędzy RBD a OOBD

Podana klasyfikacja daje możliwość wyjaśnienia tendencji rozwoju SZBD.

Łatwo zauważyć, że istnieją trzy praktyczne i teoretyczne podejścia do budowy ORMBD:

- na platformie jądra obiektowo-relacyjnego SZBD;
- na platformie „nadbudowy” nad jądrem obiektowo orientowanego SZBD;
- na platformie „nadbudowy” nad jądrem relacyjnego SZBD.

W pierwszym wypadku powinnyśmy mieć jądro SZBD, którym można sterować ze strony bazy metadanych na podstawie tabel; taka baza metadanych oprócz tego powinna przechowywać dane o typach, funkcjach, operatorach i dziedziczeniu. Wszystkie istniejące relacyjne SZBD posiadają metadane, które można opisać wyłącznie za pomocą tabel. Ponadto istniejące typy, operatory i funkcje SQL są sztywno wbudowane w jądro. Dlatego też przypadek ten może być realizowany w perspektywie długoterminowej.



Rys. 1.31. Klasyfikacja SZBD wg Stonebrakera

Druga możliwość polega głównie na rozszerzeniu jądra SQL nad językiem C++ lub innym językiem orientowanym obiektowo. Jednak i tutaj istnieją pewne istotne problemy. Między innymi bardzo trudno jest zrealizować (także ze względu na wysokie koszty) wymagany poziom ochrony danych; ponadto stworzenie nowej funkcji, nowego operatora albo typu danych wymaga przekompilowania całego programu.

Idea stworzenia „nadbudowy” między klientem a relacyjnym jądrem została praktycznie zrealizowana przez tak znane firmy jak Oracle, Informix czy IBM.

Zadania do samokontroli

1. Podać definicje terminów i pojęć:
   - model BD;
   - relacja;
   - dziedzina;
   - polimorfizm w BD;
   - dziedziczenie w BD;
   - klasa i metoda w BD;
   - hermetyzacja w BD.
2. Podać klasyfikację modeli BD.
3. Podać charakterystykę:
   - HMBD;
   - SMBD;

- RMBD;
- OOMBD;
- ORMBD.

4. Podać schemat BD *Szkoła*, przechowującej dane o *Uczniach*, *Nauczycielach*, *Klasach*, *Zajęciach*, *Salach*, *Rodzicach_Uczniów* w modelu hierarchicznym.
5. Podać schemat BD *Szkoła*, przechowującej dane o *Uczniach*, *Nauczycielach*, *Klasach*, *Zajęciach*, *Salach*, *Rodzicach_Uczniów* w modelu sieciowym.
6. Podać schemat BD *Szkoła*, przechowującej dane o *Uczniach*, *Nauczycielach*, *Klasach*, *Zajęciach*, *Salach*, *Rodzicach_Uczniów* w modelu relacyjnym.
7. Porównać schematy modeli BD z Prz. 4-6.

2. Składniki modelu relacyjnego

Relacyjny model BD składa się z trzech zasadniczych części:

- ***obiektów,*** czyli struktury danych;
- ***integralności,*** czyli spójności danych;
- ***manipulowania danymi,*** czyli operacji na danych.

W podrozdz. 1.3.3 została dość szczegółowo przeanalizowana pierwsza (podstawowa) część modelu. Teraz przechodzimy do analizy pozostałych części.

2.1. Integralność danych

2.1.1. Klucze relacyjne

Pojęcie *klucza* jest jednym z najważniejszych pojęć w analizie integralności modelu relacyjnego.

W relacyjnym modelu BD wyróżniamy trzy podstawowe typy kluczy:

- kandydujący (potencjalny), KK (ang. candidate key, CK);
- podstawowy (główny), KP (ang. primary key, PK);
- obcy, KO (ang. foreign key, FK).

Definicja 2-1
Kluczem (kandydującym) relacji (tabeli) nazywamy jeden lub większą liczbę atrybutów (kolumn), które w sposób jednoznaczny identyfikują każdy rekord.

Pole (lub pola) tabeli, pełniące wyżej opisaną funkcję, nazywa się ***polem kluczowym.***

Definicja 2-2
Zbiór atrybutów $\boldsymbol{K = (A_i, A_j,...,A_k)}$ relacji ***R*** jest ***kluczem kandydującym*** wtedy i tylko wtedy, kiedy spełnia dwa podstawowe i niezależne warunki:

- ***unikatowość***: w dowolnym czasie żadne dwa różne rekordy nie mają tej samej wartości dla $\boldsymbol{A_i, A_j,...,A_k}$, ;
- ***minimalność***: żaden z atrybutów nie może być usunięty z ***K*** bez naruszenia unikatowości.

Praktycznie wymóg *unikatowości można traktować* w następujący sposób*: klucze same w sobie muszą mieć wartości unikatowe, gdyż tylko wtedy gwarantują jednoznaczną identyfikację rekordu.*

W przypadku kluczy składających się z większej niż jeden liczby atrybutów nie każdy z nich musi być unikatowy – to połączenie atrybutów wchodzących w skład klucza kandydującego musi być unikatowe.

Minimalność może również oznaczać, że klucz kandydujący zawiera minimalną liczbę atrybutów umożliwiających jednoznaczną identyfikację rekordów. W jego skład nie wchodzą więc atrybuty nadmiarowe, stąd też nie zawiera on podzbioru właściwego umożliwiającego taką jednoznaczną identyfikację.

Z ostatnich definicji wynika, że:

***Wartości w kluczu kandydującym nie mogą być opcjonalne*, gdyż takie wartości muszą istnieć zawsze, co odnosi się do każdego z atrybutów wchodzących w skład klucza.**

Ponieważ wartość *niezdefiniowana* (null) oznacza brak wartości, nie ma możliwości odwołania się do rekordu, w którym klucz kandydujący zawierałby taką wartość.

Każdy atrybut relacji (każde pole w tabeli) musi być *funkcjonalnie zależny* od wartości klucza kandydującego (zależności funkcjonalne pomiędzy atrybutami (polami tabeli) będą szerzej omówione w dalszej części).

Definicja 2-3

Klucz składający się z dwóch lub więcej atrybutów nazywamy ***złożonym***.

Przykład 2-1

Relacja (tabela) *Studenci_Wydziału {Wydział, Nazwisko, Imię, Drugie_Imię}* posiada wartości odpowiadające tab. 2.1.

Tabela 2.1. *Studenci_Wydziału*

Wydział	*Nazwisko*	*Imię*	*Drugie_ Imię*
Matematyczny	Jezierski	Marek	Paweł
Informatyczny	Kalinowski	Rafał	Bartosz
Filozoficzny	Lipska	Ewa	Maria
Matematyczny	Jezierska	Ewa	Maria
Informatyczny	Kalinowski	Piotr	Bartosz
Matematyczny	Kalinowski	Marek	Paweł

Zakładając, że nie może być wśród studentów dwóch osób z takimi samymi nazwiskami, imionami i drugimi imionami (chociaż, w rzeczywistości, jest to możliwie), widzimy, że analizowana tabela posiada złożony KK, składający się z pól *Nazwisko, Imię, Drugie_ Imię.*

Przykład 2-2

Rozważmy relację (tabelę) *Pracownicy*, składającą się z następujących atrybutów (pól): *Id_Pracownika, Imię, Nazwisko, Data_Urodzenia, Wzrost.* Widzimy, że jedynym atrybutem, który w sposób jednoznaczny będzie identyfikował nam każdą krotkę w tej relacji, jest *Id_Pracownika* i może być on tutaj wybrany jako KK. Pozostałe atrybuty, takie jak *Imię* czy *Nazwisko*, mogą się powtarzać dla różnych pracowników, tak samo jak *Data_Urodzenia* czy *Wzrost.* Również nie ma potrzeby w tym przypadku wybierać jakiegokolwiek klucza złożonego, a więc klucza składającego się z większej liczby atrybutów, gdyż każdy z takich kluczy musiałby zawierać atrybut *Id_Pracownika*, aby jednoznacznie identyfikować każdy rekord. A ponieważ atrybut *Id_Pracownika* jest jednoznaczny samodzielnie, każdy

z kluczy zawierałby więc podzbiór właściwy, jednoznacznie identyfikujący rekordy, stąd nie spełniałby warunku *minimalności klucza kandydującego.*

Przykład 2-3

Załóżmy, że mamy relację *Wizyty*, składającą się z atrybutów: *Id_Lekarza* – jednoznaczny identyfikator lekarza przyjmującego pacjenta na wizytę, *Id_Pacjenta* – jednoznaczny identyfikator pacjenta przychodzącego na wizytę, a także *Data_Wizyty*. Atrybut *Id_Lekarza* nie może być tutaj kluczem kandydującym, ponieważ dany lekarz może przyjmować wielu pacjentów, a więc wiele rekordów będzie miało taką samą wartość w tym atrybucie. Również *Id_Pacjenta* jest złym wyborem – jeden pacjent może przyjść na wizyty do różnych lekarzy, a więc problem braku jednoznaczności wystąpi i tutaj. *Data_Wizyty* również musi być odrzucona – w danym dniu może odbywać się wiele wizyt. Jedynym wyjściem jest tutaj wybór KK złożonego, który będzie składał się ze wszystkich trzech atrybutów. Taki wybór jest oczywiście słuszny, ale tylko przy założeniu, iż każdy pacjent u danego lekarza może mieć tylko jedną wizytę w jednym dniu, co na ogół jest spełnione.

Relacja może mieć kilka KK.

Przykład 2-4

Relacja Studenci_1_roku {Id, Paszport, Indeks, Nazwisko, Imię} posiada trzy równoważne KK: Id, Paszport, Indeks.

Jeden (i tylko jeden) KK będzie pełnił funkcję *klucza podstawowego* (KP). Pozostałe będą *kluczami alternatywnymi.*

Jeżeli np. w relacji *Studenci_1_roku kluczem podstawowym* będzie atrybut *Id,* to pozostałe (*Paszport, Indeks)* są *kluczami alternatywnymi*.

Ponieważ *klucz podstawowy* jest jednym z KK, więc automatycznie musi spełniać wszystkie warunki i cechy klucza kandydującego.

Klucz podstawowy jest najważniejszym z kluczy, reprezentuje on bowiem swoją tabelę w całej strukturze danych.

Dzięki KP możemy tworzyć logiczne powiązania między tabelami, a także w sposób jednoznaczny identyfikować rekordy relacji.

Warto również podkreślić, że KP bierze udział w wymuszaniu *integralności danych* w całej bazie na różnych poziomach, co zostanie omówione w dalszej części książki.

We wspomnianym logicznym połączeniu pomiędzy relacjami (tabelami BD) bierze udział *klucz obcy*.

Definicja 2-4

Klucz obcy (KO) jest to zbiór atrybutów jednej relacji (pól jednej tabeli), będący kopią klucza podstawowego innej relacji (innej tabeli).

Mówiąc o kluczach obcych, zawsze należy mieć na uwadze, że odnoszą się one do dwóch relacji (tabel), a dokładniej – do logicznego powiązania pomiędzy danymi wpisanymi do dwóch relacji (tabel).

Zakładamy że mamy dwie tabele: *T1* i *T2,* każda z nich posiada jeden KK – odpowiednio KK_1 i KK_2.

Z formalnego punktu widzenia *klucz obcy* tabeli *T2* można zdefiniować jako zbiór pól tabeli *T1*, będących kluczem kandydującym tej tabeli (KK_1), których wartość jest dokładnie równa odpowiednim wartościom zbioru pól tabeli *T2*.

Również możliwe jest, iż kluczem obcym tabeli *T1* może być zbiór pól tabeli *T2*, będących kluczem kandydującym tej tabeli (KK_2), których wartość jest dokładnie równa odpowiednim wartościom zbioru pól tabeli *T1*.

W zależności od tego, jakiego typu jest KK (*prosty* albo *złożony*), takiego typu będzie odpowiadający mu KO. Dodatkowo: *pole lub pola wchodzące w skład KO powinny posiadać dane tego samego typu co pole lub pola składające się na KK.*

2.1.2. Definicja i typy logicznego połączenia relacji (tabel)

Jak podkreśliliśmy wcześniej, logiczne powiązanie tabel w BD (ang. *relationship*) często określa się terminem *relacja między tabelami.*

W modelu relacyjnym termin *relacja* może oznaczać tabelę (ang. *relation*) albo logiczne powiązanie tabel (ang. *relationship*).

W przeanalizowanej wyżej części modelu relacyjnego zdefiniowano pojęcia *relacji między tabelami* (Def. 1-9, ..., Def. 1-15). Tutaj dokładniej omówimy niektóre aspekty związane z relacjami.

Wiemy, że relacyjna baza danych składa się z tabel, które zawierają informacje o konkretnych obiektach. Jednak same informacje pobrane z pojedynczej tabeli są często dla nas niewystarczające i niezbędne jest pobranie (selekcja) danych zawartych w kilku tabelach. Generalnie baza składa się z wielu wzajemnie ze sobą powiązanych obiektów tworzących pewną całość. Wynika stąd konieczność wzajemnego powiązania tabel występujących w danej BD.

W relacyjnej BD *każda tabela* powinna być w relacji *przynajmniej z jedną* tabelą.

Mówiąc o relacjach między tabelami, *zawsze* analizujemy i definiujemy relację wyłącznie między dwiema tabelami.

Relacje pomiędzy tabelami są bardzo ważnym elementem relacyjnej bazy danych z następujących powodów:

- dzięki nim możemy łączyć tabele przechowujące powiązane ze sobą dane;
- pomagają w modyfikowaniu struktur tabel, a także redukują nadmiarowość – tworzenie relacji między tabelami wymaga niejednokrotnie modyfikowania struktur tabel, a także ogranicza liczbę nadmiarowych danych przechowywanych w bazie, dokonane zmiany wpływają często na większą efektywność tabel, jak i całej bazy;
- umożliwiają odczytywanie informacji jednocześnie z wielu tabel – relacje między tabelami umożliwiają łączenie pól pochodzących z różnych tabel i umieszczenie ich np. w tabeli wirtualnej (zwanej *perspektywą)*;
- poprawnie zdefiniowane pozwalają zachować *integralność referencyjną* – dzięki zachowaniu integralności referencyjnej baza jest spójna i bardziej efektywna.

Przykład 2-5

Załóżmy, że mamy tabelę *Kierownicy {Id_Kier, Nazw_Kier, Imię_Kier, Pensja_Kier}*, a także tabelę *Działy {Id_Działu, Nazwa_Działu, Nazw_Kier_Działu}*. Wtedy, rozpatrując je oddzielnie, możemy uzyskać informacje o wszystkich

kierownikach, a także działach w firmie, jednak nie będzie możliwe stwierdzenie, za jakie działy odpowiada dany kierownik. W tym celu należy dokonać powiązania między tabelami.

Takie powiązanie uzyskuje się poprzez umieszczenie w jednej tabeli (lub w większej liczbie tabel) kopii kolumny lub zbioru kolumn znajdujących się w drugiej tabeli. Te kolumny w tabeli drugiej są niczym innym jak KP tej tabeli. O tabeli, która zawiera kopię klucza podstawowego innej tabeli, powiemy, iż zawiera *klucz obcy*.

Aby powiązać tabele w Prz. 2-5, można KP tabeli *Kierownicy (Id_Kier)* umieścić w tabeli *Działy*. W tabeli *Działy* pole *Id_Kier* pełni rolę KO.

Atrybuty (pola) klucza obcego nie muszą mieć takiej samej nazwy co atrybuty (pola), których są kopiami. Ale teoria RMBD wymaga, aby KO był tego samego typu co KP, którego jest kopią.

Definicja 2-5
Tabela, której kopię KP umieszczamy w innej tabeli, nazywamy ***tabelą nadrzędną***.

W Prz. 2-5 tabelą nadrzędną jest tabela *Kierownicy*. Zwróćmy uwagę, że w tym przykładzie mamy do czynienia z relacją *jeden-do-jednego* (patrz Def. 1-13, 1-14, 1-15). W takim wypadku każda z dwóch łączonych tabel może być *tabelą nadrzędną*.

Definicja 2-6
Tabela, w której rozmieszczamy kopię KP innej tabeli, nazywamy ***tabelą podrzędną***.

W Prz. 2-5 tabelą podrzędną jest tabela *Działy*.

Jak wspomniano wcześniej, RMBD można opisać za pomocą trzech typów relacji między tabelami:

- *jeden-do-jednego* (1:1, patrz Def. 1-13, 1-14, 1-15),
- *jeden-do-wielu* (1:W, Def. 1-10, 1-11, 1-12),
- *wiele-do-wielu* (W:W, Def. 1-16, 1-17).

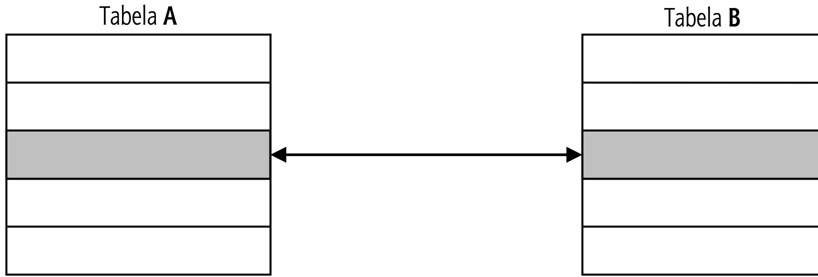
Relacja jeden-do-jednego
Modyfikując Def. 1-13, 1-14, 1-15, możemy zapisać w odniesieniu do RMBD następującą definicję:

Definicja 2-7
Relacja jeden-do-jednego pomiędzy tabelami ***A*** i ***B*** występuje wtedy, kiedy każdemu rekordowi z tabeli ***A*** jest przyporządkowany dokładnie jeden rekord z tabeli ***B*** i na odwrót – każdemu rekordowi z tabeli ***B*** jest przyporządkowany dokładnie jeden rekord z tabeli ***A***.

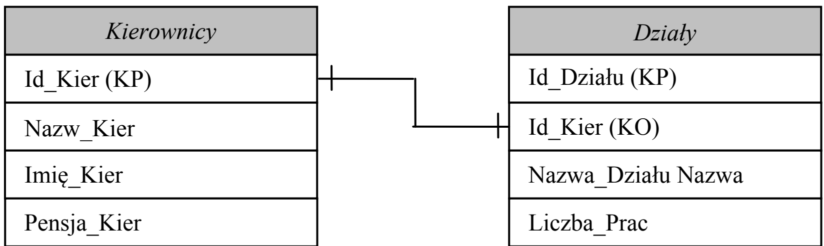
Schematycznie może być to przedstawione tak jak na rys. 2.1.

Wróćmy do Prz. 2-5. Mamy tabelę *Kierownicy* {Id_Kier, Nazw_Kier, Imię_Kier, Pensja_Kier}, a także tabelę *Działy* {Id_Działu, Nazwa_Działu, Nazw_Kier_Działu}.

Schemat połączenia tych tabel ilustruje rys. 2.2. Linie pionowe z obu stron oznaczają, że jest to relacja 1:1.



Rys. 2.1. Powiązanie między dwoma rekordami tabel w relacji *jeden-do-jednego*



Rys. 2.2. Relacja *jeden-do-jednego* między tabelami *Kierownicy* a *Działy*

Analizowane tabele z wprowadzonymi danymi mogą wyglądać tak jak na rys. 2.3. Zwróćmy uwagę, że w naszym przykładzie tabele powiązane są poprzez pola o tej samej nazwie.

Przykład 2-6

Krótko przeanalizujemy relację pomiędzy tabelami *Dyrektorzy* i *Departamenty*. Każdy dyrektor odpowiada za dokładnie jeden departament, jak również każdy departament ma dokładnie jednego dyrektora, dlatego też mamy tu do czynienia z relacją *jeden-do-jednego* (rys. 2.4).

Podsumowując, aby zdefiniować relację *jeden-do-jednego* należy kopię klucza podstawowego jednej tabeli umieścić w drugiej tabeli.

Relacja jeden-do-wielu

Najczęściej używanym typem połączenia jest relacja 1:w.

Definicja 2-8

Relacja jeden-do-wielu pomiędzy tabelami ***A*** i ***B*** występuje wtedy, kiedy pojedynczemu rekordowi z tabeli ***A*** jest przyporządkowany jeden lub wiele rekordów z tabeli ***B***, natomiast pojedynczemu rekordowi z tabeli ***B*** jest przyporządkowany dokładnie jeden rekord z tabeli ***A***.

Ogólny schemat relacji tego typu pokazany jest na rys. 2.5.

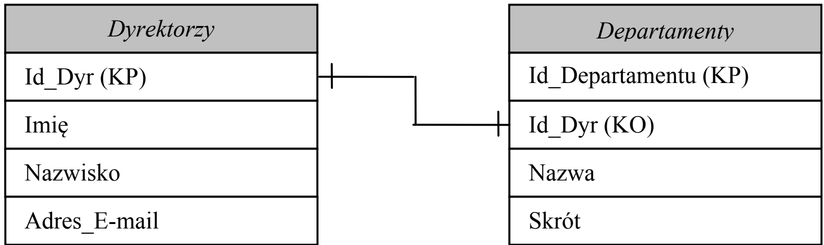
Kierownicy

Id_Kier	*Nazw_Kier*	*Imię_Kier*	*Pensja_Kier*
A1	Mojski	Jacek	7000
A2	Dobrowolska	Aldona	5500
C2	Piotrowski	Paweł	6000
C3	Nieścior	Marek	5000

Działy

Id_Działu	*Id_Kier*	*Nazwa_Działu*	*Liczba_Prac*
AU	A1	Automatyzacja	30
IN	A2	Informatyzacja	15
MI	C3	Mikroelektronika	10
MA	C2	Matematyka	25

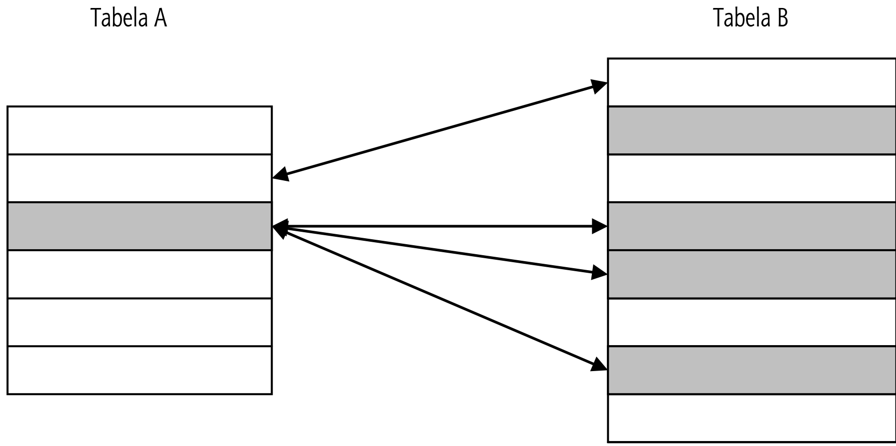
Rys. 2.3. Przykład powiązanych tabel przez relację 1:1



Rys. 2.4. Relacja *jeden-do-jednego* między tabelami *Dyrektorzy* a *Departamenty*

Relację *jeden-do-wielu* tworzymy w podobny sposób jak relację *jeden-do-jednego*. W tabeli leżącej po stronie 'wiele' należy umieścić kopię klucza podstawowego tabeli leżącej po stronie 'jeden'. Klucz ten staje się *kluczem obcym* w tabeli leżącej po stronie 'wiele'.

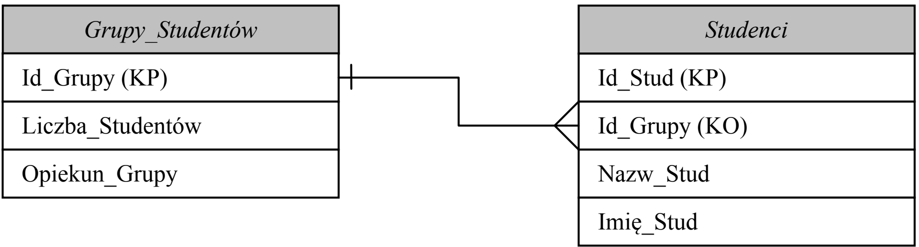
W relacji *jeden-do-wielu* tabela po stronie '*jeden*' zawsze jest tabelą *nadrzędną*.



Rys. 2.5. Ogólny schemat połączenia tabel przez relację 1:w

Przykład 2-7
Rozważmy relację między tabelami *Grupy_Studentów* a *Studenci* (została ona krótko omówiona w Prz. 1-5). Zakładamy, że pierwsza z tabel ma pola: *Id_Grupy* (KP), *Liczba_Studentów*, *Opiekun_Grupy*; druga – *Id_Stud* (KP), *Nazw_Stud*, *Imię_Stud.*

Diagram tego połączenia został pokazany na rys. 2.6.



Rys. 2.6. Diagram połączenia tabel *Grupy_Studentów* a *Studenci* przez relację 1:w

Na rys. 2.5. trzy linie wychodzące z jednego punktu przy tabeli *Studenci* oznaczają stronę 'wiele'. Na rys. 2.7 podany przykład odpowiada diagramowi na rys. 2.6.

Dzięki relacjom 1:w możemy w znacznym stopniu ograniczyć nadmiarowość danych. Wyobraźmy sobie, że chcielibyśmy przechowywać w jednej tabeli informacje o studentach oraz grupach. Wtedy tabela ta musiałaby mieć atrybuty opisujące zarówno grupy, jak i studentów. Zapisanie kilku studentów tej samej grupy wiązałoby się z koniecznością powtarzania we wszystkich rekordach tych samych informacji o grupie, co przy dużej liczbie studentów i grup byłoby niepotrzebnym rozrastaniem się rozmiarów bazy. Problem stanowiłoby również modyfikowanie czy usuwanie rekordów. Jeżeli chcielibyśmy zmodyfikować pewne dane o grupie, to musielibyśmy zmieniać rekordy opisujące studentów tej grupy. Natomiast usunięcie z bazy ostatniego studenta danej grupy powodowałoby również utracenie informacji o niej samej. Relacje *jeden-do-wielu* w znacznym stopniu ograniczają te problemy.

Relacja wiele-do-wielu

Przeformułowując Def. 1-16, 1-17, zapiszmy następującą definicję:

Definicja 2-9

Relacja wiele-do-wielu pomiędzy tabelami ***A*** i ***B*** występuje wtedy, kiedy pojedynczemu rekordowi z tabeli ***A*** jest przyporządkowany jeden lub wielu rekordów z tabeli ***B***, i na odwrót – pojedynczemu rekordowi z tabeli ***B*** jest przyporządkowany jeden lub wiele rekordów ***A***.

Ogólny schemat tego typu połączenia pokazany jest na rys. 2.8.

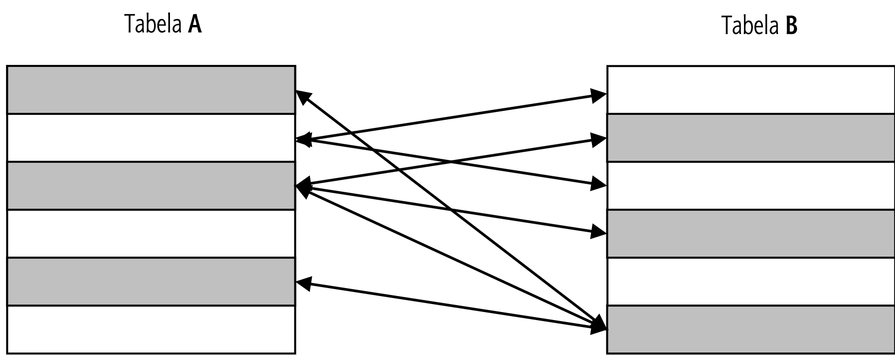
Grupy_Studentów

Id_Grupy	*Liczba_Studentów*	*Opiekun_Grupy*
34-16	4	Rybicki
44-16	3	Popławska
54-16	5	Piotrowski
64-16	4	Dąbrowski

Studenci

Id_Stud	*Id_Grupy*	*Nazw_Stud*	*Imię_Stud*
34-16_1	34-16	Mojski	Jacek
34-16_2	34-16	Dobrowolska	Aldona
34-16_3	34-16	Wiśniewski	Paweł
34-16_4	34-16	Nieścior	Marek
44-16_1	44-16	Zapała	Magdalena
		...	
54-16_5	54-16	Jakubowicz	Andrzej
64-16_1	64-16	Urbanowicz	Anna
64-16_2	64-16	Szyszka	Marek
64-16_3	64-16	Zygmunt	Stanisław
64-16_4	64-16	Kowalska	Anna

Rys. 2.7. Przykład powiązanych tabel poprzez relację 1:w



Rys. 2.8. Ogólny wygląd relacji *jeden-do-wielu*

W RMBD nie istnieje możliwość bezpośredniego zdefiniowania efektywnej relacji *wiele-do--wielu*.

Przykład 2-8

Przeanalizujemy ten typ relacji na przykładzie tabel *Wykładowcy* i *Studenci* (patrz Prz. 1.7 i rys. 1.16). Biorąc pod uwagę, że wykładowca prowadzi wykłady dla wielu studentów i student chodzi na wykłady kilku wykładowców, mamy w tym wypadku do czynienia z relacją w:w. Zakładamy również, że wykładowca ma wykłady tylko z jednego przedmiotu.

Niech tabele składają się odpowiednio z następujących pól: Wykładowcy {Id_Wyk, Nazw_Wyk, Imię_Wyk, Stopień_Naukowy, Przedmiot}, Studenci {Id_Stud, Nazw_Stud, Imię_Stud}. Kluczami podstawowymi są pola Id_Wyk i Id_Stud.

Definicja 2-10

Aby zdefiniować relację w:w, należy stworzyć ***tabelę pomocniczą*** **(*mieszającą*)**, w której powinny być rozmieszczone kopie KP obu łączonych tabel.

Na rys. 2.9 został zrealizowany w formie połączonych tabel Prz. 2.8. Tabela *Wykładowcy/Studenci* pełni rolę *tabeli pomocniczej*.

Jak widać z tego schematu, są studenci, którzy chodzą na wykłady z kilku przedmiotów. Tabela *Wykładowcy* posiada jednopolowy KP – *Id_Wyk*, tabela *Studenci* – również jednopolowy KP – *Id_Stud*. Ponadto oba te klucze (podstawowe) są kluczami obcymi w tabeli *Wykładowcy/Studenci* i jednocześnie tworzą złożony KP w tej *tabeli pomocniczej*.

2.1.3. Typy uczestnictwa tabel w relacji

Każda tabela będąca w relacji charakteryzuje się pewnym *typem uczestnictwa* w tej relacji. Typ ten określa, czy niezbędne jest istnienie jakiegoś wiersza w analizowanej tabeli dla wprowadzenia wiersza w tabeli leżącej po drugiej stronie relacji. Wyróżniamy dwa typy uczestnictwa: *obowiązkowe* i *opcjonalne*.

Definicja 2-11

Uczestnictwo tabeli w relacji jest ***obowiązkowe***, jeżeli w analizowanej tabeli niezbędne jest istnienie jakiegoś wiersza, aby wprowadzić dane do drugiej tabeli będącej w relacji.

Definicja 2-12

Uczestnictwo opcjonalne występuje w sytuacji, gdy w analizowanej tabeli nie musi istnieć żaden wiersz i dodawanie wierszy do drugiej tabeli jest dozwolone, niezależnie od liczby rekordów analizowanej tabeli.

Przykład 2-9

Przeanalizujmy teraz wcześniej przedstawiony przykład (rys. 2.4).

W tym przykładzie mieliśmy relację *jeden-do-jednego* pomiędzy tabelami *Dyrektorzy* i *Departamenty*. Aby utworzyć nowy departament w tabeli *Departamenty,* niezbędny jest wcześniej dodany rekord dyrektora odpowiedzialnego za ten departament w tabeli *Dyrektorzy*. Tak więc typ uczestnictwa tabeli *Dyrektorzy* w tej relacji jest *obowiązkowy*. Z drugiej strony możemy dowolnie dodawać rekordy dyrektorów do tabeli *Dyrektorzy*, nawet jeżeli nie stworzymy jeszcze departamentu, za który jest on odpowiedzialny. Dlatego typ uczestnictwa tabeli *Departamenty* w tej relacji jest *opcjonalny*.

Tworząc graficzny schemat połączonych tabel *Dyrektorzy* i *Departamenty,* uzgodnimy że: jeżeli typ uczestnictwa danej tabeli w relacji jest obowiązkowy, to na zakończeniu linii łączącej tabelę (na schemacie po stronie tej tabeli) umieszczamy kreskę pionową: |; natomiast w przypadku uczestnictwa opcjonalnego – kółko: ○. Mianowicie, w przykładzie pierwszym nasza relacja będzie miała oznaczenia pokazane na rys. 2.10.

Przykład 2-10

Mamy relację typu *jeden-do-wielu* między tabelami *Marki_Samochodów {Id_Marki, Nazwa, Producent, Rok_Założenia*} i *Modele* {*Id_Modelu, Pojemność, Rok_Stworzenia*}. Ponieważ każdy model samochodu musi mieć

określoną markę, nie możemy więc do tabeli *Modele* dodać rekordu, jeżeli w tabeli *Marki_Samochodów* nie ma odpowiedniego rekordu – marki, do której należy nowo dodawany model. A więc typ uczestnictwa tabeli *Marki_Samochodów* w tej relacji jest *obowiązkowy*.

Z kolei mogą istnieć nowo utworzone marki samochodów, które nie mają na swoim koncie jeszcze żadnego modelu. A więc dla dodania nowej marki nie jest niezbędny odpowiadający rekord w tabeli *Modele*. Stąd typ uczestnictwa tabeli *Modele* w tej relacji jest *opcjonalny* (patrz rys. 2.11).

Wykładowcy

Id_Wyk	*Nazw_Wyk*	*Imię_Wyk*	*Stopień_Naukowy*	*Przedmiot*
I_1-1	Składanowski	Łukasz	Dr	Informatyka
I_1-2	Rogala	Jacek	Dr_hab	Sieci komputerowe
M_1-3	Kowalska	Kamila	Dr	Matematyka
M_1-4	Pyda	Grzegorz	Dr	Matematyka

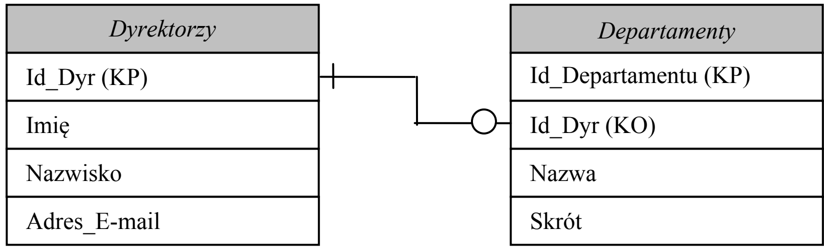
Wykładowcy/Studenci

Id_Wyk	*Id_Stud*
I_1-1	34-16_1
I_1-1	34-16_2
I_1-1	34-16_3
I_1-1	34-16_4
I_1-1	44-16_3
M_1-4	44-16_1
...	
M_1-3	54-16_5
I_1-2	64-16_1
I_1-2	64-16_2
I_1-2	64-16_3
I_1-2	64-16_4
I_1-2	54-16_4
I_1-2	54-16_3

Studenci

Id_Stud	*Nazw_Stud*	*Imię_Stud*
34-16_1	Nawrocki	Michał
34-16_2	Michalska	Grażyna
34-16_3	Pylak	Paweł
34-16_4	Matusiewicz	Krystian
44-16_1	Zapała	Magdalena
	...	
54-16_5	Rogala	Paweł
64-16_1	Urbanowicz	Anna
64-16_2	Szyszka	Marek
64-16_3	Zygmunt	Stanisław
64-16_4	Roczeń	Anna

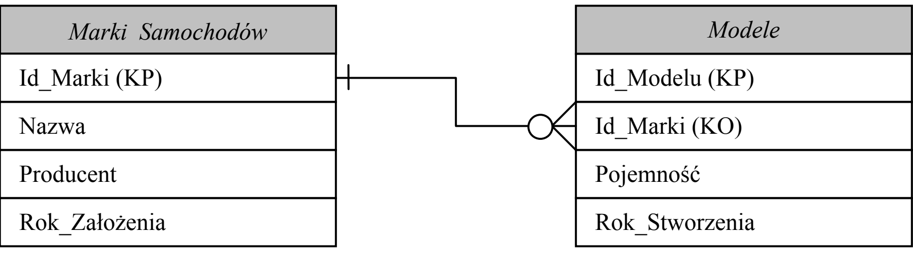
Rys. 2.9. Przykład tabel (*Wykładowcy* i *Studenci*) powiązanych w relacji w:w



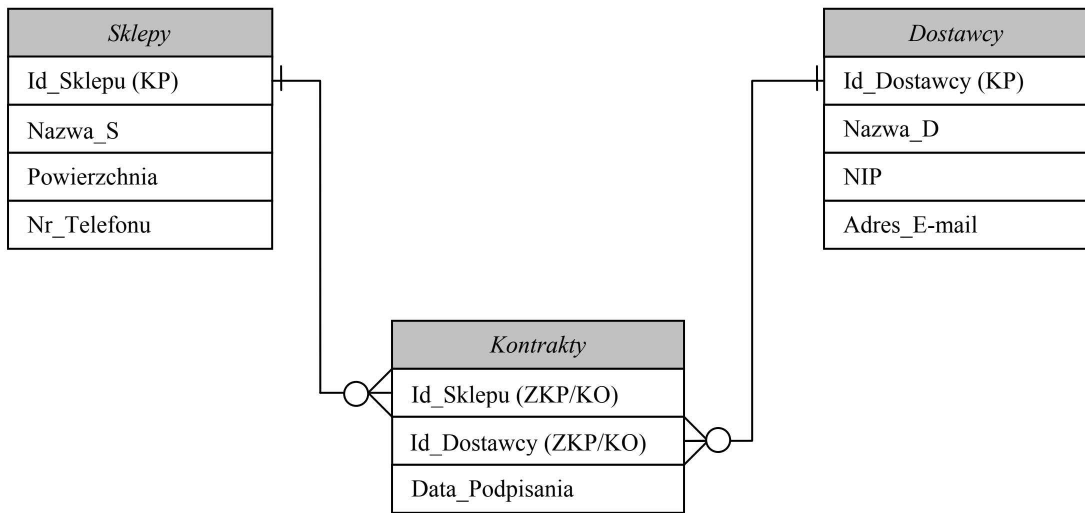
Rys. 2.10. Oznaczenie relacji *jeden-do-jednego* między dwiema tabelami z uwzględnieniem typów uczestnictwa

Przykład 2-11
W trzecim przykładzie mamy relację typu *wiele-do-wielu* pomiędzy tabelami *Sklepy* {*Id_Sklepu, Nazwa_ Sklepu, Powierzchnia, Nr_Telefonu*} i *Dostawcy* {*Id_Dostawcy, Nazwa_ Dostawcy, NIP, Adres_E-mail*}. Relację tę tworzymy, wprowadzając dodatkową tabelę łączącą *Kontrakty*. W ten sposób nasza relacje *wiele-do-wielu* została rozłożona na dwie relacje typu *jeden-do-wielu*: między tabelami *Sklepy* i *Kontrakty*, a także między tabelami *Dostawcy* i *Kontrakty*.

Aby dodać nowy kontrakt, niezbędne jest istnienie zarówno sklepu, jak i dostawcy będących stronami tego kontraktu. Tak więc typ uczestnictwa tabel *Sklepy* i *Dostawcy* w odpowiadających im relacjach typu *jeden-do-wielu* jest *obowiązkowy*. Natomiast typ uczestnictwa tabeli *Kontrakty* jest *opcjonalny*. Możemy dodawać zarówno nowe sklepy, jak i dostawców, nawet jeżeli nie istnieje żaden kontrakt, którego są one stronami. Schemat ten przedstawiony jest na rys. 2.12.



Rys. 2.11. Oznaczenie relacji *jeden-do-wielu*



Rys. 2.12. Oznaczenie relacji *wiele-do-wielu* między dwiema tabelami z uwzględnieniem typów uczestnictwa

2.1.4. Typy integralności w relacyjnych BD

Definicja 2-13

Przez ***integralność*** relacyjnej BD rozumiemy poprawność, spójność, a także dokładność przechowywanych w niej danych.

Zapewnienie integralności danych jest jednym z najważniejszych zadań, przed którymi stoją projektanci baz danych. Wyróżniamy trzy typy integralności:

- na poziomie pól;
- na poziomie tabel;
- na poziomie relacji (integralność referencyjna).

Integralność na poziomie pól gwarantuje, że struktura każdego pola jest poprawna, zawarte w nich wartości są logiczne, wszystkie pola są tego samego typu i są zdefiniowane w identyczny sposób, a zbiór dopuszczalnych wartości każdego pola mieści się w dziedzinie określonej dla odpowiadającego atrybutu tabeli.

Oznacza to, że tabela nie może zawierać pól zwielokrotnionych, a więc posiadających różne typy danych, czy pól kalkulowanych, a więc takich, których wartości możemy wyliczyć na podstawie innych pól.

Przykład 2-12

Mamy daną tabelę *Studenci* {*Id_Stud, Nazwisko, Imię, Adres*}. Przykładem wartości zwielokrotnionej może być wartość: ul. Chopina, 129, Lublin, 20-950, wpisana w odpowiednim wierszu pola *Adres,* jeżeli potraktujemy te dane jak dwuwartościowe: tekstowe i liczbowe.

Przykład 2-13

Mamy daną tabelę *Towary* {*Id_Towaru, Nazwa, Cena, Ilość, Koszt*}. Zakładając, że wartość w polu *Koszt* można wyliczyć na podstawie wartości pól *Cena* i *Ilość* (*Koszt = Cena* Ilość*), wnioskujemy, iż pole *Koszt* jest polem kalkulowanym.

Integralność na poziomie tabel gwarantuje**,** że każdy rekord w tabeli jest unikatowy, nie ma powtarzających się rekordów, a pola kluczowe w tabeli mają wartości różne od NULL, a więc zawsze mają określone wartości.

Zasada ta wynika z faktu, iż każdy rekord w tabeli musi być jednoznacznie identyfikowalny, a powtarzające się wartości kluczy lub wartości puste w kluczach nie dawałyby takiej gwarancji.

Integralność na poziomie relacji (integralność referencyjna) gwarantuje, że wszystkie relacje pomiędzy tabelami są zdefiniowane poprawnie, a dane w powiązanych tabelach są ze sobą zsynchronizowane. Inaczej rzecz ujmując, w tabeli zawierającej klucz obcy wartości tego klucza muszą odpowiadać wartościom *klucza podstawowego* w *tabeli nadrzędnej* lub być wartościami pustymi NULL. Oznacza to, że nie można dodać do tabeli rekordu z wartościami *klucza obcego* różnego od wartości *klucza podstawowego tabeli nadrzędnej*, chyba że będą to wartości NULL.

Oprócz wymienionych typów integralności administratorzy oraz użytkownicy bazy danych mogą wprowadzać własne zasady integralności, tzw. reguły integralności.

Definicja 2-14

Regułami integralności nazywamy pewne sformułowania, które ograniczają niektóre dopuszczalne wartości pól relacji lub niektóre cechy i właściwości samych relacji.

Reguły takie są specyficzne dla danej organizacji wykorzystującej bazę danych i wynikają na ogół z potrzeb tej organizacji, przepisów prawa oraz zasad w niej panujących.

Przykładowymi regułami relacji może być: „Atrybut Płeć może przyjmować tylko dwie wartości: *M* – oznaczającą mężczyznę i *K* – oznaczającą kobietę” albo „Liczba studentów przypisana do grupy nie może być mniejsza niż 12 i większa niż 20”.

Zadania do samokontroli

1. Wyjaśnić pojęcie klucza kandydującego i klucza podstawowego.
2. Mamy tabelę $R\ \{A_1, A_2, ..., A_n\}$. Ile KK i ile KP może posiadać tabela $R\ \{\}$?
3. Podać przykład tabeli posiadającej jedno-, dwu- i trzypolowy KK.
4. Definicja relacji *jeden-do-jednego*. Podać przykład.
5. Definicja relacji *jeden-do-wielu*. Podać przykład.
6. Definicja relacji *wiele-do-wielu*. Podać przykład.
7. Zdefiniować relację pomiędzy tabelami *Rodzice* i *Uczniowie*.
8. Zdefiniować relację pomiędzy tabelami *Towary* i *Klienci* w BD *Sklep*.
9. Baza danych *Przychodnia* składa się z 6 tabel. Jaką maksymalną liczbę kluczy może posiadać tabela w tej bazie?
10. Podać przykład tabeli posiadającej pole wielowartościowe.
11. Podać przykład tabeli posiadającej pole wyliczeniowe.
12. Co oznacza pojęcie *integralność* BD?
13. Scharakteryzować typ uczestnictwa tabel w relacjach z Prz. 7 i 8.

2.2. Manipulowanie danymi

Jak podkreśliliśmy na początku tego rozdziału, trzecia część modelu relacyjnego dotyczy operacji nad danymi, czyli manipulowania danymi. Ta część modelu opiera się na *algebrze relacyjnej* i określa środki, za pomocą których użytkownik może wykonywać różne operacje nad danymi.

2.2.1. Podstawy algebry relacyjnej

Ogólnie mówiąc, algebra składa się ze zbioru *operatorów* i *operandów atomowych*. Na przykład, w algebrze arytmetyki *operandami atomowymi* są zmienne typu ***x*** oraz stałe (np. *15*), operatorami zaś są zwykłe operatory arytmetyczne (dodawania, odejmowania, mnożenia i dzielenia). Dowolna algebra daje możliwość tworzenia wyrażeń (ang. *expressions*) poprzez zastosowanie operatorów do operandów i/lub innych wyrażeń.

Algebra relacyjna jest podtypem algebry. Zawiera ona następujące typy *operandów atomowych*:

- *zmienne*, odpowiadające *relacjom*;
- *stałe*, będące *relacjami końcowymi*.

Według twórcy algebry relacyjnej E. Codda wszystkie operandy oraz wyniki operacji są zbiorami. Wyrażenia tej algebry nazywamy *zapytaniami* lub *kwerendami* (ang. *queries*). Przykładem wyrażenia algebry relacyjnej może być: $\boldsymbol{R} \cup \boldsymbol{S}$, gdzie $\boldsymbol{R}$ i $\boldsymbol{S}$ – operandy atomowe, będące relacjami o dowolnym zbiorze atrybutów.

Algebra relacyjna E. Codda składa się z 8 operatorów, podzielonych na dwie równe grupy:

1) *tradycyjne operatory* do operacji nad zbiorami:

- *suma* (ang. *union*);
- *przecięcie* (ang. *intersect*);
- *różnica* (ang. *difference*);
- *iloczyn kartezjański* (ang. *cartesian product*);

2) *operatory* do wykonania *specjalnych operacji*:

- *selekcja* lub *restrykcja* (ang. *selection*);
- *projekcja* lub *rzutowanie* (ang. *projection*);
- *złączenie naturalne* (ang. *natural join*);
- *iloraz* (ang. *divide*).

Pięć ostatnich operatorów (iloczyn kartezjański, selekcja, projekcja, złączenie, iloraz) można rozpatrywać jako *prymitywne* w tym sensie, że żadnego z nich nie można wyrazić poprzez dowolny inny. Dlatego *minimalny zbiór* operatorów (i odpowiednich operacji) będzie się składać z pięciu *prymitywnych operatorów*. Pozostałe trzy operatory nie są prymitywne, bo można je wyrazić za pomocą innych operatorów (prymitywnych). Jednakże owe 3 operatory są bardzo często wykorzystywane w praktyce. Podstawowymi celami algebry relacyjnej są: umożliwienie zapisywania zdań relacyjnych (cel główny), definiowanie operacji pobierania danych, definiowanie reguł ochrony danych, definiowanie pochodnych relacji (relacji-zmiennych).

Teraz przeanalizujemy bardziej szczegółowo wymienione operatory.

2.2.2. Operatory relacyjne E. Codda

W literaturze baz danych do oznaczenia operatorów relacyjnych wykorzystuje się symbole matematyczne lub litery greckie (np. litera σ oznacza operator selekcji, litera π – operator projekcji, znak $\cup$ odpowiada operacji złączenia (sumy), symbol $\cap$ – operacji przecięcia), czasem także – słowo kluczowe JOIN, WHERE i inne. W naszych dalszych analizach będziemy korzystać z obu tych sposobów.

2.2.2.1. Operator *Sumy*

W matematyce złączeniem dwóch zbiorów (wejściowych) nazywamy zbiór, który składa się ze wszystkich elementów, należących przynajmniej do jednego z dwóch wejściowych zbiorów.

Zapis $\boldsymbol{R} \cup \boldsymbol{S}$ (lub ***R*** UNION ***S***) oznacza operację *sumy* nad zbiorami ***R*** i ***S***, której wynikiem będzie zbiór, składający się z *niepowtarzających* się elementów zbiorów wejściowych. Ograniczenie niepowtarzalności oznacza, że zbiór wynikowy nie zawsze może być relacją[1]. Jednakże, jak było wspomniane wcześniej, wynikiem każdej operacji relacyjnej musi być relacja.

Ponieważ relacja jest zbiorem, powinna istnieć możliwość utworzenia sumy dwóch relacji. Dlatego suma w algebrze relacyjnej nie jest klasyczną (w sensie matematycznym) sumą, ale jest specjalnym typem złączenia, w którym relacje wejściowe powinny mieć ten sam nagłówek, czyli muszą składać się z pasujących atrybutów (innymi słowy: typy dwóch relacji są zgodne).

Definicja 2-15

Będziemy mówić, że typy dwóch relacji są zgodne, jeżeli mają takie same nagłówki, tzn. każda z nich ma taki sam zbiór atrybutów, a odpowiadające sobie atrybuty są zdefiniowane na tej samej dziedzinie.

Definicja 2-16

Sumą (*union*) dwóch relacji ***R*** i ***S*** zgodnych typów (***R*** UNION ***S***) jest relacja, mająca ten sam nagłówek co relacje ***R*** i **S**, treść zaś złożona jest ze wszystkich krotek należących do ***R*** lub ***S***.

Przykład 2-14

Mamy dwie relacje o tych samych nagłówkach: *Studenci* {*Id, Nazwisko, Imię*}(relacja ***R***) i *Pracownicy* {*Id, Nazwisko, Imię*}(relacja ***S***).

Tabela 2.2. Relacja S*tudenci*

Id	*Nazwisko*	*Imię*
1	Jezierski	Marek
3	Kalinowski	Rafał
4	Lipska	Ewa

Tabela 2.3. Relacja *Pracownicy*

Id	*Nazwisko*	*Imię*
4	Lipska	Ewa
5	Jezierska	Ewa
6	Kalinowski	Piotr
7	Kalinowski	Marek

Realizacja operacji S*tudenci* UNION *Pracownicy* daje następującą relację wynikową:

Tabela 2.4. Relacja S*tudenci* UNION *Pracownicy*

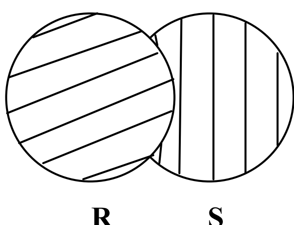
Id	*Nazwisko*	*Imię*
1	Jezierski	Marek
3	Kalinowski	Rafał
4	Lipska	Ewa

[1] Tutaj i dalej w pkt 2.2.2 *relacja* w sensie *tabela.*

5	Jezierska	Ewa
6	Kalinowski	Piotr
7	Kalinowski	Marek

Jak widzimy, relacja wynikowa (2.4) posiada tylko niepowtarzające się wartości (rekordy).

Operacja ***R*** UNION **S** może być przedstawiona w następujący sposób (rys. 2.13).



Rys. 2.13. Schematyczne przedstawienie operacji **R** UNION **S** (elementy relacji **R** i **S** znajdują się wewnątrz odpowiednich krążków; elementy relacji wynikowej – **R** UNION **S** – znajdują się wewnątrz widocznych części krążków)

W dalszej części poznamy operację sumy, której wynikiem będzie relacja, zawierająca także powtarzające się rekordy (liczba rekordów w relacji wynikowej równa będzie sumie arytmetycznej rekordów w łączonych relacjach).

2.2.2.2. Operator *Przecięcia*

Tak jak w przypadku operatora *Sumy*, typy dwóch relacji operatora *Przecięcia* powinny być zgodne. Zapis $\boldsymbol{R} \cap \boldsymbol{S}$ (lub ***R*** INTERSECT ***S***) oznacza operację *Przecięcia* nad zbiorami ***R*** i ***S***, której wynikiem będzie zbiór, składający się z *powtarzających* się elementów zbiorów wejściowych.

Definicja 2-17

Przecięciem (*intersect*) dwóch relacji ***R*** i ***S*** zgodnych typów (***R*** INTERSECT ***S***) jest relacja, mająca ten sam nagłówek co ***R*** i ***S***, złożona zaś jest ze wszystkich krotek, które należą zarówno do ***R***, jak i do ***S***.

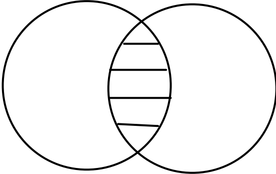
Przykład 2-15

Dla relacji z Prz. 2-14 wynikiem operacji *Studenci* INTERSECT *Pracownicy* będzie relacja pokazana w tab. 2.5.

Tabela 2.5. Relacja S*tudenci* INTERSECT *Pracownicy*

Id	*Nazwisko*	*Imię*
4	Lipska	Ewa

Operacja ***R*** INTERSECT ***S*** może być przedstawiona w następujący sposób (rys. 2.14).



Rys. 2.14. Schematyczne przedstawienie operacji **R** INTERSECT **S** (elementy relacji **R** i **S** znajdują się wewnątrz odpowiednich krążków; elementy relacji wynikowej – **R** INTERSECT **S** – znajdują się w części nałożenia krążków)

Operacja iloczynu może być przedstawiona za pomocą dwóch operacji różnicy z wykorzystaniem następującego wzoru: $\boldsymbol{R} \cap \boldsymbol{S} = \boldsymbol{R} - (\boldsymbol{R} - \boldsymbol{S})$[2].

2.2.2.3. Operator *Różnicy*

Tak jak w dwóch poprzednich sytuacjach, również dla relacyjnego operatora *Różnicy* niezbędne jest, aby jego operandy były zgodnych typów. Zapis ***R*** − ***S*** (lub ***R*** MINUS ***S***) oznacza operację *Różnicy* nad zbiorami ***R*** i ***S***, której wynikiem będzie zbiór, składający się z elementów zbioru ***R*** niepowtarzających się w zbiorze ***S***.

Definicja 2-18

Różnicą (*difference*) dwóch relacji ***R*** i ***S*** (***R*** MINUS ***S***) zgodnych typów jest relacja, mająca ten sam nagłówek (co ***R*** i ***S***), złożona ze wszystkich krotek, należących do ***R*** i nienależących do ***S***.

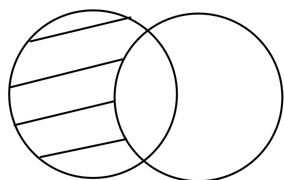
Przykład 2-16

Dla relacji z Prz. 2-14 wynikiem operacji *Pracownicy* MINUS *Studenci* będzie tab. 2.6.

Tabela 2.6. Relacja *Pracownicy* MINUS *Studenci*

Id	*Nazwisko*	*Imię*
5	Jezierska	Ewa
6	Kalinowski	Piotr
7	Kalinowski	Marek

Operacja ***R*** MINUS ***S*** może być przedstawiona graficznie w następujący sposób (rys. 2.15).



Rys. 2.15. Schematyczne przedstawienie operacji **R** MINUS **S** (elementy relacji **R** i **S** znajdują się wewnątrz odpowiednich krążków; elementy relacji wynikowej – **R** MINUS **S** – znajdują się w lewej części lewego krążka)

2.2.2.4. Operator Iloczyu kartezjańskiego

W matematyce *iloczyn kartezjański* (ang. *cartesian product*) dwóch zbiorów jest zbiorem wszystkich uporządkowanych par elementów takich, że pierwszy element każdej pary pochodzi z pierwszego zbioru a drugi element z drugiego zbioru.

Iloczyn kartezjański w algebrze relacyjnej jest rozszerzoną wersją tego operatora w matematyce i zapisywany jest jako ***R*** TIMES ***S***. Każda uporządkowana para krotek zostaje zastąpiona pojedynczą krotką powstałą z konkatenacji obu rozważanych krotek (konkatenacja oznacza tutaj sumę w sensie teorii zbiorów, a nie w sensie algebry).

Jeżeli zdarzy się, że relacje ***R*** i ***S*** posiadają atrybuty o tych samych nazwach, to przynajmniej w jednej z relacji powtarzające się nazwy należy zamienić na inne.

---

[2] Działanie operatora MINUS patrz dalej.

Definicja 2-19

Iloczynem kartezjańskim dwóch relacji ***R*** i ***S*** (***R*** TIMES ***S***), gdzie ***R*** i ***S*** nie mają wspólnych atrybutów, jest nowa relacja z nagłówkiem, składającym się z nagłówków relacji ***R*** i ***S*** i posiadającym krotki, będące konkatenacjami wszystkich możliwych par krotek, pochodzących z relacji ***R*** i relacji **S**.

Przykład 2-17

Niech relacji **R** odpowiada tabela *Studenci* {*Id_Stud, Nazwisko, Imię*} (tab. 2.7), relacji **S** – tabela *Egzaminy* {*Id_Egz, Przedm, Data_Egz, Ocena*} (tab. 2.8).

Tabela 2.7. Relacja S*tudenci*

Id_Stud	*Nazwisko*	*Imię*
1	Jezierski	Marek
3	Kalinowski	Rafał
4	Lipska	Ewa

Tabela 2.8. Relacja *Egzaminy*[3]

Id_Egz	*Przedm*	*Data_Egz*	*Ocena*
A	Matematyka	06.06.07	
B	Bazy danych	01.06.07	
C	Ekonometria	15.06.07	
D	Ochrona Informacji	10.06.07	

Operacji S*tudenci* TIMES *Egzaminy* odpowiada wynikowa relacja *Lista_Egz* (tab. 2.9):

Tabela 2.9. Relacja *Lista_Egz*

Id_Stud	*Nazwisko*	*Imię*	*Id_Egz*	*Przedm*	*Data_Egz*	*Ocena*
1	Jezierski	Marek	A	Matematyka	06.06.07	
1	Jezierski	Marek	B	Bazy danych	01.06.07	
1	Jezierski	Marek	C	Ekonometria	15.06.07	
1	Jezierski	Marek	D	Ochrona Informacji	10.06.07	
3	Kalinowski	Rafał	A	Matematyka	06.06.07	
3	Kalinowski	Rafał	B	Bazy danych	01.06.07	
3	Kalinowski	Rafał	C	Ekonometria	15.06.07	
3	Kalinowski	Rafał	D	Ochrona Informacji	10.06.07	
4	Lipska	Ewa	A	Matematyka	06.06.07	
4	Lipska	Ewa	B	Bazy danych	01.06.07	
4	Lipska	Ewa	C	Ekonometria	15.06.07	
4	Lipska	Ewa	D	Ochrona Informacji	10.06.07	

[3] Atrybut *Ocena* posiada wartości *NULL*. Patrz podrozdz. 4.1.1. Podstawowe typy danych.

Jak widać z powyższego przykładu, *k*-tą krotką (rekordem) wynikowej relacji (tabeli *Lista_Egz*) w operacji *iloczynu kartezjańskiego* będzie krotka (rekord), składająca się z *i*-tej krotki (*i*-tego rekordu, zaczynając od *i* =1), relacji *R* (tabeli *Studenci)* i *j*-tej krotki (*j*-tego rekordu, zaczynając od *j* =1) relacji *S* (tabeli *Egzaminy*).

Schemat dopasowania krotek (rekordów) wygląda następująco: na początku do pierwszej krotki relacji *R* będą kolejno dopasowywane wszystkie krotki tabeli S, następnie do drugiej krotki relacji *R* będą kolejno dopasowywane wszystkie krotki relacji *S* itd.

Jeżeli relacja *R* posiada a_r atrybutów i k_r krotek i relacja *S* posiada a_s atrybutów i k_s krotek, to po wykonaniu operacji *R* TIMES *S* relacja będzie posiadała $a_r + a_s$ atrybutów i $k_r \times k_s$ krotek. Taki sposób połączenia danych nazywa się *krzyżowym*.

2.2.2.5. Operator *Selekcji* (*Restrykcji*)

Selekcja (ang. *selection*) lub *restrykcja* (ang. *restriction*) jest bardzo ważną operacją opracowywania informacji. Jest to *unarna* operacja nad relacją.

Definicja 2-20

Operator ***Selekcji*** zastosowany do relacji ***R*** daje w wyniku relację, posiadającą podzbiór krotek ***R***, odpowiadających podanemu warunkowi ***C***, i ma ten sam nagłówek co ***R***.

Operację selekcji nad relacją *R* oznacza się jako $\sigma_c(R)$. *C* jest warunkiem podobnym do warunków używanych w tradycyjnych językach programowania. W naszym przypadku ważną cechą *C* jest to, że jego operandami są stałe lub nazwy atrybutów relacji *R*, a znakami operacji najczęściej operacje logiczne. Praktycznie *C* w $\sigma_c(R)$ jest *predykatem*.

Definicja 2-21

Jeżeli ***R*** ma nagłówek ***R {X, Y, ...}***, a symbol ⊚ oznacza dowolny skalarny operator porównania (=, >, < i itd.), to ***⊚-selekcją*** z relacji ***R*** wg atrybutów ***X*** i ***Y*** będzie relacja, mająca ten sam nagłówek co ***R*** i składająca się z krotek relacji **R**, dla których sprawdzenie warunku ***X ⊚ Y*** daje wartość logiczna prawda (ang. *true)*.

Przykład 2-18

Niech relacji ***R*** odpowiada tabela *Studenci* {*Id_Stud, Nazwisko, Imię*} (tab. 2.7). Operator selekcji

Id_Stud WHERE *Nazwisko = 'Jezierski'*

daje taki wynik:

Id_Stud	*Nazwisko*	*Imię*
1	Jezierski	Marek

Operator ten można zapisać też w innej formie:

$$\sigma_{Nazwisko = \text{'Jezierski'}}(\textbf{Studenci}).$$

W tym przykładzie warunek posiada tylko jedno porównanie, opierające się na operatorze równości. Praktycznie warunek we frazie WHERE może posiadać dowolną liczbę porównań.

Przykład 2-19
Dla relacji *Studenci* {*Id_Stud, Nazwisko, Imię*} (tab. 2.7) operator selekcji

Id_Stud WHERE *Nazwisko = 'Jezierski' or Imię = 'Ewa'*

daje wynik:

Id_Stud	*Nazwisko*	*Imię*
1	Jezierski	Marek
4	Lipska	Ewa

Operacja *selekcji* wygląda więc następująco:

$$\sigma_{Nazwisko = \text{'Jezierski'} \text{ or } Imię = \text{'Ewa'}}(\mathbf{Studenci}).$$

2.2.2.6. Operator *Projekcji* (*Rzutowania*)

Operacja *projekcji* (ang. *projection*) również jest operacją unarną.

Definicja 2-22
Załóżmy, że mamy relację ***R {A,B, ..., X, Y, ..., Z}***. ***Projekcją*** relacji ***R*** wg atrybutów ***A, B, X, Z*** (lub w innej kolejności) będzie relacja posiadająca tylko wymienione atrybuty, czyli mająca nagłówek ***R'{A,B,X,Z}***.

Zatem operator *Rzutowania* daje pionowy podzbiór danej relacji. Jest to podzbiór otrzymany przez eliminację wszystkich atrybutów niewymienionych na liście atrybutów i usunięcie powtarzających się krotek.

Definicja 2-23
Projekcją relacji ***R*** wg wszystkich atrybutów ***{A,B,..., X, Y, ..., Z}*** jest kopia relacji ***R.***

Definicja 2-24
Projekcję typu **R'=R*{ }*** nazywa się ***projekcją zerową***.

Formalnie operacja projekcji może być zapisana następująco:

$\pi_{A,B,X,Z}(R)$ (odpowiada Def. 2-22). Tutaj lista atrybutów jest przykładem ***predykatu***.

Przykład 2-20
Dla relacji *Studenci* {*Id_Stud, Nazwisko, Imię*} (tab. 2.7) operator projekcji

$\pi_{Id_Stud,\ Imię}(\mathbf{Studenci})$

daje taki wynik:

Id_Stud	*Nazwisko*
1	Jezierski
3	Kalinowski
4	Lipska

2.2.2.7. Operator *złączenia naturalnego*

Operacja złączenia ma kilka odmian – najważniejszym jest *złączenie naturalne*. Zwróćmy uwagę, że jednym z typów złączenia jest *iloczyn kartezjański*.

Złączenie naturalne (ang. *natural join*) lub (po prostu) *złączenie* (ang. *join*) wartości dwóch relacji ***R*** i ***S*** oznaczamy ***R*** JOIN ***S*** lub ***R*** ⊳⊲ ***S***.

Definicja 2-25

Rozważmy relacje ***R*** i ***S***, mające nagłówki:

$R\{X_1, X_2,..., X_n, Y_1, Y_2, ..., Y_k\}$,

$S\{Y_1, Y_2, ..., Y_k, Z_1, Z_2, ..., Z_m\}$.

Złączeniem naturalnym relacji ***R*** i relacji ***S*** jest relacja z ***nagłówkiem*** $\{X_1, X_2,..., X_n, Y_1, Y_2, ..., Y_k, Z_1, Z_2, ..., Z_m\}$.

Innymi słowy, relacje **R** i **S** mają wspólne atrybuty, w danym przypadku atrybuty $Y_1, Y_2, ..., Y_k$, które *powinny być zdefiniowane na tej samej dziedzinie*.

Operacja *złączenia naturalnego* przewiduje rozmieszczenie w relacji (tabeli) wynikowej tylko tych krotek (rekordów) relacji (tabel) R i S, ktore mają te same wartości we wspólnych atrybutach (polach).

Przykład 2-21

Dla relacji *Studenci* {*Id_Stud, Nazwisko, Imię*} (tab. 2.10) i *Egzaminy* {*Id_Egz, Id_Stud, Przedm, Data_Egz, Ocena*} (tab. 2.11) wspólnym atrybutem jest *Id_Stud*. Operacja Studenci JOIN *Egzaminy* daje tabelę *Sesja* {*Id_Stud, Nazwisko, Imię, Id_Egz, Przedm, Data_Egz, Ocena*} (tab. 2.12).

Tabela 2.10. Relacja S*tudenci*

Id_Stud	*Nazwisko*	*Imię*
1	Jezierski	Marek
3	Kalinowski	Rafał
4	Lipska	Ewa

Tabela 2.11. Relacja *Egzaminy*

Id_Egz	*Id_Stud*	*Przedm*	*Data_Egz*	*Ocena*
A	1	Matematyka	06.06.07	db
B	1	Bazy danych	01.06.07	bdb
C	4	Ekonometria	15.06.07	dost
C	3	Ekonometria	15.06.07	bdb
C	1	Ekonometria	15.06.07	db plus
D	3	Ochrona Informacji	10.06.07	db
D	4	Ochrona Informacji	10.06.07	db plus

Tabela 2.12. Relacja *Sesja*

Id_Stud	*Nazwisko*	*Imię*	*Id_Egz*	*Przedm*	*Data_Egz*	*Ocena*
1	Jezierski	Marek	A	Matematyka	06.06.07	db
1	Jezierski	Marek	B	Bazy danych	01.06.07	bdb
1	Jezierski	Marek	C	Ekonometria	15.06.07	db plus

3	Kalinowski	Rafał	C	Ekonometria	15.06.07	bdb
3	Kalinowski	Rafał	D	Ochrona Informacji	10.06.07	db
4	Lipska	Ewa	C	Ekonometria	15.06.07	dost
4	Lipska	Ewa	D	Ochrona Informacji	10.06.07	db plus

Jak widać z analizy tego przykładu, schemat tworzenia rekordów tabeli wynikowej wygląda tak samo jak w przypadku operacji *iloczynu kartezjańskiego*, z wyjątkiem tego, że do rekordów relacji (tabeli) po lewej stronie operatora JOIN (*Studenci*) będą kolejno dołączane tylko te rekordy drugiej relacji (*Egzaminy*), które mają taką samą wartość atrybutu *Id_Stud* co w relacji po lewej stronie operatora JOIN (*Studenci*).

Schemat dopasowania krotek relacji *S* do krotek relacji *R* w operacji *złączenia naturalnego R* JOIN *S* opiera się na operacji *iloczynu kartezjańskiego*.

Definicja 2-26

Θ-złączeniem relacji ***R*** i ***S*** jest relacja, w której każda krotka składa się z krotek relacji ***R*** i ***S*** pod warunkiem **Θ**.

Przykład 2-22

Dla relacji *Studenci* {*Id_Stud, Nazwisko, Imię*} (tab. 2.10) i *Egzaminy* {*Id_Egz, Id_Stud, Przedm, Data_Egz, Ocena*} (tab. 2.11) operacja *Studenci* JOIN *Egzaminy* z dodatkowym warunkiem **Θ** (*Przedmiot* = 'Bazy danych') daje relację *Sesja_1* {*Id_Stud, Nazwisko, Imię, Id_Egz, Przedm, Data_Egz, Ocena*} (tab. 2.13).

Tabela 2.13. *Sesja_1*

Id_Stud	*Nazwisko*	*Imię*	*Id_Egz*	*Przedm*	*Data_Egz*	*Ocena*
1	Jezierski	Marek	B	Bazy danych	01.06.07	bdb

Dalej zobaczymy, że operacja złączenia relacji prowadzi do pojawienia się nowej relacji, która była wejściem przy realizacji operacji *projekcji* (pkt 3.2.1).

2.2.2.8. Operator *Dzielenia* (*Iloraz*)

Operacja *dzielenia* (ang. *divide*) jest operacją binarną, tzn. wykonywaną nad dwiema relacjami (tabelami).

Definicja 2-27

Mamy relacje ***R*** i ***S*** z nagłówkami:

$R\{X_1, X_2, ..., X_n\}$,

$S\{Y_1, Y_2, ..., Y_k\}$.

Mamy również relację $C\ \{X_1, X_2, ..., X_n, Y_1, Y_2, ..., Y_k\}$. Wtedy wynikiem operacji ***dzielenia*** relacji ***R*** przez relację ***S*** wg ***C*** będzie relacja z nagłówkiem $\{X_1, X_2, ..., X_n\}$, składająca się z takich wartości relacji ***R*** (nazywane ***X***-wartościami), dla których odpowiednie ***Y***-wartości (wartości atrybutów $Y_1, Y_2, ..., Y_k$) w relacji ***C*** posiadają wszystkie wartości z relacji ***S***.

Formalnie zapis zdefiniowanej operacji ma postać:

R DIVIDEBY ***S*** PER ***C***.

Przykład 2-23

Niech relacje (tabele) ***R***, ***S*** i ***C*** będą określone, jak pokazano w tab. 2.14, tab. 2.15 i tab. 2.16.

Tabela 2.14. *R*

Id_R
R1
R2
R3
R4
R5

Tabela 2.15. *S*

Id_S
S1

Tabela 2.16. *C*

Id_R	*Id_S*
R1	S1
R1	S2
R1	S3
R1	S4
R1	S5
R1	S1
R2	S1
R3	S2
R4	S2

Operacja

R DIVIDEBY ***S*** PER ***C***
daje relację (tabelę):

Id_R
R1
R2

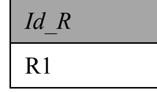
Przykład 2-24

Niech relacje (tabele) *R* i *C* będą jak w Prz. 2-23, relacja (tabela) *S* zaś wygląda następująco (tab. 2.17).

Tabela 2.17. *S*

Id_S
S3
S4

A zatem wynik będzie jak w poniższej relacji (tabeli).



Id_R
R1

W praktyce jednak często mamy do czynienia z uproszczonymi realizacjami operacji *dzielenia*. Przeanalizujmy przykład.

Przykład 2-25

Mamy relacje (tabele) *Lista_Egz* (tab. 2.18) i *Egzaminy* (tab. 2.19).

Tabela 2.18. *Lista_Egz*

Indeks	*Nazw_Stud*	*Przedm*	*Data*
1-A-07	Werner	Fizyka	10.02.08
1-A-07	Werner	Matematyka	15.02.08
2-A-07	Jabłoński	Fizyka	10.02.08
2-A-07	Jabłoński	Matematyka	15.02.08
3-A-07	Maczulska	Fizyka	10.02.08

Tabela 2.19. *Egzaminy*

Przedm	*Data*
Fizyka	10.02.08
Matematyka	15.02.08

Aby uzyskać odpowiedź na pytanie, którzy ze studentów zdali wszystkie egzaminy, należy wykonać operację *Lista_Egz* DIVIDEBY *Egzaminy*. W wyniku tej operacji mamy tabelę *Wyniki* (tab. 2.20).

Tabela 2.20. *Wyniki*

Indeks	*Nazw_Stud*
1-A-07	Werner
2-A-07	Jabłoński

W ostatnim wypadku zrealizowana została operacja ***R*** DIVIDEBY ***S***.

Dalej krótko przeanalizujemy działanie operacji *algebry relacyjnej* w bardziej skomplikowanych sytuacjach.

Przykład 2-26

Wyniki sesji egzaminacyjnej umieśćmy w trzech relacjach (tabelach): *Studenci* {*Indeks_Stud, Nazwisko_Stud, Grupa*}, *Wyniki_Egz* {*Indeks_Stud, Nazwisko_Stud, Wyniki_Egz Przedm, Ocena*}, *Egzaminy* {*Grupa, Przedm*}. Jak widać z nagłówków, pierwsza z tabel zawiera informacje o listach grup, druga – o wynikach egzaminów, trzecia – o przedmiotach, z których egzaminy powinny zdawać wszyscy studenci.

Aby uzyskać przykładowo informacje o studentach, którzy zdali egzamin z fizyki, należy wykonać operację *rzutowania* na atrybut (pole) *Nazwisko_Stud* z relacji *Wyniki_Egz* pod warunkiem *Wyniki_Egz* = ‘Fizyka’. Formalny zapis tych operacji wygląda tak:

$$\pi_{\text{Nazwisko_Stud}}(\sigma_{Przedm = \text{'Fizyka'}}(\textbf{Wyniki_Egz})).$$

Aby uzyskać listę studentów, którzy powinni zdawać egzamin z fizyki, należy wykonać operację *rzutowania* na atrybut *Nazwisko_Stud* po operacji złączenia relacji *Studenci* i *Egzaminy, pod warunkiem że Przedm* = ‘Fizyka’:

$$\pi_{\text{Nazwisko_Stud}}(Studenci \rhd\lhd \underbrace{Egzaminy\ (Przedm = \text{'Fizyka'})}_{\Theta}).$$

Zadania do samokontroli

1. Podać znaczenie terminów:
   - algebra relacyjna;
   - suma;
   - przecięcie;
   - różnica;
   - iloczyn kartezjański;
   - selekcja;
   - projekcja;
   - złączenie naturalne;
   - iloraz.
2. Podać przykłady zastosowania operacji algebry relacyjnej.
3. Przedstawić graficznie operacje nad relacjami (tam, gdzie to jest możliwe).
4. Czy istnieje różnica w wynikach wykonania operacji:
   - **R** UNION **S** a **S** UNION **R**;
   - **R** INTERSECT **S** a **S** INTERSECT **R**;
   - **R** MINUS **S** a **S** MINUS **R**;
   - **R** JOIN **S** a **S** JOIN **R**?
5. Podać formalny zapis operacji algebry relacyjnej.
6. Do Prz. 2-26 napisać operacje nad relacjami, aby uzyskać następujące informacje:
   - listę studentów nieobecnych na egzaminach;
   - listę studentów, którzy powinni zdawać egzamin z *Baz danych*;
   - listę studentów grupy A1;
   - listę studentów, którzy uzyskali niedostateczne oceny z *Matematyki*;
   - numery *Indeksów Studentów*, którzy (studenci) uzyskali pozytywne oceny z *Informatyki.*
7. Na podstawie podanych niżej relacji (tabel) *Studenci* i *Egzaminy*:

Tabela *Studenci*

Indeks	*Nazw_Stud*	*Przedm*	*Data*
1-A-07	Werner	Fizyka	10.02.08
1-A-07	Werner	Matematyka	15.02.08
2-A-07	Jabłoński	Fizyka	10.02.08
2-A-07	Jabłoński	Matematyka	15.02.08
3-A-07	Maczulska	Fizyka	10.02.08

Tabela *Egzaminy*

Przedm	*Data*
Fizyka	10.02.08
Matematyka	15.02.08

pokazać przykłady zastosowania operacji Codda.

3. Koncepcje i etapy projektowania baz danych

Ta część książki poświęcona będzie najważniejszym aspektom tworzenia BD. Powszechnie znany jest fakt, iż kluczową rolę w osiągnięciu sukcesu większości systemów informatycznych (SI) odgrywa oprogramowanie. Jednakże często zdarza się, iż tworzone oprogramowanie jest bardzo skomplikowane w obsłudze, a także charakteryzuje się niskim poziomem niezawodności.

Według badań przeprowadzonych w 1996 r. w Wielkiej Brytanii przez specjalną grupę tylko 10-20% stworzonych systemów można uważać za udane. Wyniki tych badań świadczą również o tym, że:

- ponad 80-90% systemów komputerowych nie posiada wymaganej efektywności;
- opracowanie ponad 40% systemów nie zostało w ogóle ukończone;
- około 40% stworzonych systemów wymagało istotnego podniesienia poziomu wiedzy użytkowników.

Szczegółowa analiza tej sytuacji prowadzi do ważnego wniosku: nieskuteczność opracowania oprogramowania związana jest głównie z tym, że nie zastosowano efektywnej metodologii opracowywania systemów.

Aby rozwiązać ten problem, została zaproponowana *metoda strukturalna*. Jądro tej metody to *koncepcja cyklu życia* systemów informatycznych.

3.1. Planowanie tworzenia i projektowanie baz danych

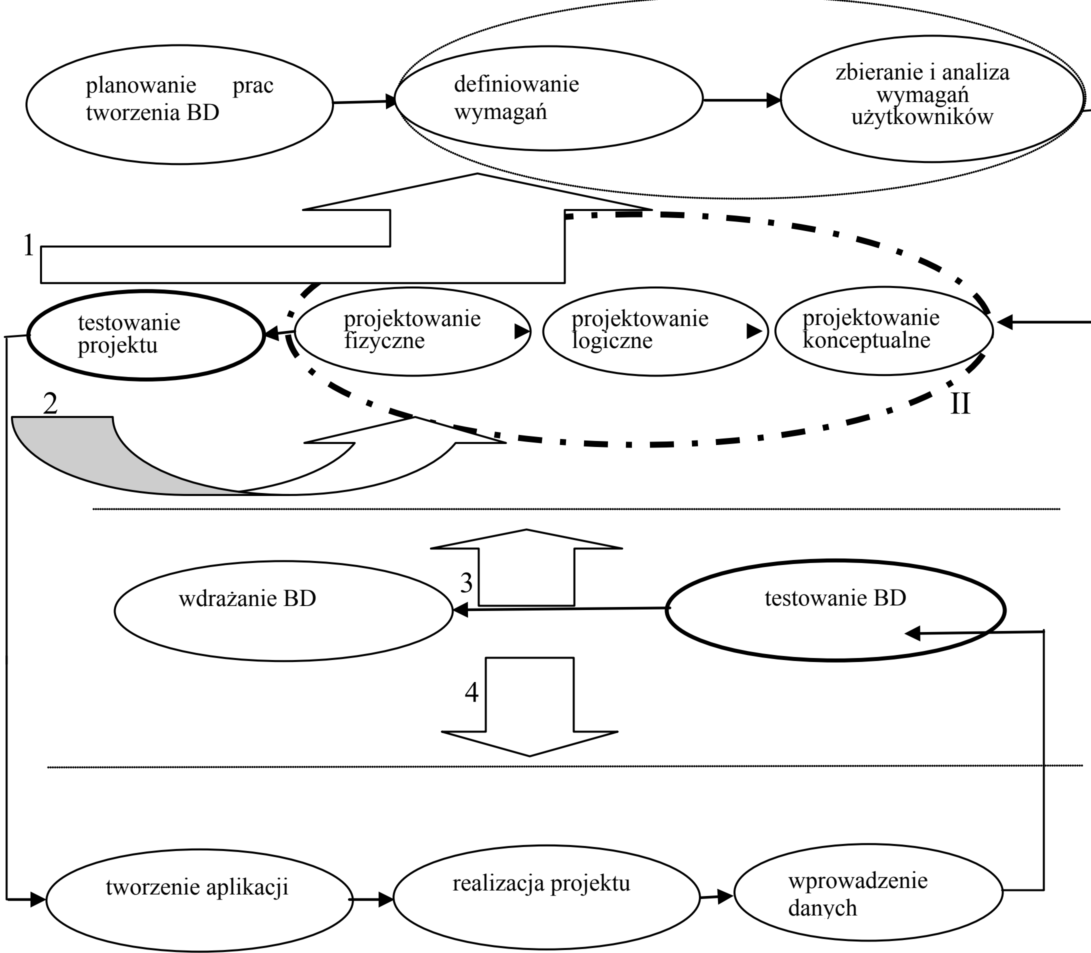
3.1.1. Cykl życia BD

Tak jak każdy inny typ oprogramowania, BD posiada własny cykl życia. Ten cykl systemu bazodanowego determinuje cykl życia całego SI, gdyż BD jest ważniejszym komponentem SI.

Cykl życia BD składa się z następujących etapów (rys. 3.1):

- planowanie prac tworzenia BD;
- definiowanie wymagań systemu;
- zbieranie i analiza wymagań użytkowników;
- projektowanie BD:

  - projektowanie koncepcyjne,
  - projektowanie logiczne,
  - projektowanie fizyczne;
- testowanie projektu;
- tworzenie aplikacji:
  - projektowanie transakcji,
  - projektowanie interfejsu użytkownika;
- realizacja całego projektu;
- wprowadzenie danych do bazy;
- testowanie systemu BD;
- wdrażanie BD.



Rys. 3.1. Schemat powiązań pomiędzy etapami cyklu życia BD

W pierwszym etapie należy opracować *strategiczny plan tworzenia BD* i korzystania z niej. Ważnymi częściami prac na tym etapie jest wybór konkretnego (lub konkretnych) modelu (modeli) SZBD i ogólny model informatyczny całego systemu. Podstawowym

celem etapu jest wstępne obliczenie kosztów realizacji projektu. Jeżeli wynik kontroli realizacji projektu będzie dodatni, należy przejść do definiowania wymagań projektu.

Na etapie *definiowania wymagań projektu* należy zdefiniować okres działań aplikacji BD, użytkowników BD, poziom ich wiedzy, możliwe dziedziny zastosowań BD (przede wszystkim z punktu widzenia wymagań różnych oddziałów instytucji (firmy), na potrzeby której tworzy się BD).

Etap *zbierania i analizy wymagań użytkowników* jest wstępnym etapem *projektowania koncepcyjnego*.

Bezpośredni cykl projektowania BD składa się z *etapów projektowania koncepcyjnego*, *logicznego* i *fizycznego* (na rys. 3.1 oznaczone cyfrą II).

Generalnie podstawowe cele czwartego etapu projektowania można sformułować następująco:

- *definiowanie danych i połączeń* między nimi, niezbędnych dla wszystkich dziedzin zastosowania aplikacji i dowolnych grup użytkowników;
- stworzenie *modelu danych*, który może wspierać wykonanie dowolnych transakcji; *optymalny model danych* powinien odpowiadać następującym kryteriom:
  - strukturalna dokładność,
  - prostota,
  - brak nadmiarowości,
  - jednolitość danych,
  - możliwość rozszerzania.

Po wypełnieniu tych działań projekt powinien być szczegółowo *przetestowany*. Najlepiej, jeżeli projekt będą testować osoby niezainteresowane (trzecia strona). Z wynikami testów powinni się zapoznać użytkownicy BD.

Zadaniem projektantów jest wprowadzenie do projektu odpowiednich poprawek (formalnie oznaczono to strzałkami 1 i 2 na rys. 3.1). Warto tutaj podkreślić, że projektowanie BD, jak i dowolne inne projektowanie jest procesem iteracyjnym.

Celem etapu *tworzenia aplikacji* jest *projektowanie transakcji* i *interfejsu użytkownika*. Ostatni etap jest najważniejszym komponentem systemu.

Podczas projektowania interfejsu należy zwrócić uwagę na:

- jasne i zrozumiałe instrukcje (punkty menu);
- zrozumiałe terminologie i skróty;
- narzędzia, informujące użytkownika o błędach;
- narzędzia korekcji błędów podczas wprowadzenia danych;
- zastosowanie „przyjaznych dla oka” kolorów;
- wizualizację ważniejszych obiektów (poprzez grafikę) i in.

Realizacja projektu powinna umożliwiać użytkownikom proste formułowanie zapytań do BD i manipulowanie danymi. Na tym etapie również realizują się narzędzia ochrony danych i integralności BD.

Po *wprowadzeniu danych* cały stworzony system musi być po raz drugi *przetestowany* z wypełnieniem niezbędnych korekcji (strzałki 3 i 4 na rys. 3.1).

3.1.2. Podstawowe etapy projektowania BD

W literaturze często drugi i trzeci etap projektowania rozpatruje się łącznie, a tak połączone etapy nazywa się etapem *analizy wymagań* (na rys. 3.1 te połączone etapy oznaczone są przez I).

W wyniku zbierania niezbędnych informacji (najczęściej na podstawie rozmów z użytkownikami lub ich ankietowania) projektanci BD muszą mieć dokładną wiedzę przede wszystkim o wykonywanych transakcjach dotyczących danych projektowanej BD.

Na podstawie tego modelu zostaną określone specyfikacje wymagań użytkowników.

Systemy bazodanowe z niepełną funkcjonalnością często mogą być dla użytkowników źródłem nieprzyjemnych niespodzianek. Dodatkowo zbyt duże zwiększanie funkcjonalności systemu prowadzi do nadmiernego stopnia jego skomplikowania. Dlatego też tak ważny jest etap zbierania i analizy wymagań użytkowników.

Rozpatrzymy teraz dwa przykłady, pozwalające zrozumieć zasygnalizowane powyżej problemy.

Przykład 3-1
Zakładamy, że mamy opracować projekt BD *Szkoła*.

Najistotniejsze wyniki realizacji *etapu analizy wymagań* mogą wyglądać następująco:

- użytkownikami BD będą nauczyciele i administracja szkoły;
- od BD wymaga się, by było możliwe wykonanie następujących operacji: dodawanie, usuwanie, modyfikacja, pobieranie danych o uczniach, nauczycielach, klasach, zajęciach, rodzicach, salach oraz przydzielanie lub odwoływanie obowiązków nauczycieli.

W ten sposób celem tworzonego projektu jest wspomaganie zarządzania szkołą.

Następny przykład przedstawia bardziej szczegółowy opis wszystkich okoliczności dotyczących etapu analizy wymagań.

Przykład 3-2
Załóżmy, iż mamy do czynienia z tworzeniem projektu BD *Wypożyczalnia filmów*.

Ponieważ użytkownikami tej bazy będą właściciel wypożyczalni, pracownicy i osoby wypożyczające filmy, musi ona umożliwiać wykonywanie następujących operacji:

- dodawanie, usuwanie filmów z bazy danych, pobranie oraz edycja informacji o danym filmie;
- dodanie klienta, odczytywanie informacji o kliencie, usunięcie klienta z bazy danych, edycja informacji o nim;
- przyjęcie nowego pracownika, pobranie oraz edycja informacji o nim, usunięcie pracownika z bazy;
- dodanie regałów, na których umieszczone są filmy, edycja informacji o nich, usunięcie danego regału z bazy;
- odczytywanie informacji o danym pokoju, w którym przechowywane są filmy, dodanie i usunięcie pokoju z bazy;
- dodanie nowego nośnika, na których przechowywane są filmy, a także usunięcie go z bazy;

- dodanie oraz usunięcie kategorii filmów oferowanych przez wypożyczalnię;
- wypożyczenie kasety oraz przechowywanie informacji o danym wypożyczeniu;
- dodawanie do bazy informacji o nowym dostawcy filmów do wypożyczalni, edycja tych informacji oraz usunięcie danego dostawcy z bazy, z którym wypożyczalnia zakończyła współpracę;
- składanie nowych zamówień u dostawców;
- gromadzenie informacji o przyjętych dostawach towarów;
- dodawanie nowych pozycji do danego zamówienia.

Ponadto stworzona baza danych musi dostarczać szereg informacji niezbędnych do prowadzenia działalności, w szczególności dane w niej zawarte muszą udzielać odpowiedzi na następujące pytania:

- Jakie obecnie wypożyczalnia oferuje klientom filmy do wypożyczenia?
- Na jakich nośnikach są dostępne poszczególne filmy?
- Jakie filmy są dostępne z danej kategorii (westerny, sensacje, komedie itd.)?
- Jakie filmy danego reżysera są dostępne?
- Na którym regale i w jakim pokoju można szukać danego filmu?
- Ile obecnie znajduje się egzemplarzy danego filmu w wypożyczalni?
- Ilu pracowników jest obecnie na danym stanowisku?
- Jaki pracownik jest odpowiedzialny za dany pokój (informacja niezbędna w przypadku zaginięcia danej kasety)?
- Ile dany klient ma na swoim koncie wypożyczonych filmów (również ile jest takich filmów, z których oddaniem się spóźnia lub spóźnił w przeszłości)?
- Jaki film cieszy się największym powodzeniem w danym okresie?
- Jakie nośniki i kategorie filmów cieszą się największym powodzeniem?
- Którzy klienci najczęściej oddają kasety po terminie?
- Który pracownik wypożyczył największą liczbę filmów w danym okresie?
- Jakie filmy zostały wypożyczone przez danego klienta w danym okresie?
- Który film przyniosł największe dochody w danym okresie?
- Ile jest złożonych zamówień u konkretnego dostawcy?
- Z jakimi filmami jest związane dane zamówienie?
- Które z zamówień zostały już zrealizowane (dostarczone) w całości, które częściowo, a które w ogóle nie zostały zrealizowane?
- Ile jeszcze dni zostało do uregulowania należności za konkretną dostawę?
- Ile jest takich dostaw, za które nie uregulowano należności?

Oprócz tego przy tworzeniu projektu zostały uwzględnione następujące uwagi:

- kilka różnych filmów może mieć ten sam tytuł;
- jeden tytuł może być dostępny na wielu nośnikach;

- dla pomieszczenia, gdzie są wystawiane filmy, jest wyznaczony jeden pracownik, który jest za nie odpowiedzialny;
- cena danego filmu nie tylko zależy od jego tytułu, lecz także od nośnika, na jakim zostaje wypożyczony;
- dane zamówienie filmów u dostawców może być zrealizowane w wielu dostawach (dane filmy mogą być dostarczane w różnych okresach, mimo że są na jednym zamówieniu);
- konkretny film na ogół jest dostępny w wypożyczalni w wielu egzemplarzach dla każdego z nośników;
- zdarza się, że dany klient kilkakrotnie wypożycza ten sam film;
- w danej chwili klient może wypożyczyć tylko jeden egzemplarz danego filmu;
- konkretna kategoria filmów niekoniecznie jest przetrzymywana na danym regale czy pokoju, tzn. filmy z danej kategorii mogą być na kilku różnych regałach i w różnych pokojach, jak i w danym pokoju i na danym regale mogą być przetrzymywane filmy z różnych kategorii;
- to samo tyczy się również nośników, na jakich są dostępne filmy;
- zdarza się, że dostawy są niezgodne ze złożonymi zamówieniami;
- pojemność regału, tzn. maksymalna liczba filmów, jaka może być przechowywana na regale, jest niezależna od tego, na jakim nośniku będą przechowywane tam filmy.

Powyższe uwagi będą miały istotny wpływ na powiązania pomiędzy tabelami, głównie na typy i stopnie uczestnictwa.

W ten sposób *celem tworzonego projektu jest dostarczenie właścicielowi i pracownikom wypożyczalni takiej bazy danych, w której będą zawarte informacje, w znaczący sposób ułatwiające i poprawiające efektywność prowadzonej działalności*.

Pierwszy krok czwartego etapu projektowania – *projektowanie konceptualne* – prowadzi do tworzenia *schematu koncepcyjnego* BD. Ten proces może być z kolei podzielony na podetapy:

- określenie struktury zależności hierarchicznych pomiędzy jednostkami analizowanego systemu, zwłaszcza w zakresie specyfikacji wymagań funkcjonalnych;
- budowa i analiza diagramu przepływu danych; ma na celu określenie przepływu danych (wejścia, wyjścia, operacje, przechowywanie) oraz elementów sterowania tym przepływem, co może być pomocne w tworzeniu aplikacji;
- wybór encji (obiektów) i ich opis; projektowanie powiązań (relacji) pomiędzy encjami; konstrukcja diagramu ER-D; jest to zasadniczy etap procesu projektowania struktury bazy danych).

Przykład 3-3

Dalszy rozwój projektu BD *Wypożyczalnia filmów* (patrz Prz. 3-1) może wyglądać następująco.

Szczegółowy opis encji oraz wchodzących w ich skład atrybutów jest przedstawiony w tab. 3.1.

Tabela 3.1. Szczegółowy opis encji dla BD *Wypożyczalnia filmów*

Encja	*Atrybut*	*Opis*
Kaseta	Tytuł	Tytuł filmu
	Reżyser	Nazwisko reżysera filmu
	Rok Produkcji	Rok produkcji filmu
	Czas	Długość filmu (w minutach)
	Liczba Sztuk	Liczba sztuk danej kasty w wypożyczalni
	Cena	Cena w złotych za dobę
	Typ Kasety	Typ kasety, na której znajduje się film
	Kategoria	Kategoria, do której należy dany film
	Regał	Regał, na którym znajduje się dana kaseta
Klient	Imię	Imię klienta
	Nazwisko	Nazwisko klienta
	Adres	Adres klienta
	Telefon	Numer telefonu klienta
	Status	Czy aktywny („ A”), czy nie („N”)
Pracownik	Imię	Imię pracownika
	Nazwisko	Nazwisko pracownika
	Adres	Adres pracownika
	Telefon	Numer telefonu pracownika
	Stanowisko	Stanowisko, jakie zajmuje pracownik
	Stawka	Stawka w złotych za godzinę
	Data Podpisania Umowy	Data, w której została zawarta umowa z danym pracownikiem
	Rodzaj Umowy	Rodzaj umowy zawartej z pracownikiem (umowa o dzieło, na czas próbny, stała)
	Status	Czy aktywny („A”), czy nie („N”)
Pokój	Powierzchnia	Powierzchnia w m^2
	Odpowiedzialny	Pracownik, który jest odpowiedzialny za dany pokój
Regał	Pojemność	Pojemność w sztukach danego regału (ile maksymalnie kaset może być na nim ułożonych)
	Pokój	Pokój, w którym znajduje się dany regał
	Opis	Dodatkowe informacje o regale
Kategoria	Nazwa Kategorii	Nazwa kategorii (np. sensacja, komedia)
	Opis	Dodatkowe informacje o kategorii
Nośnik	Nazwa Nośnika	Nazwa nośnika (np. wideo, DVD, VCD)
	Opis	Dodatkowe informacje o nośniku
Dostawca	Nazwa	Nazwa firmy dostarczającej nowe kasety do wypożyczalni
	Adres	Adres dostawcy
	Telefon	Numer telefonu dostawcy
	NIP	Numer NIP dostawcy
	Status	Czy aktywny („ A”), czy nie („N”)

Encja	*Atrybut*	*Opis*
Zamówienie	Data Złożenia	Data złożenia danego zamówienia
	Dostawca	Dostawca, u którego złożono zamówienie
Dostawa	Zamówienie	Zamówienie, którego tyczy się dana dostawa
	Data Dostawy	Data przybycia danej dostawy do wypożyczalni
	Termin Płatności	Termin płatności w dniach za daną dostawę
	Status	Czy zapłacono („Z”), czy też nie („N”)

Dalej przedstawimy również szczegółowy opis powiązań pomiędzy encjami wraz z typami uczestnictwa encji w tych relacjach, a więc *uczestnictwem obowiązkowym,* zaznaczonym jako (1,1) lub (1,n) i *opcjonalnym,* zaznaczonym jako (0,1) lub (0,n):

KASETA (0,n) ←→ (0,n) KLIENT

Dany klient może wypożyczyć wiele kaset, również dana kaseta może być wypożyczana przez wielu klientów. Jednak mogą być takie kasety, które akurat nie są wypożyczone, jak i mogą być oczywiście klienci, którzy akurat nie mają na koncie żadnej kasety.

KASETA (0,n) ←→ (0,n) PRACOWNIK

Kaseta może być wypożyczana przez wielu pracowników, również dany pracownik może wypożyczyć klientom wiele kaset. Mogą istnieć pracownicy, którzy nie wypożyczyli żadnej kasety (np. sprzątaczka).

KASETY (0,n) ←→ (1,1) NOŚNIK

Kaseta może mieć tylko jeden nośnik (zakładamy, że ten sam film, ale na dwóch różnych nośnikach stanowi dwa różne obiekty w późniejszej tabeli **KASETY**), na danym nośniku może być dostępnych wiele kaset (w znaczeniu filmów). Każda kaseta musi mieć swój nośnik (tylko jeden), natomiast mogą być nośniki, na których nie ma jeszcze żadnego filmu (np. wprowadzane nowe nośniki jeszcze w fazie testów).

KASETA (0,n) ←→ (1,1) KATEGORIA

Każdy film należy do danej kategorii (i tylko jednej), natomiast może być wiele filmów z danej kategorii. Mogą być również kategorie, dla których nie ma żadnego filmu w wypożyczalni (np. wypożyczalnia tymczasowo może wycofać filmy z danej kategorii).

KASETA (0,n) ←→ (1,1) REGAŁ

Każdej kasecie jest przyporządkowany dany regał (tylko jeden – należy pamiętać, że dany film, ale na dwóch różnych nośnikach stanowi dwa obiekty, dlatego też mogą być one na dwóch różnych regałach). Na danym regale może znajdować się wiele kaset, mogą również istnieć w wypożyczalni regały, na których nie przechowuje się kaset.

REGAŁ (0,n) ←→ (1,1) POKÓJ

Jeden regał jest umieszczony tylko w jednym pokoju, natomiast jeden pokój może mieścić wiele regałów, mogą być również pokoje w wypożyczalni, w których nie ma żadnych regałów (np. które nie służą do przechowywania kaset).

POKÓJ (0,n) ← → (0,1) PRACOWNIK

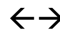
Pokój, w którym przechowywane są kasety, ma wyznaczonego pracownika, który jest za ten pokój odpowiedzialny. Istnieją pokoje, gdzie kaset nie ma, stąd nie musi też być wyznaczony odpowiedzialny pracownik. Jeden pracownik może być odpowiedzialny za kilka pokoi, istnieją też pracownicy, którzy nie są odpowiedzialni za żaden pokój.

KASETA (1,n) ←→ (0,n) **ZAMÓWIENIE**

Dana kaseta może być pozycją wielu zamówień, istnieją też kasety, które nie są już zamawiane (filmy stare). Na jednym zamówieniu może być wiele pozycji (wiele kaset). Nieskładane są zamówienia, na których nie ma żadnej pozycji, a więc każde zamówienie ma przynajmniej jedną (pozycję).

KASETA (1,n) ←→ (0,n) **DOSTAWA**

Dana kaseta może być dostarczana w wielu dostawach, istnieją kasety, które nie są akurat dostarczane. W jednej dostawie może być wiele pozycji, a także istnieją dostawy, w których nie ma żadnej pozycji, a zatem każda dostawa ma przynajmniej jedną pozycję.

DOSTAWA (0,n)  (1,1) **ZAMÓWIENIE**

Jedna dostawa może dotyczyć tylko jednego zamówienia, każda dostawa musi też być związana z jakimś zamówieniem. Dane zamówienie może być zrealizowane w kilku dostawach, istnieją również zamówienia jeszcze niezrealizowane w ogóle.

ZAMÓWIENIE (0,n) (1,1) **DOSTAWCA**

Każde zamówienie musi być skierowane do jakiegoś dostawcy (i tylko jednego). Do konkretnego dostawcy może być skierowanych wiele zamówień, istnieją również dostawcy, do których akurat nie skierowano żadnego zamówienia.

Na rys. 3.2 została przedstawiona konstrukcja ER-D dla analizowanej BD. Właśnie projekt konceptualny BD najczęściej ma wygląd ER-D.

Etap *logicznego projektowania* często określa się terminem *modelowanie danych* i obejmuje następujące kroki:

- projektowanie tabel, kluczy, indeksów itd. na podstawie zdefiniowanego diagramu ER-D; na tym etapie następuje „sprecyzowanie" struktury bazy danych wraz ze szczegółami technicznymi;
- stworzenie słownika danych; specyfikacja słownika danych;
- analiza *zależności funkcyjnych* i *normalizacja tabel* (*dekompozycja* do jednej z *postaci normalnych*); na tym etapie dokonuje się sprawdzenia, czy tabele spełniają warunki zakładanych postaci normalnych i ewentualnie dokonuje się dekompozycji w celu normalizacji;
- projektowanie operacji na danych: zdefiniowanie kwerend dla realizacji funkcji wyspecyfikowanych w projekcie; (zgodnie z wymaganiami użytkownika; na tym etapie mogą one zostać uszczegółowione bądź zmodyfikowane); projekt w języku SQL.

Poniższy przykład ilustruje możliwy początkowy wygląd jednej z tabel.

Przykład 3-4

Mamy następujący początkowy wygląd tabeli o nazwie *Wypożyczenia* z BD *Wypożyczalnia filmów* (Prz. 3-1).

Tabela 3.2. Wypożyczenia

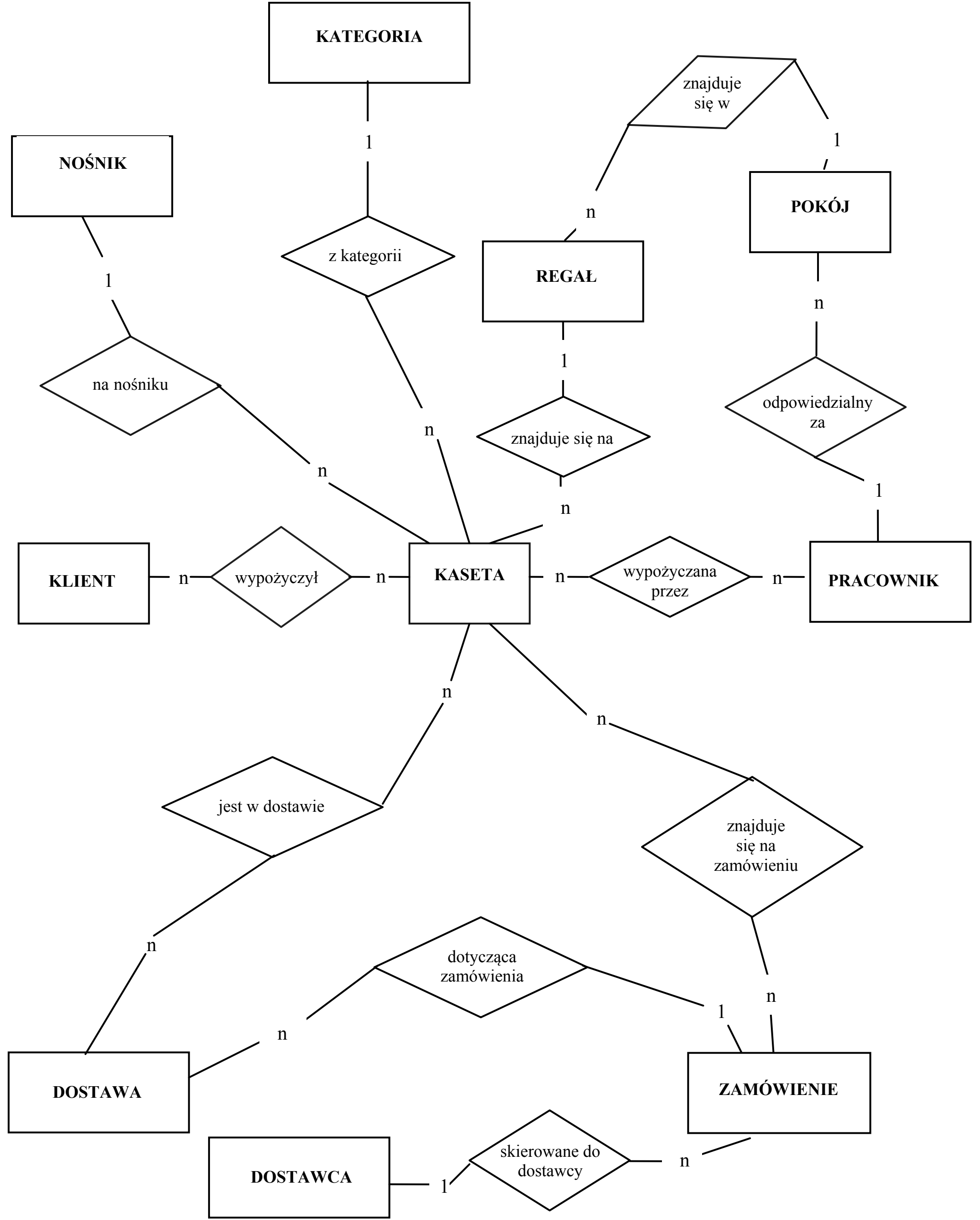
Id_Kasety	*Id_Klienta*	*Data_Wypoż*	*Id_Prac*	*Okres*	*Tytuł*	*Reżyser*	*Imię_Prac*	*Nazwisko_Prac*
1 3	5	05.02.2006 05.04.2005	15	3 2	„Matrix” „Obcy”	Wachowscy Scott	Roman	Wójcik
1	8	12.03.2006	3	5	„Matrix”	Wachowscy	Maria	Nowak
8	2	14.04.2006	2	1	„Predator”	McTiernan	Krzysztof	Kowalski
12 13	4	15.03.2006 16.03.2005	15 18	1	„The ring” „Taxi”	Nakata Pires	Roman Hanna	Wójcik Malinowska
4	14	18.04.2006	3	2	„Kiler”	Machulski	Maria	Nowak

Kluczem podstawowym tej tabeli jest klucz złożony, składający się z pól *Id_Kasety*, *Id_Klienta* oraz *Data_Wypoż*. Wybór takiego klucza wynika z tego, iż dana kaseta może być wypożyczana wiele razy, dany klient może również wypożyczać daną kasetę wiele razy, jednak w danej chwili jeden klient może wypożyczyć daną kasetę tylko raz. Dlatego też kombinacja tych trzech pól w sposób jednoznaczny będzie nam identyfikowała każdy rekord relacji.

Modelowanie danych jest etapem kluczowym całego projektu i wymaga wiedzy o *normalizacji danych*, a także związanych z nią pojęć, takich jak *zależności funkcyjne*.

Zadania do samokontroli

1. Przedstawić cykl życia BD i jego podstawowe etapy.
2. Wymienić główne cele procesu projektowania BD.
3. Podać przykłady konceptualnych modeli BD:
   - *Biblioteka*;
   - *Apteka*;
   - *Restauracja*;
   - *Przychodnia*;
   - *Szpital*;
   - *Szkoła podstawowa*;
   - *Wydział uczelni wyższej*;
   - *Stacja paliw*;
   - *Wypożyczalnia samochodów*;
   - *Wydawnictwo*;
   - *Księgarnia*.
4. Zaprojektować tabele i zdefiniować w nich klucze dla BD z pkt 3.



Rys. 3.2. Konstrukcja ER-D dla BD *Wypożyczalnia filmów*

3.2. Normalizacja BD

3.2.1. Cele i podstawowe pojęcia normalizacji

Normalizacja jest bardzo ważnym etapem logicznego projektowania BD.

Przeanalizujmy następujący przykład.

Przykład 3-5

Mamy relację (tabelę) *Producenci_Komputerów* {***Id_Prod***, *Miasto*, ***Id_Komp***, *Cena_Komp*} (nazwy atrybutów (pól) kluczowych są pogrubione). Ta tabela posiada złożony klucz podstawowy (ZKP), składający się z pól *Id_Prod* i *Id_Komp*. Tabela *Producenci_Komputerów* wygląda następująco (tab. 3.3).

Tabela 3.3. Producenci_Komputerów

Id_Prod	*Miasto*	***Id_Komp***	*Cena_Komp*
P1	Lublin	K1	3000
P1	Lublin	K2	2500
P1	Lublin	K3	3600
P2	Bydgoszcz	K1	3000
P2	Bydgoszcz	K3	3600
B1	Mińsk	K2	2500
B1	Mińsk	K3	3600

Z analizy tab. 3.3 widać, że w tabeli tej występuje dość istotna wada, związana z powtarzającymi się danymi, którą określa się w literaturze jako *nadmiarowość (redundancja) danych*. Mówiąc dokładniej, w każdym wierszu tabeli z *Id_Prod* = P1 mamy powtarzającą się wartość w polu *Miasto*: *Miasto* = 'Lublin', przy *Id_Prod* = P2 odpowiednio *Miasto* = 'Bydgoszcz' itd. Istnienie w tabeli tego typu powtarzających się wartości jest dość istotną wadą. Mianowicie, podczas wprowadzenia danych w polu *Miasto* (dla *Id_Prod* = P1) może mieć miejsce sytuacja, w której w jednym wierszu będzie wpisana wartość *Lublin*, a w drugim pomyłkowo wpisana przykładowo wartość *Mińsk*. Dodatkowo, formalnie tab. 3.3 posiada 7 wierszy i 4 kolumny, czyli razem 28 wartości; jeżeli jednak tab. 3.3 podzielimy (wykonując operację *dekompozycji*) na dwie tabele (załóżmy *Producenci*{***Id_Prod***, *Miasto*} i *Komputery*{***Id_Prod***, *Miasto*}, tak jak pokazano w tab. 3.4 i 3.5), to każda z nowych tabel nie posiada nadmiarowych danych i łączna ilość wartości w obu tabelach wynosi 12.

Tabela 3.4. Tabela *Producenci*

Id_Prod	*Miasto*
P1	Lublin
P2	Bydgoszcz
B1	Mińsk

Tabela 3.5. Tabela *Komputery*

Id_Komp	*Cena_Komp*
K1	3000
K2	2500
K3	3600

Z formalnego punktu widzenia rozdzielenie tabeli na dwie mniejsze dało możliwość realizacji podstawowego wymagania dotyczącego struktury tabeli: *jeden fakt (jedna wartość) w jednym miejscu*. Ponadto mniejszy rozmiar tabel daje możliwość zwiększenia

szybkości wyszukiwania danych. Podsumowując: *poprawa jakości funkcjonowania tabel (w kontekście: unikania nadmiarowości oraz poprawy efektywności) jest podstawowym celem normalizacji tabel*.

Inną niepożądaną cechą, którą można usunąć w procesie normalizacji, są powtarzające się lub *niespójne zależności*[1] pomiędzy danymi, utrudniające dostęp do nich.

Sytuacja, która została przeanalizowana w Prz. 3-5, powoduje, iż wykorzystywana jest dużo większa liczba pamięci dyskowej w celu przechowywania danych z tabeli *Producenci_Komputerów*. Może ona również być przyczyną problemów nazywanych *anomaliami aktualizacji*, które dzielą się na *anomalie wstawiania (dołączania)*, *anomalie usuwania* oraz *anomalie modyfikacji*.

Przeanalizujmy problemy anomalii na dodatkowym przykładzie.

Przykład 3-6

Mamy relację (tabelę) *Modele_Marki* {***Id_Modelu***, *Nazwa_Modelu*, *Rok_Produkcji*, *Id_Marki*, *Nazwa_Marki*}.

Tabela 3.6. Relacja *Modele_Marki*

Id_Modelu	*Nazwa_Modelu*	*Rok_Produkcji*	*Id_Marki*	*Nazwa_Marki*
AS	Astra	1995	OP	Opel
VE	Vectra	1996	OP	Opel
CA	Carina	2003	TO	Toyota
PA	Panda	1994	FI	Fiat

Anomalia wstawiania (na podstawie powyższego przykładu):

- przy wstawianiu nowego modelu samochodu musimy wstawić również wszystkie informacje dotyczące marki samochodu, do której ten model należy. Co więcej, wstawione dane dotyczące marki muszą być zgodne z wcześniej wstawionymi informacjami o tej marce dla każdego z pozostałych modeli tej marki. A więc musimy wstawić więcej informacji, niż wydawałoby się to potrzebne;
- wprowadzenie nowej marki, dla której nie ma przypisanego żadnego modelu, staje się niemożliwe, ponieważ pole *Id_Modelu* jest kluczem relacji. Stąd próba wstawienia rekordu z pustą wartością w polu *Id_Modelu* będzie naruszała integralność na poziomie tabeli, która mówi o tym, iż pola kluczowe nie mogą być puste.

Anomalia usuwania dotyczy sytuacji, gdy po usunięciu z relacji ostatniego modelu danej marki utracimy wszystkie informacje o tej marce. Na przykład usunięcie modelu o identyfikatorze CA spowoduje jednocześnie usunięcie informacji o marce Toyota.

Anomalia modyfikacji dotyczy sytuacji, gdy zmianę danych o jakiejś marce (np. nazwy marki) musimy dokonać w każdym rekordzie zawierającym informacje o tej marce, a więc dla każdego modelu należącego do tej marki. Jeżeli taka zmiana nie zostałaby dokonana w każdym z tych rekordów, mogłoby się okazać, że jedną markę opisywałyby dwa zupełnie różne zestawy informacji (np. jedna marka miałaby dwie różne nazwy), co byłoby oczywiście sytuacją niedopuszczalną, naruszającą poprawność danych w bazie. Wobec tego w celu dokonania aktualizacji danych musimy dokonać tej samej operacji kilka razy.

[1] Zależności pomiędzy danymi zostaną przeanalizowane w podrozdz. 3.2.2.

Tych anomalii możemy uniknąć dzięki *dekompozycji* naszej relacji (tabeli) na dwie mniejsze relacje: *Modele* (tab. 3.7) i *Marki* (tab. 3.8).

Tabela 3.7. *Modele*

Id_Modelu	*Nazwa_Modelu*	*Rok_Produkcji*	*Id_Marki*
AS	Astra	1995	OP
VE	Vectra	1996	OP
CA	Carina	2003	TO
PA	Panda	1994	FI

Tabela 3.8. *Marki*

Id_Marki	*Nazwa_Marki*
OP	Opel
TO	Toyota
FI	Fiat

Widzimy, że po podzieleniu tabeli na dwie mniejsze, odpowiadające modelom i markom samochodów, pozbyliśmy się *anomalii aktualizacji*. Teraz wystarczy tylko raz wprowadzić markę samochodu, a modele tej marki będziemy mogli dołączać całkowicie niezależnie – to świadczy o usunięciu *anomalii dodawania*. W każdej chwili możemy dowiedzieć się, jaką nazwę ma marka danego modelu dzięki istnieniu związku między tabelami. W tabeli *Modele* oprócz klucza podstawowego, którym jest ***Id_Modelu***, mamy również klucz obcy *Id_Marki*, odpowiadający kluczowi podstawowemu tabeli *Marki*. Usunięcie ostatniego modelu danej marki nie spowoduje usunięcia informacji o tej marce, gdyż trzymane są one w oddzielnej relacji, a więc nie ma już też *anomalii usuwania*. Również w przypadku potrzeby aktualizacji informacji o marce musimy dokonać tego tylko raz – w tabeli *Marki*, co świadczy o usunięciu *anomalii modyfikacji*.

Normalizacja relacji (tabeli) polega na rozdzieleniu (dekompozycji) tej relacji (tabeli) na dwie lub więcej mniejszych relacji (tabel).

Normalizacja opiera się na koncepcji *postaci normalnych* (PN) (ang. *normal form*). Początkowo E.F. Codd zdefiniował trzy PN (1PN, 2PN, 3PN) i udowodnił, że podczas projektowania BD należy dążyć do stworzenia takiej relacji (tabeli), żeby odpowiadała ona 3PN.

Mówimy, że relacja (tabela) znajduje się w *i*-tej PN (*i*>1), jeżeli znajduje się ona w *i-1* PN i spełnia dodatkowe warunki.

Następnie zaproponowano bardziej dokładną definicję 3PN[2]. W dzisiejszej literaturze mówi się, że relacja (tabela) odpowiadająca tej definicji znajduje się w *postaci normalnej Boyce'a-Codda* (PNBC). Z kolei R. Fagin[3] zdefiniował jeszcze dwie PN: 4PN i 5PN. W ten sposób obecnie mamy 6 (podstawowych) postaci normalnych.

---

[2] E.F. Codd, *Resent Envestigation into Relational Data Base System*, Proc IFIP Congres, Stockholm 1974.

[3] R. Fagin, *Multivalued dependencies and a new normal form for relational databases*, ACM Trans. on Database System (1977), pp. 262-278.

Przy podziale większych relacji na mniejsze muszą być zachowane następujące zasady: *bezstratnego złączenia* i *zachowania zależności*.

Definicja 3-1

Zasada bezstratnego złączenia (***dekompozycja bez strat***) oznacza, iż każdy stan oryginalnej relacji (tabeli) może być odtworzony ze stanów mniejszych relacji (tabel).

Definicja 3-2

Zasada zachowania zależności oznacza, że wszelkie zależności pomiędzy atrybutami relacji (polami tabeli) większej muszą być utrzymane poprzez wymuszenie pewnych zależności w relacjach (tabelach) mniejszych.

Podstawy *zależności funkcyjnych* przeanalizujemy w dalszej części, teraz natomiast wyjaśnimy za pomocą przykładów tylko te elementy, które dotyczą *dekompozycji relacji (tabeli) bez strat danych.*

Przykład 3-7

Mamy tabelę *Studenci* (tab. 3.9).

Tabela 3.9. *Studenci*

Id_Stud	*Nazwisko*	*Imię*
1	Jezierski	Marek
3	Kalinowski	Rafał
4	Lipska	Ewa
5	Jezierska	Ewa
6	Kalinowski	Piotr
7	Kalinowski	Marek

Możliwe warianty dekompozycji powyższej tabeli na dwie mniejsze tabele:

1. Studenci_1_a{Id_Stud} i Studenci_1_b{Nazwisko, Imię};
2. Studenci_2_a{Id_Stud, Nazwisko} i Studenci_2_b{Nazwisko, Imię};
3. Studenci_3_a{Id_Stud, Nazwisko} i Studenci_3_b{ Id_Stud, Imię}.

Jak widać z tego i z poprzednich przykładów, *dekompozycja tabel* jest efektem *pionowego* podziału tabeli. Teoretycznie istnieje i inny sposób rozdzielenia tabeli: *poziomy*, polegający na operacji *selekcji* (patrz: Prz. 2-18). Jednakże mówiąc o dekompozycji relacji (tabel), rozumiemy tę operację jako rozdzielenie pionowe.

Dekompozycja relacji (tabeli) praktycznie oznacza _pionowe_ rozdzielenie tej relacji (tabeli).

Wróćmy do trzech podanych wyżej wariantów dekompozycji tabeli *Studenci.*

Przypomnijmy sobie, iż dopasowanie dwóch relacji z różnymi schematami polega na operacji *złączenia* na podstawie *wspólnego atrybutu*.

W związku z tym pierwszy wariant dekompozycji w ogóle nie jest możliwy, bo w obu łączonych relacjach (tabelach *Studenci_1_a*{*Id_Stud*} i *Studenci_1_b*{*Nazwisko*, *Imię*}) nie ma wspólnego atrybutu (pola). W ostatnim wariancie dekompozycji taki wspólny element istnieje. Jednakże operacja *złączenia* w drugim wariancie nie jest operacją jednoznaczną, bo wspólny atrybut (*Nazwisko*) posiada powtarzające się wartości.

Podsumowując, podkreślmy, że *dekompozycja* relacji polega na zastosowaniu *operacji projekcji (rzutowania)*.

3.2.2. Zależności funkcyjne w BD

Proces normalizacji relacyjnych baz danych wykorzystuje głównie pojęcie *klucza* (głównego i kandydującego), jak i *zależności funkcyjnych* występujących między atrybutami relacji.

Pierwsze z tych pojęć zostało przeanalizowane w rozdz. I.

Definicja 3-3

Zależność funkcyjna (ang. *funcional dependence*) opisuje związek między atrybutami w relacji. Jeżeli ***A*** i ***B*** będą atrybutami relacji ***R***, to mówimy, że atrybut ***B*** jest zależny funkcyjnie od atrybutu ***A***, co będziemy formalnie zapisywać w następujący sposób:

$A \rightarrow B$,

jeżeli z każdą wartością atrybutu ***A*** jest powiązana dokładnie jedna wartość atrybutu ***B***.

Niech ***R*(*U*)** oznacza dowolną relację ***R*** z nagłówkiem ***U***.

Definicja 3-4

Zależnością funkcyjną nazywamy każdy zapis postaci $A \rightarrow B$, gdzie $A; B \subseteq U$. Mówimy wówczas, że ***B zależy funkcyjnie od A*** lub że ***A determinuje funkcyjnie B***.

Jeżeli w określonej relacji występuje *zależność funkcyjna* pomiędzy atrybutami ***A*** i ***B***, to w każdej krotce zawierającej tę samą wartość atrybutu ***A*** będziemy mieli taką samą wartość atrybutu ***B***. Ale dla danej wartości atrybutu ***B*** mogą istnieć różne wartości atrybutu ***A***, a więc *zależność odwrotna nie musi być spełniona*.

Przykład 3-8

W dalszej analizie wykorzystamy relację (tabelę) *Egzaminy* (tab. 3.10).

Zwróćmy teraz uwagę na pewne istotne elementy. Wraz z upływem czasu, gdy w tabeli będą pojawiać się nowe wiersze, relacja ta będzie się zmieniała, a więc w różnych momentach zapis *Egzaminy* {*Id_Stud, Nazwisko, Przedm, Ocena*} będzie oznaczał różne relacje. Jednak wszystkie te relacje, aby mogły być traktowane jako reprezentacje pewnej informacji o rozważanej rzeczywistości, muszą mieć pewne cechy stałe, niezmienne w czasie. Posiadanie tych cech jest tylko warunkiem koniecznym, a zatem nie oznacza to wcale, że każda relacja je posiadająca odwzorowuje poprawnie dane informacje.

Mogą istnieć relacje spełniające wszystkie wymagane warunki spójności (zwane też warunkami niesprzeczności lub więzami integralności), które nie odwzorowują żadnej poprawnej informacji o świecie rzeczywistym. Jeśli jednak któraś z relacji nie spełnia jakiegoś warunku niesprzeczności, to można z całą pewnością orzec, że jest ona *niepoprawna semantycznie*, tzn. że nie odpowiada jej żaden stan rzeczywistości, o którym informacje mogłaby ona reprezentować.

We wspomnianej relacji *Egzaminy* takimi niezmiennymi w czasie właściwościami, które wynikają z właściwości odwzorowywanego świata rzeczywistego, są np.:

- każdy *Id_Stud* ma jednoznacznie przyporządkowane *Nazwisko*, tzn. z jednym identyfikatorem studenta nie mogą być związane dwa różne nazwiska. Odwrotna sytu-

acja może jednak występować, tzn. z jednym nazwiskiem mogą być związane dwie różne osoby (lub jedna osoba, która studiuje na dwóch kierunkach), a więc dwa różne identyfikatory studenta;

- każdej parze *(Id_Stud, Przedm)* przyporządkowana jest jednoznacznie *Ocena*.

Tabela 3.10. *Egzaminy*

Id_Stud	*Nazwisko*	*Przedm*	*Ocena*
1	Jezierski	Matematyka	db
1	Jezierski	Bazy danych	bdb
1	Jezierski	Ekonometria	db plus
3	Kalinowski	Ekonometria	bdb
3	Kalinowski	Ochrona Informacji	db
4	Lipska	Ekonometria	dost
4	Lipska	Ochrona Informacji	db plus

Powyższe warunki wyrażają pewne semantyczne właściwości relacji, przy czym źródłem tych właściwości jest analiza rozpatrywanego fragmentu rzeczywistości. Mianowicie, pierwsza właściwość wyraża fakt, że dwaj różni studenci mogą mieć jednakowe nazwiska, muszą mieć jednak zawsze przyporządkowane różne identyfikatory studentów, druga natomiast mówi, że każdej parze: student i przedmiot, z którego student zdał egzamin, przyporządkowana jest jednoznacznie ocena.

Podane powyżej przykłady niezmiennych właściwości relacji noszą nazwę *zależności funkcyjnych*. Nazwa ta podkreśla istnienie pewnej funkcyjnej zależności między podzbiorami atrybutów w relacji. W omawianym przykładzie zależność taka istnieje w przypadku pierwszej właściwości między *Id_Stud* oraz *Nazwisko*, a w przypadku drugiej między *Id_Stud*, *Przedm* oraz *Ocena*. Pozostaje do wyjaśnienia kwestia, dlaczego mówimy o istnieniu zależności funkcyjnej, a nie po prostu o istnieniu funkcji. Otóż pojęcie *funkcja* oznacza istnienie stałego przyporządkowania między elementami zbioru. Tak może być np. między zbiorem identyfikatorów studentów i zbiorem nazwisk, gdzie możemy powiedzieć, że każdemu identyfikatorowi studenta przypisane jest na stałe pewne nazwisko, tzn. studenci mają przyporządkowane identyfikatory w ten sposób, że pozostają one niezmienione we wszystkich stanach bazy danych. Nie ma natomiast takiego stałego przyporządkowania między zbiorem par (*Id_Stud*, *Przedm*) a zbiorem *Ocen*, ponieważ w jednym stanie bazy danych pewnej parze (*Id_Stud*, *Przedm*) może być przypisana *Ocena: o*, a w innym, gdy np. student zdał egzamin poprawkowy, tej samej parze może być przypisana ocena o', $o \neq o'$.

Na koniec tych rozważań ostatecznie wnioskujemy, iż w relacji (tabeli) *Egzaminy* {*Id_Stud, Nazwisko, Przedm, Ocena*} istnieją następujące zależności funkcyjne:

Id_Stud → *Nazwisko* oraz *Id_Stud*, *Przedm* → *Ocena*.

W pierwszym przypadku bowiem równość dwóch różnych krotek dla atrybutu *Id_Stud* pociąga za sobą ich równość dla atrybutu *Nazwisko*.

Przykład 3-9

W relacji *Modele* {*Id_Modelu, Nazwa_Modelu, Rok_Produkcji, Id_Marki*} (tab. 3.7)

występuje m.in. następująca zależność funkcyjna:

Id_Modelu → *Id_Marki.*

Atrybut *Id_Marki* jest zależny funkcyjnie od atrybutu *Id_Modelu*, gdyż każdej wartości atrybutu *Id_Modelu* odpowiada dokładnie jedna wartość atrybutu *Id_Marki* (każdy model może należeć tylko do jednej marki). Ale zależność odwrotna nie występuje. Atrybut *Id_Modelu* nie jest zależny funkcyjnie od atrybutu *Id_Marki*, dla jednej marki bowiem może istnieć (i na ogół istnieje) wiele modeli samochodów.

W kontekście tej analizy warto wrócić do problemu rozkładalności (dekompozycji) relacji.

Twierdzenie Heatha

Niech ***R{A, B, C}*** będzie relacją, a ***A, B, C*** – zbiorami atrybutów tej relacji. Jeżeli ***R*** spełnia zależność funkcyjną

$A \rightarrow B$,

to ***R*** równa jest złączeniu projekcji **{*A, B*}** i **{*A,C*}**.

Twierdzenie to jest kolejnym potwierdzeniem rozważań i konkluzji zawartych w poprzednim podrozdziale.

Przykład 3-9

Niech *U* = {*Przedmiot*, *Indeks, Ocena*, *Egzamin*, *Godzina*, *Sala*}, gdzie znaczenie poszczególnych atrybutów jest następujące: *Przedmiot* – przedmiot, z którego jest zdawany egzamin, *Indeks* – numer indeksu studenta, *Ocena* – ocena uzyskana z egzaminu, *Egzamin* – numer ewidencyjny egzaminatora, *Godzina* – godzina egzaminu, *Sala* – sala, w której odbywa się egzamin. Zbiór *F* zależności funkcyjnych na tym zbiorze atrybutów może być następujący.

F = {*Przedmiot* → *Godzina, Sala; Godzina, Sala* → *Przedmiot; Przedmiot, Indeks* → *Ocena; Godzina, Indeks* → *Przedmiot, Sala; Przedmiot, Godzina, Sala* → *Egzamin*}.

Okazuje się jednak, że np. zamiast jednej zależności funkcyjnej *Przedmiot* → *Godzina, Sala* można zapisać dwie: *Przedmiot* → *Godzina* i *Przedmiot* → *Sala*, które razem są równoważne tej pierwszej.

Zauważmy ponadto, że jeśli *Przedmiot* → *Godzina, Sala* jest zależnością funkcyjną, to również zależnością funkcyjną jest np. *Przedmiot*, *Egzamin* → *Godzina, Sala*.

Prz. 3.9. ilustruje problem *generowania nowych zależności funkcyjnych*, gdy dany jest pewien ich zbiór ***F***. Powstaje pytanie: czy można rozpatrywać zależności funkcyjne, w tym wyprowadzanie jednych z drugich, bez odwoływania się do relacji, w których są one spełnione? Pozytywną odpowiedź na to pytanie udzielił Armstrong, który stworzył teorię tych zależności.

Definicja 3-5

Niech dany będzie zbiór atrybutów ***U*** i niech ***F*** będzie podzbiorem zbioru wszystkich zależności funkcyjnych o postaci $A \rightarrow B$, gdzie $A; B \subseteq U$, tzn. $F \subseteq \{ A \rightarrow B; A \subseteq U; B \subseteq U \}$.

Najmniejszy zbiór zależności funkcyjnych, który zawiera zbiór ***F*** i jest ***zamknięty*** ze względu na następujące reguły wyprowadzeń, nazywanych ***aksjomatami Armstronga,*** gdzie **A, B, C** są dowolnymi podzbiorami zbioru ***U***:

$B \subseteq A \Rightarrow A \rightarrow B \subseteq F^{+}$ (*zwrotność*);

$A \rightarrow B \in F^{+} \Rightarrow AC \rightarrow BC \in F^{+}$ (*rozszerzenie*);

$A \to B \in F^+ \wedge B \to C \in F^+ \Rightarrow A \to C \in F^+$ (*przechodniość*);

nazywamy ***najmniejszym domknięciem zbioru F*** lub ***pełną rodziną zależności funkcyjnych generowaną przez F*** i oznaczamy F^+.

Tutaj symbole $\in$, $\Rightarrow$, ^ oznaczają odpowiednio „należy do", „implikacja", „i".

Inaczej mówiąc, aksjomaty Armstronga można potraktować w następujący sposób:

- **zwrotność** – jeżeli ***B*** jest podzbiorem ***A***, to ***A→B***, a więc od atrybutu ***A*** zależny jest zawsze dowolny podzbiór ***A***;
- **rozszerzenie** – jeśli ***A→B***, to ***A, C→B, C***, a więc do obu stron zależności funkcyjnych można dołączyć dowolny podzbiór atrybutów relacji i zależność powstała w ten sposób będzie prawdziwa;
- **przechodniość** – jeśli ***A→B*** oraz ***B→C***, to ***A→C*** (można także nazwać ją *zależnością tranzytywną*).

Z wymienionych *aksjomatów Armstronga* można wyprowadzić jeszcze kilka reguł, znacznie ułatwiających wyznaczanie zbiorów zależności funkcyjnych, a mianowicie:

- **samookreślenie –** ***A→A***, a więc każdy zbiór atrybutów jest od siebie zależny funkcyjnie;
- **rozkład** – jeśli ***A→B, C***, to ***A→B*** i ***A→C***, a więc jeżeli dwa zbiory atrybutów są zależne funkcyjnie od zbioru atrybutów ***A***, to każdy z nich oddzielnie jest zależny funkcyjnie od tego zbioru;
- **suma** – jeśli ***A→B*** i ***A→C***, to również ***A→B***, ***C***, a więc jeśli dwa zbiory atrybutów oddzielnie są zależne funkcyjnie od zbioru atrybutów ***A***, to są również razem zależne od tego zbioru atrybutów;
- **złożenie** – jeśli ***A→B*** oraz ***C→D***, to również ***A, C→B, D***, gdzie ***D*** jest również zbiorem atrybutów relacji ***R***. Reguła ta mówi, iż możemy połączyć zbiór zależności rozłącznych w jedną poprawną zależność.

Definicja 3-6

Niech dany będzie zbiór ***F*** zależności funkcyjnych nad ***U***. Taki najmniejszy podzbiór F_0 zbioru ***F*** ($F_0 \subseteq F$), dla którego $F_0 = F^+$, nazywamy ***minimalnym generatorem zbioru*** F^+. Jeżeli ponadto wszystkie zależności $A \to B \in F^+$ są tego rodzaju, że ich lewe strony są możliwie najmniejsze, a prawe strony są pojedynczymi atrybutami, to F_0 nazywamy ***minimalnym zredukowanym generatorem zbioru*** F^+.

Przykład 3-10

Minimalnym zredukowanym generatorem pełnej rodziny zależności funkcyjnych F^+ z Prz. 3-9 mogą być zbiory:

$F_{10} = \{Przedmiot \to Sala;\ Godzina,\ Sala \to Przedmiot;\ Przedmiot,\ Indeks \to Ocena;\ Godzina,\ Indeks \to Przedmiot;\ Przedmiot \to Egzamin\}$

lub

$F_{20} = \{Przedmiot \to Godzina;\ Przedmiot \to Sala;\ Godzina,\ Sala \to Przedmiot;\ Przedmiot,\ Indeks \to Ocena;\ Godzina,\ Indeks \to Sala;\ Przedmiot \to Egzamin\}$.

Jak widać z powyższego przykładu ani minimalny zredukowany generator zbioru F^+, ani jego liczebność nie muszą być określone jednoznacznie. Problem wyznaczania minimal-

nego generatora zbioru F^+ należy do najważniejszych problemów w procesie projektowania schematu relacyjnej bazy danych.

Definicja 3-7

Niech dany będzie zbiór atrybutów ***U*** i niech ***F*** będzie zbiorem zależności funkcyjnych nad ***U***. Parę uporządkowaną ***R = (U; F)*** nazywamy ***schematem relacyjnym*** o zbiorze atrybutów ***U*** i ze zbiorem ***F*** zależności funkcyjnych nad ***U***.

Ostatnia definicja jest rozszerzeniem Def. 1.27.

Przykład 3-11

Niech dana będzie relacja *Egzaminy* = {*Indeks*, *Nazwisko_Stud*, *Przedmiot*, *Ocena*}.

Egzaminy = ({*Indeks, Nazwisko_Stud, Przedmiot, Ocena}, {Indeks* → *Nazwisko_Stud; Indeks, Przedmiot* → *Ocena*}).

3.2.3. Proces normalizacji BD

Jak już wspomniano wcześniej, proces normalizacji polega w zasadzie na transformacji relacji (tabel) z niższej PN w wyższą PN przez realizację operacji dekompozycji relacji na podstawie operatora projekcji.

Normalizacja relacji (tabeli) może być rozpatrywana względem zależności funkcyjnych pomiędzy jej atrybutami (polami). W związku z tym warto przypomnieć, że w praktycznym użytkowaniu relacyjnej bazy danych, a także w jej projektowaniu ważną rolę odgrywa wyodrębnienie kluczy w schematach relacyjnych, tzn. tych minimalnych podzbiorów zbioru atrybutów, które mają cechę jednoznacznego (unikalnego) identyfikowania krotek w każdej relacji (będącej przypadkiem danego schematu relacyjnego (patrz podrozdz. 2.1.1)). Wykorzystując pojęcie *zależności funkcyjnych*, można podać nową definicję *klucza*.

Definicja 3-8

Niech dany będzie schemat relacyjny ***R*** = ***(U, F)***. Zbiór atrybutów ***K*** $\subseteq$ ***U*** nazywamy ***kluczem*** schematu ***R*** wtedy i tylko wtedy, kiedy zbiór ten spełnia następujące warunki:

- wszystkie atrybuty należące do zbioru ***U*** muszą być funkcyjnie zależne od klucza;
- kluczem może być tylko taki zbiór atrybutów, którego żaden podzbiór właściwy nie ma cechy jednoznacznej identyfikowalności.

Przykład 3-12

Niech dany będzie schemat relacyjny z Prz. 3.11.

Cechę jednoznacznej identyfikowalności mają zbiory: {*Indeks, Przedmiot*}, {*Indeks, Nazwisko_Stud, Przedmiot*}, {*Indeks, Nazwisko_Stud, Przedmiot*, *Ocena*}. Najmniejszym z nich jest jednak pierwszy z wymienionych, i to on właśnie jest kluczem schematu.

Analiza zależności funkcyjnych między atrybutami kluczowymi a pozostałymi atrybutami umożliwia stwierdzanie posiadania przez schemat pewnych pożądanych właściwości. Okazuje się bowiem, że w schematach relacyjnych mogą występować pewne anomalie. Anomalie te (*anomalia dołączania*, *redundancja*, *anomalia aktualizacji*, *anomalia usuwania*) mogą być usuwane przez rozkładanie schematów relacyjnych na schematy

bardziej elementarne. Proces ten również często nazywany jest *procesem normalizacji* i został zaproponowany przez E. Codda.

3.2.3.1. Pierwsza postać normalna

Dowolna poprawnie zaprojektowana relacja (tabela) znajduje się przynajmniej w 1PN. Poprawność ta oznacza, że wszystkie atrybuty (pola) spełniają następujące warunki:

- *nie ma wśród nich atrybutów wielowartościowych (składających się z dwóch i więcej wartości);*
- *nie są one atrybutami wyliczeniowymi (wartość atrybutu w każdej krotce (rekordzie) jest wynikiem operacji nad wartościami w innych atrybutach tej krotki);*
- *każda wartość tego samego atrybutu ma jednakowy typ (danych).*

Przykład 3-13
Spójrzmy jeszcze raz na relację (tabelę) *Wypożyczenia* z Prz. 3.4.

Tabela ta przedstawia filmy wypożyczone przez klientów. Jednak wyraźnie widać, iż nie spełnia ona pierwszej postaci normalnej, która mówi, iż tabela musi mieć na przecięciu kolumn i wierszy wyrażenia atomowe (jednowartościowe), a więc nie może posiadać pól o wielu wartościach (pól zwielokrotnionych). W tabeli *Wypożyczenia* takie pola istnieją, np. w pierwszym wierszu pola *Id_Kasety*, *Data_Wypoż*, *Okres*, *Tytuł*, *Reżyser*. Aby tabelę tę doprowadzić do 1PN, zastosujemy technikę zwaną „spłaszczaniem" tabeli, a więc podzielimy rekordy z wielokrotnymi polami na rekordy, które już takich pól nie będą zawierać.

Nowa relacja *Wypożyczenia* nie ma już pól wielowartościowych, co oznacza, że na przecięciu się kolumn i wierszy są jedynie wartości pojedyncze (atomowe).

3.2.3.2. Druga postać normalna

Relacja (tabela) jest w drugiej postaci normalnej(2PN), jeżeli jest ona w pierwszej postaci normalnej i dodatkowo nie występują w niej częściowe zależności funkcyjne między kluczem głównym a atrybutami (polami) niekluczowymi.

Inaczej mówiąc, relacja jest w drugiej postaci normalnej, jeżeli wszystkie zależności funkcyjne między kluczem głównym i atrybutami niekluczowymi są zależnościami generowanymi przez cały klucz.

Tabela *Wypożyczenia*

Id_Kasety	***Id_Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*	*Tytuł*	*Reżyser*	*Imię_Prac*	*Nazwisko_Prac*
1 3	5	05.02.2006 05.04.2005	15	3 2	„Matrix" „Obcy"	Wachowscy Scott	Roman	Wójcik
1	8	12.03.2006	3	5	„Matrix"	Wachowscy	Maria	Nowak
8	2	14.04.2006	2	1	„Predator"	McTiernan	Krzysztof	Kowalski
12 13	4	15.03.2006 16.03.2005	15 18	1	„The ring" „Taxi"	Nakata Pires	Roman Hanna	Wójcik Malinowska
4	14	18.04.2006	3	2	„Kiler"	Machulski	Maria	Nowak

Technika „spłaszczania" pozwoli nam uzyskać następującą relację (tab. 3.11).

Tabela 3.11. Tabela *Wypożyczenia* doprowadzona do 1PN

Id_Kasety	***Id_Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*	*Tytuł*	*Reżyser*	*Imię_Prac*	*Nazwisko_Prac*
1	5	05.02.2006	15	3	„Matrix"	Wachowscy	Roman	Wójcik
3	5	05.04.2005	15	2	„Obcy"	Scott	Roman	Wójcik
1	8	12.03.2006	3	5	„Matrix"	Wachowscy	Maria	Nowak
8	2	14.04.2006	2	1	„Predator"	McTiernan	Krzysztof	Kowalski
12	4	15.03.2006	15	1	„The ring"	Nakata	Roman	Wójcik
13	4	16.03.2005	18	1	„Taxi"	Pires	Hanna	Malinowska
4	14	18.04.2006	3	2	„Kiler"	Machulski	Maria	Nowak

Definicja 3-9

Niech ***A*** i ***B*** będą dowolnymi zbiorami atrybutów relacji ***R***. Zbiór atrybutów ***B*** jest ***w pełni funkcyjnie zależny*** od zbioru atrybutów ***A***, jeżeli ***A→B*** oraz nie istnieje podzbiór właściwy atrybutów ***A'*** zbioru ***A***, taki że ***A'→B***, czyli ***B*** nie jest funkcyjnie zależny od żadnego właściwego podzbioru zbioru ***A***.

Stąd wynika, iż po usunięciu dowolnego atrybutu z *A* zależność *A→B* nie będzie już zachodzić.

Definicja 3-10

Zbiór atrybutów ***B*** jest ***częściowo funkcyjnie zależny*** od zbioru atrybutów ***A*** relacji ***R***, jeżeli istnieje taki właściwy podzbiór ***A'*** zbioru ***A***, że zachodzi zależność ***A'→B***.

Jest to równoważne temu, że istnieje taki atrybut w zbiorze ***A***, który możemy usunąć z tego zbioru i nadal będzie zachodzić zależność funkcyjna ***A→B***.

Najpierw na przykładzie pokażemy, jakie anomalie ma schemat relacyjny, który nie jest w 2PN, a następnie przedstawimy sposób usunięcia tych anomalii.

Przykład 3-14

Niech dany będzie schemat relacyjny (lub po prostu relacja):

Egzaminy = ({***Indeks***, *Nazwisko_Stud, Kierunek_Studiów, Adres_Stud*, ***Przedmiot***, *Ocena*}, {*Indeks → Nazwisko_Stud, Kierunek_Studiów, Adres_Stud; Indeks, Przedmiot → Ocena*}).

Relacja (tabela) ta posiada klucz złożony: ***Indeks, Przedmiot*** (atrybuty należące do tego klucza zostały pogrubione). Zauważmy, że ma ona następujące anomalie:

- **Anomalia dołączania**. Zakładamy, że relacja (tabela) posiada informacje tylko o tych studentach, którzy zdali już co najmniej jeden egzamin. W takim razie inni studenci nie mogą być opisani w tej relacji, gdyż wówczas klucz *Indeks, Przedmiot* nie byłby pełny. Trudność tę można by przezwyciężyć, dopuszczając możliwość stosowania pustych wartości dla atrybutu *Przedmiot* (wartości NULL). W przypadku atrybutów klucza głównego jest to jednak niemożliwe, gdyż żaden z atrybutów klucza głównego nie może przyjmować wartości NULL.

- **Redundancja**. Informacje o studencie, takie jak jego nazwisko i adres, przechowywane są wielokrotnie. A zatem niepotrzebnie zajmowana jest pamięć oraz dodatkowo może być to źródłem problemów związanych z aktualizacją. Trudno też wymagać, aby przy wprowadzaniu informacji o zdanym egzaminie przez studenta wprowadzać każdorazowo jego nazwisko i adres.
- **Anomalia aktualizacji**. Przypuśćmy, że jakiś student w trakcie sesji zmienił adres zamieszkania. Ponieważ zdał on już jakieś egzaminy, więc jego adres musi być aktualizowany w różnych krotkach relacji. W ogólnym przypadku aktualizacja tego rodzaju wymagać będzie przeglądania dużej, często zmiennej w czasie, liczby krotek w relacji. Może to spowodować, że przed zakończeniem procesu aktualizacji baza danych będzie dostarczała sprzecznych informacji.
- **Anomalia usuwania**. Przypuśćmy, że jakiś student zdał do tej pory tylko jeden egzamin i że egzamin ten z jakiegoś powodu został unieważniony. Usunięcie z relacji krotki o wartości klucza spowoduje jednak jednoczesną utratę informacji o studencie.

Powyższe anomalie wynikają stąd, że niektóre atrybuty (*Ocena*) są funkcyjnie zależne od całego klucza, inne natomiast (*Nazwisko_Stud, Kierunek_Studiów, Adres_Stud*) zależne są tylko od części tego klucza (*Indeks*). Mówimy wówczas, że istnieje w schemacie relacyjnym *niepełna zależność funkcyjna atrybutów od klucza*. W definicji 2PN wymagamy, aby nie było w schemacie tego rodzaju niepełnych zależności funkcyjnych.

Przykład 3-15

Rozpatrzmy relację (tabelę) *Wypożyczenia* (tab. 3.11) w postaci, w której spełnia ona 1PN.

Aby ta relacja spełniała 2PN, nie mogą w niej występować *częściowe zależności funkcyjne* atrybutów niekluczowych od klucza podstawowego (przypomnimy sobie, że relacja posiada ZKP: *Id_Kasety, Id_Klienta, Data_Wypoż)*. Okazuje się jednak, że w naszej relacji występują tego typu zależności. Mianowicie, mamy do czynienia z następującymi zależnościami funkcyjnymi:

Id_Kasety → Tytuł oraz *Id_Kasety → Reżyser.*

Niewątpliwie atrybuty *Tytuł* oraz *Reżyser* są atrybutami niekluczowymi. Atrybut *Id_Kasety* wchodzi w skład KP, a więc jest on jego podzbiorem właściwym. Mamy więc tu do czynienia z zależnościami funkcyjnymi atrybutów niekluczowych od podzbioru właściwego klucza podstawowego. Stąd atrybuty niekluczowe są *częściowo zależne funkcyjnie od klucza podstawowego*. Dlatego też nasza relacja nie znajduje się w drugiej postaci normalnej.

Formalnie rozszerzona definicja 2PN może być sformułowana następująco:

Definicja 3-11

Schemat relacyjny ***R = (U, F)*** jest w **2PN**, jeśli każdy niekluczowy atrybut $A \in U$ jest w ***pełni funkcyjnie zależny od każdego klucza*** tego schematu.

ALGORYTM sprowadzania relacji (w sensie ogólnym – schematu relacyjnego) z 1PN do 2PN:

Wejście: Ω – schemat relacyjny *R = (U, F).*

Wyjście: Γ – zbiór schematów relacyjnych w 2PN.

Kroki:

1) na początek niech jedynym elementem zbioru Ω będzie schemat relacyjny $R = (U, F)$, a zbiór Γ niech będzie zbiorem pustym;
2) czy Ω jest zbiorem pustym?
   - tak, koniec algorytmu, w zbiorze Γ znajdują się schematy relacyjne w 2PN;
   - nie, przejdź do kroku 3;
3) weź dowolny schemat relacyjny ze zbioru Ω i oznacz go przez $R = (U, F)$. Czy istnieje taki klucz $K \subseteq U$ schematu R, że dla pewnego $K' \subset K, K' \to X \in F^+$ (X dowolny podzbiór U)?
   - tak, przejdź do kroku 4;
   - nie, schemat R dołącz do zbioru Γ i usuń go ze zbioru Ω, przejdź do kroku 2;
4) rozłóż schemat R na dwie projekcje: $R1 = R[K', X]$ i $R2 = R[K',(U - X)]$. Ze zbioru Ω usuń schemat R i dołącz do niego schematy $R1$ i $R2$, przejdź do kroku 2.

Przy przeprowadzaniu schematu relacyjnego do 2PN należy brać pod uwagę następujące fakty:

- schemat jest już w 2PN, jeśli każdy jego klucz jest jednoelementowy;
- przeprowadzanie schematu relacyjnego do 2PN nie jest procesem jednoznacznym, tzn. dla jednego schematu relacyjnego może istnieć wiele równoważnych informacyjnie układów projekcji tego schematu w 2PN.

Spośród zbioru wszystkich układów schematów relacyjnych uzyskanych w wyniku procesu normalizacji pewnego schematu relacyjnego wybieramy ten, który zawiera najmniej elementów (nie musi on być określony jednoznacznie). Nazwiemy go *układem minimalnym*. Układ minimalny jest więc pewnego rodzaju układem optymalnym, ale optimum to jest lokalne, a nie globalne. Ponadto kryterium tej optymalności jest na tyle usprawiedliwione, jak dalece prawdą jest, że wraz ze wzrostem liczby relacji na poziomie logicznym bazy danych wzrasta złożoność procesów manipulowania danymi.

Praktyczne wyniki zastosowania tego algorytmu wyglądają tak jak w Prz. 3.16 i 3.17.

Przykład 3-16

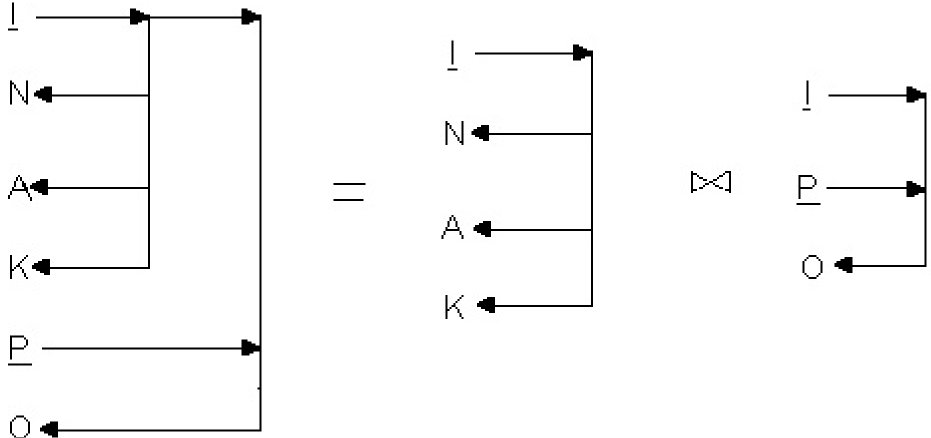
Schemat relacyjny (relacja) *Egzaminy* z Prz. 3.14 nie jest w 2PN. Schemat ten można rozłożyć bez straty danych na dwa takie schematy (*Egzaminy_1* i *Egzaminy_2)*, z których każdy będzie w 2PN.

Egzaminy_1 = ({***Indeks***, *Nazwisko_Stud*, *Kierunek_Studiów*, *Adres_Stud*},{*Indeks* → *Nazwisko_Stud*, *Kierunek_Studiów*, *Adres_Stud*}),

Egzaminy_2 = ({***Indeks***, ***Przedmiot***, *Ocena*},{*Indeks*, *Przedmiot* → *Ocena*}).

Relacja *Egzaminy_1* posiada jednoatrybutowy KP (*Indeks*), relacja *Egzaminy_2* – ZKP (*Indeks, Przedmiot*).

Schematy te (poprzez operację złączenia) przedstawimy następująco (rys. 3.3):



Rys. 3.3. Graficzne przedstawienie wyniku bezstratnego rozkładu schematu relacyjnego *Egzaminy* do 2PN

Na rys. 3.3 oznaczono: I – *Indeks*, N – *Nazwisko_Stud*, A – *Adres_Stud*, K – *Kierunek_Studiów*, P – *Przedmiot*, O – *Ocena*.

Przykład 3-17
Rozpatrzmy relację (tabelę) *Wypożyczenia* w postaci, w której znajduje się ona w 1PN (tab. 3.11).

Aby relacja spełniała wymagania 2PN, nie mogą w niej występować *częściowe zależności funkcyjne atrybutów niekluczowych od klucza podstawowego*, jednak w naszej relacji zależności takie występują. Zauważmy, iż mamy w niej do czynienia z następującymi zależnościami funkcyjnymi:

Id_Kasety → Tytuł oraz *Id_Kasety → Reżyser.*

Niewątpliwie atrybuty *Tytuł* oraz *Reżyser* są atrybutami niekluczowymi. Atrybut *Id_Kasety* wchodzi w skład klucza podstawowego, a więc jest on jego podzbiorem właściwym. Mamy więc tu do czynienia z zależnościami funkcyjnymi atrybutów niekluczowych od podzbioru właściwego klucza podstawowego, stąd atrybuty niekluczowe są częściowo zależne funkcyjnie od klucza podstawowego. Dlatego też relacja *Wypożyczenia* nie spełnia drugiej postaci normalnej.

Aby relację doprowadzić do 2PN, należy atrybuty niekluczowe częściowo zależne funkcyjnie od klucza podstawowego, a więc *Tytuł* i *Reżyser*, przenieść do drugiej relacji (niech nazwą tej relacji będzie nazwa *Kasety*) wraz z wyznacznikiem tych częściowych zależności, a więc atrybutem *Id_Kasety*, który stanie się kluczem podstawowym nowej relacji. Po przeprowadzeniu tej procedury otrzymamy następujące dwie tabele.

Tabela 3.12. Tabela *Wypożyczenia* doprowadzona do 2PN

Id_ Kasety	***Id_ Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*	*Imię_Prac*	*Nazwisko_ Prac*
3	5	05.04.2005	15	2	Roman	Wójcik
3	5	05.04.2005	15	2	Roman	Wójcik
1	8	12.03.2006	3	5	Maria	Nowak
8	2	14.04.2006	2	1	Krzysztof	Kowalski
12	4	15.03.2006	15	1	Roman	Wójcik
13	4	16.03.2005	18	1	Hanna	Malinowska
4	14	18.04.2006	3	2	Maria	Nowak

Tabela 3.13. Tabela *Kasety* w 2PN

Id_Kasety	*Tytuł*	*Reżyser*
1	„Matrix”	Wachowscy
3	„Obcy”	Scott
4	„Kiler”	Machulski
8	„Predator”	McTiernan
12	„The ring”	Nakata
13	„Taxi”	Pires

Powstałe relacje (tabele) są już w 2PN. Pozbyliśmy się również *anomalii aktualizacji*, występujących w poprzedniej relacji *Wypożyczenia* przed doprowadzeniem jej do 2PN. Wcześniej zmiana tytułu danego filmu powodowała, iż musieliśmy tej zmiany dokonywać w każdym rekordzie, w którym kaseta ulegająca zmianie występowała, a więc zmiany trzeba było dokonać dla każdego wypożyczenia tej kasety. Po normalizacji wystarczy dokonać zmiany tylko w jednym rekordzie tabeli *Kasety*.

Wcześniej usunięcie wszystkich rekordów reprezentujących wypożyczenie danej kasety powodowało utratę wszystkich informacji o niej, teraz informacje o kasetach przechowywane są w tabeli *Kasety* całkowicie niezależnie od tabeli *Wypożyczenia*. Wraz z wyeliminowaniem *anomalii aktualizacji* pozbyliśmy się *nadmiarowości danych*, która polegała na konieczności powtarzania tytułu i reżysera danego filmu za każdym razem, gdy ten film wypożyczał klient. Jednak nowa tabela *Wypożyczenia* nie jest całkowicie pozbawiona anomalii aktualizacji, stąd konieczna jest dalsza normalizacja.

Tabela znajdująca się w pierwszej postaci normalnej i posiadająca *prosty klucz podstawowy*, a więc składający się z jednego pola, *będzie zawsze w drugiej postaci normalnej*, nie możemy bowiem z klucza wybrać podzbioru właściwego, od którego byłyby zależne funkcyjnie atrybuty niekluczowe. Stąd problem spełniania przez tabele wymagań drugiej postaci normalnej odnosi się tylko do tych tabel, które mają złożony klucz podstawowy.

3.2.3.3 Trzecia postać normalna

Relacja (tabela) występuje w trzeciej postaci normalnej (3PN), jeżeli jest ona w drugiej postaci normalnej i dodatkowo nie ma w niej zależności tranzytywnych (przechodnich) atrybutów (pól) niekluczowych od klucza podstawowego relacji (tabeli).

Posiadanie przez schemat relacyjny właściwości 2PN nie oznacza automatycznie, że w relacjach będących jego przypadkami nie występują te anomalie (dołączania, redundancji, aktualizacji i usuwania), które występowały, gdy schemat relacyjny nie miał tej właściwości. Pokażemy to na przykładzie.

Definicja 3-12

Jeżeli ***A***, ***B*** i ***C*** są atrybutami relacji ***R*** oraz istnieją zależności ***A→B*** oraz ***B→C***, to mówimy, że atrybut ***C*** jest ***tranzytywnie (przechodnio)*** zależny od atrybutu ***A***, pod warunkiem że ***A*** nie jest zależny ani od atrybutu ***B***, ani ***C***.

Pokażemy na przykładach, jakie anomalie ma schemat relacyjny, który nie jest w 3PN.

Przykład 3-18

Niech dany będzie schemat relacyjny:

Wykonawcy_Projektu = ({***Wykonawcy***, *Adres_Wykonawcy, Projekt, Data_Ukończenia*}, {***Wykonawcy*** → *Adres_Wykonawcy, Projekt, Data_Ukończenia; Projekt* → *Data_Ukończenia*}).

Podany zbiór zależności funkcyjnych oddaje następujące właściwości semantyczne odwzorowywanej rzeczywistości:

- każdy wykonawca ma jednoznacznie określony adres;
- każdy wykonawca może realizować tylko jeden projekt, natomiast jeden projekt może być realizowany przez wielu wykonawców;
- dla wykonawcy określony jest jednoznaczny termin ukończenia projektu;
- termin ukończenia konkretnego projektu jest taki sam dla wszystkich wykonawców biorących udział w jego realizacji.

Schemat relacyjny jest w 2PN, ponieważ jedyny klucz (*Wykonawcy*) tego schematu jest jednoelementowy. Relacje będące jego przypadkami charakteryzują się jednak podobnymi anomaliami jak relacje niebędące w 2PN, a mianowicie:

- **Anomalia dołączania**. Nie można zapamiętać informacji o projekcie i dacie jego zakończenia, dopóki nie zostanie określony co najmniej jeden wykonawca tego projektu.
- **Redundancja**. Dane o projekcie powtarzane są wielokrotnie, bo jeden projekt może być realizowany przez wielu wykonawców.
- **Anomalia aktualizacji**. Zmiana daty ukończenia któregoś z projektów wymaga dokonywania zmian w dużej liczbie krotek relacji; może to spowodować udzielanie sprzecznych informacji z bazy danych.
- **Anomalia usuwania**. Zaniechanie realizowania któregoś z projektów, tj. usunięcie odpowiadającej mu krotki z relacji, spowoduje utratę informacji o wykonawcach. Podobnie gdy jakiś projekt realizowany jest tylko przez jednego wykonawcę i wykonawca ten wycofuje się z tej realizacji, następuje utrata informacji o projekcie.

Wymienione anomalie wynikają stąd, że niektóre atrybuty (*Adres_Wykonawcy* i *Projekt*) są funkcyjnie zależne tylko od klucza, natomiast inne (*Data_Ukończenia*) również od atrybutu niekluczowego (*Projekt*).

Przykład 3-19

Rozpatrzmy relację (tabelę) *Wypożyczenia* wcześniej doprowadzoną do 2PN (tab. 3.12). Jej kluczem jest klucz złożony, składający się z atrybutów (pól) *Id_Kasety, Id_Klienta, Data_Wypoż.* Ponadto występują w tej relacji m.in. następujące zależności funkcyjne:

Id_Kasety, Id_Klienta, Data_Wypoż → *Id_Prac*,

bo każde wypożyczenie jednoznacznie określa nam pracownika, który dokonał tego wypożyczenia:

Id_Prac → *Imię_Prac*;

Id_Prac → *Nazwisko_Prac*.

Zauważmy ponadto, iż nasz KP nie jest zależny funkcyjnie ani od atrybutu *Id_Prac* (bowiem jeden pracownik na ogół dokonuje wielu wypożyczeń), ani od żadnego z atrybutów *Imię_Prac, Nazwisko_Prac* (z tego samego powodu). Stąd atrybuty niekluczowe *Imię_Prac* oraz *Nazwisko_Prac* są tranzytywnie zależne od klucza podstawowego relacji *Wypożyczenia*:

Id_Kasety, Id_Klienta, Data_Wypoż→Id_Prac→ Imię_Prac;

Id_Kasety, Id_Klienta, Data_Wypoż→Id_Prac→Nazwisko_Prac.

Z tego wnioskujemy, iż relacja *Wypożyczenia* nie jest w trzeciej postaci normalnej.

Tak jak w definicji 2PN korzysta się z pojęcia pełnej zależności funkcyjnej, tak w przypadku 3PN wykorzystuje się pojęcie *zależności tranzytywnej (przechodniej).*

Formalnie definicja 3PN może być sformułowana następująco.

Definicja 3-13

Schemat relacyjny ***R = (U, F)*** jest w **3PN**, jeżeli jest on w 2PN i dodatkowo żaden atrybut spoza klucza podstawowego tej relacji nie jest ***tranzytywnie zależny*** od tego klucza.

ALGORYTM sprowadzenia schematu relacyjnego z 2PN do 3PN:

Wejście: $\boldsymbol{\Omega}$ – schemat relacyjny ***R = (U, F)***.

Wyjście: $\boldsymbol{\Gamma}$ – zbiór schematów relacyjnych w 3PN.

Kroki:

1) na początek niech jedynym elementem zbioru $\boldsymbol{\Omega}$ będzie schemat relacyjny ***R* = (*U*, *F*),** a zbiór $\boldsymbol{\Gamma}$ niech będzie zbiorem pustym;
2) czy $\boldsymbol{\Omega}$ jest zbiorem pustym?
   - tak, koniec algorytmu, w zbiorze $\boldsymbol{\Gamma}$ znajdują się schematy relacyjne w 3PN,
   - nie, przejdź do kroku 3;
3) weź dowolny schemat relacyjny ze zbioru $\boldsymbol{\Omega}$ i oznacz go przez ***R* = (*U*, *F*)**. Czy istnieją rozłączne zbiory atrybutów $\boldsymbol{X, Y \subset U}$, gdzie ***X*** nie jest kluczem, a ***Y*** jest zbiorem atrybutów niekluczowych, dla których $\boldsymbol{X \rightarrow Y \in F^{+}}$?
   - tak, przejdź do kroku 4,
   - nie, schemat ***R*** dołącz do zbioru $\boldsymbol{\Gamma}$ i usuń go ze zbioru $\boldsymbol{\Omega}$, przejdź do kroku 2;
4) rozłóż schemat ***R*** na dwie projekcje: ***R1 = R[X, Y]*** i ***R2 = R[X, (U – Y)]***. Ze zbioru $\boldsymbol{\Omega}$ usuń schemat ***R*** i dołącz do niego schematy ***R1*** i ***R2***, przejdź do kroku 2.

Teraz przeanalizujemy praktyczne zastosowanie tego algorytmu.

Przykład 3-20

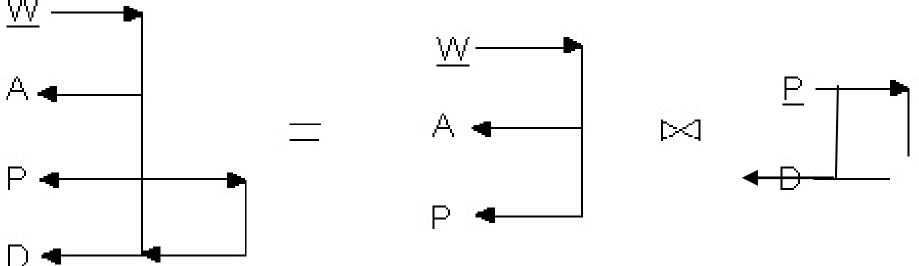
Schemat relacyjny z Prz. 3.18 nie jest w 3PN. Można go jednak rozłożyć na takie dwa schematy, z których każdy będzie w 3PN:

Wykonawcy_Projektu_1 = ({***Wykonawcy****, Adres_Wykonawcy, Projekt*}, {***Wykonawcy*** → *Adres_Wykonawcy, Projekt*}),

oraz

Wykonawcy_Projektu_2 = ({***Projekt****, Data_Ukończenia; Projekt* → *Data_Ukończenia*}).

Schematy te (poprzez operację złączenia) przedstawimy następująco (rys. 3.4).



Rys. 3.4. Graficzne przedstawienie wyniku rozkładu schematu relacyjnego *Wykonawcy_ Projektu* do 3PN

Na rys. 3.4. użyte oznaczenia (literowe) są pierwszymi literami nazw atrybutów.

Przykład 3-21

Rozpatrzmy relację (tabelę) *Wypożyczenia* w postaci, w której jest ona w 2PN (tab. 3.12). Aby doprowadzić ją do 3PN, należy atrybuty niekluczowe tranzytywnie zależne od KP (a więc *Imię_Prac* oraz *Nazwisko_Prac*) przenieść do drugiej relacji (*Pracownicy*) wraz z wyznacznikiem pośrednim zależności tranzytywnej (a więc atrybutem występującym w środku schematu zależności tranzytywnej) *Id_Prac*, który to atrybut będzie kluczem podstawowym nowej relacji.

Stosując tę procedurę, otrzymamy następujące relację (tab. 3.14 i tab. 3.15).

Tabela 3.14. Tabela *Wypożyczenia* doprowadzona do 3PN

Id_Kasety	***Id_Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*
3	5	05.04.2005	15	2
3	5	05.04.2005	15	2
1	8	12.03.2006	3	5
8	2	14.04.2006	2	1
12	4	15.03.2006	15	1
13	4	16.03.2005	18	1
4	14	18.04.2006	3	2

Tabela 3.15. Tabela *Pracownicy* w 3PN

Id_Prac	*Imię_Prac*	*Nazwisko_Prac*
15	Roman	Wójcik
3	Maria	Nowak
2	Krzysztof	Kowalski
18	Hanna	Malinowska

Powyższe relacje (tabele) są już w 3PN, a więc nie występują w nich tranzytywne zależności funkcyjne atrybutów niekluczowych od klucza głównego.

W tabeli *Wypożyczenia* kluczem podstawowym pozostała kombinacja *Id_Kasety, Id_Klienta* i *Data_Wypoż*, natomiast pole *Id_Prac* jest kluczem obcym (atrybutem ogólnym), odpowiadającym KP relacji *Pracownicy* o tej samej nazwie. Zauważmy, iż doprowadzając tabelę *Wypożyczenia* do 3PN, pozbyliśmy się kolejnych anomalii aktualizacji, a przy tym nadmiarowych danych. Wcześniej, gdy chcieliśmy zmienić imię bądź nazwisko danego pracownika, musieliśmy zmienić wszystkie krotki reprezentujące

wypożyczenia obsłużone przez tego pracownika. Problem był również z wprowadzaniem do bazy pracownika – by było to możliwe, konieczne było dodanie choć jednego rekordu z wypożyczeniem realizowanym przez tego pracownika. Usunięcie natomiast z relacji ostatniego wypożyczenia danego pracownika powodowało całkowite usunięcie danych o tym pracowniku. Dzięki doprowadzeniu relacji do 3PN anomalie te są usunięte – dane o pracownikach są przetrzymywane w oddzielnej relacji, w której można modyfikować, usuwać i dodawać dane o pracownikach, całkowicie niezależnie od relacji *Wypożyczenia*.

3.2.3.4. Postać normalna Boyce'a-Codda

Proces normalizacji najczęściej przeprowadza się do trzeciej postaci normalnej, gdyż w większości przypadków relacje (tabele) w tej postaci są pozbawione nadmiarowych danych. Jednak występują przypadki, gdy konieczne staje się dalsze usuwanie nadmiarowości (choć są one bardzo rzadkie). Prześledźmy hipotetyczną sytuację, która teoretycznie mogłaby się zdarzyć przy projektowaniu bazy.

Postać normalna Boyce'a-Codda (PNBC) jest postacią silniejszą niż 3PN. Przy doprowadzaniu schematu relacyjnego (relacji) do drugiej, a później do trzeciej postaci normalnej badaliśmy, czy w relacji nie ma żadnych częściowych ani przechodnich zależności atrybutów niekluczowych od KP relacji. Jednak nie były badane żadne zależności pomiędzy kluczami, atrybutami kluczowymi, a także nie wykluczaliśmy istnienia zależności, których wyznacznikami byłyby atrybuty niekluczowe. Takie kwestie są rozpatrywane przy doprowadzaniu relacji do postaci normalnej Boyce'a-Codda.

Definicja 3-14

Schemat relacyjny ***R = (U, F)*** jest w ***PNBC***, jeżeli jest on w 3PN i dodatkowo wszystkie wyznaczniki zależności funkcyjnych w tej relacji są kluczami kandydującymi.

Ta postać normalna wymusza, aby w relacji nie istniały ani zależności pomiędzy atrybutami wchodzącymi w skład klucza, ani zależności, których wyznacznikami byłyby atrybuty niekluczowe. Wszystkie zależności występujące w relacji muszą być determinowane przez klucze kandydujące, a więc atrybuty lub zbiory atrybutów, które w sposób jednoznaczny identyfikują wszystkie krotki relacji.

Przykład 3-22

Dany schemat relacyjny *Wykłady_Studentów:*

Wykłady_Studentów = ({***Id_Studenta***, ***Przedmiot***, *Wykładowca*}, {*Id_Studenta, Przedmiot* $\rightarrow$ *Wykładowca; Wykładowca* $\rightarrow$ *Przedmiot*}).

Zależności funkcyjne zawarte w schemacie reprezentują następujące ograniczenia semantyczne odwzorowanego fragmentu rzeczywistości:

- student może uczęszczać na określony przedmiot tylko do jednego wykładowcy;
- każdy wykładowca prowadzi dokładnie jeden przedmiot.

Istnieje w tej relacji zależność funkcyjna atrybutu kluczowego (*Przedmiot)* od atrybutu niekluczowego (*Wykładowca*). Jest to źródłem pewnych anomalii schematu. Nie można przypisać wykładowcy do konkretnego przedmiotu, zanim nie przypisze się do tego przedmiotu co najmniej jednego studenta (*anomalia dołączania*). Wystąpi również *anomalia usuwania*, gdy usunięte mają być informacje o ostatnim studencie uczęszczającym na dany przedmiot.

Omawiane wyżej anomalie wynikają stąd, że schemat nie jest w postaci normalnej Boyce'a-Codda. Łatwo jest zdekomponować schemat relacyjny będący w 3PN do zbioru schematów w PNBC.

Dla rozważanego przykładu schematu relacyjnego dekompozycja ta doprowadza do układu złożonego z dwóch schematów relacyjnych (*Wykłady* i *Studenci_Wykładowcy*), z których każdy (z punktu widzenia anomalii) znajduje się w PNBC (rys. 3.5, gdzie symbole mają następujące oznaczenia: I – *Id_Studenta*, P – *Przedmiot*, W – *Wykładowca*):

Studenci_Wykładowcy = ({***Id_Studenta***, ***Wykładowca***}, {Ø})

i

Wykłady = ({***Wykładowca***, *Przedmiot*}, {*Wykładowca* → *Przedmiot*}).

I P W = I W ⋈ P W

Rys. 3.5. Graficzne przedstawienie wyniku transformacji schematu relacyjnego *Wykłady_Studentów* do PNBC

Tę nową postać normalną schematów relacyjnych możemy zdefiniować następująco.

Definicja 3-15

Schemat relacyjny $\boldsymbol{R = (U; F)}$ jest w ***PNB–C*** wtedy i tylko wtedy, kiedy z istnienia zależności funkcyjnej $\mathbf{A \rightarrow B \in F^+}$ dla $\boldsymbol{B \subseteq U - A}$ wynika, że $\boldsymbol{A \rightarrow U \in F^+}$.

Definicja 3-15 mówi, że w schematach będących w PNBC, jeśli od zbioru atrybutów *B* zależny jest funkcyjnie jakikolwiek atrybut rozłączny z *A*, to od *A* zależne są funkcyjnie wszystkie inne atrybuty w tym schemacie.

Innymi słowy, wszystkie zależności funkcyjne w schemacie relacyjnym w PNBC determinowane są przez klucze.

Przy rozkładzie pokazanym na rys. 3.5 powstaje jednak istotna trudność: nie można dołączyć krotki do relacji *Studenci_Wykładowcy* bez jednoczesnej aktualizacji relacji *Wykłady*. Powodem omawianej trudności jest to, iż mimo *że proponowany rozkład zachowuje dane, jednak nie zachowuje zależności*. Jak widać, sprowadzanie schematu relacyjnego do PNBC nie zawsze jest pożądane. W rozważanym przez nas przykładzie najlepszym rozwiązaniem byłoby jednoczesne istnienie wyjściowego schematu relacyjnego *Wykłady_Studentów* oraz schematu *Wykłady*. Dzięki temu możliwe byłoby przezwyciężenie *anomalii dołączania* i *usuwania*, a także uniknięcie omawianych powyżej trudności.

Przykład 3-23

Załóżmy, iż w naszej wypożyczalni filmów (patrz Prz. 3.3 i rys. 3.2) możemy złożyć kilka zamówień dziennie, jednak z takim zastrzeżeniem, iż dana kaseta może być zamówiona tylko raz dziennie – czyli występować tylko na jednym zamówieniu. Każde zamówienie może zawierać kilka pozycji, a więc może być na nim kilka różnych kaset. Taką sytuację reprezentuje relacja (tabela) *Zamówione_Kasety* (tab. 3.15).

Tabela 3.15. Relacja *Zamówione_Kasety*

Id_Kasety	***Data***	*Id_Zamówienia*
1	2006.04.08	12
2	2006.05.23	13
1	2006.05.23	13
3	2005.06.01	5

Kluczem podstawowym relacji (tabeli) jest klucz złożony, składający się z atrybutów (pól) *Id_Kasety* i *Data* – ta para jednoznacznie identyfikuje *Id_Zamówienia*, gdyż dana kaseta w danym dniu może występować tylko na jednym zamówieniu. Występuje więc zależność funkcyjna:

Id_Kasety, Data → *Id_Zamówienia.*

Oprócz KP wymienionego powyżej (który jest oczywiście z definicji również kluczem kandydującym, KK) występuje KK złożony z atrybutów *Id_Kasety* i *Id_Zamówienia* – ta para jednoznacznie identyfikuje wartość atrybutu (pola) *Data*. Jednak zauważmy jeszcze, iż występuje także zależność funkcyjna *Id_Zamówienia* → *Data* – dane zamówienie jest składane tylko w określonym dniu.

Mamy więc do czynienia z zależnością funkcyjną, której wyznacznikiem nie jest KK relacji, gdyż pojedynczy atrybut *Id_Zamówienia* nie określa chociażby *Id_Kasety* (na jednym zamówieniu może być wiele kaset). Stąd możemy wywnioskować, iż nasza relacja nie spełnia PNBC, która mówi, iż jedynymi wyznacznikami zależności funkcyjnych mogą być KK.

Aby doprowadzić relację *Zamówione_Kasety* do postaci znajdującej się w PNBC rozbijemy ją na dwie mniejsze relacje (tab. 3.16 i tab. 3.17).

W nowych relacjach kluczami podstawowymi są *Id_Zamówienia* dla relacji *Zamówienie_Data* oraz *Id_Zamówienia, Id_Kasety* dla relacji *Zamówienie_Kasety*. Nie ma już w nich zależności funkcyjnych, których wyznacznikiem nie byłyby KK, stąd relacje te spełniają wymagania postaci normalnej Boyce'a-Codda.

Tabela 3.16. Relacja *Zamówienie_Data*

Id_Zamówienia	*Data*
2	2006.04.08
13	2006.05.23
5	2005.06.01

Tabela 3.17. Relacja *Zamówienie_Kasety*

Id_Zamówienia	***Id_Kasety***
12	1
13	2
13	1
5	3

Zauważmy, że w naszej relacji przed doprowadzeniem jej do postaci normalnej Boyce'a-Codda występowały *anomalie aktualizacji*, np. *anomalia usuwania,* polegająca na tym, iż usunięcie ostatniej kasety znajdującej się na zamówieniu powodowało również usunięcie i tego zamówienia. Po doprowadzeniu tej relacji do PNBC zamówienia możemy dodawać niezależnie od kaset (do tabeli *Zamówienie_Data*).

Czasem PNBC nie jest spełniana w relacjach zawierających więcej niż jeden złożony KK. Tak jest też w naszym przypadku – relacja pierwotna *Zamówione_Kasety* posiadała dwa złożone KK: *Id_Kasety, Data* i *Id_Kasety, Id_Zamówienia*. Ponadto klucze te miały atrybut wspólny – *Id_Kasety*, co też może w ogólnych przypadkach prowadzić do naruszenia przez tabelę PNBC.

Istnieje prosty warunek, krórego spełnienie gwarantuje brak anomalii. Ten warunek nazywa się *postacią normalną Boyce'a-Codda*.

3.2.3.5. Czwarta postać normalna

Dzięki doprowadzeniu relacji do postaci normalnej Boyce'a-Codda możemy się pozbyć nadmiarowości danych spowodowanych zależnościami funkcyjnymi. Jednak oprócz zależności funkcyjnych w relacjach możemy mieć do czynienia z tzw. *zależnościami wielowartościowymi*, z którymi ma do czynienia czwarta postać normalna (4PN). Przypomnimy tutaj, że pojęcie atrybutów (pól) wielowartościowych analizowaliśmy i opisywaliśmy przy okazji 1PN.

Definicja 3-16

Z zależnością wielowartościową (ang. *multivalued dependency*) pomiędzy zbiorami atrybutów ***A*** i ***B*** (oznaczenie ***A*** $\twoheadrightarrow$ ***B***) w relacji składającej się ze zbiorów atrybutów ***A***, ***B*** i ***C*** mamy do czynienia, gdy zbiór wartości ***B*** odpowiadający parze ***A*** i ***C*** w relacji ***R*** zależy jedynie od wartości ***A*** i jest niezależny od wartości ***C***.

Rozróżniamy *trywialne* i *nietrywialnie* zależności wielowartościowe.

Definicja 3-17

Mówimy, że zależność wielowartościowa ***A*** $\twoheadrightarrow$ ***B*** jest ***trywialna***, gdy spełniony jest jeden z poniższych warunków:

- ***B*** jest podzbiorem zbioru ***A***;
- ***A*** i ***B*** zawierają wspólnie wszystkie atrybuty relacji ***R***.

Zależność wielowartościową natomiast nazywamy ***nietrywialną***, gdy nie jest spełniony żaden z powyższych warunków.

Aby lepiej zrozumieć problematykę sprowadzania relacji do 4PN, przyjrzyjmy się następującemu przykładowi.

Przykład 3-24

Rozważmy następującą relację (tabelę): *Osoby* {***Id_Osoby***, ***Obywatelstwo***, ***Kurs***}.

Tabela 3.18. Relacja *Osoby*

Id_Osoby	*Obywatelstwo*	*Kurs*
1	Polskie	MCSA
1	Angielskie	MCSE
1	Polskie	MCSE
1	Angielskie	MCSA
2	Rosyjskie	MCP
2	Hiszpańskie	MCP
3	Włoskie	MCDA

Relacja *Osoby* przedstawia obywatelstwo i informatyczne kursy ukończone przez konkretne osoby. Zawarte są w niej dwie zależności wielowartościowe: *Id_Osoby* $\twoheadrightarrow$ *Obywatelstwo* oraz *Id_Osoby* $\twoheadrightarrow$ *Kurs*.

Dana osoba może mieć kilka obywatelstw i jednocześnie może ukończyć kilka kursów informatycznych. Zbiór wartości atrybutu *Obywatelstwo* dla danej pary *Id_Osoby* i *Kurs* jest zależny wyłącznie od atrybutu *Id_Osoby*, tak samo sytuacja wygląda z atrybutem

Kurs – zbiór wartości tego atrybutu dla danej pary *Id_Osoby* i *Obywatelstwo* jest zależny wyłącznie od atrybutu *Id_Osoby*. Są to też zależności *wielowartościowe nietrywialnie*, gdyż nie jest spełniony żaden z warunków podanych przy definicji zależności wielowartościowych trywialnych. Mamy tu więc do czynienia z dwiema *nietrywialnymi zależnościami wielowartościowymi*, stąd relacja *Osoby* nie jest w 4PN.

Taka sytuacja powoduje, że w tabeli jest sporo nadmiarowych danych. Dla każdego obywatelstwa osoby musimy dodać tyle wierszy, ile ona skończyła kursów. A więc dla osoby o dwóch obywatelstwach i czterech skończonych kursach potrzebne będzie aż osiem wierszy. Jeżeli dana osoba nie skończyła żadnego kursu, nie można jej dodać do tej tabeli. Obecne są tu więc *anomalie aktualizacji*.

W ten sposób mówimy, że relacja znajduje się w *czwartej postaci normalnej*, jeżeli jest ona w postaci normalnej Boyce'a-Codda oraz występuje w niej nie więcej niż jedna *nietrywialna zależność wielowartościowa*.

***Czwarta postać normalna* dopuszcza więc w tabeli występowanie co najwyżej jednej nietrywialnej zależności wielowartościowej.**

Aby taką sytuację zmienić, należy naszą tabelę rozłożyć na dwie mniejsze tabele, w których należy umieścić wyznacznik zależności wielowartościowej, a także po jednym atrybucie wielowartościowo zależnym od tego wyznacznika.

W naszym przypadku relacja (tabela) *Osoby* zostanie rozłożona na dwie mniejsze w sposób pokazany w tab. 3.19 i tab. 3.20.

W powstałych w ten sposób relacjach *Obywatelstwa* i *Kursy* jest już tylko po jednej zależności wielowartościowej: odpowiednio *Id_Osoby* $\twoheadrightarrow$ *Obywatelstwo* oraz *Id_Osoby* $\twoheadrightarrow$ *Kurs*. Są to dodatkowo zależności *wielowartościowe trywialne*, dla każdej bowiem z tych relacji w zależności uczestniczą wszystkie atrybuty. Dlatego relacje te są w 4PN, co pozwoliło nam pozbyć się nadmiarowości spowodowanych *nietrywialnymi zależnościami wielowartościowymi* i związanych z nimi *anomaliami aktualizacji*.

Tabela 3.19. Relacja *Obywatelstwa*

Id_Osoby	*Obywatelstwo*
1	Polskie
1	Angielskie
2	Rosyjskie
2	Hiszpańskie
3	Włoskie

Tabela 3.20. Relacja *Kursy*

Id_Osoby	*Kurs*
1	MCSA
1	MCSE
2	MCP
3	MCDA

Przykład 3-25

Niech w BD *Wypożyczalnia filmów* tabela *Dostawy_Pokoje* ma wygląd tab. 3.21.

Tabela 3.21. Tabela *Dostawy_Pokoje*

Id_Pracownika	*Id_Pokoju*	*Id_Dostawy*
1	3	2
1	3	3
1	4	2
1	4	3
2	5	6
2	5	7

Załóżmy, że dany pracownik może być odpowiedzialny za jeden lub większą liczbę pokoi, jak również za dany pokój może być odpowiedzialny jeden lub większa liczba pracowników. Taka sama relacja (w sensie powiązania) występuje między pracownikiem a dostawą – jeden pracownik może przyjąć jedną lub większą liczbę dostaw, jak również dana dostawa może być przyjęta przez jednego lub większą liczbę pracowników.

Z tabeli *Dostawy_Pokoje* wynika, iż pracownik 1 jest odpowiedzialny za pokoje 3 i 4, a także przyjął dostawy 2 i 3, natomiast pracownik 2 jest odpowiedzialny za pokój 5 i przyjął dostawy 6 i 7. Zauważmy również, że zbiór wartości atrybutu *Id_Pokoju* jest zależny wyłącznie od atrybutu *Id_Pracownika* i jest niezależny od atrybutu *Id_Dostawy*. Z taką samą sytuacją mamy do czynienia w przypadku atrybutu *Id_Dostawy* – zbiór wartości tego atrybutu jest zależny tylko od atrybutu *Id_Pracownika* i niezależny od atrybutu *Id_Pokoju*. Są to więc zależności wielowartościowe:

Id_Pracownika $\twoheadrightarrow$ *Id_Pokoju oraz Id_Pracownika* $\twoheadrightarrow$ *Id_Dostawy.*

Ponadto są to *zależności wielowartościowe nietrywialne* (bowiem żadna cecha zależności wielowartościowej trywialnej nie jest spełniona). Nasza tabela nie spełnia więc wymagań 4PN, ponieważ występuje w niej więcej niż jedna wielowartościowa zależność nietrywialna.

Aby doprowadzić naszą tabelę do czwartej postaci normalnej, należy ją rozbić na tabele mniejsze (zob. tab. 3.22 i tab. 3.23).

W otrzymanych tabelach mamy już tylko po jednej zależności wielowartościowej: odpowiednio *Id_Pracownika* $\twoheadrightarrow$ *Id_Pokoju* w tabeli *Pracownik_Pokój* oraz *Id_Pracownika* $\twoheadrightarrow$ *Id_Dostawy* w tabeli *Pracownik_Dostawa*. Jednak są to zależności wielowartościowe trywialne – suma w sensie teorii mnogości atrybutów wchodzących w skład tych zależności będzie zbiorem składającym się ze wszystkich atrybutów odpowiednich tabel. A więc nasze tabele będą spełniały wymagania 4PN.

Tabela 3.22. Tabela *Pracownik_Pokój*

Id_Pracownik	*Id_Pokoju*
1	3
1	4
2	5

Tabela 3.23. Tabela *Pracownik_Dostawa*

Id_Pracownika	*Id_Dostawy*
1	2
1	3
2	6
2	7

Zauważmy, że tabela *Dostawy_Pokoje* przed normalizacją do 4PN posiadała *anomalie aktualizacji*, a przez to i *nadmiarowe* dane. Dla przykładu, jeżeli chcielibyśmy pracownikowi 1 przypisać nowy pokój, za który byłby odpowiedzialny, to do tabeli *Dostawy_Pokoje* musielibyśmy wstawić dwa zamiast jednego rekordu – dla każdej dostawy, którą ten pracownik przyjął, po jednym rekordzie. Jest to widoczna nadmiarowość, której pozbyliśmy się, rozbijając tabelę na dwie mniejsze. Po normalizacji wystarczy wstawić tylko jeden rekord do tabeli *Pracownik_Pokój*.

3.2.3.6. Piąta postać normalna

Aby dokładniej omówić *piątą postać normalną* (5PN), należy na początku zapoznać się z pojęciem *zależności złączenia bezstratnego*.

Definicja 3-18

Relacja ***R*** z podzbiorami atrybutów ***A***, ***B***, ***C***, ***...***, ***Z*** spełnia ***zależność złączeniową*** *(ang. join dependency)* wtedy i tylko wtedy, kiedy każdy prawidłowy stan relacji ***R*** jest równy złączeniu rzutów na ***A***, ***B***, ***C***, ..., ***Z***.

Zależność złączenia bezstratnego opisuje własność rozkładu, która gwarantuje, iż nie pojawią się żadne nieautentyczne, czyli nowe krotki, ani żadna krotka nie zostanie utracona w czasie ponownego połączenia za pomocą operacji *łączenia naturalnego* relacji, które powstały z relacji pierwotnej w wyniku rozkładu (*dekompozycji*).

Przeprowadzając proces normalizacji, rozkładamy relację na takie relacje, z których będziemy mogli dostać niezmienioną relację pierwotną w wyniku łączenia relacji mniejszych.

Definicja 3-19

Piąta postać normalna, zwana często ***zależnością rzutu-złączenia***, określa relację, która występuje w czwartej postaci normalnej i dodatkowo nie zawiera ***zależności złączeniowych***.

Aby lepiej zrozumieć istotę zależności złączenia i piątej postaci normalnej, rozważmy następujący przykład:

Przykład 3-26

Przeanalizujemy stan relacji (tabeli) *Sprzedawcy* (tab. 3.24).

Relacja *Sprzedawcy* przedstawia przedstawicieli, którzy sprzedają określone produkty dla określonych firm. Przy tym musi być spełniony warunek*: jeżeli dany sprzedawca **S** pracuje dla firmy **F**, firma **F** produkuje produkt **P** i sprzedawca **S** sprzedaje produkt **P**, to sprzedawca **S** sprzedaje produkt **P** dla firmy **F**.*

Tabela 3.24. Relacja *Sprzedawcy*

Sprzedawca	***Firma***	***Typ***
Kowalski	Fiat	Osobowy
Kowalski	Fiat	Ciężarowy
Kowalski	Toyota	Osobowy
Kowalski	Toyota	Ciężarowy
Nowak	Fiat	Osobowy

Naszą relację możemy więc odzyskać z relacji mniejszych – *Sprzedawca_Firmy* (tab. 3.25; określającej, dla jakiej firmy pracuje dany sprzedawca), *Sprzedawca_Produkt* (tab. 3.26; określającej, jakie produkty sprzedaje dany sprzedawca) oraz *Firma_Produkt* (tab. 3.27; określającej, jakie produkty wytwarza dana firma) poprzez odpowiednie *złączenie naturalne tych relacji*.

Każdy właściwy stan relacji *Sprzedawcy* (właściwy, a więc odpowiadający wymogowi stawianemu przez wcześniej określony warunek) jest równy złączeniu jej mniejszych rzutów. Stąd możemy wnioskować, że nasza relacja spełnia (zawiera) *zależność złączenia*, a więc nie jest ona w 5PN. Musi być rozłożona na mniejsze relacje (tab. 3.25 – 3.27), które będą spełniały już piątą postać normalną i z których będzie można odzyskać naszą wyjściową relację poprzez *złączenie naturalne*.

Doprowadzenie relacji do 5PN pozwala wyeliminować niepotrzebną *nadmiarowość* danych. Wyobraźmy bowiem sobie, że dodajemy nowego sprzedawcę do relacji *Sprzedawcy*, który będzie sprzedawał trzy produkty dla trzech firm, przy założeniu, że każda z tych firm produkuje każdy z trzech produktów. Wtedy będziemy musieli dodać dodatkowe dziewięć krotek – dla każdej ***F*** i każdego ***P*** po jednej krotce. Natomiast po uzyskaniu mniejszych relacji potrzebne będzie dodanie tylko sześciu krotek – po trzy do re-

lacji *Sprzedawca_Firmy* i *Sprzedawca_Produkt*. Redukcja nadmiarowych danych jest więc dość znaczna.

Tabela 3.25. Relacja *Sprzedawca_Firmy*

Sprzedawca	*Firma*
Kowalski	Fiat
Kowalski	Toyota
Nowak	Fiat

Tabela 3.26. Relacja *Sprzedawca_Produkt*

Sprzedawca	*Produkt*
Kowalski	Osobowy
Kowalski	Ciężarowy
Nowak	Osobowy

Tabela 3.27. Relacja *Firma_Produkt*

Firma	*Produkt*
Fiat	Osobowy
Fiat	Ciężarowy
Toyota	Osobowy
Toyota	Ciężarowy

Przykład 3-27

Załóżmy, że pracownicy (wracamy się do BD *Wypożyczalnia filmów*) są odpowiedzialni za składanie zamówień. Jeden pracownik może składać zamówienia u jednego lub wielu dostawców i może zamawiać filmy na jednym lub wielu nośnikach. Dany dostawca może również oferować filmy na jednym lub wielu nośnikach. Przy czym musi być spełniony następujący warunek: *jeżeli pracownik **P** składa zamówienia u dostawcy **D**, składa zamówienia dotyczące nośnika **N** oraz dostawca **D** oferuje filmy na nośniku **N**, to pracownik **P** składa zamówienia dotyczące nośnika **N** u dostawcy **D***.

Sytuację opisaną w Prz. 3.27. przedstawia tabela *Dostawcy* (tab. 3.28).

Tabela ta pokazuje, jacy pracownicy jakie nośniki zamawiają u jakich dostawców. Jeżeli dany dostawca rozszerzy swoją ofertę o kolejny nośnik, to do naszej tabeli dojdzie tyle rekordów, ilu jest pracowników zamawiających ten nośnik i jednocześnie zamawiających u tego dostawcy. Warunek opisujący naszą sytuację powoduje, że tabela jest złączeniem trzech mniejszych tabel: *Pracownik_Dostawca* (tab. 3.29), *Pracownik_Nośnik* (tab. 3.30) oraz *Dostawca_Nośnik* (tab. 3.31).

Tabela 3.28. Tabela *Dostawcy*

Pracownik	*Dostawca*	*Nośnik*
Kowalski	Ambro	DVD
Kowalski	Wars	VCD
Kowalski	Ambro	VCD
Malinowski	Ambro	VCD
Wójcik	Colorado	VHS
Malinowski	Colorado	VHS
Wójcik	Colorado	DVD

Tabela 3.29. Tabela *Pracownik_Dostawca*

Pracownik	*Dostawca*
Kowalski	Ambro
Kowalski	Wars
Malinowski	Ambro
Malinowski	Colorado
Wójcik	Colorado

Tabela 3.30. Tabela *Pracownik_Nośnik*

Pracownik	*Nośnik*
Kowalski	DVD
Kowalski	VCD
Malinowski	VCD
Malinowski	VHS
Wójcik	DVD
Wójcik	VHS

Każdy poprawny stan wyjściowej tabeli *Dostawcy* (poprawny, a więc odpowiadający warunkowi postawionemu wcześniej) jest równy złączeniu naturalnemu jej rzutów, a więc tabel *Pracownik_Dostawca*, *Pracownik_Nośnik* oraz *Dostawca_Nośnik*.

Tabela 3.31. Tabela *Dostawca_Nośnik*

Dostawca	*Nośnik*
Ambro	VCD
Ambro	DVD
Wars	VCD
Wars	VHS
Colorado	DVD
Colorado	VHS

Złączenie tych trzech tabel przebiegałoby następująco: tabelę *Pracownik_Dostawca* łączymy z tabelą *Pracownik_Nośnik* po polu *Pracownik*, tabelę *Pracownik_Nośnik* z tabelą *Dostawca_Nośnik* po polu *Nośnik* oraz tabelę *Pracownik_Dostawca* z tabelą *Dostawca_Nośnik* po polu *Dostawca*. Dlatego możemy wnioskować, iż tabela *Dostawcy* spełnia zależność złączenia, a więc nie spełnia 5PN.

Aby naszą tabelę doprowadzić do 5PN, rozbijamy ją na trzy mniejsze tabele (jej rzuty) w sposób pokazany wyżej. Każda z tych tabel spełnia już 5PN, ponieważ nie jest równa *złączeniu naturalnemu* swoich rzutów (nie spełnia *zależności złączenia*).

Wyniki modelowania danych i optymalizacji BD (*jak widać, nie wszystkie tabele są w 5PN, a nawet w 4PN*) najczęściej przedstawiane są w formie zaprezentowanej na rys. 3.6 dla BD *Wypożyczalnia filmów* (niektóre aspekty przeprowadzonych operacji nad tym projektem zostały opisane w Prz. 3-2 – 3-4 oraz na rys. 3.2).

Na schemacie (rys. 3.6) oprócz samych tabel zostały pokazane również połączenia pomiędzy tabelami oraz typy i stopnie uczestnictwa. Atrybuty zakończone przyrostkiem _PK (od *primary key*) są kluczami podstawowymi tabel lub częściami kluczy podstawowych złożonych. Linie łączące tabele ze sobą wskazują na *klucze podstawowe tabel nadrzędnych* oraz *klucze obce tabel podrzędnych*. Atrybuty niebędące *kluczami podstawowymi* relacji, a będące *kluczami obcymi*, są oznaczone przyrostkiem _FK (od *foreign key*).

W schemacie mamy do czynienia z następującymi powiązaniami między tabelami:

– uczestnictwo obowiązkowe o stopniu jeden;

– uczestnictwo obowiązkowe o stopniu wiele;

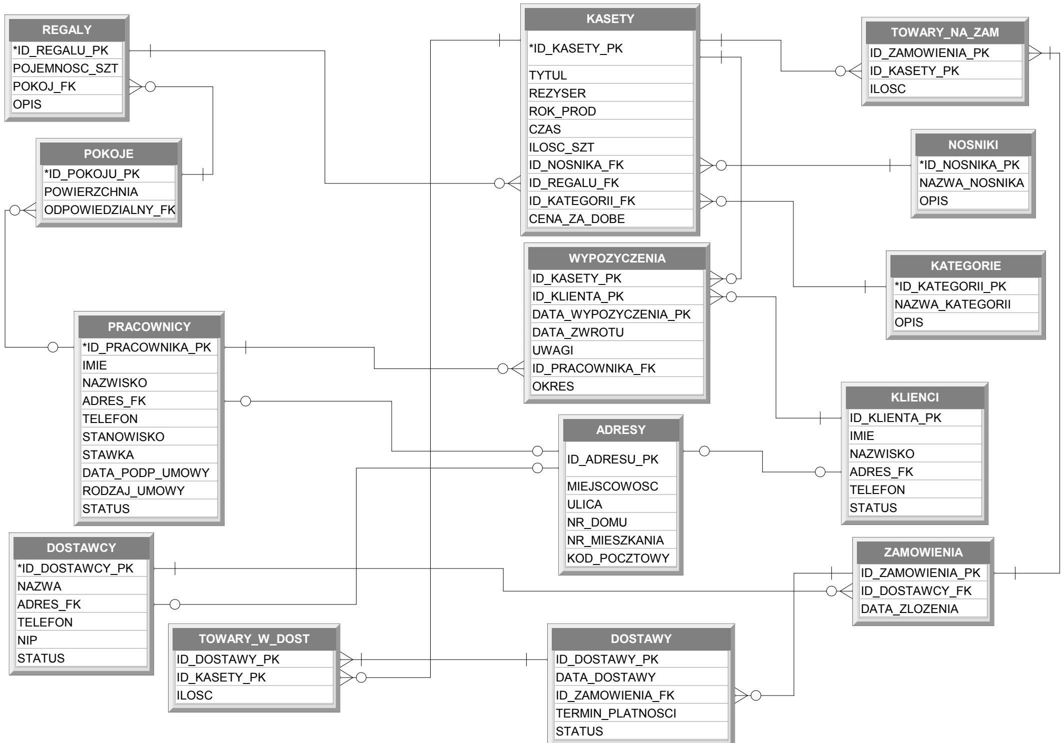
– uczestnictwo opcjonalne o stopniu jeden;

– uczestnictwo opcjonalne o stopniu wiele.

Podsumowując przeprowadzone w tym podrozdziale analizy, podkreślmy następujące fakty:

- normalizacja jest procesem, pozwalającym wyeliminować niespójne zależności oraz nadmiarowość danych w relacji, które na ogół prowadzą do anomalii aktualizacji: anomalii wstawiania, usuwania oraz modyfikacji. Proces ten składa się z szeregu kroków, w których relacja (schemat relacyjny) jest badana (-y) pod względem spełniania kryteriów odpowiadających kolejnym postaciom normalnym;
- w normalizacji kluczowym pojęciem jest zależność funkcyjna pomiędzy atrybutami: atrybut (lub zbiór atrybutów) ***B*** jest zależny funkcyjnie od atrybutu (lub zbioru atrybutów) ***A***, jeżeli z każdą wartością ***A*** powiązana jest dokładnie jedna wartość ***B***;
- wyróżniamy pięć postaci normalnych. Aby relacja znajdowała się w danej postaci normalnej, musi znajdować się we wcześniejszych postaciach normalnych i spełniać dodatkowe warunki:
  - relacja znajduje się w pierwszej postaci normalnej (1PN), jeżeli na przecięciu każdej kolumny i wiersza występują wartości atomowe, a więc żadna wartość nie jest wielowartościowa,
  - druga postać normalna (2PN) zabrania występowania w relacji częściowych zależności funkcyjnych między kluczem podstawowym a atrybutami niekluczowymi,
  - trzecia postać normalna (3PN) zabrania występowania zależności tranzytywnych (przechodnich) między kluczem podstawowym a atrybutami niekluczowymi. Na ogół doprowadzenie relacji do trzeciej postaci normalnej wyeliminowuje nadmiarowość danych, jednak czasem istnieje potrzeba doprowadzenia relacji do wyższych postaci normalnych,
  - w relacji spełniającej postać normalną Boyce'a-Codda (PNBC) wyznacznikiem wszystkich zależności funkcyjnych muszą być klucze kandydujące,
  - czwarta postać normalna (4PN) wymaga, aby w relacji była co najwyżej jedna nietrywialna zależność wielowartościowa pomiędzy atrybutami,
  - piąta postać normalna (5PN) natomiast wymaga, aby relacja nie spełniała zależności złączeniowych.

Sytuacje, w których relacja (tabela) narusza postać normalną Boyce'a-Codda, czwartą czy piątą postać, są bardzo rzadkie, gdyż do ich naruszenia niezbędne jest spełnienie warunków, które w rzeczywistości występują bardzo rzadko i które są eliminowane najczęściej w momencie projektowania bazy.



Rys. 3.6. Struktura i elementy BD *Wypożyczalnia filmów*

Zadania do samokontroli

1. Scharakteryzować cele i istotę normalizacji BD.
2. Podać definicje następujących pojęć:
   - zależność funkcyjna;
   - pełna rodzina zależności funkcyjnych;
   - minimalny zredukowany generator zbioru zależności funkcyjnych;
   - tranzytywna zależność pomiędzy atrybutami relacji;
   - zależność wielowartościowa;
   - zależność złączenia.
3. Wyjaśnić znaczenie terminów:
   - nadmiarowość danych;
   - anomalie wstawiania (dołączania), anomalie usuwania oraz anomalie modyfikacji;
   - dekompozycja;
   - pierwsza, druga i trzecia postaci normalne, postać normalna Boyce'a-Codda, czwarta i piąta postaci normalne.
4. Podać przykłady dekompozycji bez strat i ze stratą danych w relacjach (atrybuty odpowiadające KP są pogrubione):
   - Czytelnicy {**Id_Czyt**, Nazwisko_Czyt, Imię_Czyt, Adres_Czyt};
   - Pacjenci {**Id_Pac**, Nazwisko_Pac, Imię_Pac};
   - Studenci {**Nazwisko_Stud, Imię_Stud**}.

5. Jakie zależności funkcyjne występują w relacji *Studenci_Wydziału* (określić atrybuty kluczowe):

Studenci_Wydziału

Wydział	*Nazwisko*	*Imię*	*Imię_Drugie*
Matematyczny	Jezierski	Marek	Paweł
Informatyczny	Kalinowski	Rafał	Bartosz
Filozoficzny	Lipska	Ewa	Maria
Matematyczny	Jezierski	Ewa	Maria
Informatyczny	Kalinowski	Piotr	Bartosz
Matematyczny	Kalinowski	Marek	Paweł

6. Podać przykłady relacji niespełniających wymagań 1PN.
7. Podać przykłady relacji spełniających:
   - 1PN;
   - 2PN;
   - 3PN;
   - PNBC;
   - 4PN;
   - 5PN.
8. Podać algorytmy sprowadzenia relacji:
   - z 3PN do PNBC;
   - z PNBC do 4PN;
   - z 4PN do 5PN.
9. Podać schemat graficzny zależności funkcyjnych w następujących relacjach:

Producenci_Komputerów

Id_Prod	*Miasto*	***Id_Komp***	***Cena_Komp***
P1	Lublin	K1	3000.00
P1	Lublin	K2	2500.00
P1	Lublin	K3	3600.00
P2	Bydgoszcz	K1	3000.00
P2	Bydgoszcz	K3	3600.00
B1	Mińsk	K2	2500.00
B1	Mińsk	K3	3600.00

Wypożyczenia

Id_Kasety	***Id_Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*
3	5	05.04.2005	15	2
3	5	05.04.2005	15	2
1	8	12.03.2006	3	5

8	2	14.04.2006	2	1
12	4	15.03.2006	15	1
13	4	16.03.2005	18	1
4	14	18.04.2006	3	2

Modele_Marki

Id_Modelu	*Nazwa_Modelu*	*Rok_Produkcji*	*Id_Marki*	*Nazwa_Marki*
AS	Astra	1995	OP	Opel
VE	Vectra	1996	OP	Opel
CA	Carina	2003	TO	Toyota
PA	Panda	1994	FI	Fiat

Zamówione_Kasety

Id_Kasety	***Data***	*Id_Zamówienia*
1	2006.04.08	12
2	2006.05.23	13
1	2006.05.23	13
3	2005.06.01	5

Osoby

Id_Osoby	***Obywatelstwo***	***Kurs***
1	Polskie	MCSA
1	Angielskie	MCSE
1	Polskie	MCSE
1	Angielskie	MCSA
2	Rosyjskie	MCP
2	Hiszpańskie	MCP
3	Włoskie	MCDA

Zamówienie_Kasety

Id_Zamówienia	***Id_Kasety***
12	1
13	2
13	1
5	3

10. Podać opis i strukturę BD:
    - *Biblioteka*;
    - *Apteka*;
    - *Restauracja*;
    - *Przychodnia*;
    - *Szpital*;
    - *Szkoła podstawowa*;
    - *Wydział uczelni wyższej*;
    - *Stacja paliw*;
    - *Wypożyczalnia samochodów*;
    - *Wydawnictwo*;
    - *Księgarnia*;

    w takiej formie jak na rys. 3.6.

Część II
Definiowanie oraz manipulowanie danymi

4. Manipulowanie danymi w SQL

4.1. Typy danych, frazy i polecenia w SQL

4.1.1. Podstawowe typy danych

Ważnym narzędziem posługiwania się danymi, jak również środkiem logicznego projektowania BD (o czym była mowa w podrodz. 3) jest *strukturalny język zapytań*, SQL (ang. *Structured Query Language*).

SQL składa się z dwóch części. Jedna z tych części – *język manipulowania danymi* (ang. *Data Manipulation Language*, DML) przeznaczona jest do wykonywania operacji na danych (manipulowania danymi). Część druga – *język definiowania danych* (ang. *Data Definition Language*, DDL) przeznaczona jest do wykonywania operacji nad obiektami BD (w tym do projektowania BD).

Za pomocą poleceń DDL możemy tworzyć nie tylko tabele, lecz także inne obiekty serwerów SQL. Obiekty definiowane za pomocą DDL nazywamy *metadanymi*. DDL umożliwia tworzenie, modyfikację oraz usuwanie obiektów serwera SQL.

DDL jest najbardziej zależną od implementacji częścią języka SQL. Jest tak dlatego, że producenci baz danych używają bardzo różnych obiektów, mających na celu rozszerzenie możliwości, a tym samym ułatwienie pracy z tego typu serwerami.

Warto również podkreślić, że SQL posiada możliwości wykonania wielu instrukcji zarządzania BD.

Podstawy języka SQL w większym stopniu związane są ze strukturą i typami używanych danych. Dlatego w tym rozdziale skupimy się na podstawach i analizie używania języka do operacji nad danymi. Przy czym w trakcie rozważań zwrócimy uwagę na niektóre różnice w składni tych samych zdań SQL występujących w różnych dialektach (środowiskach, czyli SZBD) języka.

W dalszej części książki (w rozdz. 5) wrócimy do DDL.

Początki SQL sięgają definicji języka o nazwie S*tructured English Query Language*, SEQUEL (z 1974 r.), na podstawie którego został realizowany pierwszy prototyp relacyjnej BD firmy IBM – System R.

Początkowa wersja SQL pojawiła się w 1986 r., a w 1989 r. Międzynarodowy Komitet Standaryzacyjny (ISO, International Standards Organization) opublikował pierwszy międzynarodowy standard języka SQL – SQL1 (lub SQL/89), który został opracowany przez Amerykański Państwowy Instytut Standardów (ANSI, American National Standards Institute). Ostatnio pojawiły się nowe wersje standardu: SQL2 (SQL/92) i SQL3 (SQL/99). Do tej pory są kontynuowane prace nad nowym standardem.

W odróżnieniu od języków teoretycznych, opisanych przez E. Codda, i algebry relacyjnej, przeznaczonych w większości do realizacji *zapytań* kierowanych do BD, SQL jest językiem pełnym, umożliwiającym realizację trzech rodzajów operacji.

Chociaż SQL nie jest językiem programowania, opiera się na podstawowych elementach takich języków, które teraz krótko scharakteryzujemy.

Mówiąc o typach i strukturze danych w SQL, najczęściej wykorzystuje się terminy nie z algebry relacyjnej, lecz z BD: *tabela*, *pole*, *rekord*. Tych samych reguł będziemy trzymać się i my.

W standardzie SQL1 zostały zdefiniowane typy danych języka przedstawione w tab. 4.1.

Tabela 4.1. Typy danych w standardzie SQL1

Typ	*Opis*
`CHARACTER(n)` lub `CHAR(n)`	Typy o stałym rozmiarze pola, które przechowują zawsze *n* znaków; typ oznacza *rekord stałej długości* (ang. *fixed-length string*). Jeżeli w pole tego typu będzie wpisany tekst o liczbie znaków mniejszej niż *n*, to rekord będzie dopełniony odpowiednią liczbą spacji; np. jeżeli komponentowi typu `CHAR(8)` będzie odpowiadała wartość 'Nowak', to w rzeczywistości mamy 'Nowak' (z trzema spacjami). Przy operacjach porównania rekordów dodatkowe spacje *nie będą* wzięte pod uwagę
`NUMERIC[(n,m)]`	Liczby stałoprzecinkowe; *n* – ogólna liczba cyfr w liczbie, *m* – liczba cyfr w części ułamkowej liczby
`DECIMAL[(n,m)]` lub `DEC[(n,m)]`	Liczby stałoprzecinkowe; *n* – ogólna liczba cyfr w liczbie, *m* – liczba cyfr w części ułamkowej liczby
`INTEGER` lub `INT`	Liczby całkowite; ten typ ma podobny sens co `int` w języku C
`SMALLINT`	Liczby całkowite o mniejszym zakresie
`REAL`	Typ rzeczywisty, który odpowiada liczbom w formacie *zmiennoprzecinkowym* (ang. *floating-point numbers*), lecz o dokładności mniejszej niż `DOUBLE PRECISION`
`FLOAT [(n)]`	Liczby w formacie zmiennoprzecinkowym, gdzie *n* jest liczbą bitów przeznaczonych do przechowywania mantysy liczby
`DOUBLE PRECISION`	Specyficzny typ danych, który charakteryzuje się dokładnością większą niż `REAL`

W standardzie SQL2 zostały dodatkowo wprowadzone nowe typy danych (tab. 4.2).

Tabela 4.2. Dodatkowe typy danych w standardzie SQL2

Typ	*Opis*
`VARCHAR(n)`	Typ łańcuchowy o zmiennym rozmiarze pola, który przechowuje co najwyżej `n` znaków; komponenty tego pola mogą posiadać rekordy z liczbą symboli od 0 do *n*
`NCHAR(n)`	Typ łańcuchowy lokalizowanych symboli stałej długości
`NCHAR VARYING(n)`	Typ łańcuchowy lokalizowanych symboli różnej długości
`BIT(n)`	Ciąg bitów stałej długości *n*
`DATE`	Typ daty
`TIME`	Typ czasu
`TIMESTAMP`	Okres czasowy

Między innymi w standardzie zostały zdefiniowane *stałe łańcuchowe* (*stringowe*), które ujmujemy w apostrofy:

Nazwisko_Stud = ‘Nowak’.

W rozdziale piątym pokażemy konkretne przykłady zastosowania i korzystania z wymienionych typów danych na platformie *InterBase*.

W tym samym standardzie pojawiła się możliwość wykonania operacji dodawania (+) i odejmowania (-) w odniesieniu do dat i zdefiniowano 16 dodatkowych wbudowanych funkcji, np. `CURRENT_DATE` – bieżąca data, `BIT_LENGTH` (wyrażenie typu łańcuchowego) – liczba bitów w wyrażeniu, `CHAR_LENGTH` (wyrażenie typu łańcuchowego) – liczba znaków w wyrażeniu.

W standardzie SQL3 wprowadzono nowe funkcje. Użytkownik ma możliwość tworzenia skomplikowanych typów danych, co daje z kolei możliwość organizowania obiektowo-orientowanej pracy z danymi.

Rozszerzony i bardziej szczegółowy opis typów danych (wykorzystywanych w *InterBase*) będzie podany niżej (w podrozdz. 5.2.5).

SQL umożliwia przypisywanie polom specjalnych wartości NULL. W dalszej części taką wartość będziemy wykorzystywać do analizy niektórych operatorów SQL (m.in. *operatorów logicznych*). Dlatego teraz warto przeanalizować wartości NULL bardziej szczegółowo.

Wartość NULL

Wartość NULL może być potraktowana na różne sposoby i ma zastosowanie w kilku typowych sytuacjach wymienionych niżej:

- **wartość nie jest znana**; np. w polu *Data_Urodzenia_Stud* powinna być wpisana odpowiednia wartość, ale nie wiadomo jaka;
- **wartość nie może być zastosowana**; np. w polu *Nazwisko_Małżonka* nie można nadać żadnej wartości osobom niebędącym w związku małżeńskim;
- **wartość nie jest ogólnie dostępna**; np. w polu *Numer_Tel* wartość numeru telefonu nie jest zamieszczana w ogólnodostępnej liście numerów telefonów.

Dla przykładu: *wartość NULL* została po raz pierwszy wspomniana w Prz. 2-17. Warto tutaj jeszcze raz podkreślić, że *wartość NULL* i *wartość zerowa* (0) są dwiema różnymi wartościami. Dla przykładu poniższa tabela ilustruje te różnice.

Tabela 4.3. Tabela *Studenci*

Id_Stud	*Nazw_Stud*	*Nazw_Małżonka*	*Data_Urodzenia_Stud*	*Numer_Tel*	*Stypendium*
1	Kuchta	Mróz	15.06.1986	234-16-25	400
2	Lipska		10.09.1983	756-35-89	0
3	Piotrowski		10.19.1983	756-45-76	0
4	Zieliński	Lato	05.03.1983		800

Wartości zerowe wpisano w polu *Stypendium*.

4.1.2. Struktura poleceń w SQL

Warto podkreślić, że SQL jest językiem bardzo rozbudowanym. Dokument opisujący standard SQL3 posiada powyżej tysiąc stron. Dlatego w naszej książce nie jest możliwe opisanie wszystkich aspektów tego języka, zostaną jedynie przeanalizowane te ważniejsze.

Praktycznie *zdanie* w SQL jest samodzielnym „programem", umożliwiającym zrealizowanie jednej lub więcej operacji nad danymi, lub nad elementami struktur danych w bazie.

Główną jednostką strukturalną w języku SQL jest *zdanie* (*polecenie*), które powinno się kończyć znakiem *średnika* lub naciśnięciem *klawisza ENTER* (w niektórych wersjach SQL zgodnie z regułami standardu SQL2).

Każde zdanie można semantycznie podzielić na składniki, czyli *frazy*. Całe zdanie, jak i pojedyncza fraza w SQL zaczynają się od *słowa kluczowego* (*zarezerwowanego*). Oprócz tego składa się ona z *operatorów* i/lub *funkcji* oraz innych informacji pomocniczych.

Operatory i *funkcje* w SQL pełnią rolę taką jak operatory i funkcje w klasycznych językach programowania.

Początkowa fraza w każdym zdaniu nazywa się również *zdaniem*, *poleceniem* lub *klauzulą.*

W tab. 4.4 podane są nazwy podstawowych poleceń definiowania danych[1].

Tabela 4.4. Podstawowe polecenia definiowania danych w SQL

Typ	*Wyjaśnienie*
`CREATE TABLE`	Tworzenie tabeli
`CREATE VIEW`	Tworzenie widoku (perspektywy)
`ALTER TABLE`	Modyfikacja tabeli
`ALTER VIEW`	Modyfikacja widoku
`DROP TABLE`	Usuwanie tabeli
`DROP VIEW`	Usuwanie widoku

Wymienione w tab. 4.4 polecenia zastosujemy do opisania struktur wykorzystywanych danych.

Kolejne tabele posiadają informacje o podstawowych poleceniach manipulowania danymi (tab. 4.5), zarządzania danymi i transakcjami (tab. 4.6).

1 W niektórych książkach polecenia nazywane są operatorami, chociaż z punktu widzenia semantyki języka nie jest to do końca poprawne.

Tabela 4.5. Podstawowe polecenia manipulowania danymi w SQL

Typ	*Wyjaśnienie*
`SELECT`	Pobieranie danych z tabel[a]
`INSERT`	Wstawianie danych (rekordów) do tabeli
`UPDATE`	Modyfikacja wartości jednego lub kilku pól w jednym lub wielu rekordach
`DELETE`	Usunięcie rekordów

Tabela 4.6. Podstawowe polecenia administrowania danymi i zarządzania transakcjami w SQL

Typ	*Wyjaśnienie*
`CREATE DATABASE`	Tworzenie BD
`CREATE DBAREA`	Tworzenie nowego obszaru przechowywania danych
`GRANT`	Przydzielenie prawa do wykonywania operacji na obiektach BD
`ALTER DATABASE`	Zmiana podstawowych obiektów w BD, jak również ograniczeń dotyczących całej bazy
`ALTER DBAREA`	Zmiana istniejącego obszaru przechowywania danych
`ALTER PASSWORD`	Zmiana hasła dla całej BD
`DROP DATABASE`	Usunięcie istniejącej BD
`DROP DBAREA`	Usunięcie obszaru przechowywania danych
`REVOKE`	Odebranie prawa do wykonywania operacji na obiektach BD
`COMMIT`	Zakończenie transakcji
`ROLLBACK`	Zapisanie zmian dokonanych przez transakcję w bazie oraz zakończenie transakcji
`SAVEPOINT`	Rozwiązanie zmian dokonanych przez transakcję przed zapisaniem ich w bazie oraz zakończenie transakcji

[a] Polecenie SELECT czasem rozpatrywane jest osobno jako nienależące do grupy poleceń manipulowania danymi; nazywa się je *poleceniem pobierania (selekcji) danych.*

W większości komercyjnych SZBD wprowadzono dodatkowe polecenia (np. polecenia definicji i usunięcia *indeksów*, inicjalizacji *procedur*, definicji *wyzwalaczy* (ang. *trigger*). Natomiast nie wszystkie z tych SZBD operują poleceniami zdefiniowanymi w standardach SQL.

Kompilatory poleceń SQL w SZBD są niewrażliwe na wielkość liter (ang. *case insensitive*). W ten sposób napisy `SELECT`, `Select`, `select`, `sElEcT` niczym się nie różnią. Jednak jest tradycyjnie przyjęte, aby *słowa kluczowe* w zdaniach SQL zapisywać dużymi literami. *Kompilatory również nie są wrażliwe na wielkość liter w nazwach pól, nazwach tabel, nazwach pseudonimów* (ang. *alias*).

Wielkość liter jest ważna tylko w stałych łańcuchowych, dlatego '*SELECT*' i '*Select*' mogą być potraktowane jako różne wartości.

Kompilatory poleceń SQL są *wrażliwe na wielkość liter* tylko w odniesieniu do danych typu łańcuchowego.

Dla poleceń SQL dopuszczalny jest dowolny format. Oznacza to, że dane polecenie może składać się z jednego albo z kilku wierszy. Jednakże tekst polecenia wygląda bardziej

estetycznie i jest łatwiejszy do analizy i zrozumienia, gdy *każda fraza zaczyna się od nowego wiersza*.

W dalszej analizie zapisów konkretnych zdań będziemy używać również tradycyjnych znaczków:

- linia pionowa „|", która wskazuje, że należy wybrać jedną wartość z kilku;
- nawiasy {} definiują element obowiązkowy;
- nawiasy [] definiują element opcjonalny.

Zadania do samokontroli

1. Wymienić i wyjaśnić sens poleceń każdej z następujących grup w standardzie SQL:
   - definiowania danych;
   - manipulowania danymi;
   - administrowania danymi i zarządzania transakcjami.
2. Scharakteryzować typy danych przyjętych w SQL. Podać przykłady.
3. Wyjaśnić znaczenie wartości NULL. Podać przykłady.
4. Wyjaśnić podstawy zapisu i kompilacji poleceń SQL.

4.2. Zdanie SELECT

Z punktu widzenia technologii projektowania BD analizę SQL należałoby zacząć od tej jego części, która ma do czynienia z *definiowaniem danych* (patrz: tab. 4.4). Jednakże znaczna większość użytkowników BD rozpoczyna poznawanie SQL od poleceń *manipulowania danymi*, a dokładniej – od pobierania danych z BD. Dlatego też naukę podstaw języka warto zacząć od zdań wykonujących te operacje.

4.2.1. Struktura i składnia polecenia

Najprostszym i najczęściej używanym *zdaniem* SQL jest zdanie `SELECT`, umożliwiające pobieranie danych z tabeli (tabel). Analizę tego zdania warto zacząć od konstrukcji *wybrać-z-pod_warunkiem*. Jak widać, polecenie to składa się z trzech części, inaczej mówiąc z trzech *fraz*. Pierwszą frazą jest sama fraza `SELECT` (wybierz; podkreśliliśmy już wcześniej, że *pierwsza fraza w każdym poleceniu jednocześnie występuje w roli nazwy całego zdania*), drugą – fraza `FROM` (z), trzecią – fraza `WHERE` (pod_warunkiem).

Schematycznie format zapytania `SELECT` ma następującą postać:

```
SELECT L
FROM R
WHERE W,
```

gdzie ***L*** oznacza pewną listę (najczęściej listę pobieranych pól), ***R*** – nazwę tabeli (nazwy tabel), ***W*** – predykat, warunek.

Składnia polecenia wygląda następująco:

```
SELECT [DISTINCT | ALL] {*|[<lista pól>]};
FROM {<lista tabel>};
[WHERE <predykat-warunek wyboru albo połączenia>];
[GROUP BY <lista pól wyniku>];
[HAVING <predykat-warunek dla grupy>];
[ORDER BY <lista pól, wg których należy posortować pobrane dane
  [ASC|DESC]>];
```

Listing 4.1. Składnia polecenia SELECT

W podanym listingu nawiasy [ ] oznaczają opcjonalny charakter ujętej w nie części zdania, a nawiasy {} oznaczają charakter obowiązkowy. Linia pionowa oznacza wybór.

Podana *kolejność fraz* w listingu 4.1 *nie może być inna*, lecz nie każda z tych fraz jest zawsze obowiązkowa. Do *obowiązkowych* należą tylko frazy *SELECT* i *FROM*.

Dla lepszego zrozumienia dalszych rozważań ważne jest, aby pamiętać o następujących zagadnieniach:

- **edytor SQL** to narzędzie, służące do wprowadzania i uruchamiania poleceń SQL;
- SQL jest produktem niezależnym od platformy (czyli od typu serwera i systemu bazodanowego);
- istnieje tzw. problem nieistotnych różnic pomiędzy *dwiema rodzinami* czy *dialektami* implementacji SQL:
  - **rodzina Sybase**: MS SQL (*Server*), Sybase SQL (*Server*) i in.,
  - **rodzina ANSI**: *InterBase*, *Oracle* i większość pozostałych SZBD;
- razem z serwerem *InterBase* zwykle stosuje się edytor zwany *Windows Interactive SQL* (WI SQL). Natomiast w *Oracle* edytorem SQL jest SQL**Plus* (PL/SQL). Za pomocą PL/SQL użytkownik może tworzyć własne obiekty baz danych. SQL i SQL*Plus są bardzo podobne. Interbase i Oracle wymagają, aby każde polecenie SQL kończyło się średnikiem. Edytor WI SQL wymaga, aby ciąg poleceń SQL kończył się słowem kluczowym *GO*;
- implementacje frazy `FROM` różnią się na tych dwóch platformach: w *Oracle* i *InterBase frazy* `FROM` *używamy nawet wtedy, kiedy fraza* `SELECT` *nie wybiera żadnych danych z tabeli, natomiast w Sybase SQL Server,* MS SQL *Server fraza* `FROM` *wymagana jest tylko przy pobieraniu danych znajdujących się w tabeli*[2]. Istnieją również różnice w zastosowaniu fraz `ORDER BY, GROUP BY, HAVING.` Zastosowanie tych fraz wiąże się z zastosowaniem frazy `WHERE`.

Wyjaśnimy teraz każdą frazę polecenia w list. 4.1.

Fraza `SELECT`

Fraza `SELECT` charakteryzuje się następującymi elementami:

- słowo kluczowe `DISTINCT` przeznaczone jest do tego, aby w tabeli wynikowej nie pojawiły się powtarzające się rekordy, inaczej mówiąc, *tabela wynikowa powinna odpowiadać podstawom teorii relacyjnej*;

---

[2] Patrz: Struktura i składnia polecenia.

Fundamentalnym obiektem danych w języku SQL jest dokładnie nie relacja, a tabela. *SQL-tabele nie są zbiorami, a multizbiorami rekordów* (w multizbiorach dopuszczalne jest powtarzanie się elementów).

- słowo kluczowe `ALL` (domyślnie) oznacza, że w tabeli wynikowej mogą pojawić się powtarzające się rekordy (na to jeszcze raz zwrócimy uwagę podczas analizowania operatora SQL: `UNION ALL`);
- symbol '*' określa często spotykaną sytuację, gdy wynikowa tabela składa się ze wszystkich pól tabeli wejściowej (podanej we frazie `FROM`);
- zamiast gwiazdki we frazie może być podana dokładna lista kolumn (i/lub wyrażeń); wtedy nazwy tych kolumn (pól) powinny być oddzielone przecinkami (po nazwie ostatniej kolumny nie stawiamy przecinka).

Fraza `FROM`

W tej frazie podajemy nazwy jednej lub kilku tabel, z których należy pobrać wartości, zgodnie z frazą `SELECT`. Jak podkreśliliśmy wcześniej, najprostsze polecenie do pobierania danych składa się z fraz `SELECT` i `FROM`.

Przykład 4-1

Mamy tabelę (wejściową) *Modele_Marki* (tab. 4.7).

Tabela 4.7. *Modele_Marki*

Id_Modelu	*Nazwa_Modelu*	*Rok_Produkcji*	*Id_Marki*	*Nazwa_Marki*
AS	Astra	1995	OP	Opel
VE	Vectra	1996	OP	Opel
CA	Carina	2003	TO	Toyota
PA	Panda	1994	FI	Fiat

Polecenie:

```
SELECT*
FROM Modele_Marki;
```

w wyniku daje kopię tabeli 4.7.

Id_Modelu	*Nazwa_Modelu*	*Rok_Produkcji*	*Id_Marki*	*Nazwa_Marki*
AS	*Astra*	*1995*	*OP*	*Opel*
VE	*Vectra*	*1996*	*OP*	*Opel*
CA	*Carina*	*2003*	*TO*	*Toyota*
PA	*Panda*	*1994*	*FI*	*Fiat*

Polecenie:

```
SELECT Id_Marki, Nazwa_Marki
FROM  Modele_Marki;
```

umożliwia pobranie wartości dwóch kolumn:

Id_Marki	*Nazwa_Marki*
OP	*Opel*
OP	*Opel*
TO	*Toyota*
FI	*Fiat*

Natomiast zdanie:

```
SELECT DISTINCT Id_Marki, Nazwa_Marki
FROM  Modele_Marki;
```

daje wynik:

Id_Marki	*Nazwa_Marki*
OP	*Opel*
TO	*Toyota*
FI	*Fiat*

Jak widzimy w powyższych przykładach, polecenie *SELECT* w SQL jest najbliższe znaczeniowo *operacji projekcji,* π (ang. *projection*; patrz Def. 2-24) w algebrze relacyjnej.

W wyświetlonych wynikach nagłówki kolumn są nazwami, które zostały użyte w `liście pól`. Aby nie używać nie zawsze zrozumiałych nazw, można nadać kolumnom (polom) w tabeli wyjściowej bardziej czytelne nagłówki (np. *Id_Marki_Samochodu*) poprzez wprowadzenie *aliasów* (pseudonimów; ang. *alias*) nagłówków kolumn.

Używając standardu ANSI SQL, ostatnie polecenie może być zapisane następująco:

```
SELECT DISTINCT Id_Marki AS Id_Marki_Samochodu, Nazwa_Marki
FROM  Modele_Marki;
```

a zatem wynik będzie następujący:

Id_Marki_Samochodu	*Nazwa_Marki*
OP	*Opel*
TO	*Toyota*
FI	*Fiat*

Słowo kluczowe `AS` nie jest wymagane. Następujące wyrażenie zwraca takie same informacje jak ostatnie zapytanie:

```
SELECT DISTINCT Id_Marki Id_Marki_Samochodu, Nazwa_Marki
FROM  Modele_Marki;
```

W niektórych implementacjach SQL (np. w MS SQL *Server*) korzystanie z pseudonimów może się różnić od podanych przykładów: jeżeli używany *alias* zawiera spacje lub jest słowem kluczowym SQL, należy ująć *alias* w pojedynczy cudzysłów lub nawiasy kwadratowe, oznaczające identyfikator danego typu SQL. W poniższym przykładzie wykorzystujemy spacje i nawiasy kwadratowe:

```
SELECT DISTINCT Id_Marki AS 'Id_Marki_Samochodu', Nazwa_Marki AS
[Nazwa_Marki_Samochodu]
FROM  Modele_Marki;
```

Fraza `WHERE`

We frazie `WHERE` definiujemy warunek lub warunki *selekcji rekordów*, lub warunki łączenia rekordów tabel wejściowych (podobnie jak w operacji łączenia w algebrze relacyjnej).

Fraza *WHERE* w zapytaniach SQL jest najbardziej zbliżona znaczeniowo do *operacji selekcji (wyboru;* ang. *selection),* σ (patrz Def. 2-20; Prz. 2-18) w algebrze relacyjnej

Warunki-predykaty we frazie `WHERE` są podobne do tych, które stosuje się w kodach programistycznych, napisanych przykładowo w języku *C* lub *Java*.

Zapytanie SQL, odpowiadające schematowi

```
SELECT L
FROM R
WHERE W,
```

odpowiada wyrażeniu algebry relacyjnej: $\pi_L\ (\sigma_W\ (R))$.

Jako warunki selekcji danych z tabel mogą być zastosowane *predykaty* z użyciem *operatorów* i *funkcji* języka SQL.

Zadania do samokontroli

1. Zapisać schematycznie wszystkie możliwe warianty zdania `SELECT`.
2. Podać odpowiedniki frazy `WHERE` i frazy `FROM` wśród operatorów E. Codda.
3. Które z następujących poleceń SQL jest niepoprawne (jeżeli nie jest poprawne, to wyjaśnić dlaczego; jeżeli poprawne, to podać wynik polecenia):

- ```
  Select distinct Id_marki
  as Id_marki_samochodu,
  nazwa_marki
  from
  modele_marki;
  ```
- ```
  SELECT Id_Marki, Nazwa_Marki
  FROM Modele_Marki
  ```
- ```
  SELECT *,*
  FROM Modele_Marki;
  ```
- ```
  SELECT Id_Marki Nazwa_Marki
  FROM Modele_Marki;
  ```
- ```
  SELECT DISTINCT
  FROM Modele_Marki
  Id_Marki AS Id_Marki_Samochodu', Nazwa_Marki AS
  [Nazwa_Marki_Samochodu];
  ```
- ```
  SELECT *, Id_Marki
  FROM Modele_Marki;
  ```

- ```
  SELECT Id_Marki, ID_MARKI
  FROM Modele_Marki;
  ```

4. Na przykładzie poniższej tabeli *Studenci_Wydziału*:

   zapisać polecenia SQL pobierające następujące dane:

   - *Imiona* i *Nazwiska* wszystkich studentów;
   - nazwy *Wydziałów*;
   - zapisać dwa powyższe polecenia, wykorzystując operatory algebry relacyjnej.

Tabela *Studenci_Wydziału*

Wydział	*Nazwisko*	*Imię*	*Drugie_Imię*
Matematyczny	Jezierski	Marek	Paweł
Informatyczny	Kalinowski	Rafał	Bartosz
Filozoficzny	Lipska	Ewa	Maria
Matematyczny	Jezierska	Ewa	Maria
Informatyczny	Kalinowski	Piotr	Bartosz
Matematyczny	Kalinowski	Marek	Paweł

4.2.2. Operatory SQL

4.2.2.1. Operatory porównania

Rozróżniamy 8 podstawowych operatorów (porównania): = (równe), <> (różne; w niektórych dialektach SQL ten operator wygląda tak: !=), < (mniejsze), > (większe), <= (mniejsze lub równe), >= (większe lub równe), !> (nie więcej niż), !< (nie mniej niż).

W roli *operandów*, porównywanych za pośrednictwem wymienionych operatorów, mogą występować *stałe*, *identyfikatory-nazwy pól* tabel, wspomnianych we frazie `WHERE`, jak również *funkcje*, o których powiemy niżej.

Przykład 4-2

Tabela wejściowa *Towary* wygląda następująco (tab. 4.8):

Tabela 4.8. *Towary*

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	Aaa	20.19	50
2	Vvv	19.96	5
4	Ccc	20.03	25
5	Ppp	10.00	100

Zapytanie:

```
SELECT Cena_Towaru
FROM Towary
WHERE Cena_Towaru > 15.00;
```

daje tabelę wyjściową:
```

Cena_Towaru
20.19
19.96
20.03

We frazie `WHERE` mogą być zastosowane nazwy pól niewymienione we frazie `SELECT`.

Przykład 4-3

Polecenie:

```
SELECT Id_Towaru, Nazwa_Towaru
FROM   Towary
WHERE Cena_Towaru < 15.00;
```

dla tabeli z poprzedniego przykładu daje wynik:

Id_Towaru	*Nazwa_Towaru*
5	*Ppp*

4.2.2.2. Operatory arytmetyczne

Rozróżniamy 5 podstawowych operatorów arytmetycznych: + (dodawanie), - (odejmowanie), * (mnożenie), / (dzielenie), % (modulo).

Ten typ operatorów najczęściej stosuje się we frazie `SELECT`.

Przykład 4-4

Polecenie:

```
SELECT Cena_Towaru*Ilość_Towaru
FROM   Towary;
```

dla tab. 4.8 daje wynik:

*Cena_Towaru*Ilość_Towaru*
1009.50
99.80
550.75
1000.00

Stosując *alias* w ostatnim poleceniu, możemy zmienić nazwę kolumny w tabeli wynikowej:

```
SELECT Cena_Towaru*Ilość_Towaru AS Koszt_Towaru
FROM   Towary;
```

Pierwsze 4 z wyżej podanych operatorów można używać niemal ze wszystkimi typami danych z wyjątkiem typów `DATY` i `CZASU`. Ostatni operator dodatkowo nie może być używany z danymi typów `DECIMAL`, `FLOAT`, `REAL`, `MONEY` i in.

Podkreślmy, że operator odejmowania oprócz swego podstawowego przeznaczenia może zmieniać znak wartości:

```
SELECT  -Ilość_Towaru
FROM  Towary;
```

W wyniku mamy hipotetyczne wartości:

- Ilość_Towaru
- 50
- 5
- 25
- 100

Poniższy przykład ilustruje zastosowanie operatora *modulo*.

Przykład 4-4

Polecenie (hipotetyczne):

```
SELECT  Ilość_Towaru%4
FROM  Towary;
```

daje taką jednokolumnową tabelę:

Ilość_Towaru%4
2
1
1
0

W przypadku zastosowania kilku operatorów pierwszeństwo wykonania operacji jest takie samo, jak i w arytmetyce klasycznej. Oczywiście można zmienić kolejność działań poprzez wykorzystanie nawiasów zwykłych. Najpierw obliczane są najbardziej zagnieżdżone wyrażenia.

4.2.2.3. Operatory logiczne

Operatory logiczne (np. OR, AND, NOT) najczęściej wykorzystuje się we frazie `WHERE` do budowy bardziej skomplikowanych warunków (porównań).

Wynikiem operacji porównania jest wartość typu `BOOLEAN`, która przyjmuje tylko dwie wartości: TRUE albo FALSE. Oprócz tych wartości SQL wspiera jeszcze jedną wartość logiczną: UNKNOWN. Aby wyjaśnić sens tego typu wartości, przypomnimy znaczenie wartości NULL (patrz tab. 4.3) i przeanalizujemy reguły jej wykorzystania. Warto pamiętać o dwóch rzeczach:

- *jeżeli wartość NULL (razem z dowolną inną wartością) jest argumentem binarnego operatora arytmetycznego (*****, **+**, **-** albo /), wynikiem działania tego operatora będzie wartość NULL*;

- *porównanie wartości NULL z dowolną inną wartością (w tym – wartością NULL) pośrednictwem operatorów porównania w wyniku daje wartość UNKNOWN.*

W ten sposób zastosowanie operatorów logicznych polega na wykorzystaniu *trójwartościowej logiki*. Poniższa tabela (tab. 4.9) daje możliwość zrozumienia podstaw działania operatorów logicznych na wartościach ***x*** i ***y***.

Warunek we frazie *WHERE* zapytania SQL stosuje się w odniesieniu do każdego rekordu tabeli wymienionej we frazie FROM. Dla każdego rekordu będzie wyliczona jedna z trzech możliwych wartości: TRUE, FALSE, UNKNOWN. *Do tabeli wynikowej będą dołączone tylko rekordy, którym odpowiada wartość* TRUE.

Tabela 4.9. Tabela prawdy logiki trójwartościowej

x	***y***	***x** AND **y***	***X** OR **y***	*NOT **x***
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Podstawy zastosowania operatorów logicznych przeanalizujemy na kilku przykładach.

Operator OR

Przykład 4-5

Polecenie:

```
SELECT Cena_Towaru,Ilość_Towaru
FROM  Towary
WHERE (Cena_Towaru < 15.00 OR Nazwa_Towaru = 'Aaa')[3];
```

dla tab. 4.8 daje taki wynik:

Cena_Towaru	*Ilość_Towaru*
20.19	50
10.00	100

[3] W niektórych dialektach SQL zastosowanie nawiasów w takiej sytuacji nie jest konieczne.

Operator AND

Przykład 4-6

Zdanie:

```
SELECT*
FROM  Towary
WHERE Cena_Towaru <= 50.00 AND Id_Towaru > 2;
```

w odniesieniu do tej samej tab. 4.8 prowadzi do pobrania dwóch rekordów, których porównanie odpowiednich wartości na podstawie warunku we frazie WHERE odpowiada wartości logicznej TRUE:

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
2	Vvv	19.96	5
4	Ccc	20.03	25

Przykład 4-7

Sytuacja z zastosowaniem kilku operatorów może wyglądać następująco:

```
SELECT *
FROM  Towary
WHERE (Cena_Towaru <= 50.00 OR Id_Towaru > 2) AND Nazwa_Towaru =
'Aaa';
```

W tym poleceniu istotne jest umieszczenie nawiasów. W naszym wypadku wynikowa tabela składa się tylko z jednego rekordu:

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	Aaa	20.19	50

Jeżeli przesuniemy nawiasy:

```
SELECT *
FROM  Towary
WHERE Cena_Towaru <= 50.00 OR (Id_Towaru > 2 AND Nazwa_Towaru =
'Aaa');
```

to mamy inny wynik:

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	Aaa	20.19	50
2	Vvv	19.96	5
4	Ccc	20.03	25

Aby sprawdzić, czy ***x*** posiada wartość NULL, należy skorzystać ze specjalnej konstrukcji SQL:

x IS NULL or ***x IS NOT NULL***.

Wyrażenie ***x IS NULL*** odpowiada wartości TRUE, jeżeli *x* rzeczywiście jest równe NULL. Natomiast wyrażenie ***x IS NOT NULL*** w wyniku daje wartość TRUE, jeżeli *x* nie jest wartością typu NULL.

Przykład 4-8

Zapytanie:

```
SELECT*
FROM  Towary
WHERE Cena_Towaru IS NULL;
```

nie pobiera żadnego rekordu z tabeli *Towary*.

Inne polecenie

```
SELECT*
FROM  Towary
WHERE Cena_Towaru IS NOT NULL;
```

daje kopię tej tabeli.

4.2.2.4. Operatory znakowe

Standard ANSI definiuje dwa operatory, umożliwiające porównanie danych łańcuchowych z wzorcem (*operator LIKE*) i dopasowania wartości stringowych pochodzących z różnych pól i ich dalszego rozmieszczenia w jednym (wynikowym) polu (operator *konkatenacji*, mający wygląd dwóch równoległych linii pionowych: || ; w niektórych dialektach SQL operatorowi odpowiada symbol +).

Operator LIKE (albo NOT LIKE*) jest w rzeczywistości predykatem* wykorzystywanym we frazie WHERE i wymagającym zadania szablonu, z którym będzie wykonane porównanie podanych wartości.

Ogólny wygląd formalnego zapisu tej operacji można przedstawić następująco:

x* LIKE *y ,

gdzie ***x*** jest testowanym ciągiem symboli, ***y*** – szablonem (wzorcem), czyli ciągiem symboli, składającym się z konkretnych znaków alfabetycznych, cyfr i symboli specjalnych, jak również nieobowiązkowych symboli-szablonów: **'%'** i '_'.

*Pierwszy z tych symboli w **y** odpowiada zerowej lub większej liczbie dowolnych symboli w **x**, a znak podkreślenia – dokładnie jednemu dowolnemu symbolowi w **x**.*

Wynikiem porównania (*x* LIKE *y*) będzie TRUE, jeżeli łańcuch *x* pasuje do wzorca *y*. Natomiast wyrażenie (*x* NOT LIKE *y*) daje TRUE, jeśli łańcuch *x* nie pasuje do wzorca *y*.

Działanie *operatora LIKE* polega na sprawdzeniu, czy wyrażenie znakowe jest identyczne z danym wzorcem. Stosuje się go przede wszystkim we frazie WHERE.

Przeanalizujemy działanie operatorów na przykładzie danych z tab. 4.8.

Operator LIKE

Przykład 4-9

Zdanie:

```
SELECT *
FROM  Towary
WHERE Nazwa_Towaru LIKE 'aaa';
```

Żaden z warunków nie jest równy TRUE, czyli nie będzie wybrany żaden z rekordów.

Teraz zmienimy warunek:

```
SELECT *
FROM   Towary
WHERE Nazwa_Towaru LIKE 'Aaa';
```

Mamy wynik:

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	Aaa	20.19	50

Przykład 4-10

Niech szablon posiada symbole '%' i '_', tabelą wejściową będzie tab. 4.10 (*Towary_1*).

Tabela 4.10. *Towary_1*

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	AAA	20.19	50
2	Vvaaa	19.96	5
4	Cac	20.03	25
5	app	10.00	100
6	AAa	2.12	480

Polecenie:

```
SELECT Nazwa_Towaru
FROM   Towary_1
WHERE Nazwa_Towaru LIKE '%a%';
```

zwraca wartości:

Nazwa_Towaru
Vvaaa
Cac
app
AAa

We frazie `WHERE` *operator LIKE* może być używany razem z innymi operatorami, np. logicznymi. Spójrzmy na kolejny przykład.

Przykład 4-11

Zapytanie:

```
SELECT Nazwa_Towaru
FROM   Towary_1
WHERE Nazwa_Towaru LIKE '%a%'OR Cena_Towaru=20.19;
```

daje nową tabelę:

Nazwa_Towaru
AAA
Vvaaa
Cac
app
AAa

Operator konkatenacji

W odróżnieniu od *operatora LIKE operator konkatenacji* częściej wykorzystywany jest we frazie `SELECT`.

Przykład 4-12

Na podstawie danych tab. 4.11 można połączyć (ale *to nie jest operacja trwałego połączenia*) w jednym polu wartości kilku pól.

Tabela 4.11. *Studenci_Wydziału*

Wydział	*Nazwisko*	*Imię*	*Imię_Drugie*
Matematyczny	Jezierski	Marek	Paweł
Informatyczny	Kalinowski	Rafał	Bartosz
Filozoficzny	Lipska	Ewa	Maria
Matematyczny	Jezierska	Ewa	Maria
Informatyczny	Kalinowski	Piotr	Bartosz
Matematyczny	Kalinowski	Marek	Paweł

Najprostsze polecenie z *operatorem konkatenacji* może mieć następujący wygląd:

```
SELECT Nazwisko || Imię
FROM  Studenci_Wydziału;
```

Wynik operacji będzie następujący:

Nazwisko‖Imię
JezierskiMarek
KalinowskiRafał
LipskaEwa
JezierskaEwa
KalinowskiPiotr
KalinowskiMarek

Dopasowywane wartości można oddzielić spacją lub innym symbolem stworzonym za pomocą dodatkowego *operatora konkatenacji*.

Przykład 4-13

Zapytanie:

```
SELECT     Imię||'     '||Nazwisko AS Imię_Nazwisko
FROM       Studenci_Wydziału;
```

jak widać, tworzy wspomniany sztuczny obszar (składa się z 5 znaków spacji; w tym obszarze można umieszczać dowolne symbole, np. przecinki), a wynik przedstawia jako jedno pole o nazwie *Imię_Nazwisko*:

Imię_Nazwisko
Marek Jezior
Rafał Kalinowski
Ewa Lato
Ewa Jezior
Piotr Kalinowski
Marek Kalinowski

Liczba pól, których wartości można połączyć poprzez *operator konkatenacji*, może być też większa, jednakże technologia dopasowania będzie taka sama.

4.2.2.5. Operatory dodatkowe

Istnieją dwa podstawowe operatory tej grupy: IN i BETWEEN, jak również ich odmiany: NOT IN, NOT BETWEEN. Operatory te, jak i *operator LIKE* są predykatami, czyli częściami frazy `WHERE`.

Formalny zapis pierwszego z operatorów dodatkowych jest następujący:

***x* IN (*y*, ..., *z*),**

gdzie ***x*** jest wartością testową, a ***y***, ..., ***z*** to podany zbiór wartości (najczęściej stałych). W tym przypadku wartość ***x*** będzie porównywana z każdą wartością zbioru.

Wynikiem porównania (*x* IN (*y*, ..., *z*)) będzie wartość TRUE, jeżeli *x* jest elementem zbioru (*y*, ..., *z*). Natomiast wyrażenie (*x NOT* IN (*y*, ..., *z*)) zwraca wartość TRUE, gdy *x* nie jest elementem zbioru (*y*, ..., *z*).

Operator IN

Przykład 4-14

Wróćmy do tabeli *Towary_1*.

Zapytanie:

```
SELECT *
FROM  Towary_1
WHERE Nazwa_Towaru IN ('AAA', 'Vvaaa', 'Cac');
```

daje wynik:

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	AAA	20.19	50
2	Vvaaa	19.96	5
4	Cac	20.06	25

Zbiór może składać się z wartości innych typów. Na przykład:

```
SELECT *
FROM  Towary_1
WHERE Id_Towaru IN(1,2,4).
```

W przykładzie tym zostaną pobrane te same rekordy co w poprzednim zapytaniu. Jeżeli w ostatnim poleceniu *operator IN* zamienimy na *operator NOT IN*, to wyjściowa tabela będzie posiadać dwa ostatnie rekordy tabeli wejściowej.

Zwróćmy uwagę, że ostatnie polecenie można przedstawić w inny sposób:

```
SELECT *
FROM  Towary_1
WHERE Id_Towaru=1 OR Id_Towaru=2 OR Id_Towaru=4;
```

Teraz pokażemy działanie innych operatorów tej grupy.

Formalny zapis *operatora BETWEEN* jest następujący:

***x* BETWEEN *y* *AND* *z*,**

gdzie ***x*** jest wartością testową, a ***y*** i ***z*** to wartości (najczęściej stałe) definiujące dolną i górną granice zbioru wartości. W tym przypadku wartość ***x*** będzie porównywana z każdą wartością zbioru.

Wynikiem porównania (*x* BETWEEN *y AND z*) będzie wartość *TRUE,* jeżeli *x* jest elementem przedziału, którego dolną i górną granicę definiują odpowiednio *y* i *z*. Natomiast wyrażenie (*x NOT* BETWEEN *y AND z*) zwraca wartość *TRUE* w przeciwnym przypadku.

Operator BETWEEN

Przykład 4-15

Zdanie z operatorem BETWEEN:

```
SELECT *
FROM  Towary_1
WHERE Id_Towaru BETWEEN 1 AND 4;
```

daje wynikową tabelę, dokładnie odpowiadającą wynikowi ostatniego polecenia w Prz. 4-14.

Ostatnie zapytanie można przedstawić w innej formie:

```
SELECT *
FROM  Towary_1
WHERE Id_Towaru >=1 AND Id_Towaru <=4;
```

Zdanie z operatorem NOT BETWEEN:

```
SELECT *
FROM   Towary_1
WHERE Id_Towaru NOT BETWEEN 1 AND 4;
```

daje w wyniku:

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
5	app	10.00	100
6	AAa	2.12	480

W standardzie SQL2 wprowadzono dwa nowe operatory, podobne semantycznie do *operatora IN*, które można sklasyfikować raczej jako operatory porównania: *operator ANY* i *operator ALL*. Operatory te wykorzystujemy do porównania wartości jakiegoś wyrażenia z polem danych, które zwraca *podzapytanie*. Przykłady zastosowania tych operatorów będą podane w podrozdz. 4.3.1.

4.2.2.6. Operatory mnogościowe

Język SQL posiada operatory do bezpośredniej realizacji takich operacji algebry relacyjnej, jak *suma*, *przecięcie* i *różnica.* Tym operacjom w SQL odpowiadają operatory: UNION, INTERSECT, MINUS (lub EXCEPT).

Warto wspomnieć (patrz: Def. 2-15 – 2-18), że te operatory (operacje) pobierają dane z dwóch (lub więcej) tabel, mających jednakowe nagłówki, a dane w odpowiednich polach są zgodnych typów.

Składnia poleceń z operatorami danej grupy jest bardzo podobna do siebie: *każdy z operatorów znajduje się pomiędzy tekstami dwóch zapytań,* które należy ująć w nawiasach (to wymaganie nie jest obowiązkowe we wszystkich dialektach (implementacjach) SQL).

Dalsze analizy przeprowadzimy, uwzględniając dane znajdujące się w tabelach *Studenci* {*Nazwisko*, *Imię*, *Imię_Drugie*} (tab. 4.11) i *Pracownicy* {*Nazwisko*, *Imię*, *Imię_Drugie*} (tab. 4.12).

Tabela 4.11. *Studenci*

Nazwisko	*Imię*	*Imię_Drugie*
Jezierski	Marek	Paweł
Kalinowski	Rafał	Bartosz
Lipska	Ewa	Maria
Jezierska	Ewa	Maria
Kalinowski	Piotr	Bartosz
Kalinowski	Marek	Paweł

Tabela 4.12. *Pracownicy*

Nazwisko	*Imię*	*Imię_Drugie*
Rombalski	Krzysztof	Wacław
Kalinowski	Rafał	Bartosz
Michalska	Maria	Ewa
Jezierska	Ewa	Maria
Bednarz	Edyta	Magdalena
Kalinowski	Marek	Paweł
Struś	Piotr	Zbigniew
Jezierski	Marek	Paweł
Lipska	Ewa	Maria

Operator UNION

Przykład 4-16

Zdanie:

```
(SELECT *
FROM  Studenci)
UNION
(SELECT *
FROM  Pracownicy);
```

pobiera niepowtarzające się rekordy z tabel:

Nazwisko	*Imię*	*Imię_Drugie*
Jezierski	Marek	Paweł
Kalinowski	Rafał	Bartosz
Lipska	Ewa	Maria
Jezierska	Ewa	Maria
Kalinowski	Piotr	Bartosz
Kalinowski	Marek	Paweł
Rombalski	Krzysztof	Wacław
Michalska	Maria	Ewa
Bednarz	Edyta	Magdalena
Struś	Piotr	Zbigniew

Przy zmianie kolejności zdań `SELECT` w ostatnim poleceniu:

```
(SELECT *
FROM Pracownicy)
UNION
(SELECT *
FROM Studenci);
```

mamy inną kolejność rekordów w tabeli wynikowej:

Nazwisko	*Imię*	*Imię_Drugie*
Rombalski	Krzysztof	Wacław
Kalinowski	Rafał	Bartosz
Michalska	Maria	Ewa
Jezierska	Ewa	Maria
Bednarz	Edyta	Magdalena
Kalinowski	Marek	Paweł
Struś	Piotr	Zbigniew
Jezierski	Marek	Paweł
Lipska	Ewa	Maria
Kalinowski	Piotr	Bartosz

Każde ze zdań może również posiadać dodatkowe warunki jak w poniższym poleceniu:

```
(SELECT Nazwisko, Imię
FROM  Pracownicy
WHERE Imię LIKE 'M%')
UNION
(SELECT  Nazwisko, Imię
FROM  Studenci
WHERE Nazwisko LIKE 'K%');
```

Wynik będzie wyglądał następująco:

Nazwisko	*Imię*
Michalska	Maria
Kalinowski	Marek
Jezierski	Marek
Kalinowski	Piotr

O ile tabela jest strukturą trochę inną niż relacja w algebrze relacyjnej, SQL daje możliwości pobierania danych z tabel *różniących się nagłówkami* i łączenia je w jedną tabelę *powtarzających się danych*.

Przykład 4-17

Niech tabela *Studenci* ma ten sam nagłówek, tabela *Pracownicy* zaś inny: *Pracownicy* {*Nazwisko_Prac*, *Imię_Prac*, *Imię_Drugie_Prac*}. W takiej sytuacji można zastosować *alias* (chociaż w większości serwerów SQL nazwy kolumn drugiej tabeli nie mają wpływu na nazwy kolumn tabeli wynikowej). Wtedy ostatnie polecenie z poprzedniego przykładu będzie miało postać:

```
(SELECT Nazwisko_Prac, Imię_Prac
FROM  Pracownicy
WHERE Imię_Prac LIKE 'M%')
```

```
UNION
    (SELECT  Nazwisko AS Nazwisko_Prac, Imię AS Imię_Prac
    FROM  Studenci
    WHERE Nazwisko LIKE 'K%');
```

Tabela wynikowa będzie wyglądać następująco:

Nazwisko_Prac	*Imię_Prac*
Michalska	Maria
Kalinowski	Marek
Jezierski	Marek
Kalinowski	Piotr

Operator UNION ALL

Jak już wiemy, *realizacja polecenia* SELECT *nie przewiduje domyślnego usuwania z tabeli wyjściowej powtarzających się wartości.* To usuwanie może być inicjowane tylko poprzez zastosowanie słowa kluczowego DISTINCT (mającego takie samo znacznie jak *operator* σ algebry relacyjnej). W odróżnieniu od tego, w sytuacjach zastosowania *operatorów mnogościowych* w celu wyłączenia usuwania powtórzeń odpowiedni operator należy uzupełnić słowem kluczowym ALL.

Uwzględniając powyższe rozważania, warto zaznaczyć, że odpowiednia operacja będzie wykonana na podstawie reguł *multizbiorów* (ale nie zbiorów).

Przykład 4-18

Polecenie:

```
(SELECT *
FROM  Studenci)
UNION ALL
(SELECT *
FROM  Pracownicy);
```

w odniesieniu do tab. 4.11 i 4.12 posiada nowe słowo kluczowe (ALL) i również daje nowy wynik:

Nazwisko	*Imię*	*Imię_Drugie*
Jezierski	Marek	Paweł
Kalinowski	Rafał	Bartosz
Lipska	Ewa	Maria
Jezierska	Ewa	Maria
Kalinowski	Piotr	Bartosz
Kalinowski	Marek	Paweł
Rombalski	Krzysztof	Wacław
Kalinowski	Rafał	Bartosz
Michalska	Maria	Ewa
Jezierska	Ewa	Maria

Nazwisko	*Imię*	*Imię_Drugie*
Bednarz	Edyta	Magdalena
Kalinowski	Marek	Paweł
Struś	Piotr	Zbigniew
Jezierski	Marek	Paweł
Lipska	Ewa	Maria

Operatory INTERSECT, INTERSECT ALL

Przykład 4-19

Polecenie:

```
(SELECT *
FROM   Studenci)
INTERSECT
(SELECT *
FROM   Pracownicy);
```

pobiera 5 rekordów z pierwszej tabeli:

Nazwisko	*Imię*	*Imię_Drugie*
Jezierski	Marek	Paweł
Kalinowski	Rafał	Bartosz
Lipska	Ewa	Maria
Jezierska	Ewa	Maria
Kalinowski	Marek	Paweł

Natomiast zapytanie:

```
(SELECT *
FROM Pracownicy)
INTERSECT
(SELECT *
FROM   Studenci);
```

daje inną kolejność tych samych rekordów:

Nazwisko	*Imię*	*Imię_Drugie*
Kalinowski	Rafał	Bartosz
Jezierska	Ewa	Maria
Kalinowski	Marek	Paweł
Jezierski	Marek	Paweł
Lipska	Ewa	Maria

Oczywiste jest, że *operator* `INTERSECT ALL` w naszym wypadku nic nie zmieni.

Operatory MINUS i MINUS ALL (EXCEPT i EXCEPT ALL)

Dla tych samych tabel te dwa operatory dają również ten sam wynik.

Przykład 4-20

```
(SELECT *
FROM  Studenci)
MINUS
(SELECT *
FROM  Pracownicy);
```

Wynik:

Nazwisko	*Imię*	*Imię_Drugie*
Kalinowski	Piotr	Bartosz

Natomiast polecenie:

```
(SELECT *
FROM Pracownicy)
MINUS
(SELECT *
FROM  Studenci);
```

daje odpowiednio:

Nazwisko	*Imię*	*Imię_Drugie*
Rombalski	Krzysztof	Wacław
Michalska	Maria	Ewa
Bednarz	Edyta	Magdalena
Struś	Piotr	Zbigniew

Zadania do samokontroli

1. Wymienić podstawowe grupy operatorów SQL standardu ANSI.
2. Scharakteryzować przeznaczenie operatorów każdej z grup. Podać ogólną postać składni odpowiedniego polecenia.
3. Dla tabeli *Towary*:

Tabela *Towary*

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*	*Producent*
1	Aaa	20.19	50	AZLK
2	Vvv	19.96	5	MORE
4	Ccc	20.03	25	TARA
5	Ppp	10.00	100	MARA

podać wyniki działania poleceń:

- ```
  SELECT Cena_Towaru*Ilość_Towaru Koszt
  FROM  Towary;
  ```
- ```
  SELECT (Cena_Towaru*Ilość_Towaru)- 10
  FROM  Towary;
  ```
- ```
  SELECT Cena_Towaru+Nazwa_Towaru
  FROM  Towary;
  ```
- ```
  SELECT Cena_Towaru, Nazwa_Towaru
  FROM  Towary
  WHERE Nazwa_Towaru > 'c';
  ```
- ```
  SELECT Cena_Towaru, Nazwa_Towaru
  FROM  Towary
  WHERE Nazwa_Towaru > 'c' OR Id_Towaru >= 2;
  ```
- ```
  SELECT *
  FROM  Towary
  WHERE Nazwa_Towaru > 'c' AND Id_Towaru NOT LIKE '_ _ _ _';
  ```
- ```
  SELECT *
  FROM Towary
  WHERE Nazwa_Towaru LIKE '_ % _ ' AND Nazwa_Towaru > 'c';
  ```
- ```
  SELECT Producent||','||Nazwa_Towaru
  FROM  Towary;
  ```

4. Dla tabeli *Towary* z polecenia 3 przepisać podane niżej zdania z wykorzystaniem innych operatorów SQL:

- ```
  SELECT *
  FROM Towary
  WHERE Nazwa_Towaru LIKE '_ % _ ' OR Nazwa_Towaru LIKE  'c';
  ```
- ```
  SELECT *
  FROM  Towary
  WHERE Nazwa_Towaru LIKE '_ % _ ' OR Nazwa_Towaru NOT LIKE  'c';
  ```
- ```
  SELECT *
  FROM  Towary
  WHERE Id_Towaru = 1 OR Id_Towaru = 2 OR Id_Towaru = 10;
  ```
- ```
  SELECT *
  FROM  Towary
  WHERE Id_Towaru >= 1 AND Id_Towaru <=10;
  ```

5. Podać przykłady (tabeli i poleceń SQL) z zastosowaniem operatorów mnogościowych.

6. Podać przykłady tabel, działań poleceń SQL z Prz. 4-17 i 4-18, które prowadzą do różnych wyników.

7. Podać przykłady poleceń SQL na podstawie operatorów `NOT IN`, `NOT BETWEEN`, `INTERSECT ALL`, `MINUS ALL`.

4.2.3. Funkcje w języku SQL

Funkcje SQL, jak i dowolne inne funkcje posiadają *argumenty* lub *parametry* (w odróżnieniu od operatorów). Wszystkie funkcje, jak i operatory można warunkowo podzielić na grupy. Dalej przeanalizujemy ważniejsze funkcje.

4.2.3.1. Funkcje agregujące (grupowe)

Funkcje agregujące zwracają podsumowania dla całej tabeli lub dla grup wierszy w tabeli. Funkcje te są używane z frazami `GROUP BY` i `HAVING`; *argumentem najczęściej jest lista kolumn*. Najważniejsze funkcje tej grupy, ich parametry oraz wyniki działania pokazuje tab. 4.13.

Tabela 4.13. Ważniejsze funkcje agregujące

Funkcja	*Wynik*
`AVG ([ALL|DISTICT] {<wyrażenie>})`	Zwraca ***średnią*** z wartości wyrażenia ***numerycznego*** obliczonego dla wszystkich wierszy tabeli/grupy. Klauzula `DISTINCT` wymusza usunięcie powtórek przed obliczeniem średniej. Wartości NULL są ignorowane. Jeśli wszystkie obliczone wyrażenia mają wartość NULL, to AVG również zwraca NULL
`COUNT ([ALL|DISTICT] {<wyrażenie>})`	Zwraca ***liczbę*** wybranych wierszy
`MIN ({<wyrażenie>})`	Zwraca ***najmniejszą wartość*** w zbiorze wynikowym; działa nad polami ***typu liczbowego*** i ***typu tekstowego***
`MAX ({<wyrażenie>})`	Zwraca ***największą wartość*** w zbiorze wynikowym; działa nad polami ***typu liczbowego*** i ***typu tekstowego***
`SUM([ALL|DISTINCT] {<wyrażenie>})`	Zwraca ***sumę*** kolumn ze zbioru wynikowego; działa nad polami ***typu liczbowego***

Przeanalizujmy zastosowanie tych funkcji na przykładzie tabeli wejściowej *Pracownicy*, która ma postać tab. 4.14.

Tabela 4.14. Tabela *Pracownicy*

Id_Prac	*Imię_Prac*	*Nazwisko_Prac*
15	Roman	Wójcik
7	Maria	Nowak
3	Krzysztof	Kowalski
19	Hanna	Malinowska

Przykład 4-21

Zdanie:

```
SELECT COUNT(*)
FROM   Pracownicy;
```

daje wynik:

COUNT(*)
4

Przykład 4-22

Polecenie:

```
SELECT MIN (Id_Prac) Minim_Id, MAX (Id_Prac) Maxim_Id,
MIN (Imię_Prac), MAX (Imię_Prac)
FROM Pracownicy;
```

zwraca następującą tabelę wynikową:

Minim_Id	*Maxim_Id*	*MIN (Imię_Prac)*	*MAX (Imię_Prac)*
3	19	Hanna	Roman

Stosując funkcje agregujące, należy pamiętać o kilku ważnych kwestiach.

1. **W procesie grupowania (agregowania) wartość NULL nie będzie brana pod uwagę.**
2. **Funkcja COUNT(*) zawsze zwraca ogólną liczbę rekordów w tabeli, natomiast funkcja COUNT(*a*) zwraca liczbę rekordów, dla których wartość wyrażenia *a* nie jest równa NULL.**

4.2.3.2. Funkcje matematyczne

Funkcje matematyczne umożliwiają takie operacje, jak wyznaczanie wartości bezwzględnej, wartości trygonometrycznych, wyliczanie pierwiastków kwadratowych i podnoszenie wartości do potęgi. Tab. 4.15 wymienia niektóre z funkcji tej grupy.

Tabela 4.15. Funkcje matematyczne

Funkcja	*Wynik*
ABS {(<*kolumna*> | *wartość*)}	Zwraca wartość bezwzględną typu ***numerycznego***
COS | SIN | TAN {(<*kolumna*>| *wartość*)}	Zwraca *Cos*, *Sin* lub *Tg* kąta (w radianach); wartości typu ***zmiennoprzecinkowego***
EXP {(<*kolumna*> | *wartość*)}	Zwraca liczbę *e* podniesioną do potęgi równej podanej wartości
LOG {(<*kolumna*> | *wartość*)}	Zwraca logarytm naturalny wartości {(<*kolumna*> | *wartość*)}
POWER {(<*kolumna*> | | *wartość, y*)}	Zwraca wartość (<*kolumna*>|*wartość*) podniesioną do potęgi *y*
SQRT {(<*kolumna*> | | *wartość*)}	Zwraca pierwiastek kwadratowy określonej wartości {(<*kolumna*> | *wartość*)}
CEIL {(<*kolumna*> | | *wartość*)}	Zwraca najmniejszą liczbę całkowitą większą lub równą podanemu argumentowi
FLOOR {(<*kolumna*> | | *wartość*)}	Zwraca największą liczbę całkowitą mniejszą lub równą podanemu argumentowi

Przykład 4-23
Niech tabela wejściowa wygląda następująco.

Tabela 4.16. Tabela *Liczby*

A	*B*	*C*
1	5.5	–0.1
3	3.4	4.5
4	7.8	9.8
10	–1.2	–11.0

Zapytanie:

```
SELECT ABS(B), ABS(C*2), POWER (A,2)
FROM Liczby;
```

daje następującą tabelę wynikową:

ABS(B)	*ABS (C*2)*	*POWER (A,2)*
5.5	0.2	1
3.4	4.5	9
7.8	9.8	4
–1.2	11.0	100

Inne zapytanie:

```
SELECT CEIL(B), FLOOR(C), ABS(FLOOR(C))
FROM Liczby;
```

daje w wyniku:

CEIL(B)	*FLOOR (C)*	*ABS(FLOOR (C))*
6	- 1	1
4	4	4
8	9	9
- 1	- 11	11

Jak widzimy, w ostatnim przypadku argumentem jednej funkcji (`ABS()`) jest inna funkcja (`FLOOR()`).

W przypadku korzystania z funkcji matematycznych z *walutowymi* typami danych większość implementacji SQL wymaga, by wartość walutowa była poprzedzona znakiem dolara ($).

4.2.3.3. Funkcje znakowe

Większość tych funkcji operuje jedynie na typach danych `VARCHAR(n)`, `CHAR(n)`, `NCHAR(n)`, `NCHAR VARYING(n)`. Inne typy danych muszą być najpierw *skonwertowane* za pomocą odpowiednich funkcji (patrz podrozdz. 4.2.3.5. Funkcje konwersji danych.

Niektóre z funkcji znakowych podano w tab. 4.17.

Tabela 4.17. Funkcje znakowe

Funkcja	*Wynik*
CHAR{(<*kolumna*>| *wartość*)}	Zwraca odpowiednik znakowy wartości kodu ASCII
LOWER{(<*kolumna*>| *wartość*)}	Zwraca małe litery
UPPER{(<*kolumna*>| *wartość*)}	Zwraca duże litery
CONCAT{(<*lista wyrażeń*>)}	Daje wynik podobny do wyniku działania operatora konkatenacji[a]
REPLACE{('*string1*', '*string2*', '*string3*')}	Zastępuje wszystkie wystąpienia łańcucha *string2* przez łańcuch *string3* w łańcuchu *string1*

[a] W niektórych implementacjach SQL (np. Transact-SQL, T-SQL w MS SQL Server) słowo kluczowe CONCAT zastępuje znak +.

Przykład 4-24

Niech tabelą wejściową będzie tabela *Pracownicy* (tab. 4.18).

Tabela 4.18. Tabela *Pracownicy*

Imię_Prac	*Nazwisko_Prac*
Roman	Wójcik
Maria	Nowak
Krzysztof	Kowalski
Hanna	Malinowska

Wynikiem polecenia:

```
SELECT  Imię_Prac, UPPER(Nazwisko_Prac), LOWER (Imię_Prac)
FROM    Pracownicy;
```

będzie tabela:

Imię_Prac	*UPPER (Nazwisko_Prac)*	*LOWER (Imię_Prac)*
Roman	WÓJCIK	roman
Maria	NOWAK	maria
Krzysztof	KOWALSKI	krzysztof
Hanna	MALINOWSKA	hanna

Zdanie:

```
SELECT CONCAT (Imię_Prac, Nazwisko_Prac) Full_Name
FROM Pracownicy;
```

daje inny układ wierszy (por. z Prz. 4-12 i 4-13) niż w przypadku zastosowania *operatora konkatenacji*.

W naszym przypadku wynik wygląda następująco:

Full_Name
RomanWójcik
MariaNowak
KrzysztofKowalski
HannaMalinowska

Podstawowa różnica w zastosowaniu operatora znakowego *konkatenacji* i funkcji znakowej `CONCAT()` polega na przedstawieniu wyników polecenia. W pierwszej sytuacji odległość pomiędzy dopasowywanymi wartościami, tworzącymi osobny wiersz w tabeli wynikowej, zależy tylko od frazy `SELECT`, w drugiej – od wartości parametru `n` zmiennej `CHAR(n)`, definiującej typ danych w odpowiednim polu.

Jeżeli np. dla pola *Imię_Prac* został ustalony rozmiar n=10, to pomiędzy wartością '*Roman*' a wartością '*Wójcik*' będzie odstęp równy 10 spacji, gdyż długość wartości '*Roman*' jest równa 5 znakom.

4.2.3.4. Funkcje daty i czasu

Funkcje tej grupy mogą być zastosowane w liście kolumn, we frazie `WHERE` lub w innych wyrażeniach. Za pomocą tych funkcji można operować na danych typu `DATE`, `TIME`, `TIMESTAMP` (w niektórych implementacjach SQL – typu `DATETIME`). Warto podkreślić, że wymienione wartości w niektórych dialektach należy umieszczać jako parametry w pojedynczych lub podwójnych cudzysłowach.

Tabela 4.19 pokazuje niektóre z analizowanych funkcji.

Tabela 4.19. Funkcje daty i czasu

Funkcja	*Wynik*
`ADD MONTHS {(<wyrażenie>, number)}`	Funkcja dodaje podaną liczbę miesięcy `number` do wartości wyrażenia *`wyrażenie`* typu `date` i w wyniku zwraca otrzymaną datę
`DATEADD{(datepart, number, <wyrażenie>)}`	Funkcja dodaje ilość `number` jednostek czasu `datepart` do wartości wyrażenia *`wyrażenie`* typu `date`[a] i w wyniku zwraca otrzymaną datę
`MONTHS BETWEEN{(date|<kolumna1>, date|<kolumna2>)}`	Zwraca różnicę w miesiącach pomiędzy wartościami pierwszego i drugiego parametru typu `date`
`DATEDIFF{(datepart, <date1>, <date2>)}`	Zwraca ilość jednostek `datepart` pomiędzy dwoma datami

[a] W MS SQL *Server*.

Przykład 4-25

Mamy tabelę *Etapy_Projektu* (tab. 4.20).

Tabela 4.20. Tabela *Etapy_Projektu*

Etap	*Pocz_Etapu*	*Koniec_Etapu*
1	01.01.05	15.02.05
2	16.02.05	05.03.05

Polecenie:

```
SELECT Etap, Pocz_Etapu, Koniec_Etapu, ADD MONTHS (Koniec_Etapu, 1)
Nowy_Term_Zak
FROM Etapy_Projektu;
```

zwraca następującą tabelę:

Etap	*Pocz_Etapu*	*Koniec_Etapu*	*Nowy_Term_Zak*
1	01.01.05	15.02.05	15.03.05
2	16.02.05	05.03.05	05.04.05

4.2.3.5. Funkcje konwersji danych

Ponieważ, jak widzimy, różne funkcje wymagają danych w określonym formacie lub typie danych, może zachodzić potrzeba konwersji jednego typu w drugi. Aby zrealizować taką modyfikację danych, można użyć odpowiednich funkcji. Ważniejsze z nich przedstawiono w tab. 4.21.

Tabela 4.21. Funkcje konwersji danych

Funkcja	*Wynik*
`TO_CHAR {(wartość| <kolumna>)}`	Konwertuje argument w ciąg znaków tekstowych
`CONVERT {(datatype [(length)], expression [, style])}`	Konwertuje dane typu `datatype` w `expression`[a]
`TO_NUMBER {(wartość| <kolumna>)}`	Konwertuje argument do typu liczbowego

[a] W MS SQL *Server*.

Przykład 4-26

Tabelą wejściową będzie już znana tabela *Liczby*.

A	*B*	*C*
1	5.5	- 0.1
3	3.4	4.5
4	7.8	9.8
10	- 1.2	- 11.0

Zdanie:

```
SELECT CONCAT (TO_CHAR (A,B,C)) Połączenie
FROM Liczby;
```

Na początku zmieni typ danych (z liczbowego na tekstowy) i połączy wartości tekstowe w jednym polu o nazwie *Połączenie*:

	Połączenie	
1	5.5	- 0.1
3	3.4	4.5
4	7.8	9.8
10	- 1.2	- 11.0

Przykład 4-27

Następujące zapytanie konwertuje bieżącą datę do łańcucha znaków o długości 8 i stylu danych ANSI:

```
SELECT CONVERT (CHAR (8), GETDATE, 2)
```

Wynik:

………

16.09.07

4.2.3.6. Funkcje systemowe

Przypomnijmy, że w niektórych dialektach zastosowanie frazy `FROM` nie jest obowiązkowe (patrz podrozdz. 4.2.1). Taka osobliwość występuje w przypadku funkcji systemowych, spośród których ważniejsze zostały podane w tab. 4.22.

Tabela 4.22. Funkcje systemowe

Funkcja	*Wynik*
`SYSDATE`	Zwraca bieżącą wartość daty systemowej
`USERNAME` lub `USER` `SUSER_NAME ([‘server/id’])`	Zwraca nazwę użytkownika Zwraca nazwę użytkownika (serwera)[a]
`OBJECT_ID (‘object_name’)`	Zwraca ID obiektu BD
`COL_NAME (table_id, column_id)`	Zwraca nazwę kolumny tabeli
`DB_NAME ([database_id])`	Zwraca nazwę BD

[a] W MS SQL *Server*.

Przykład 4-28

Zapytanie do bazy danych *Interbase/Firebird*:

```
SELECT USER
FROM RDB$DATABASE;
```

zwraca nazwę bieżącego użytkownika.

Przykład 4-29

Zapytanie do bazy danych MS SQL *Server* używa dwóch funkcji systemowych, aby uzyskać nazwę pierwszej kolumny tabeli *Pracownicy*:

```
SELECT COL_NAME (OBJECT_ID (‘Pracownicy’), 1);
```

Tabela *Pracownicy*

Imię_Prac	*Nazwisko_Prac*
Roman	Wójcik
Maria	Nowak
Krzysztof	Kowalski
Hanna	Malinowska

Polecenie zwraca wynik:

Imię_Prac

Zwróćmy uwagę, że ostatnie polecenie nie używa frazy `FROM`.

Zadania do samokontroli

1. Wyjaśnić zastosowanie następujących funkcji:
   - agregujących;
   - matematycznych;
   - znakowych;
   - daty i czasu;
   - konwertujących;
   - systemowych.
2. Dla tabeli *Wypożyczenia*:

Tabela *Wypożyczenia*

Id_Kasety	***Id_Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*	*Tytuł*	*Reżyser*	*Imię_Prac*	*Nazwisko_Prac*
1	5	05.02.2006	15	3	„Matrix”	Wachowscy	Roman	Wójcik
3	5	05.04.2005	15	2	„Obcy”	Scott	Roman	Wójcik
1	8	12.03.2006	3	5	„Matrix”	Wachowscy	Maria	Nowak
8	2	14.04.2006	2	1	„Predator”	McTiernan	Krzysztof	Kowalski
12	4	15.03.2006	15	1	„The ring”	Nakata	Roman	Wójcik
13	4	16.03.2005	18	1	„Taxi”	Pires	Hanna	Malinowska
4	14	18.04.2006	3	2	„Kiler”	Machulski	Maria	Nowak

napisać polecenia SQL do pobierania następujących danych:

- tytuły filmów do wypożyczenia;
- nazwiska i imiona pracowników, połączone w jednym polu (*Nazw_i_Imię*);
- średni, maksymalny i minimalny okres;
- nazwiska reżyserów, zaczynające się od litery ‘W’ lub ‘S’;
- liczbę wypożyczonych filmów Scotta;

- tytuły filmów wypożyczonych w marcu 2006 r.;
- numery tekstowych pól tabeli.

Podać wyniki realizacji poleceń.

3. Co będzie wynikiem następujących zapytań? Jeżeli polecenie nie zadziała, wyjaśnić przyczynę:

- ```
  SELECT COUNT(Nazwisko_Prac), Nazwisko_Prac
  FROM  Pracownicy;
  ```
- ```
  SELECT COUNT(Nazwisko_Prac) Nazwisko_Prac
  FROM  Pracownicy;
  ```
- ```
  SELECT COUNT(Nazwisko_Prac)
  FROM  Pracownicy;
  ```
- ```
  SELECT COUNT(Nazwisko_Prac) AND COUNT (Imię_Prac)
  FROM  Pracownicy;
  ```
- ```
  SELECT COUNT(*)
  FROM Pracownicy
  WHERE Nazwisko_Prac LIKE '%ó%' AND Imię_Prac LIKE '%o%';
  ```
- ```
  SELECT *
  FROM Pracownicy
  WHERE Nazwisko_Prac >= 'Nowak';
  ```
- ```
  SELECT COUNT(DISTINCT(Reżyser))
  FROM  Pracownicy;
  ```

4.2.4. Pomocnicze frazy zdania SELECT

Oprócz trzech podstawowych fraz (SELECT, FROM i WHERE) całe zdanie SELECT (pobierające dane z jednej tabeli) posiada często i inne frazy. Warto w tym miejscu przypomnieć ogólną postać pełnego polecenia SELECT, podanego wcześniej w listingu 4.1.

```
SELECT [DISTINCT | ALL] {*|[<lista pól>]}
FROM {<lista tabel>}
[WHERE <predykat-warunek wyboru albo połączenia>]
[GROUP BY <lista pól wyniku>]
[HAVING <predykat-warunek dla grupy>]
[ORDER BY <lista pól, wg których należy posortować pobrane dane
  [ASC|DESC] >];
```

Kolejność tych sześciu (lub mniejszej liczby) fraz jest stała.

4.2.4.1. Fraza GROUP BY

W najprostszej sytuacji fraza FROM posiada tylko jedną nazwę tabeli, której rekordy można podzielić na grupy ze względu na zawarte w nich wartości. Jeżeli zdanie SELECT posiada funkcje agregujące, to zastosowanie frazy GROUP BY sprawi, że te funkcje będą wykorzystane tylko w odniesieniu do każdej ze wspomnianych wcześniej grup (wewnątrz grupy).
```

Fraza `GROUP BY` grupuje dane względem wartości podanego pola, które spełniają warunki zawarte we frazie `WHERE` i zostają zwrócone w postaci pojedynczych wierszy.

Załóżmy, że mamy tabelę *Studenci* (tab. 4.23).

Tabela 4.23. Tabela *Studenci*

Nazwisko	*Imię*	*Wydział*	*Rok_Stud*	*Stypendium*
Rombalski	Krzysztof	Matematyczno-Przyrodniczy	2	300
Kalinowski	Rafał	Informatyczny	1	1000
Michalska	Maria	Informatyczny	1	300
Jezierska	Ewa	Informatyczny	3	200
Bednarz	Edyta	Historyczny	5	800
Kalinowski	Marek	Matematyczno-Przyrodniczy	5	600
Struś	Piotr	Filozoficzny	1	500

Rozważmy przykład.

Przykład 4-30

Zapytanie:

```
SELECT SUM (Stypendium)
FROM Studenci;
```

jak wiadomo, zwraca pojedynczą wartość (pojedynczy wiersz) równą 3700. Jest ona sumą stypendiów wypłacanych wszystkim studentom wpisanym do tej tabeli.

Aby obliczyć sumy wypłacane studentom każdego wydziału, należy wykorzystać frazę `GROUP BY`:

```
SELECT Wydział, SUM (Stypendium)
FROM Studenci
GROUP BY Wydział;
```

Wynik może wyglądać następująco:

Wydział	*SUM (Stypendium)*
Matematyczno-Przyrodniczy	900
Informatyczny	1500
Historyczny	800
Filozoficzny	500

Lista frazy `GROUP BY` może się składać z więcej niż jednego elementu.

Przykład 4-31

Polecenie:

```
SELECT Wydział, SUM (Stypendium), Rok_Stud
FROM Studenci
GROUP BY Wydział, Rok_Stud;
```

daje wynik:

Wydział	*SUM (Stypendium)*	*Rok_Stud*
Matematyczno-Przyrodniczy	300	2
Matematyczno-Przyrodniczy	600	5
Informatyczny	1300	1
Informatyczny	200	3
Historyczny	800	5
Filozoficzny	500	1

Należy pamiętać o następującym fakcie.

We frazie `SELECT` zdania z frazą `GROUP BY` oprócz funkcji agregujących dopuszczalne są tylko nazwy tych pól, które zostały wymienione we frazie `GROUP BY`.

Analizowana fraza może być wykorzystywana także w zapytaniach, które obejmują jednocześnie kilka tabel.

4.2.4.2. Fraza HAVING

Zakładamy, że polecenie z Prz. 4-31 należy zmodyfikować w taki sposób, aby zostali wzięci pod uwagę tylko studenci pierwszego i drugiego roku. W takim wypadku należy wykorzystać frazę `HAVING`. Wtedy wspomniany przykład można zapisać następująco.

Przykład 4-32

Zmodyfikowane polecenie:

```
SELECT Wydział, SUM (Stypendium), Rok_Stud
FROM Studenci
GROUP BY Wydział, Rok_Stud
HAVING Rok_Stud <= 2;
```

daje wynik:

Wydział	*SUM (Stypendium)*	*Rok_Stud*
Matematyczno-Przyrodniczy	300	2
Informatyczny	1300	1
Filozoficzny	500	1

1. **Fraza `HAVING` daje ten sam efekt we frazie `GROUP BY`, `co` fraza `WHERE` w poleceniu `SELECT`.**
2. **Fraza `HAVING` może być zastosowana tylko w zdaniu z frazą `GROUP BY`.**

4.2.4.3. Fraza ORDER BY

Wyniki pobierania danych można posortować za pomocą frazy `ORDER BY` polecenia `SELECT`.

Podstawowa składnia polecenia z wykorzystaniem tej frazy jest następująca (listing 4.2):

```
SELECT [DISTINCT | ALL] {*|[<lista pól>]}
FROM {<lista tabel>}
ORDER BY {<lista pól, wg których należy posortować pobrane dane
  [ASC|DESC] >};
```

Listing 4.2

Lista we frazie `ORDER BY` może zawierać dowolną liczbę kolumn (ale np. w serwerze MySQL nie może przekroczyć 900 bajtów).

Prz. 4-33 zwraca tabelę *Studenci* (tab. 4.23), w której rekordy zostały posortowane według pola *Nazwisko.*

Przykład 4-33

Polecenie:

```
SELECT *
FROM Studenci
ORDER BY Nazwisko;
```

Wynikowa tabela:

Nazwisko	*Imię*	*Wydział*	*Rok_Stud*	*Stypendium*
Bednarz	Edyta	Historyczny	5	800
Jezierska	Ewa	Informatyczny	3	200
Kalinowski	Rafał	Informatyczny	1	1000
Kalinowski	Marek	Matematyczno-Przyrodniczy	5	600
Michalska	Maria	Informatyczny	1	300
Rombalski	Krzysztof	Matematyczno-Przyrodniczy	2	300
Struś	Piotr	Filozoficzny	1	500

Wynik będzie taki sam, jeżeli ostatnia fraza będzie wyglądała następująco:

```
...
ORDER BY Nazwisko ASC;
```

Słowo kluczowe `ASC` jest (ang. *ascending*, rosnący) domyślną częścią frazy. Natomiast zastosowanie słowa `DESC` (ang. *descending*, malejący) daje w wyniku kolejność odwrotną: będą one posortowane malejąco.

Następujący przykład ilustruje sortowanie rekordów względem dwóch pól.

Przykład 4-34

Sortowanie według wartości pola *Nazwisko* ma większy priorytet i wykonywane jest rosnąco, z kolei sortowanie według wartości pola *Imię* ma mniejszy priorytet i wykonywane jest malejąco:

```
SELECT Nazwisko, Imię
FROM Studenci
ORDER BY Nazwisko, Imię DESC;
```

co daje tabelę wynikową:

Nazwisko	*Imię*
Bednarz	Edyta
Jezierska	Ewa
Kalinowski	Rafał
Kalinowski	Marek
Michalska	Maria
Rombalski	Krzysztof
Struś	Piotr

Jak widzimy, wynik (w wybranych polach) się nie zmienił (w stosunku do poprzedniego), bo powtarzającym się wartościom w polu *Nazwisko* ('Kalinowski') odpowiadają wartości w polu *Imię*, ale tym razem posortowane malejąco ('Rafał' > 'Marek').

Jeżeli sortowanie odbywa się względem pola zawierającego wartości NULL i wybrana jest kolejność ASC, wiersze zawierające wartości NULL wyświetlane będą na początku.

We frazie ORDER BY można również używać zamiast nazw pól odpowiadające polom numery. Na tej zasadzie polecenie z Prz. 4-34 możemy zapisać w następujący sposób:

```
SELECT Nazwisko, Imię
FROM Studenci
ORDER BY 1, 2 DESC;
```

I na koniec przedstawimy przykład, który ilustruje zastosowanie trzech przeanalizowanych w tym podrozdziale fraz.

Przykład 4-35

Zapytanie:

```
SELECT Wydział, SUM (Stypendium), Rok_Stud
FROM Studenci
GROUP BY Wydział, Rok_Stud
HAVING SUM (Stypendium) > 500
ORDER BY Wydział DESC;
```

daje w wyniku tabelę:

Wydział	*SUM (Stypendium)*	*Rok_Stud*
Matematyczno-Przyrodniczy	600	5
Historyczny	800	5
Informatyczny	1300	1

4.2.4.4. Fraza STARTING WITH

Frazę `STARTING WITH` najczęściej wykorzystuje się jako część frazy `WHERE` i działa prawie tak samo jak operator `LIKE`.

Przeanalizujemy zastosowanie frazy `STARTING WITH` na podstawie poprzednio podanych przykładów (4-10).

Niech tabelą wejściową będzie tabela *Towary_1* (tab. 4.10 z Prz. 4-10).

Tabela *Towary_1*

Id_Towaru	*Nazwa_Towaru*	*Cena_Towaru*	*Ilość_Towaru*
1	AAA	20.19	50
2	Vvaaa	19.96	5
4	Cac	20.03	25
5	app	10.00	100
6	AAa	2.12	480

Przykład 4-36

Przypomnijmy, że polecenie:

```
SELECT Nazwa_Towaru
FROM  Towary_1
WHERE Nazwa_Towaru LIKE '%a%';
```

zwraca wartości:

Nazwa_Towaru
Vvaaa
Cac
app
AAa

Przede wszystkim podkreślmy, że *we frazie* STARTING WITH *nie stosujemy masek* (znaczników postaci: '%' i '_').

Po wykonaniu polecenia:

```
SELECT Nazwa_Towaru
FROM  Towary_1
WHERE Nazwa_Towaru STARTING WITH ('a');
```

mamy wynik:

Nazwa_Towaru
app

Fraza `STARTING WITH` może być zastosowana razem z operatorem `LIKE`. Spójrzmy na kolejny przykład.

Przykład 4-37

Zapytanie:

```
SELECT Nazwa_Towaru
FROM  Towary_1
WHERE Nazwa_Towaru LIKE '%a%'OR Nazwa_Towaru STARTING WITH ('AA');
```

daje nową tabelę:

Nazwa_Towaru
AAA
Vvaaa
Cac
app
AAa

Zadania do samokontroli

1. Wyjaśnić przeznaczenie fraz:
   - `STARTING WITH;`
   - `GROUP BY;`
   - `ORDER BY;`
   - `HAVING.`
2. Jaki operator działa podobnie jak fraza `STARTING WITH`? Podać przykłady.
3. Jakie wyniki zwracają następujące polecenia (o ile są one poprawne):
   - ```
     SELECT Wydział, SUM (Stypendium), Rok_Stud
     FROM Studenci
     HAVING SUM (Stypendium) > 500;
     ```
   - ```
     SELECT Wydział, SUM (Stypendium), Rok_Stud
     FROM Studenci
     GROUP BY Wydział, Rok_Stud
     ORDER BY Wydział DESC
     HAVING SUM (Stypendium) > 500;
     ```
4. Dla tabeli *Wypożyczenia* napisać polecenia SQL do pobierania i umieszczenia w jednej tabeli wynikowej następujących danych:
   - średni (również minimalny i maksymalny) okres wypożyczenia zrealizowany przez pracowników;

Tabela *Wypożyczenia*

Id_Kasety	***Id_Klienta***	***Data_Wypoż***	*Id_Prac*	*Okres*	*Imię_Prac*	*Nazwisko_Prac*
3	5	05.04.2005	15	2	Roman	Wójcik
3	5	05.04.2005	15	2	Roman	Wójcik
1	8	12.03.2006	3	5	Maria	Nowak
8	2	14.04.2006	2	1	Krzysztof	Kowalski
12	4	15.03.2006	15	1	Roman	Wójcik
13	4	16.03.2005	18	1	Hanna	Malinowska
4	14	18.04.2006	3	2	Maria	Nowak

- nazwiska pracowników rozmieszczone w kolejności zmniejszenia (zwiększenia) liczby wykonanych wypożyczeń;
- rekordy tabeli posortowane według *Daty_Wypoż*;
- średni (również minimalny i maksymalny) okres wypożyczenia zrealizowany przez pracowników, pod warunkiem że wypożyczenie zostało realizowane od początku 2006 r.;
- liczbę wypożyczeń realizowanych przez każdego pracownika każdemu klientowi;
- liczbę wypożyczeń realizowanych przez pracownika, którego nazwisko zaczyna się od ‘N’ lub ‘M’ lub imię posiada literę ‘a’.

5. Co będzie wynikiem działania następujących zapytań (o ile są one poprawne)?:

- ```
  SELECT Imię_Prac Imię, Nazwisko_Prac Nazwisko
  FROM Wypożyczenia
  ORDER BY 1, 2 DESC;
  ```
- ```
  SELECT *
  FROM Wypożyczenia
  ORDER BY Data_Wypoż DESC
  GROUP BY Imię_Prac
  HAVING Okres > 1;
  ```
- ```
  SELECT avg (Okres), Nazwisko_Prac
  FROM Wypożyczenia
  ORDER BY Data_Wypoż DESC
  GROUP BY Imię_Prac
  HAVING Okres > 1;
  ```
- ```
  SELECT avg (Okres), Nazwisko_Prac, Imię_Prac
  FROM Wypożyczenia
  GROUP BY Nazwisko_Prac
  HAVING Okres > 0
  ORDER BY Data_Wypoż DESC;
  ```
- ```
  SELECT Nazwisko_Prac, Imię_Prac
  FROM Wypożyczenia
  WHERE  Nazwisko_Prac STARTING WITH (‘N’);
  ```
- ```
  SELECT SUM (Id_Prac), Nazwisko_Prac
  FROM Wypożyczenia;
  ```

4.3. Inne operacje manipulowania danymi

4.3.1. Zapytania do kilku tabel

Znaczna część algebry relacyjnej w dużym stopniu związana jest z możliwościami połączenia kilku relacji (tabel) za pomocą operacji *sumy* (ang. *union*), *przecięcia* (ang. *intersect*), *różnicy* (ang. *difference*), *iloczynu kartezjańskiego* (ang. *cartesian product*), *złączenia* (ang. *join*), opisanych w podrozdz. 2.2.2.

Operacje *sumy*, *przecięcia* i *różnicy* realizowane są bezpośrednio w języku SQL (patrz podrozdz. 4.2.2.6). Teraz skupimy się na analizie podstaw realizacji *iloczynu kartezjańskiego* i *złączenia* za pomocą konstrukcji `SELECT-FROM-WHERE`.

Standard SQL umożliwia w prosty sposób realizację, w jednym zdaniu, zapytania odnoszącego się do kilku tabel jednocześnie. Wystarczy wymienić nazwy tabel we frazie `FROM`, a we frazach `SELECT` i `WHERE` padać nazwy pól.

Załóżmy, że mamy dwie tabele: *Towary* {*Id_Tow*, *Id_Produc*, *Nazwa_Tow*, *Cena_Tow*} (tab. 4.24) i *Producenci* {*Nazwa_Prod*, *Id_Prod*, *Adres_Prod*}(tab. 4.25).

Tabela 4.24. Tabela *Towary*

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
54	B1	Chleb	1.50
24	B7	Cukierki	10.00
10	B1	Mąka	2.20
11	B2	Mąka	2.30
76	B5	Cukier	3.50
25	B8	Cukierki	15.50
23	B7	Cukierki	19.20

Tabela 4.25. Tabela *Producenci*

Nazwa_Prod	*Id_Prod*	*Adres_Prod*
Laa	B1	Lublin
Waa	B2	Warszawa
Wab	B3	Warszawa
Lab	B4	Lublin
Baa	B6	Bydgoszcz
Bbb	B7	Bydgoszcz
Taa	B8	Toruń
Lbb	B5	Lublin

Rozważmy przykład.

Przykład 4-38

Załóżmy, że chcemy znaleźć nazwę producenta towaru o cenie 15.50. Nazwy producentów zawarte są w tabeli *Producent.* Natomiast druga część predykatu (informacja o cenie) znajduje się w tabeli (*Towary*).

Najprostsze polecenie SQL, które pomoże nam znaleźć niezbędne dane, wygląda następująco:

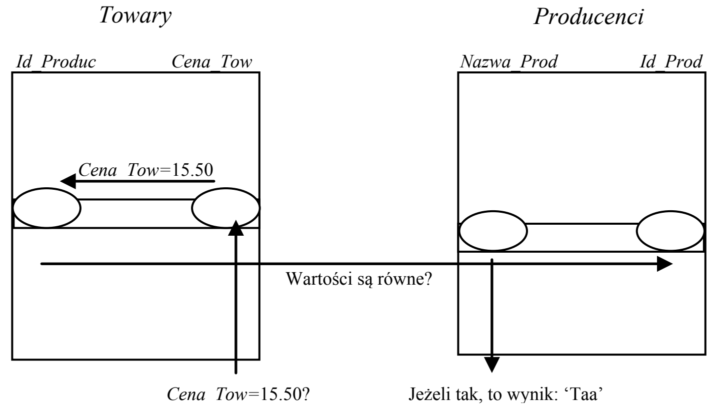
```
SELECT Nazwa_Prod
FROM Towary, Producenci
WHERE Cena_Tow = 15.50 AND Id_Produc = Id_Prod;
```

Listing 4.3

Schematycznie procedura opracowania tego zapytania wygląda tak, jak pokazano na rys. 4.1.

Jeżeli w ostatnim poleceniu SQL nie będzie frazy `WHERE`, to otrzymamy konstrukcję polecenia realizującego *iloczyn kartezjański.*

Może zaistnieć potrzeba połączenia w jednym zapytaniu kilku tabel z powtarzającymi się nazwami pól. W takich sytuacjach powinna być możliwość, pozwalająca jednoznacznie identyfikować każde pole. W SQL takim środkiem jest konstrukcja '*pełnej nazwy pola'*, posiadająca jako **prefiks** nazwę tabeli i symbol '.'. Przykładowo, wzór ***R.A*** oznacza pole ***A*** tabeli ***R***.



Rys. 4.1. Schemat kolejności operacji realizujących zapytanie kierowane do dwóch tabel

Do skrócenia zapisu nazw tabel stosujemy tzw. prefiks (*alias*). Składa się on najczęściej z części nazwy tabeli (a nawet tylko z pierwszej litery). Prefiks podajemy po nazwie odpowiedniej tabeli we frazie `FROM`.

Przykład 4-39

Zakładamy, że w tabelach z poprzedniego przykładu identyfikatory producentów będą oznaczone tak samo: *Id_Produc.* W takim wypadku listing 4.3 może wyglądać następująco:

```
SELECT Producenci.Nazwa_Produc
FROM Towary, Producenci
WHERE Towary.Cena_Tow = 15.50
AND Towary.Id_Produc =  Producenci.Id_Produc;
```

lub tak:

```
SELECT P.Nazwa_Produc
FROM Towary T, Producenci P
WHERE T.Cena_Tow = 15.50
AND T.Id_Produc =  P.Id_Produc;
```

Bardziej szczegółowo operacje *join* (różnego typu) w SQL przeanalizujemy poniżej.

4.3.2. Podzapytania w SQL

Standard języka umożliwia umieszczenie zdania `SELECT` w ciele innego zdania `SELECT`. W takiej sytuacji możemy rozpatrywać zapytania *zewnętrzne* i *wewnętrzne*. Ogólna struktura najprostszego polecenia z dwoma słowami `SELECT` ma postać:

- `SELECT [DISTINCT | ALL] {*|[<lista pól>]}`
- `FROM {niezbędne dane}`
- `WHERE {niezbędne dane}`
- `{operator(y) porównania}`
- `(SELECT [DISTINCT | ALL] {*|[<lista pól>]}`
- `FROM {niezbędne dane}`
- `[WHERE {niezbędne dane}]);`

Listing 4.4

W ostatnim listingu wiersze 1-3 posiadają tekst zwykłego zapytania, które nazywa się *zewnętrznym*, wiersze 5-7 – tekst zapytania *wewnętrznego* (umieszczanego w nawiasach). W ten sposób*, zapytanie wewnętrzne generuje wartości, które porównuje się w predykacie z odpowiednimi wartościami zapytania zewnętrznego.*

Wewnętrzne zapytanie nazywa się *podzapytaniem*. *Predykat* występuje we frazie `WHERE` lub we frazie `HAVING`.

Wyróżniamy kilka typów podzapytań:

- **podzapytanie skalarne**: zwraca jedną wartość; podzapytanie może być umieszczone w dowolnym miejscu frazy `WHERE`, gdzie można podać nazwę pola lub wyrażenie stałe;
- **podzapytanie tablicowe**: zwraca wartość tablicową; wartość ta może być wykorzystana we frazie `WHERE`;
- **podzapytanie tabelaryczne**: zwraca wartość w formie tabeli; podzapytanie takie może być użyte we frazie `FROM` zapytania głównego.

Realizacja zapytania z podzapytaniem wymaga spełnienia kilku ważnych reguł.

1. **W podzapytaniach nie powinno być frazy `ORDER BY`.**
2. **Lista we frazie może posiadać tylko nazwy pól i złożone z nich wyrażenia.**
3. **Domyślnie nazwy pól w podzapytaniu są polami tabeli wymienionej we frazie `FROM`.**
4. **Podzapytanie można definiować bezpośrednio po operatorze porównania (=, <, <=, >, >=, <>).**

Do wykonania dalszej analizy należy przedstawić niektóre ważne elementy, dotyczące wcześniej (podrozdz. 4.2.2.5) wspomnianych operatorów `ANY` i `ALL`.

Warunek (***s*** > `ALL` ***R***) jest równy TRUE wtedy i tylko wtedy, kiedy wartość ***s*** jest większa od wszystkich wartości w ***R***. Zamiast operatora > może być użyty dowolny inny operator porównania; oczywiście odpowiednio zmieni się sens warunku. Warunki (***s*** <> `ALL` ***R***) i (***s*** `NOT IN` ***R***) są równoznaczne.

Warunek (***s*** > `ANY` ***R***) jest równy TRUE wtedy i tylko wtedy, kiedy wartość ***s*** jest większa od przynajmniej jednej wartości w ***R***. Zamiast operatora > może być wpisany dowolny inny operator porównania; oczywiście odpowiednio zmieni się sens warunku. Warunki (***s*** = `ANY` ***R***) i (***s*** `IN` ***R***) są równoznaczne.

Przeanalizujemy zapytania z każdym z wymienionych typów podzapytań.

Podzapytanie skalarne

Załóżmy, że mamy pobrać z tabeli *Wypożyczenia_1* (tab. 4.26) dane o pracownikach, których operacje wypożyczenia przewyższają średni koszt wypożyczeń.

Tabela 4.26. Tabela *Wypożyczenia_1*

Id_Kasety	*Id_Klienta*	*Data_Wypoż*	*Id_Prac*	*Okres*	*Koszt_Wypoż*	*Nazwisko_Prac*
3	5	05.04.2005	15	2	10.00	Wójcik
1	8	12.03.2006	3	5	20.00	Nowak
8	2	14.04.2006	2	1	4.50	Kowalski
13	4	16.03.2005	18	1	5.50	Malinowska

Ponieważ standard języka nie pozwala na wykorzystanie funkcji agregujących we frazie `WHERE`, musimy wartości w polu *Koszt_Wypoż* porównać z wynikiem podzapytania, które zwraca średni koszt wypożyczenia (równy 10.00).

Przykład 4-38

Polecenie:

```
SELECT Id_Prac, Koszt_Wypoż, Nazwisko_Prac
FROM Wypożyczenia_1
WHERE  Koszt_Wypoż >
(SELECT AVG (Koszt_Wypoż)
FROM Wypożyczenia_1);
```

daje wynik:

Id_Klienta	*Koszt_Wypoż*	*Nazwisko_Prac*
1	20.00	Nowak

Ostatni przykład ilustruje zastosowanie funkcji agregujących w podzapytaniach.

Podzapytanie tablicowe

W bardziej skomplikowanych (od poprzedniego) wypadkach zapytanie może wykorzystywać wiele wartości zwróconych przez podzapytanie. Przeanalizujmy następujący przykład.

Przykład 4-39

Z tabeli *Towary* mamy pobrać informacje o towarach, których cena jest większa od ceny chociażby jednego towaru o nazwie '*Mąka*'.

Tabela *Towary*

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
54	B1	Chleb	1.50
24	B7	Cukierki	10.00
10	B1	Mąka	2.20

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
11	B2	Mąka	2.30
76	B5	Cukier	3.50
25	B8	Cukierki	15.50
23	B7	Cukierki	19.20

Niezbędne dane możemy pobrać, stosując operator ANY (można także i w inny sposób):

```
SELECT Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow
FROM Towary
WHERE  Cena_Tow > ANY
(SELECT Cena_Tow
FROM Towary
WHERE Nazwa_Tow = 'Mąka');
```

Wynik zapytania:

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
24	B7	Cukierki	10.00
11	B2	Mąka	2.30
76	B5	Cukier	3.50
25	B8	Cukierki	15.50
23	B7	Cukierki	19.20

Teraz zmodyfikujemy warunek: mamy pobrać informacje o towarach, których cena jest większa od ceny dowolnego towaru o nazwie '*Mąka*'.

Dane możemy uzyskać, realizując polecenie:

```
SELECT Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow
FROM Towary
WHERE  Cena_Tow > ALL
(SELECT Cena_Tow
FROM Towary
WHERE Nazwa_Tow = 'Mąka');
```

Wynikowa tabela wygląda następująco:

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
24	B7	Cukierki	10.00
76	B5	Cukier	3.50
25	B8	Cukierki	15.50
23	B7	Cukierki	19.20

Podzapytanie tabelaryczne

Rozpatrzmy przykład.

Przykład 4-40
Załóżmy, że mamy trzy tabele: *Filmy* {*Id_Film, Id_Reż, Nazwa_Film*}, *Reżyserzy* {*Id_Reż, Nazwisko_Reż, Adres_Reż*} i tabelę *Aktorzy* {*Id_Aktor, Nazwisko_Aktor, Nazwa_Film*}.

Pierwsza z tych tabel posiada informacje o filmach, druga – o reżyserach tych filmów, trzecia – o aktorach grających w filmach. Zakładamy, że w tych tabelach nie ma powtarzających się nazw filmów i że między tabelami są zdefiniowane następujące typy relacji (być może nawet sztuczne):

Filmy ← → *Reżyserzy*: **1:wielu**;

Filmy ← → *Aktorzy*: **wiele:wielu**;

Reżyserzy ← → *Aktorzy*: **wiele:wielu**.

Zakładamy, że mamy odszukać dane o reżyserach filmów, w których wystąpiła aktorka Beata Tyszkiewicz. Listing (4.5) odpowiedniego zapytania może wyglądać następująco:

```
1) SELECT Reżyserzy.Nazwisko_Reż
2) FROM Reżyserzy, (SELECT Filmy.Id_Reż
3)                  FROM Filmy, Aktorzy
4)                  WHERE Filmy.Nazwa_Film = Aktorzy.Nazwa_Film AND
5)                  Aktorzy.Nazwisko_Aktor = 'Beata Tyszkiewicz'
6)                  ) Tab_Pom
7) WHERE Reżyserzy.Id_Reż = Tab_Pom.Id_Reż;
```

Listing 4.5

Jak widać, wiersze 2-7 tego listingu posiadają opis frazy `FROM` innego (zewnętrznego) zapytania. Lista tej frazy posiada nazwę tabeli *Reżyserzy*, jak również tekst podzapytania (w nawiasach). Podzapytanie łączy tabele *Filmy* i *Aktorzy* poprzez porównanie wartości ich pól (wierszy 3-4), określa nazwisko aktora (wiersz 5) i zwraca tabelę z identyfikatorami reżyserów. Tabela ta ma przypisaną nazwę *Tab_Pom*. Całe zapytanie zwraca zbiór wartości pola *Nazwisko_Reż* tabeli *Reżyserzy*, którym odpowiada wartość pola tabeli *Tab_Pom*.

4.3.3. Złączenie tabel w SQL

Najprostszą formą zdania SQL, które realizuje operację łączenia danych z dwóch tabel, jest *złączenie krzyżowe* (ang. *cross join*) i jest ono odpowiednikiem *iloczynu kartezjańskiego* (podrozdz. 2.2.2.4). Ten typ łączenia w podstawowej formie wykorzystuje się dość rzadko. Przykładowo: realizacja tej operacji nad tabelami *Towary* i *Producenci* (patrz Prz. 4-38) daje nową tabelę (np. *Towary_Producenci*), składającą się z 7 pól (4+3) i 56 wierszy (7*8).

4.3.3.1. Θ-złączenie i naturalne złączenie tabel

Dużo szersze zastosowanie ma operacja **Θ-**_złączenia tabel_ (podrozdz. 2.2.2.7).

Przypomnijmy, że wynikiem operacji **Θ-**_złączenia_ na relacjach (tabelach) ***S*** i ***R*** będzie relacja (tabela) ***RS***, zawierająca krotki (rekordy) _iloczynu kartezjańskiego_ ***R*** i ***S*** , spełniające warunek ***F*** (predykat ***F***).

W praktyce **Θ** jest operatorem porównania.

Przykład 4-41

Zdanie umożliwiające pobieranie potrzebnych danych z tabel _Towary_ i _Producenci_ pod warunkiem równości wartości w polach _Id_Prod_ i _Id_Prod_ (predykat _F_):

```
SELECT Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow, Nazwa_Prod, Id_Prod
FROM Towary, Producenci
WHERE  Id_Produc = Id_Prod;
```

daje tabelę wynikową:

_Id_Tow_	_Id_Produc_	_Nazwa_Tow_	_Cena_Tow_	_Nazwa_Prod_	_Id_Prod_
54	B1	Chleb	1.50	Laa	B1
24	B7	Cukierki	10.00	Bbb	B7
10	B1	Mąka	2.20	Laa	B1
11	B2	Mąka	2.30	Waa	B2
76	B5	Cukier	3.50	Lbb	B5

W tym przykładzie w tabelach nie ma powtarzających się nazw pól. Dlatego nie musimy stosować przed nazwą pola odpowiedniego _aliasu_ tabeli.

Tutaj operatorem porównania jest operator równości. Operator ten realizuje _równozłączenie tabel_. Jeżeli w takim zdaniu będzie zastosowany inny operator porównania, to będziemy mieli do czynienia z _nierównozłączeniem tabel_.

Operacja ***Θ-złączenia*** różni się od operacji _złączenia naturalnego_ (podrozdz. 2.2.2.7) tym, że w złączeniu naturalnym:

- porównujemy wartości pól o tych samych nazwach;
- jedno pole każdej pary pól o tej samej nazwie będzie usunięte z tabeli wynikowej.

Przykład 4-42

Załóżmy, że analizowana wyżej tabela _Towary_ ma ten sam nagłówek ({_Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow_}) i zawiera te same dane (Prz. 4-38). Mamy jeszcze jedną tabelę: _Zamówienia_ {_Id_Zam, Id_Tow, Ilość_Tow_} (tab. 4.27).

Tabela 4.27. Tabela _Zamówienia_

_Id_Zam_	_Id_Tow_	_Ilość_Tow_
Z1	54	10
Z1	23	1
Z2	23	2
Z3	10	10
Z3	54	20

Jak widać, w obu tabelach występują pola o tej samej nazwie (*Id_Tow*). Na tej podstawie możemy realizować *złączenie naturalne* (dodatkowo stosując *aliasy*):

```
SELECT T.Id_Tow, T.Id_Produc, T.Nazwa_Tow, T.Cena_Tow, Z.Id_Zam,
       Z.Ilość_Tow
FROM Towary T, Zamówienia Z
WHERE  T.Id_Tow = Z.Id_Tow;
```

Mamy więc następujący wynik:

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*	*Id_Zam*	*Ilość_Tow*
54	B1	Chleb	1.50	Z1	10
54	B1	Chleb	1.50	Z3	20
10	B1	Mąka	2.20	Z1	1
23	B7	Cukierki	19.20	Z1	1
23	B7	Cukierki	19.20	Z2	2

Możemy zmodyfikować ostatnie zapytanie w celu podliczenia kosztu sprzedanych towarów:

```
SELECT T.Id_Tow, T.Id_Produc, T.Nazwa_Tow, T.Cena_Tow, Z.Id_Zam,
       Z.Ilość_Tow, T.Cena_Tow*Z.Ilość_Tow Koszt
FROM Towary T, Zamówienia Z
WHERE  T.Id_Tow = Z.Id_Tow;
```

Do poprzedniej tabeli wynikowej będzie dodane jedno pole:

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*	*Id_Zam*	*Ilość_Tow*	*Koszt*
54	B1	Chleb	1.50	Z1	10	15.00
54	B1	Chleb	1.50	Z3	20	30.00
10	B1	Mąka	2.20	Z1	1	2.20
23	B7	Cukierki	19.20	Z1	1	19.20
23	B7	Cukierki	19.20	Z2	2	38.40

Jak widać, w Prz. 4-41 i 4-42 *operator porównania* (=) używa się we frazie `WHERE`.

Tabela wymieniona po lewej stronie znaku '=' nazywa się tabelą *zewnętrzną*, a po prawej stronie *wewnętrzną*. Ze względu na ich pozycję według znaku równości często używane są określenia lewa i prawa strona, a ten typ złączenia nazywa się lewa do prawej lub *złączenie lewostronne*. Jest to najczęściej używany typ złączenia tabel. Od strony formalnej jest to typ *wewnętrzny*.

W standardzie ISQL firmy Borland zdanie wewnętrznego złączenia ma inną postać i opiera się na konstrukcji:

R `JOIN` S `ON` C.

Konstrukcja ta polega na tym, iż na początku oblicza się *iloczyn kartezjański* (***R*** x ***S***), po czym stosuje się operację wyboru według kryterium ***C*** podanego po słowie kluczowym `ON`. Zgodnie z tym zdanie z Prz. 4-41 może być zapisane następująco:

```
SELECT Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow, Nazwa_Prod, Id_Prod
FROM Towary JOIN Producenci
ON  Id_Produc = Id_Prod;
```

W niektórych dialektach SQL złączenie wewnętrzne wymaga zastosowania dodatkowego słowa kluczowego: `INNER`. Wtedy podana wyżej konstrukcja wygląda następująco:

R INNER JOIN S ON C.

4.3.3.2. Złączenie zewnętrzne

Standard SQL2 rozszerzył pojęcie *złączenia warunkowego* (*wewnętrznego*). To rozszerzenie związane jest z tworzeniem tzw. *złączeń otwartych* lub *złączeń zewnętrznych*.

Jak już mówiliśmy wcześniej, *złączenie naturalne* polega na tym, że w tabeli wynikowej nie będą umieszczane rekordy niemające swoich 'odpowiedników' – rekordów z różnymi wartościami w polu o tej samej nazwie. Takie niedopasowane rekordy nazywają się *wiszące* (ang. *dangling*). *Złączenie zewnętrzne* pozwala na dopasowanie do tabeli wynikowej rekordów wiszących.

> ***Wiszące rekordy*** **łączą się z innymi rekordami drugiej tabeli, przyjmując w odpowiednich polach *wartości NULL*.**

Podstawową konstrukcję *złączenia zewnętrznego* można przedstawić następująco:

R OUTER JOIN S ON C.

Rozróżniamy trzy typy złączenia zewnętrznego:

- **lewostronne** (ang. *left outer join*) – wypełnia wartościami NULL wiszące rekordy tabeli ***R***; odpowiada mu poniższa konstrukcja:

  R* LEFT OUTER JOIN *S* ON *C;

- **prawostronne** (ang. *right outer join*) – wypełnia wartościami NULL wiszące rekordy tabeli ***S***; odpowiada mu poniższa konstrukcja:

  R* RIGHT OUTER JOIN *S* ON *C;

- **obustronne, pełne** (ang. *full outer join*) – wypełnia wartościami NULL wiszące rekordy zarówno tabeli ***R***, jak i tabeli ***S***; odpowiada mu poniższa konstrukcja:

  R* FULL OUTER JOIN *S* ON *C.

Technologie realizacji złączenia zewnętrznego przeanalizujemy na prostych przykładach.

Załóżmy, że mamy dwie tabele: *R* i *S*.

Tabela *R*

A	*B*	*C*
A1	B2	C3
A4	B5	C6
A7	B8	C9

Tabela *S*

B	*C*	*D*
B2	C3	D10
B2	C3	D11
B6	C7	D12

Następnie chcemy dokonać złączenia tych tabel według wspólnego pola *C* (na podstawie standardu SQL3).

Przykład 4-43

Realizacja złączenia lewostronnego:

```
SELECT R.A, R.B, R.C, S.D
FROM R
LEFT OUTER JOIN S
ON R.C = S.C;
```

Wynik:

R.A	*R.B*	*R.C*	*S.D*
A1	B2	C3	D10
A1	B2	C3	D11
A4	B5	C6	NULL
A7	B8	C9	NULL

Przykład 4-44

Realizacja złączenia prawostronnego:

```
SELECT R.A, R.B, R.C, S.D
FROM R
RIGHT OUTER JOIN S
ON R.C = S.C;
```

Wynik:

R.A	*R.B*	*R.C*	*S.D*
A1	B2	C3	D10
A1	B2	C3	D11
NULL	B6	C7	D12

Przykład 4-45

Realizacja pełnego złączenia:

```
SELECT R.A, R.B, R.C, S.D
FROM R
FULL OUTER JOIN S
ON R.C = S.C;
```

Wynik:

R.A	*R.B*	*R.C*	*S.D*
A1	B2	C3	D10
A1	B2	C3	D11
A4	B5	C6	NULL
A7	B8	C9	NULL
NULL	B6	C7	D12

Jak widzimy, typ złączenia w zdaniu SQL jest zależny wyłącznie od jednego słowa kluczowego: `LEFT` – dla lewostronnego, `RIGHT` – dla prawostronnego i `FULL` – dla pełnego.

W środowisku SQL *Server* ostatni wiersz każdego zdania może wyglądać następująco:

```
... JOIN S
USING (C);
```

Zadania do samokontroli

1. Wyjaśnić schemat realizacji operacji pobierania danych z dwóch tabel.
2. Zapisać zapytanie do tab. 4.25, posiadające podzapytanie skalarne na podstawie funkcji `AVG (Koszt_Wypoż)`. Co będzie wynikiem tego zapytania?
3. Co będzie wynikiem polecenia (patrz Prz. 4.39):

   ```
   SELECT Id_Tow, Nazwa_Tow
   FROM Towary
   WHERE  Cena_Tow > ANY
   (SELECT Cena_Tow
   FROM Towary
   WHERE Nazwa_Tow STARTING WITH ('Cu');
   ```

   Jaki typ podzapytania występuje w powyższym przykładzie?
4. Stworzyć tabele z Prz. 4.39 i wypełnić je danymi (o 8-10 rekordów). Podać wynik polecenia z listingu 4.5.
5. Podać przykłady równo- i nierównozłączenia tabel.
6. Jaki będzie wynik poleceń skierowanych do tabel *Towary* (Prz. 4.39) i *Zamówienia* (tab. 4.27)?

   - ```
     SELECT Z.Id_Zam, Z.Ilość_Tow, T.Id_Tow, T.Id_Produc,
          T.Nazwa_Tow, T.Cena_Tow
     FROM Zamówienia Z, Towary T
     WHERE  Z.Id_Tow = T.Id_Tow;
     ```
   - ```
     SELECT Z.Id_Zam, Z.Ilość_Tow, T.Id_Tow, T.Id_Produc,
          T.Nazwa_Tow, T.Cena_Tow
     FROM Zamówienia Z
     LEFT OUTER JOIN Towary T
     ON  Z.Id_Tow = T.Id_Tow;
     ```
   - ```
     SELECT Z.Id_Zam, Z.Ilość_Tow, T.Id_Tow, T.Id_Produc,
          T.Nazwa_Tow, T.Cena_Tow
     FROM Zamówienia Z
     RIGHT OUTER JOIN Towary T
     ON  Z.Id_Tow = T.Id_Tow;
     ```
   - ```
     SELECT T.Id_Produc, T.Nazwa_Tow, T.Cena_Tow,
          Z.Id_Zam, Z.Ilość_Tow, T.Id_Tow
     FROM Towary T
     ```

```
FULL OUTER JOIN Zamówienia Z
ON  T.Id_Tow = Z.Id_Tow;
```

- ```
  SELECT T.Id_Produc, T.Nazwa_Tow, T.Cena_Tow,
      Z.Id_Zam, Z.Ilość_Tow, T.Id_Tow
  FROM Towary T
  RIGHT INNER JOIN Zamówienia Z
  ON  T.Id_Tow = Z.Id_Tow;
  ```
- ```
  SELECT T.Id_Produc, T.Nazwa_Tow, T.Cena_Tow,
      Z.Id_Zam, Z.Ilość_Tow, T.Id_Tow
  FROM Towary T
  LEFT INNER JOIN Zamówienia Z
  ON  T.Id_Tow = Z.Id_Tow.
  ```

4.4. Modyfikacja danych w SQL

W poprzedniej części tego rozdziału omówione zostały operacje pobierania danych za pomocą polecenia `SELECT`.

Niniejszy podrozdział skupia się na analizie operacji modyfikacji danych w tabelach przy użyciu poleceń (zdań) `INSERT`, `UPDATE`, `DELETE`.

4.4.1. Zdanie INSERT

Istnieją dwa typy tego rodzaju zdania: `INSERT...VALUES` i `INSERT...SELECT`.

Za pomocą tych poleceń można wstawiać rekordy do tabeli.

Ogólna struktura polecenia `INSERT...VALUES` ma postać:

```
INSERT INTO {nazwa tabeli}[(<lista pól>)]
VALUES (<lista wartości>);
```

Listing 4.6

Zdanie na listingu 4.6 realizuje **wstawianie tylko jednego rekordu**. Podanie listy pól nie jest obowiązkowe, gdy lista wartości posiada dane, które mamy wpisać po kolei do wszystkich pól tabeli (w tej samej kolejności, tzn. w kolejności pól w nagłówku tabeli).

Przykład 4-46

Polecenie:

```
INSERT INTO Towary (Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow)
VALUES (37,'B10','Chleb',2.10);
```

jak też polecenie:

```
INSERT INTO Towary
VALUES (37,'B10','Chleb',2.10);
```

dają taki sam wynik: dodaje do tabeli *Towary* (z Prz. 4.39) jeden rekord. Po tej operacji wynikowa tabela będzie miała następujący wygląd (tab. 4.28):

Tabela 4.28. *Towary*

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
54	B1	Chleb	1.50
24	B7	Cukierki	10.00
10	B1	Mąka	2.20
11	B2	Mąka	2.30
76	B5	Cukier	3.50
25	B8	Cukierki	15.50
23	B7	Cukierki	19.20
37	B10	Chleb	2.10

Pomiędzy listą nazw pól i listą wartości powinny istnieć następujące zależności:

1. **Liczba elementów w obu listach powinna być taka sama (w przypadku używania drugiego z podanych w ostatnim przykładzie poleceń).**
2. **Kolejność elementów w obu listach powinna być taka sama.**
3. **Typy danych odpowiadających sobie elementów powinny być jednakowe i pochodzić z tej samej domeny.**

Zwróćmy uwagę, że wartości typu znakowego muszą być ujęte w cudzysłów. Wartości liczbowe nie ujmujemy w cudzysłów.

Kiedy aplikacje typu *Klient* łączą się z BD, aby dodać dane, używają omówionych powyżej poleceń.

Najczęściej mamy do czynienia z sytuacjami, gdy podczas dodawania nowego wiersza każde z pól posiada określoną wartość. Jeżeli jednak użytkownik nie określił tej wartości, SZBD musi 'podjąć jedną z dwóch decyzji': albo odrzucić polecenie, albo wpisać wartość domyślną. W drugim przypadku najczęściej pole dopuszcza wprowadzenie wartości NULL lub spełnia inne warunki.

Ogólna struktura polecenia `INSERT...SELECT` ma postać:

```
INSERT INTO {nazwa tabeli}[(<lista pól>)]
[AS] SELECT <lista wartości>
FROM {nazwa innej tabeli};
```

Listing 4.7

Zdanie na listingu 4.7 realizuje **wstawienie dowolnej liczby rekordów pochodzących z innej tabeli, perspektywy (widoku)**. Słowo `AS` wykorzystuje się w niektórych dialektach SQL.

Rozważmy przykład.

Przykład 4-47

Mamy tabelę *Skład* {*Id_Towar*, *Nazwa_Towar*}. Niech pierwsze z pól tabeli będzie typu liczbowego, drugie typu znakowego. Zakładamy również, że tabela ta nie posiada żadnych danych (jest tabelą pustą).

Wtedy polecenie:

```
INSERT INTO Skład
SELECT Id_Tow, Nazwa_Tow
FROM Towary;
```

prowadzi do następującego wyniku (polecenie skierowane do wynikowej tabeli z poprzedniego przykładu):

Tabela *Skład*

Id_Tow	*Nazwa_Tow*
54	Chleb
24	Cukierki
10	Mąka
11	Mąka
76	Cukier
25	Cukierki
23	Cukierki
37	Chleb

4.4.2. Zdanie UPDATE

Polecenie oparte na tym zdaniu pozwala zmienić wartości w istniejącym wierszu lub wierszach.

Zdanie UPDATE posiada następującą składnię:

```
UPDATE {nazwa tabeli}
SET
Pole1 = {wartość1|(operacja selekcji)|wzór1}
[,Pole2 = {wartość2|(operacja selekcji)|wzór2},
... ]
[WHERE {niezbędne dane}];
```

Listing 4.8

Klauzula SET określa pola, które mają być aktualizowane.

Przeanalizujmy działanie tego zdania na konkretnych przykładach.

Przykład 4-48

Niech na wejściu dana będzie tabela *Towary* (tab. 4.28). Następujące polecenie ustawia wartość w polu *Id_Produc* na wartość B1 (w każdym z wierszy tabeli):

```
UPDATE Towary
SET
ID_Produc = 'B1';
```

polecenie zaś:

```
UPDATE Towary
SET
ID_Produc = 'B1'
WHERE Nazwa_Tow = 'Chleb';
```

zamieni wartości pola *Id_Produc* (na wartość B1) tylko w tych wierszach, w których wartość pola *Nazwa_Tow* jest równa 'Chleb'.

Polecenie:

```
UPDATE Towary
SET
Cena_Tow = Cena_Tow * 1.01;
```

zwiększy wpisane ceny wszystkich towarów o jeden procent, natomiast polecenie:

```
UPDATE Towary
SET
Cena_Tow = Cena_Tow * 0.99
WHERE Nazwa_Tow = 'Chleb';
```

zmniejszy o jeden procent wartości w tym samym polu tylko dla jednego typu towaru.

Standardowe zdanie `UPDATE` umożliwia wprowadzenie zmian wierszy w pojedynczej tabeli.

Zmianie może ulec dowolna liczba wierszy, jak i dowolna liczba pól w wyselekcjonowanych wierszach.

Są przypadki, gdy żaden wiersz nie zostanie zmodyfikowany (zależy to oczywiście od frazy `WHERE`).

4.4.3. Zdanie DELETE

Składnia zdania `DELETE` ma postać:

```
DELETE
[FROM] {nazwa tabeli}
[WHERE {niezbędne dane}];
```

Listing 4.9

Frazy `FROM` i `WHERE` mogą być opcjonalne, jednak fraza `FROM` w standardzie ANSI jest obowiązkowa.

Przykład 4-49

Załóżmy, że wykonujemy operacje nad tą samą tabelą (tab. *Towary*).

Polecenie:

```
DELETE
FROM Towary;
```

jak też polecenie:

```
DELETE
FROM Towary
WHERE Id_Tow >=10;
```

usuną wszystkie rekordy z tabeli *Towary*. Ale nie oznacza to usunięcia samej tabeli. Wynikiem będzie po prostu pusta tabela.

Polecenie:

```
DELETE
FROM Towary
WHERE Id_Tow =10;
```

usuwa tylko jeden (trzeci) wiersz,
a zdanie:

```
DELETE
FROM Towary
WHERE Nazwa_Tow LIKE 'M%';
```

prowadzi do usunięcia dwóch rekordów.

Za pomocą zdania DELETE można usunąć jeden, część albo wszystkie rekordy z tabeli.

Zdanie DELETE nie usuwa samej tabeli (struktury tabeli) z katalogu systemowego. Wykonanie tej operacji można dokonać za pomocą polecenia DROP TABLE.

Zadania do samokontroli

1. Zapisać ogólną postać poleceń SQL, wstawiających dane do tabeli, modyfikujących dane w tabeli oraz usuwających dane z tabeli.
2. Na czym polega różnica w działaniu zdań `INSERT...VALUES` i `INSERT...SELECT`. Podać przykłady.
3. Czy poprawne są następujące polecenia:

- ```
  INSERT INTO Towary (Id_Tow, Id_Produc, Nazwa_Tow, Cena_Tow)
  VALUES (37,'B10','Chleb');
  ```
- ```
  INSERT INTO Towary (Id_Tow, Id_Produc, Nazwa_Tow,)
  VALUES (37,'B10','Chleb',2.10);
  ```
- ```
  INSERT INTO Towary
  VALUES (37,'B10','Chleb',2.10);
  ```
- ```
  INSERT INTO Skład
  SELECT (Id_Tow, Nazwa_Tow, Cena_Tow)
  FROM Towary;
  ```
- ```
  INSERT INTO Skład
  SELECT (Id_Tow)
  FROM Towary;
  ```

jeżeli tabele *Towary* i *Skład* odpowiadają tabelom opisanym w Prz. 4.46 i Prz. 4.47?
  ```

Jeżeli tak, to co będzie wynikiem ich działania? Jeżeli nie są one poprawne, to należy wyjaśnić dlaczego?

4. Zapisać polecenia skierowane pod adresem tabeli *Skład* (z Prz. 4.47), tak aby:
   - *cukierki* miały bardziej szczegółową *nazwę* (różne dla różnych identyfikatorów),
   - wyrobom z *mąki* odpowiadały wartości *identyfikatorów* nie mniejsze niż 100.
5. Zapisać różne typy poleceń skierowanych do tabeli *Towary* (z Prz. 4.46), aby w wyniku został usunięty tylko jeden rekord.

5. Definiowanie danych

Wraz z *Delphi* i *Builderem* (które właśnie będziemy używać w dalszej części książki) dostarczane są narzędzia służące do tworzenia tabel, które są podstawowymi obiektami BD. Programy te pozwalają również w prosty sposób nimi zarządzać. Narzędziami tymi są *Database Desktop* oraz SQL *Explorer*. Pierwsze z nich służy głównie do tworzenia nowych tabel oraz modyfikacji już istniejących. Nie zawiera ono żadnych narzędzi, pozwalających na tworzenie zaawansowanych obiektów, wykorzystywanych poprzez serwery SQL, takich jak: *wyzwalacze*, *procedury zapamiętane*, *generatory* itp. Dlatego *Database Desktop* służy głównie do tworzenia tabel lokalnych baz danych (np. *Paradox*, *dBase* itp.).

SQL *Explorer* umożliwia przeglądanie, tworzenie oraz modyfikowanie zawartości bazy danych zarządzanych przez serwery SQL. Możliwości te nie ograniczają się tylko do tabel, lecz także rozciągają się na pozostałe obiekty serwerów SQL, takie jak: domeny, widoki, procedury, funkcje, generatory oraz wyjątki. Mimo tych zaawansowanych możliwości narzędzie to nie jest wykorzystywane zbyt często. Dzieje się tak dlatego, że większość serwerów baz danych dostarcza własne, często bardzo zaawansowane i wygodne narzędzia, służące do tworzenia i zarządzania całymi strukturami baz danych (włączając w to wszystkie obsługiwane przez dany serwer obiekty).

W tej części przeanalizujemy operacje nad obiektami BD na podstawie możliwości i mechanizmów środowiska Builder 6.0. Takimi obiektami, jak podkreślaliśmy wcześniej, są: sama BD, tabele BD, kolumny tabeli, użytkownicy BD itd.

5.1. Tworzenie obiektów BD w Database Desktop

Pierwsze praktyczne kroki w kierunku zapoznania się z BD warto rozpocząć od definiowania tabel. W tym celu możemy się posłużyć narzędziem *Database Desktop*.

Jak powiedzieliśmy we wstępie, narzędzie *Database Desktop* służy głównie do tworzenia tabel lokalnych BD, takich jak *Paradox* czy *dBase*. W naszym opisie zajmiemy się bazami typu *Paradox*. Każdy element bazy (tj. tabele, indeksy, więzy integralności itd.) przechowywany jest w osobnym pliku.

5.1.1. Tworzenie struktury tabel BD

Przedstawimy na początku, jakie pliki generowane są przez *Database Desktop* w trakcie tworzenia bazy danych. Pliki te zostały opisane w tab. 5.1.

Tabela 5.1. Opis plików *Database Desktop* przeznaczonych do tworzenia BD

Rozszerzenie pliku	*Opis pliku*
.CFG	Plik konfiguracyjny
.DB	Tabela bazy danych typu *Paradox*
.DBF	Tabela bazy danych typu *dBase*
.DBT	Plik przechowujący zawartość pól typu `Memo` dla tabel *dBase*
.FAM	Plik przechowujący powiązania do plików (takich jak pliki TV)
.INI	Plik konfiguracyjny
.LCK	Plik blokad
.MB	Plik przechowujący zawartość pól typu `Memo` dla tabel *Paradox*
.MDX	Plik samoaktualizującego się indeksu dla tabel typu *dBase*
.NDX	Plik zwykłego indeksu dla tabel typu *dBase*
.PX	Plik indeksu głównego (klucza) dla tabel typu *Paradox*
.QBE	Plik zawierający zapytanie utworzone za pomocą narzędzia QBE (Query By Example)
.SQL	Plik z kodem zapytania SQL
.TV	Plik z ustawieniami wyglądu tabeli typu *Paradox*
.TVF	Plik z ustawieniami wyglądu tabeli typu *dBase*
.TVS	Plik z ustawieniami wyglądu dla danych SQL
.VAL	Plik poprawności danych dla pól oraz więzów integralności tabeli typu *Paradox*
.Xnn	Indeks założony na pojedyncze pole tabeli typu *Paradox*
.Ynn	Indeks założony na pojedyncze pole tabeli typu *Paradox*
.XGn	Indeks założony na kilka pól tabeli typu *Paradox*
.Ygn	Indeks założony na kilka pól tabeli typu *Paradox*

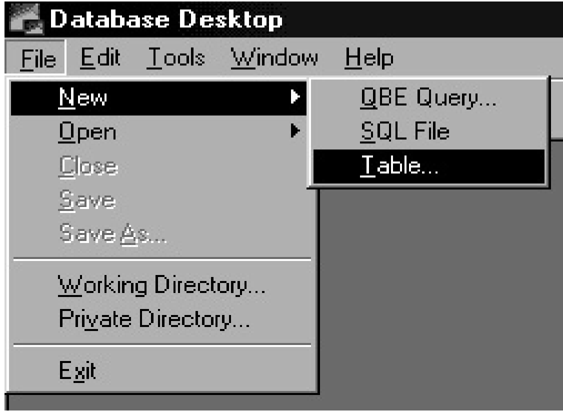
Jako typ tabel wybraliśmy tabele typu *Paradox*, gdyż są one bardziej uniwersalne od innych lokalnych typów tabel. Poza tym charakteryzują się następującymi właściwościami:

- posiadają kontrolę poprawności wprowadzanych danych;
- pozwalają na zdefiniowanie maski wprowadzania danych;
- mają możliwość tworzenia wielu indeksów;
- umożliwiają nakładanie więzów integralności referencyjnej;
- pozwalają na wybór języka tabeli;
- umożliwiają zabezpieczenia tabeli hasłem.

Będziemy omawiać poszczególne etapy wraz z uwzględnieniem ich funkcjonalności pod kątem współpracy z Builderem.

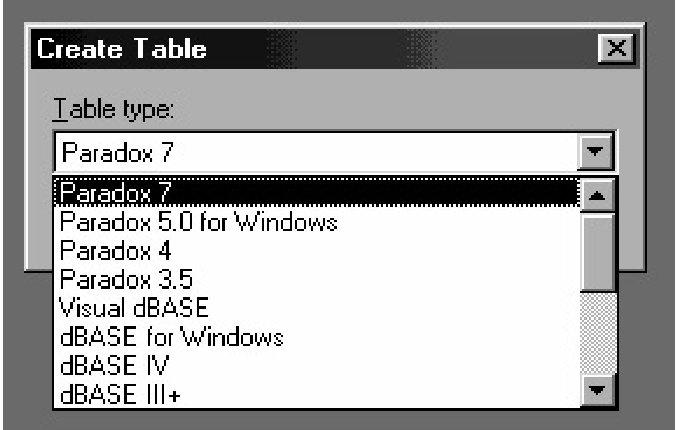
Proces tworzenia tabel typu *Paradox*, wykorzystując narzędzie *Database Desktop*, prześledzimy na przykładzie tabeli *Student*. Tabela ta będzie posiadała następujące pola: *Nr_Albumu*, *Imię*, *Nazwisko*, *Rok_Studiów*, *Ulica*, *Nr_Domu*, *Nr_Mieszkania*, *Miejscowość*, *Kod_Pocztowy*, *Uwagi*.

Przejdźmy teraz do tworzenia tak przedstawionej tabeli. Po uruchomieniu *Database Desktop* z menu (rys. 5.1) wybieramy opcje *File/New/Table*.



Rys. 5.1. Główne okno *Database Desktop*

Z okienka *Create Table* (rys. 5.2) wybieramy typ tworzonej tabeli. W naszym przypadku niech będzie to *Paradox 7* (czyli pozostajemy przy ustawieniu domyślnym).



Rys. 5.2. Okno *Create Table*

Po wciśnięciu przycisku *OK* na ekranie pokaże się właściwe okno (rys. 5.3) dialogowe, służące do definiowania struktury tabeli.

W przedstawionym oknie dialogowym widzimy lewą stronę oznaczoną nagłówkiem *Field roster*. Pozwala ona na dodawanie, edycję, usuwanie i zmianę typu pola oraz rozmiaru. Pierwsze pole określa numer pola i jest ono automatycznie zwiększane w trakcie dodawania nowych pól. Drugie pole *Field Name* określa nazwę pola tabeli. Pole to musi być wypełnione. Nazwa pola musi stosować się do poniższych reguł:

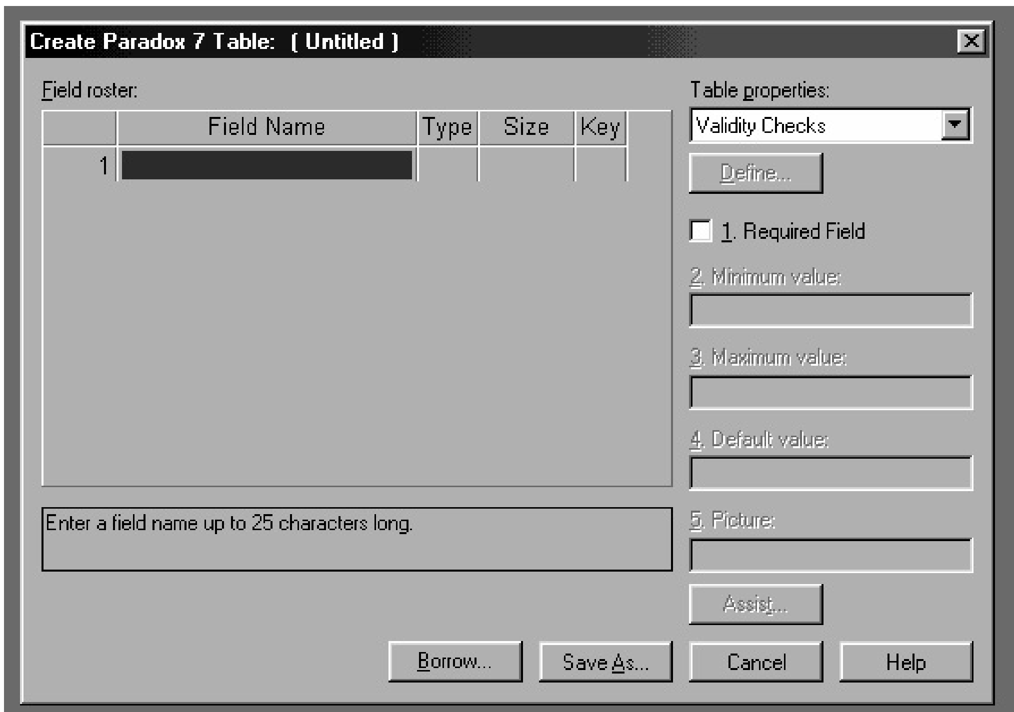
- maksymalna długość nazwy pola to 25 znaków;
- nazwa musi być unikalna w obrębie tabeli.

Może zawierać litery, cyfry oraz dowolne znaki, z wyjątkiem: cudzysłowu, nawiasów kwadratowych, nawiasów klamrowych, nawiasów okrągłych, znaku `#`, a także kombinacji `->`. Przedstawione restrykcje są konsekwencją starszych wersji *Paradox* dla systemu DOS, które zostały zachowane ze względów kompatybilności.

Takie znaki, jak: kropka, przecinek, znak pionowej kreski (`|`), wykrzyknik są dozwolone, ale niezalecane. Znaki te bowiem stanowią elementy składniowe poleceń SQL, co może powodować problemy w prawidłowym ich używaniu. Nazwa pola może zawierać spację, ale nie może się od niej rozpoczynać. Jednakże większość systemów baz danych, szczególnie klient/serwer, nie pozwala na używanie w nazwach pól spacji. Dlatego nie powinno się ich używać w nazwach kolumn, jeżeli planujemy w przyszłości przeniesienie naszych danych do innego formatu (w związku z tym polecamy zamiast spacji postawić znak podkreślenia).

Tabela bazy typu *Paradox* może zawierać maksymalnie 255 pól. Każde pole może być jednym z 17 obsługiwanych przez *Paradox* typów. Wszystkie typy zostały opisane w tab. 4.2.

Tabele typu *Paradox* używają stałej długości dla każdego pola, nawet jeżeli nie zawiera ono żadnych danych. Pola typu `Number` zawsze zajmują 8 bajtów, a pola typu `Alpha` zajmują `n` bajtów, nawet wtedy, kiedy zawartość pola jest krótsza niż `n`. Kolumna *Size* (rozmiar) pokazuje, ile bajtów zajmuje dane pole w jednym rekordzie. Kolumna czwarta zawiera informacje o klasach reprezentujących dany typ pola w Builderze. Każda z tych klas dziedziczy po klasie `TField` (o czym będzie w dalszej części rozdziału). Kolejna kolumna zawiera klasę kontrolki, jaka jest standardowo używana do wyświetlania zawartości pola danego typu. Ostatnia kolumna zawiera uwagi dotyczące poszczególnych pól.



Rys. 5.3. Okno definiowania struktury tabeli

Tabela 5.2. Typy pól w tabeli BD *Paradox*

Typ	*Skrót*	*Rozmiar (w bajtach)*	*Klasa pola w Builderze*	*Komponent w Builderze*	*Pojaśnienia*
`Alpha`	A	1 <= n <= 255	`TStringField`	`TDBEdit`	
`Number`	N	8	`TFloatField`	`TDBEdit`	15 cyfr znaczących z zakresu od -10e-307 do 10e308
`Money`	$	8	`TCurrencyField`	`TDBEdit`	Zakres taki jak w `number`, tylko wyświetlany z dwoma miejscami po przecinku
`Short`	S	2	`TSmallIntField`	`TDBEdit`	16-bitowa liczba całkowita ze znakiem z zakresu od 32767 do 32767; $00 oznacza wartość NULL
`LongInt`	I	4	`TIntegerField`	`TDBEdit`	32-bitowa liczba całkowita ze znakiem z zakresu od -2147483647 do 2147483647; $0000 oznacza wartość NULL
`Date`	D	4	`TDateField`	`TDBEdit`	Dowolna poprawna data od 1.1.9999 p.n.e. do 31.12.9999
`Time`	T	4	`TTimeField`	`TDBEdit`	Liczba milisekund, jakie minęły od północy dnia dzisiejszego
`TimeStamp`	@	8	`TDateTimeField`	`TDBEdit`	Połączenie daty i czasu, z których każdy zajmuje 4 bajty
`Memo`	M	1 <= n <= 240	`TMemoField`	`TDBMemo`	Maksymalny rozmiar to 64 MB
`FormattedMemo`	F	0 <= n <= 240	`TBlobField`	`TDBRichEdit`	Maksymalny rozmiar to 64 MB
`Graphic`	G	1 <= n <= 240	`TGraphicField`	`TDBGraphic`	Maksymalny rozmiar to 64 MB
`OLE`	O	2 <= n <= 240	`TBlobField`	`TDBImage`	Maksymalny rozmiar to 64 MB
`Binary`	B	0 <= n <= 240	`TBlobField`	`TDBImage`	Maksymalny rozmiar to 64 MB
`Bytes`	Y	1 <= n <= 255	`TBytesField`	`TDBEdit`	
`Logical`	L	1	`TBooleanField`	`TDBCheckBox`	Dozwolone wartości to: `true`, `false` i wartość pusta
`BCD`	#	1 <= n <= 32	`TBCDField`	`TDBEdit`	Pole używane w celu wyeliminowania błędów związanych z precyzją obliczeń oraz z zaokrągleniami. Liczba `n` oznacza liczbę cyfr dziesiętnych
`AutoIncrement`	+	4	`TAutoIncField`	`TDBEdit`	Wewnętrznie reprezentowany jak `LongInt`. W tabeli może być tylko jedno pole tego typu

Omówimy teraz dokładnie pewne istotne szczegóły dotyczące przedstawionych typów. Dla pól typu `Memo` *Database Desktop* przechowuje `n` pierwszych znaków w pliku `*.DB`.

Cała zaś zawartość jest przechowywana w plikach *.MB, chyba że owa zawartość jest mniejsza niż n. Z właściwym określeniem rozmiaru pól typu Memo wiąże się kilka trudności:

- pierwsze n znaków w każdym polu Memo jest przechowywana dwa razy. Raz w pliku *.DB i drugi raz w pliku *.MB. Jeżeli n jest duże, duplikowanie danych może znacznie zwiększyć rozmiar bazy;
- z drugiej strony, jeżeli pola Memo są zazwyczaj bardzo krótkie i tylko w jednostkowych przypadkach są dłuższe, to większy rozmiar n pozwala przechowywać całą zawartość Memo tylko w pliku *.DB, z wyjątkiem kilku dłuższych wartości;
- jeżeli często następuje przeszukiwanie względem kilku pierwszych znaków pola Memo, to należy rozmiar n ustalić na zbliżonym do liczby tych znaków poziomie. W takim przypadku BDE będzie korzystał tylko z danych przechowywanych w pliku *.DB, pomijając wykorzystanie zawartości w pliku *.MB.

Tabele typu *Paradox* nie posiadają żadnej możliwości ograniczenia ilości danych, jakie mogą być zapisane w polach Memo. Taką możliwość daje jednak *Builder*. Ustawiając właściwość MaxLength komponentu TDBMemo, możemy ograniczyć liczbę możliwych do wpisania znaków.

W odniesieniu do pól typu Formatted Memo, Graphic, OLE, Binary liczba bajtów, jaka ma być przechowywana w tabeli, jest opcjonalna. Zalecanym rozwiązaniem jest pozostawienie pustego rozmiaru lub wpisanie wielkości 0, co daje ten sam efekt.

Builder nie dostarcza żadnych komponentów do wyświetlania zawartości pól typu OLE, Binary i Bytes. Zawartość pól typu OLE jest zazwyczaj wykorzystywana przez aplikacje używające obiektów OLE, które zapisują w nich tego typu obiekty. Dostęp do pól typu Binary i Bytes jest realizowany poprzez strumienie, które wymagają specjalnych metod, pozwalających na prawidłową współpracę z takimi polami.

Pole typu Autoincrement automatycznie zwiększają swoją wartość dla kolejnych rekordów. Pole to zawiera liczbę Long integer, która jest tylko do odczytu. Pierwszą wartością pola jest wartość 1, jednakże można ją zmienić, definiując odpowiednią wartość w polu *Minimum value*. Usunięcie rekordu o danym numerze nie daje możliwości wykorzystania zwolnionego identyfikatora. Nowa wartość pola tego typu generowana jest w momencie zatwierdzania zawartości danego rekordu.

Znając już wszystkie typy, możemy przejść do tworzenia tabeli *Student*. Definiujemy więc kolejne pola tej tabeli, używając odpowiednich typów. Pozostaje do omówienia ostatnie pole okienka dialogowego o nazwie *Key*. Za pomocą gwiazdki oznaczamy dane pole jako *pole kluczowe*. Gwiazdkę możemy wpisać poprzez wciśnięcie dowolnego klawisza lub podwójne kliknięcie w pole. Możemy również zdefiniować klucz złożony, w takim przypadku gwiazdką należy oznaczyć kilka pól. Pola będące kluczami muszą występować kolejno, rozpoczynając od pierwszego pola. Utworzoną tabelę *Studenci* przedstawia rys. 5.4.

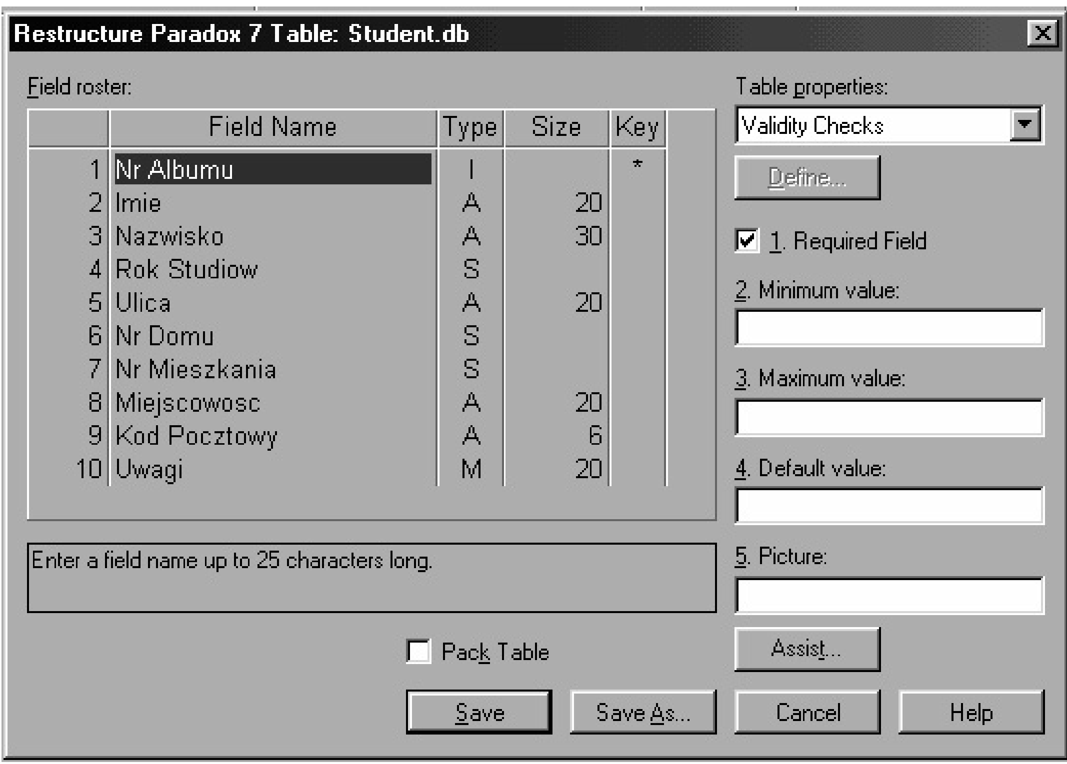
5.1.2. Kontrola poprawności danych

Zwróćmy uwagę na prawą część okna (rys. 5.4) – *Table properties* (właściwości tabeli). Pierwszą z tych właściwości jest *Validity Checks* – kontrola poprawności.

Tabele typu *Paradox* umożliwiają zdefiniowanie reguł poprawności wprowadzanych danych. Oznacza to, że użytkownik nie będzie mógł nadać wartości polu, dla którego

wprowadzana wartość nie będzie spełniała nałożonych ograniczeń. *Paradox* obsługuje następujące typy ograniczeń:

- *Required Field* – pole jest wymagane. Oznacza to, że jego wartość nie może pozostać pusta (nie może posiadać wartości NULL). W przypadku próby zatwierdzenia rekordu, z pustą wartością dla pola takiego typu, zostanie zgłoszony błąd i rekord nie zostanie dodany. Opcja ta może być zastosowana dla pola dowolnego typu. Builder dostarcza właściwości `Required`, która przyjmuje wartość `true` dla pól z zaznaczoną opcją *Required Field,* natomiast przyjmuje ona wartość `false` dla pozostałych pól. Dla naszej tabeli takimi polami prawdopodobnie powinny być pola *Nr_Albumu*, *Imię*, *Nazwisko*.



Rys. 5.4. Okno z opisem struktury i właściwości tabeli

- *Minimum value* – wartość minimalna. Opcja ta pozwala na zdefiniowanie minimalnej wartości dla pól następujących typów: `Alpha`, `Number`, `Short`, `Long Integer`, `Money`, `TimeStamp`, `Time` oraz `Date`. Builder dostarcza właściwości `MinValue` odpowiadającej ograniczeniu *Minimum value*. Jednakże wartość ta nie jest importowana z tabeli zawierającej dane pole i przyjmuje wartość zero. Oczywiście jest ona do wykorzystania, ale zmusza programistę do przypisania odpowiedniej wartości dla tej właściwości. Nadanie wartości właściwości `MinValue` daje korzyści wynikające z informacji płynących ze zgłoszonego wyjątku. Mianowicie, jeżeli wartości te nie są ustalone, to w przypadku wprowadzenia błędnej wartości otrzymamy wyjątek klasy `EDBEngineError` z komunikatem „Minimum validity check failed. Field: <nazwa>". Natomiast jeżeli ustawimy ograniczenie pola poprzez właściwość `MinValue`, otrzymamy wyjątek klasy `EDatabaseError` z komunikatem „X is not a valid value for field <nazwa>. The allowed range is <minimum> to <maksimum>". W naszym przykładzie można zadać minimalną wartość dla pola *Rok_studiow*.

- *Maximum value* – wartość maksymalna. Opis analogiczny, jak w poprzednim punkcie.
- *Default value* – wartość domyślna. Pole zostanie wypełnione wartością domyślną w momencie wstawiania rekordu niezawierającego wypełnionej wartości dla tego pola. Wartość domyślna może być przypisana polom następujących typów: `Alpha`, `Number`, `Short`, `Long Integer`, `Money`, `Logical`, `Date`, `Time`, `TimeStamp`. Builder w pełni wspiera wartości domyślne dla wielu typów pól. Tę właściwość korzystne nadawać polom liczbowym i logicznym.
- *Picture* – maska poprawności. Maski pozwalają na zdefiniowanie szablonu, według którego będą sprawdzane wartości wprowadzane dla danego pola. Wymagają one szerszego omówienia, co zostanie uczynione poniżej.

5.1.3. Maski poprawności danych

Definicja 5-1

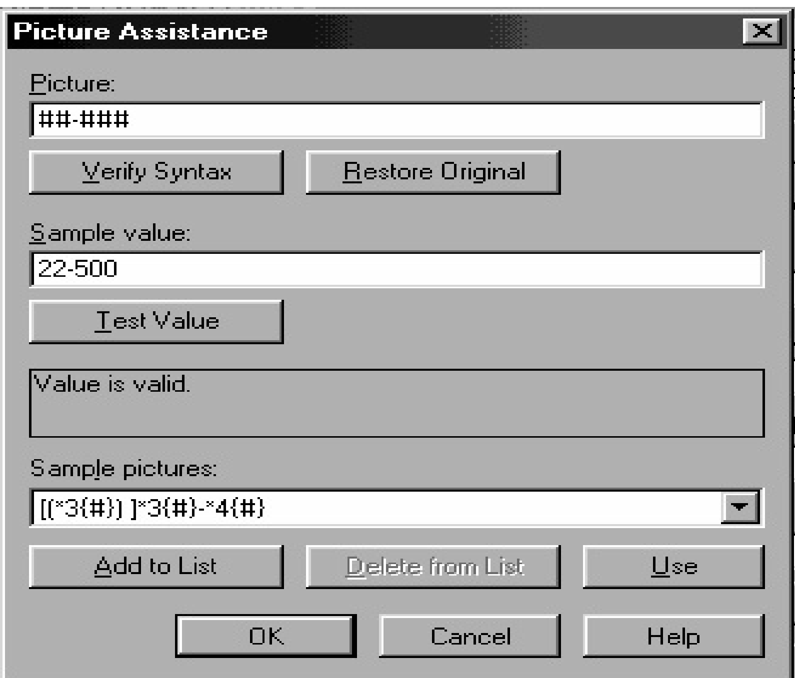
Maska poprawności danych jest łańcuchem znaków, który definiuje, w jakim formacie mają być przechowywane wartości danego pola.

Tabela 5.3 przedstawia znaki, jakie mogą zostać użyte przy definiowaniu maski.

Maskę poprawności możemy zdefiniować w polu *Picture* bądź za pomocą asystenta masek *Assist*.

Tabela 5.3. Opis znaków używanych przy definiowaniu maski

Znak	*Znaczenie*
#	Dowolna cyfra (0-9)
?	Dowolna litera (mała lub duża)
&	Dowolna litera (konwertowana na dużą)
~	Dowolna litera (konwertowana na małą)
@	Dowolny znak
!	Dowolny znak (litera konwertowana na dużą)
;	Interpretuje kolejny znak jako dosłowny
*	Dowolna liczba powtórzeń następnego znaku
*<cyfra>	<cyfra> powtórzeń następnego znaku równa
[a,b,c]	Opcjonalnie a, b lub c
{a,b,c}	Obligatoryjnie a, b lub c
Pozostałe znaki	Traktowane dosłownie



Rys. 5.5. Okno *Picture Assistance*

W polu *Picture* (rys. 5.5) wpisujemy ciąg znaków, który będzie maską. Za pomocą przycisku *Verify Syntax* możemy sprawdzić poprawność maski. W przypadku gdy maska będzie poprawna, w etykiecie tekstowej zostanie wyświetlony komunikat „The picture is correct". Przycisk *Restore Original* przywraca poprzednią maskę pola.

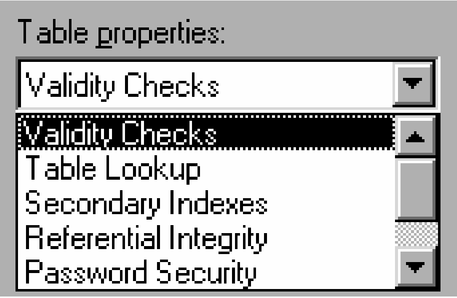
Pole edycyjne *Sample value* pozwala na sprawdzenie przykładowej wartości w kontekście aktualnej maski. Poprawność wprowadzonej wartości testujemy za pomocą przycisku *Test Value*. Informacja ta wyświetlona zostanie w etykiecie tekstowej poniżej przycisku. Dodatkowo w liście rozwijanej zostały zdefiniowane przykładowe maski gotowe do wykorzystania. Dla przykładu pokażemy trzy maski:

- ##-### – kod pocztowy (22-500);
- {Mężczyzna, Kobieta} – płeć (Kobieta);
- *11# – PESEL (01020345678) .

Builder nie dostarcza żadnego wsparcia dla masek zdefiniowanych w tabelach typu *Paradox*. Jednakże dostarcza kilku właściwości specyficznych dla każdego typu pól, pozwalających wykorzystać podobne możliwości:

- klasy: `TFloatField`, `TCurrencyField`, `TSmallIntField`, `TIntegerField`, `TBCDField` i `TAutoIncrementField` dostarczają dwóch właściwości: `DisplayFormat` oraz `EditFormat` (`EditFormat` jest ignorowana dla pól typu `Autoincrement`);
- klasy: `TDateField`, `TTimeField` i `TTimeStampField` dostarczają właściwości: `DisplayFormat` oraz `EditMask`;
- klasa `TStringField` posiada właściwość `EditMask`;
- klasa `TBooleanField` posiada właściwość `DisplayValues` (określa ona sposób reprezentacji wartości `true` i `false`).

Wszystkie omówione opcje dotyczyły kontroli poprawności danych. Możemy z nich korzystać po wybraniu opcji *Validity Checks* z rozwijanej listy *Table properties* (okno na rys. 5.6). Podana lista przedstawia też kilka innych pozycji, które zostaną omówione poniżej.



Rys. 5.6. Okno listy *Table properties*

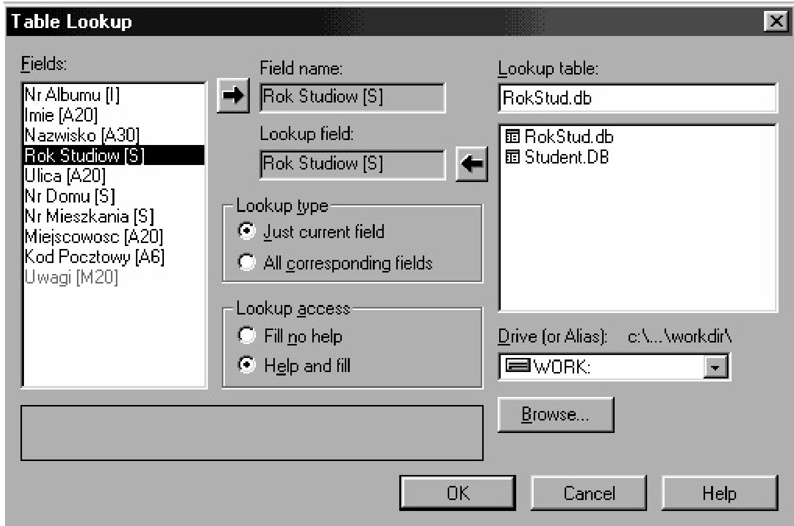
5.1.4. Odnośnik tabeli

Opcja ta za pomocą przycisku *Define...* pozwala na zdefiniowanie odnośnika tabeli do wybranego pola. Odnośnik tabeli (*Table Lookup*) jest tabelą, która zawiera poprawne wartości dla danego pola (*Lookup field*). Określenie odnośnika tabeli umożliwia:

- odwołanie się do tabeli w celu uzyskania listy akceptowalnych wartości;
- automatyczne kopiowanie wartości z odnośnika tabeli do tabeli, która jest edytowana;
- sprawdzanie, czy wartość wpisana do pola *Lookup field* znajduje się w pierwszym polu odnośnika tabeli.

Zdefiniowanie odnośnika tabeli oznacza, że dane pole może zawierać tylko wartości istniejące w pierwszym polu odnośnika tabeli.

Po naciśnięciu przycisku *Define...* ukazuje się okienko przedstawione na rys. 5.7.



Rys. 5.7. Okno *Table Lookup*

Lista *Fields* wyświetla wszystkie pola w danej tabeli. Pola z tej listy można dodać za pomocą strzałki do etykiety *Field name*. To dodane pole (*Lookup field*) będzie powiązane z odnośnikiem tabeli, który wybieramy z listy *Lookup table*. Za pomocą strzałki dodajemy go do etykiety *Lookup field*. Objawi się to wyświetleniem w niej nazwy pierwszego jej pola. Tak skonfigurowane okienko jest gotowe do zatwierdzenia, a tym samym do ustalenia powiązania pomiędzy wybranym polem a tabelą. W naszym przykładzie takim wybranym polem będzie *Rok_Studiow*.

Oprócz tych podstawowych operacji mamy do dyspozycji dwie grupy opcji. Pierwsza opcja *Lookup type* pozwala na określenie typu połączenia pomiędzy tabelami. Jeżeli wybierzemy pierwszą opcję *Just current field*, to tylko pole *Lookup field* pobierze wartość z pola tabeli odnośnika. Jeżeli natomiast wybierzemy opcję drugą (*All corresponding field*), to wtedy oprócz pola *Lookup field* zostaną wypełnione także wszystkie pola korespondujące.

Pola korespondujące to pola posiadające taką samą nazwę w obu tabelach. Druga opcja *Lookup access* pozwala na określenie możliwości użycia pomocy w przypadku wypełniania pola *Lookup field* w *Database Desktop*. Jeżeli wybierzemy pierwszą opcję *Fill no help*, to użytkownik musi bezbłędnie wpisać wartość pola, nie uzyskując żadnej podpowiedzi. Jeżeli zaznaczymy opcję drugą *Help and fill*, to po naciśnięciu kombinacji klawiszy *Ctrl + spacja*, wyświetlone zostanie okienko zawierające wszystkie dostępne dla tego pola wartości. Dzięki temu możemy w łatwy sposób wybrać poprawną wartość.

Podczas tworzenia odnośników tabel należy stosować się do poniższych reguł:

- pole z odnośnika tabeli (*Table lookup*), które zawiera wartości dla pola *Lookup field*, musi być jej pierwszym polem;
- pole *Lookup field* musi być tego samego typu i rozmiaru co pierwsze pole tabeli odnośnika tabeli (*Table lookup*);
- w celu zwiększenia efektywności pierwsze pole odnośnika tabeli, zawierające listę dostępnych wartości, powinno być kluczem;
- możliwe jest użycie odnośnika tabeli (*Table lookup*) znajdującego się w innym katalogu. *Database Desktop* zachowuje pełną ścieżkę dostępu do katalogu. Jeżeli odnośnik tabeli zostanie przeniesiony, należy zmienić tę ścieżkę.

Builder jednak nie daje wsparcia dla obsługi pól *Lookup field*. Opcje *Lookup type* oraz *Lookup access* nie są obsługiwane. W przypadku wprowadzenia niepoprawnej wartości, w momencie zatwierdzenia danego rekordu, dostaniemy wyjątek `EDBEngineError` z niewiele mówiącym komunikatem o błędzie „Field value out of lookup table range". Z tego powodu nie powinno się stosować tabel odnośników. Zamiast tego można użyć integralności referencyjnej, która zapewnia większe możliwości kontroli.

5.1.5. Definiowanie indeksów (dodatkowych)

Definicja 5-2

Indeksy (tutaj – indeksy dodatkowe; ang. *secondary indexes*) są obiektami bazy tworzonymi w celu poprawienia efektywności opracowania danych (jak poprzednio podkreśliliśmy, w relacyjnym modelu BD pojęcie indeksu nie zostało określone).

Indeks można przedstawić jako posortowaną listę. Podczas tworzenia indeksu *Database Desktop* zakłada plik zawierający wartości pola (na które nakładamy indeks) dla każdego rekordu, włączając duplikaty. Wartości te są sortowane w porządku alfabetycznym lub numerycznym (data i czas są przechowywane wewnętrznie jako liczby). Do każdego rekordu dołączony jest wskaźnik, którego wartość określa aktualną pozycję w tabeli.

W tak określonej strukturze do wyszukiwania określonego rekordu wykorzystuje się metodę *wyszukiwania binarnego*, która pozwala w znaczny sposób skrócić czas przeszukiwania.

Podsumowując, indeksy dodatkowe tworzymy w celu:

- zdefiniowania alternatywnego porządku sortowania tabeli; wykorzystując indeks, możemy zmienić domyślny (określony wartościami klucza) porządek sortowania;
- zoptymalizowania połączeń pomiędzy tabelami poprzez nałożenie indeksów na pola definiujące połączenie; indeks z reguły nakładamy na pole będące *kluczem obcym* w danej relacji (dla *klucza głównego* indeks zostaje założony automatycznie); takie postępowanie zapewnia szybsze wykonywanie złączeń tabel oraz umożliwia utworzenie w *Builderze* związku *Master/Detail* (*Nadrzędny/Podrzędny*);
- zwiększenia efektywności przeszukiwania tabeli; odnosi się to do takich operacji, jak: wyszukiwanie, filtrowanie, definiowanie zakresów.

Tabele typu *Paradox* umożliwiają zdefiniowanie więcej niż jednego indeksu na wszystkich typach pól z wyjątkiem: `Memo`, `Formatted Memo`, `Binary`, `OLE`, `Graphic`, `Logical`, `Bytes`. Użycie indeksu nie zmienia fizycznego porządku rekordów.

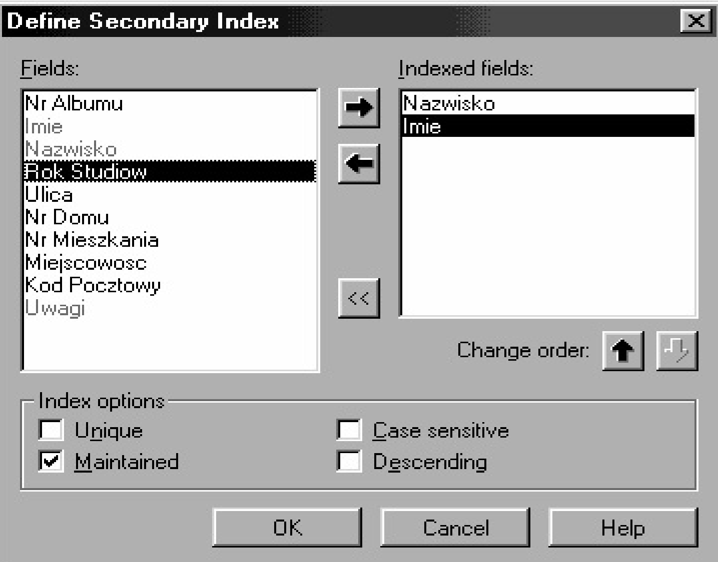
Indeks definiujemy, wykorzystując opcję *Secondary Indexes* z listy rozwijanej *Table properties*. Po wybraniu tej opcji (rys. 5.8) wciskamy przycisk *Define...*, co powoduje pojawienie się okna *Define Secondary Index*.

Okno to pozwala na założenie indeksu wraz z określeniem jego opcji. Opiszemy teraz dokładniej wszystkie dostępne elementy.

Pierwsza lista *Fields* zawiera wykaz dostępnych pól. Pola, które są wypisane szarym kolorem, nie są aktualnie dostępne. Może to być spowodowane tym, że na dane pole indeks został już nałożony lub typ pola uniemożliwia założenie indeksu (patrz dalej: opcja *Unique*). Druga lista *Indexed fields* zawiera pola wchodzące w skład aktualnego indeksu. Kolejność tych pól jest istotna ze względu na porządek sortowania. Do zmiany kolejności możemy wykorzystać strzałki poniżej listy. Tak więc tworzenie indeksu polega na przenoszeniu odpowiednich pól z lewej do prawej listy. Operacje tę wykonujemy za pomocą strzałek znajdujących się pomiędzy listami.

Załóżmy, że w naszej tabeli indeksujemy pola *Nazwisko*, *Imię*, *Rok_Studiów*. W tab. 5.4 podane są warianty indeksów, które mogą być użyteczne w różnych zastosowaniach.

Okienko (rys. 5.8) zawiera także cztery opcje zgrupowane w obrębie grupy *Index options*.



Rys. 5.8. Okno *Definiowania indeksów*

Tabela 5.4. Warianty indeksów dla tabeli *Student*

Nazwa indeksu	*Pola*	*Wyjaśnienie*
Nazw_Imię	*Nazwisko, Imię*	Sortowanie listy studentów alfabetycznie; studenci z tym samym nazwiskiem będą posortowani względem imion
Rok_Nazw_Imię	*Rok_Studiów, Nazwisko, Imię*	Studenci będą uporządkowani względem roku, a wewnątrz roku alfabetycznie

Przykład 5.1

Jeżeli np. zdefiniujemy indeks *Fullname* dla pól *Nazwisko_Stud* i *Imię_Stud* tabeli *Studenci_WSG* {*Id_Stud, Nazwisko_Stud, Imię_Stud, Adres_Stud*}, to kolejność wierszy będzie taka jak w tab. 4.5.

Tabela 5.5. *Studenci_WSG*

Id_Stud	*Nazwisko_Stud*	*Imię_Stud*	*Adres_Stud*
s5	BIENIARZ	ALEKSANDRA	KRAKÓW
s4	CYWIŃSKA	ANNA	POZNAŃ
s7	KOWALSKI	ADAM	LUBLIN
s2	KOWALSKI	PAWEŁ	WARSZAWA
s6	WYSOCKI	KAZIMIERZ	GDAŃSK

Jak widać, kolejność rekordów tab. 5.5 może być taka sama jak po wykonaniu operacji sortowania poprzez zastosowanie frazy ... ORDER BY *Nazwisko_Stud, Imię_Stud*.

Opcja Unique

Opcja *Unique* określa, czy dwa lub więcej rekordy mogą zawierać tę samą wartość dla pól wchodzących w skład indeksu. Jeżeli opcja ta jest zaznaczona, to wstawianie rekordu o zduplikowanych wartościach w obrębie pól wchodzących w skład indeksu nie powiedzie się. Z drugiej strony nie jest również możliwe utworzenie indeksu z zaznaczoną

opcją *Unique* dla tabeli, w której już istnieją takie duplikaty. W takiej sytuacji należy przed założeniem indeksu usunąć powtarzające się wartości.

Nałożenie tej opcji na indeks *Nazw_Imię* zablokuje pojawienie się w tabeli dwóch studentów z powtarzającym się nazwiskiem i imieniem.

Opcja Maintained

Opcja *Maintained* określa, czy indeks będzie automatycznie aktualizowany. Kiedy jest ona włączona, indeks będzie automatycznie aktualizowany wtedy, kiedy dane w tabeli ulegną zmianie. Może być ona zaznaczona tylko dla tabel posiadających *klucz*.

Indeks z odznaczoną opcją *Maintained* będzie aktualizowany wtedy, kiedy wykonywana operacja będzie używać tego indeksu. Operacje z wykorzystaniem takiego indeksu są nieco wolniejsze, dlatego też powinno się używać ich tylko w odniesieniu do tabel *read-only* – tylko do odczytu.

Opcja Case sensitive

Opcja *Case sensitive* określa, w jaki sposób wielkość liter wpływa na kolejność sortowania. Jeżeli jest ona zaznaczona, to wielkość liter będzie rozróżniana. Litery wielkie będą – w porządku sortowania – występować przed małymi. W przeciwnym przypadku małe i wielkie litery nie będą rozróżniane.

Database Desktop automatycznie nazywa indeksy nakładane na pojedyncze pola z zaznaczoną opcją *Case sensitive*. Nazwa tego indeksu będzie tożsama z nazwą pola. Jeżeli indeks jest zakładany z odznaczoną opcją *Case sensitive,* to w takim przypadku użytkownik będzie musiał podać jego nazwę. Umożliwia to utworzenie dwóch indeksów nałożonych na to samo pole: jeden wrażliwy na wielkość liter, drugi niewrażliwy.

Opcja Descending

Opcja *Descending* określa porządek sortowania. Jeżeli jest ona zaznaczona, to porządek sortowania jest malejący (*Desc*). W przeciwnym przypadku (domyślnie) porządek sortowania jest rosnący (*Asc*).

5.1.6. Integralność referencyjna

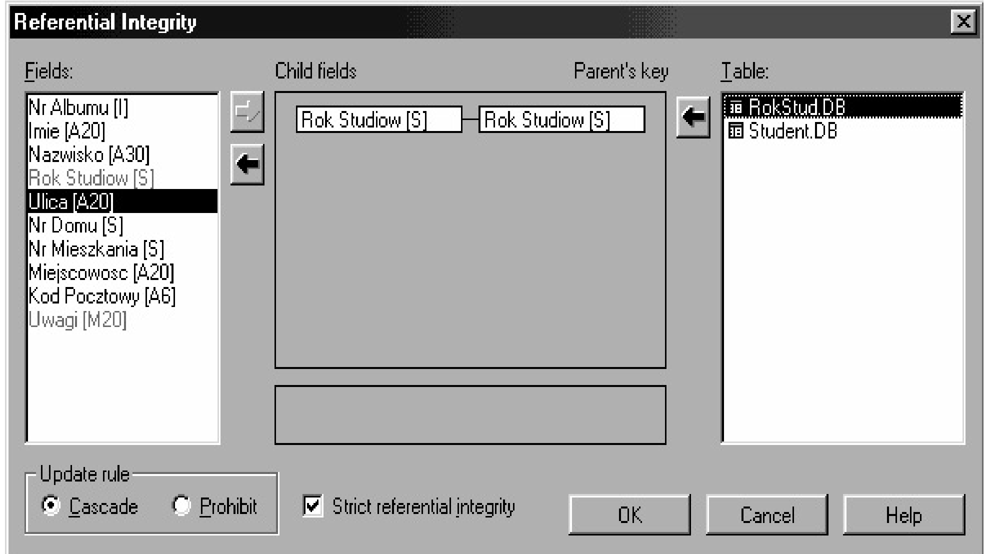
Integralność referencyjna (ang. *referential integrity*) pozwala na zdefiniowanie mechanizmu kontroli poprawności połączeń pomiędzy tabelami (kontrola integralności na poziomie relacji). Jak podkreśliliśmy wyżej (podrozdz. 2.1.2), połączenie takie realizowane jest pomiędzy *kluczem obcym tabeli podrzędnej* a *kluczem głównym* (podstawowym) *tabeli nadrzędnej*. Integralność referencyjna zapobiega istnieniu niepoprawnych powiązań. Oznacza to, że nie może istnieć wartość *klucza obcego* (KO) w *tabeli podrzędnej*, która nie ma odpowiednika w *kluczu podstawowym* (KP) *tabeli nadrzędnej*.

Przejdźmy teraz do szczegółów praktycznej realizacji definiowania integralności referencyjnej.

Przed rozpoczęciem procedury definiowania integralności powinnyśmy mieć dwie tabele, które mamy połączyć. Niech jedną z tych tabel będzie tabela *Student*, która posiada następujące pola: *Nr_Albumu*, *Imię*, *Nazwisko*, *Rok_Studiów*, *Ulica*, *Nr_Domu*, *Nr_Mieszkania*, *Miejscowość*, *Kod_Pocztowy*, *Uwagi,* a drugą tabelą – *RokStud*.

Na początku należy wybrać katalog roboczy (menu *File/Working Directory....*), który posiada obydwie tabele. Później uruchomić tabelę podrzędną (menu *File/Open*) i w trybie restrukturyzacji (*Table/Restructure...)* w oknie *Table properties* wybrać rozdział

Referential Integrity. Po wciśnięciu przycisku *Define...* zostanie wyświetlone stosowne okienko (rys. 5.9).



Rys. 4.11. Okno dialogowe *Integralność referencyjna*

W lewej liście *Fields* wyświetlane są wszystkie pola *tabeli podrzędnej* (w naszym przykładzie jest to tabela *Student*). Nie można utworzyć integralności referencyjnej dla pól typu: `Memo`, `Formatted Memo`, `Graphic`, `Binary`, `OLE`, `Logical`, `Autoincrement`, `BCD`, dlatego te pola są niedostępne (wyświetlane na szaro). Z listy tej wybieramy pole, które jest kluczem obcym. Wybrane pole będzie widoczne w etykiecie *Child fields*. W liście prawej (*Table*) wyświetlane są tabele z katalogu roboczego *Working Directory*. Katalog roboczy możemy zmienić, wybierając z menu opcję *File/Working Directory...*. Wybierając tabelę z tej listy, w etykiecie *Parent's key* zostanie wyświetlona nazwa jej klucza głównego. Po wybraniu obu obiektów możemy utworzyć integralność referencyjną. Do dyspozycji mamy także opcje *Update rule*. Określają one, jakie działania mają być podjęte, kiedy zostanie zmieniona lub usunięta wartość KP *tabeli nadrzędnej*.

Pierwsza opcja *Cascade* oznacza, że jakiekolwiek zmiany KP *tabeli nadrzędnej* mają bezpośredni wpływ na zawartość *tabeli podrzędnej*. Mianowicie, jeżeli zostanie usunięty rekord z tabeli nadrzędnej, to automatycznie zostaną usunięte wszystkie rekordy z tabeli podrzędnej, których wartość KO odnosi się do wartości KP usuniętego rekordu. Działanie takie nie jest dostępne dla tabel typu *Paradox*.

Z drugiej strony, jeżeli zostanie zmodyfikowana wartość KP, to wtedy wartości KO, odnoszące się do zmienianej wartości klucza tabeli nadrzędnej, zostaną zastąpione nową wartością. Działanie to jest wspierane dla tabel typu *Paradox*.

Druga opcja *Prohibit* uniemożliwia usunięcie lub zmianę KP tabeli nadrzędnej, dla którego istnieją powiązania z odpowiednimi wartościami KO tabeli podrzędnej. Opcja ta jest w pełni wspierana dla tabel typu *Paradox*.

Do dyspozycji pozostaje jeszcze jedna opcja *Strict referential integrity*. Opcja ta pochodzi z czasów, kiedy *Paradox* dla DOS i *Paradox* dla Windows były używane jednocześnie. *Paradox* dla DOS nie obsługiwał integralności referencyjnej. Zaznaczenie tej opcji zabezpieczało tabele przed dostępem aplikacji *Paradox* dla DOS, która mogłaby zniszczyć nałożoną integralność referencyjną. Pozwalała ona natomiast na dostęp aplikacji *Paradox* dla Windows. Ustawienie to jest ignorowane przez *Database Desktop* (lub *Borland Database Engine*, BDE).

Po utworzeniu integralności referencyjnej zostaje także dodany indeks dodatkowy nałożony na pole będące kluczem obcym. Przyjmie on nazwę tożsamą z nazwą pola oraz dwiema opcjami: *Maintained* i *Case sensitive*.

Builder w pełni obsługuje implementacje integralności referencyjnej dla tabel typu *Paradox*. W trakcie próby dodania nowej wartości do pola będącego KO, która nie posiada swojego odpowiednika w KP, Builder zgłosi wyjątek klasy `EDBEngineError` z komunikatem „Master record missing”. W komunikacie tym nie dostaniemy informacji o polu, którego wartość naruszyła reguły integralności referencyjnej. Jeżeli nastąpi próba usunięcia lub zmiany wartości klucza głównego, na co nie pozwala reguła integralności, zostanie zgłoszony wyjątek klasy `EDBEngineError` z komunikatem „Master has detail records. Cannot delete or modify”. W tym przypadku Builder nie podaje nazwy tabeli podrzędnej, której dotyczy powstałe naruszenie reguł integralności.

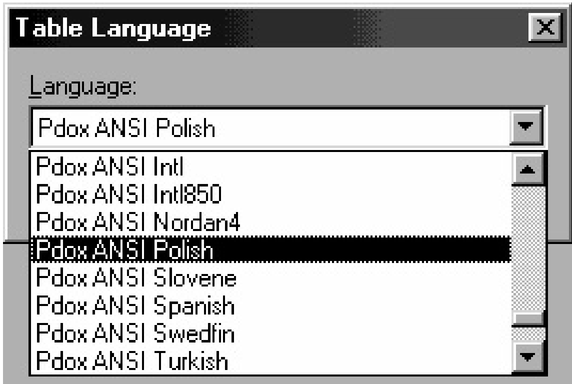
5.1.7. Język tabeli

Język tabeli (ang. *table language*) pozwala określić sposób kodowania znaków w tabelach typu *Paradox*. Domyślny język tabeli pobierany jest od BDE (*Borland Database Engine*). Możemy go ustawić korzystając z narzędzia BDE *Administrator*. Następnie wybieramy zakładkę *Configuration*. Z okienka *TreeView* przechodzimy kolejno poprzez: *Configuration*, *Drivers*, *Native*, *Paradox*. Pozostaje tylko zmienić pozycję LANGDRIVER. BDE wspiera ponad 100 różnych sterowników języków.

Język tabeli określa, które znaki traktowane są jako część alfabetu. Zawiera on również reguły, jak należy konwertować małe litery na ich duże odpowiedniki, a także reguły sortowania danego alfabetu.

BDE przechowuje bieżący sterownik języka na poziomie sesji we właściwości *Locale*. Jeżeli zostaną otwarte dwie tabele używające różnych języków, to właściwość ta przyjmie wartość odpowiadająca językowi używanemu przez ostatnio otwartą tabelę. Może to powodować problemy (np. podczas kopiowania danych pomiędzy tymi tabelami). Tabele typu *Paradox* używają 8 bitów do przechowywania jednego znaku. Dotyczy to pól typu `Alpha` i `Memo`.

Język tabeli wybieramy, używając przycisku *Define...*. W okienku *Table Language* możemy wybrać odpowiedni sterownik. Dla języka polskiego należy użyć sterownika *Pdox ANSI Polish* (rys. 5.10).



Rys. 5.10. Okno *Table Language*

Ostatnia opcja to *Dependent Table*. Wyświetla ona wszystkie tabele, które zależą od bieżącej tabeli. Nie daje ona żadnych bezpośrednich możliwości modyfikacji owych zależności.

Po wypełnieniu wszystkich opisanych procedur i operacji dotyczących struktury tabeli należy kliknąć przycisk *Save as...* i przed nami pojawi się okienko. To okno różni się od standardowego tym, że posiada listę *Alias*, która zawiera pseudonimy wszystkich BD. Z tej listy można wybrać BD, do której będzie wpisana nasza tabela.

Proces tworzenia i zarządzania tabelami w innych SZBD może się nieco różnić.

Zadania do samokontroli

1. Wyjaśnić przeznaczenie *Database Desktop* oraz SQL *Explorer*.
2. Scharakteryzować tabele typu *Paradox*.
3. Za pomocą *Database Desktop* zdefiniować po 2-3 tabele typu *Paradox* (lub *dBase*) z BD:
   - *Biblioteka;*
   - *Apteka;*
   - *Restauracja;*
   - *Przychodnia;*
   - *Szpital;*
   - *Szkoła podstawowa;*
   - *Wydział uczelni wyższej;*
   - *Stacja paliw;*
   - *Wypożyczalnia samochodów;*
   - *Wydawnictwo;*
   - *Księgarnia.*
4. Stworzyć maski poprawności dla tabel BD z zadania nr 3.
5. Wyjaśnić rolę indeksów. Zdefiniować indeksy dla tabel BD z zadania nr 3.
6. Dla tabel BD z zadania nr 3 stworzyć poprawne związki za pomocą integralności referencyjnej.

5.2. Język definiowania danych

Jak powiedzieliśmy we wstępie, do tworzenia BD możemy wykorzystać narzędzia *Database Desktop* oraz SQL *Explorer*. Do tworzenia lokalnych baz danych wykorzystujemy *Database Desktop*, natomiast do tworzenia zdalnych baz danych zastosujemy SQL *Explorer*. Jednakże każdy z producentów serwerów SQL dostarcza własnych, często bardzo zaawansowanych i wyspecjalizowanych narzędzi, służących do tworzenia i zarządzania wszystkimi obiektami BD.

Językiem używanym do tworzenia BD jest język SQL, a dokładniej jego odmiana DDL. Za pomocą jego poleceń możemy tworzyć nie tylko tabele, lecz także wszystkie inne obiekty serwerów SQL. Obiekty definiowane za pomocą DDL nazywamy ***metadanymi***. DDL umożliwia tworzenie, modyfikację oraz usuwanie obiektów serwera SQL.

DDL jest najbardziej zależną od implementacji częścią języka SQL. Jest tak dlatego, że producenci BD używają bardzo różnych obiektów, mających na celu rozszerzenie możliwości, a tym samym ułatwienie pracę z takimi serwerami.

Ciągły postęp w rozwoju BD i wprowadzanie nowych rozwiązań poprawia funkcjonalność, bezpieczeństwo, a także wydajność tak tworzonych baz danych. Jednak w wyraźny sposób utrudnia standaryzację.

Częścią wspólną języka DDL są trzy główne polecenie: `CREATE` (tworzy obiekt), `ALTER` (modyfikuje obiekt) i `DROP` (usuwa obiekt).

Polecenia tworzenia baz danych omówimy na przykładzie serwera *InterBase* 6.0. Wersja ta jest rozprowadzana wraz z narzędziami firmy Borland (Delphi, C++ Builder itp.). Możemy również używać bezpłatnego serwera *FireBird*, który jest klonem serwera *InterBase*, a zatem udostępnia te same funkcje.

Podstawowymi elementami każdego serwera SQL są bazy danych i tabele. Tak więc podstawowe polecenia serwera to:

- tworzenie bazy danych – `CREATE DATABASE`;
- tworzenie tabel – `CREATE TABLE`;
- modyfikacja tabel – `ALTER TABLE`;
- usuwanie tabel – `DROP TABLE`.

Wszystkie *metadane* przechowywane są w tabelach systemowych. Są one zbiorem tabel automatycznie tworzonymi podczas tworzenia BD. Nazwy wszystkich tabel systemowych rozpoczynają się od przedrostka RDB$. Dla przykładu: systemowa tabela RDB$RELATIONS zawiera dane o wszystkich tabelach w BD.

Do tworzenia obiektów BD możemy korzystać z dwóch narzędzi. Pierwsze to ISQL, drugie zaś to *IBConsole* (w środowisku *InterBase*). Narzędzie ISQL jest typową aplikacją konsolową, służącą do wykonywania poleceń języka SQL. Zazwyczaj uprzednio przygotowany plik tekstowy, zawierający polecenia tworzenia bazy i wszystkich jej obiektów, ładujemy do okna konsoli i wykonujemy go. Użycie pliku tekstowego jest wskazane przy tworzeniu każdej BD oraz przy pracy z każdym serwerem BD.

5.2.1. Zdanie CREATE

Polecenie tworzenia bazy danych – CREATE DATABASE

Bazę danych tworzymy za pomocą polecenia `CREATE DATABASE`. Polecenie to ma następującą składnię:

```
CREATE {DATABASE | SCHEMA} 'NazwaPliku'
  [USER 'NazwaUzytkownika' [PASSWORD 'haslo']]
  [PAGE_SIZE [=] int]
  [LENGTH [=] int [PAGE[S]]]
  [DEFAULT CHARACTER SET charset]
  [<dodatkowy_plik>];
```

```
<dodatkowy_plik> = FILE 'NazwaPliku' [<plik_info>] [<dodatkowy_plik>]
<plik_info> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
[<plik_info>]
```

Listing 5.1

Nazwy plików, nazwy użytkowników oraz hasła powinny być umieszczane w pojedynczych apostrofach. Wszystkie polecenia języka SQL będą zapisywane w podobnej konwencji. Zwróćmy jeszcze raz uwagę na widoczne w list. 5.1 symbole:

- {a | b} – alternatywa (a albo b);
- [a] – opcjonalność (parametr może wystąpić, ale nie musi);
- <a> – a jest wyrażeniem o bardziej skomplikowanej konstrukcji, która wymaga osobnej definicji;
- <a> = – definicja wyrażenia.

Przejdźmy teraz do omówienia szczegółów powyższego polecenia. Jak widzimy, zawiera ono liczne parametry, z których tylko *NazwaPliku* jest parametrem obowiązkowym (pozostałe są opcjonalne). Tak więc *NazwaPliku* jest parametrem, określającym ścieżkę dostępu do nowo utworzonej bazy danych. Zawiera ona nazwę dysku, ścieżkę katalogów i nazwę bazy danych. Domyślnie baza tworzona jest jako jeden plik.

Przykład 5.2
Oto przykład polecenia tworzącego bazę *Firma*:

```
CREATE DATABASE 'c:\Firma.gdb';
```

Plik ten, oprócz wszystkich tabel będzie zawierał także i inne obiekty należące do tej bazy (indeksy, wyzwalacze, procedury składowane itp.).

Dwa kolejne parametry `USER` i `PASSWORD` są opcjonalne. Parametr `USER` określa użytkownika bazy, natomiast `PASSWORD` jego hasło. Jeżeli łączymy się ze zdalnym serwerem, powyższe parametry nie są opcjonalne. Dodajmy jeszcze, że dla haseł tylko 8 pierwszych znaków jest znaczących (tzn. są brane pod uwagę).

Przykład 5.3
Przykład polecenia wykorzystującego powyższe parametry USER i PASSWORD:

```
CREATE DATABASE 'Firma.gdb' USER 'Jan_Kowalski' PASSWORD 'Ajh80%#_';
```

Parametr `PAGE_SIZE` pozwala zmienić domyślny rozmiar strony (1024 bajty). Może on przyjmować następujące wartości: 1024, 2048, 4096, 8192. Zwiększenie wielkości strony może w niektórych przypadkach zwiększyć wydajność bazy danych:

- indeksy są bardziej efektywne (przy większym rozmiarze strony), gdyż zmniejsza się ich głębokość;
- przechowywanie dużych rekordów na pojedynczej stronie jest bardziej efektywne (jeżeli pewien rekord jest zbyt duży, aby zmieścić się w rozmiarze jednej strony, to podczas operacji odczytu czy zapisu wykorzystywana jest więcej niż jedna strona);
- wykorzystanie danych typu `BLOB` jest bardziej efektywne, gdy dane te zajmują jedną stronę (jeżeli rozmiar danych zawiera się pomiędzy 1KB a 2KB, to domyślny rozmiar strony powinien przyjąć wartość 2048).

Jeżeli przeważająca większość transakcji wykorzystuje tylko kilka wierszy, to mniejszy rozmiar strony może wpłynąć na większą wydajność bazy. Wynika to z faktu, że podczas odczytywania i zapisywania danych będą wykorzystane mniejsze ilości danych.

Przykład 5.4

Przykład wykorzystania parametru PAGE_SIZE:

```
CREATE DATABASE 'Firma.gdb' PAGE_SIZE 2048;
```

Kolejny parametr LENGTH związany jest z podziałem BD na kilka plików. Domyślnie cała baza jest przechowywana w jednym pliku, zwanym *plikiem głównym*. W takim przypadku parametr LENGTH określa liczbę stron, na jakich będzie przechowywana baza danych.

Przykład 5.5

Poniższy przykład tworzy bazę danych, której zawartość będzie zajmować 1000 stron:

```
CREATE DATABASE 'Firma.gdb' LENGTH 1000;
```

Jeżeli jednak rozmiar bazy zwiększy się ponad podaną w parametrze LENGTH wartość, to *InterBase* zwiększy rozmiar pliku głównego. Rozmiar pliku będzie się zwiększał aż do zapełnienia całej przestrzenie dysku lub do osiągnięcia rozmiaru 4GB (maksymalny rozmiar pliku BD dla serwera *InterBase* 6.0). W celu zabezpieczenia się przed takim działaniem można przechowywać zawartość bazy w kilku plikach. Pierwszy plik jest plikiem głównym, natomiast pozostałe to pliki drugorzędne. Zazwyczaj każdy z plików umieszcza się na oddzielnym dysku. *InterBase* rozpoczyna wprowadzanie danych od pliku głównego. Jeżeli jego rozmiar osiągnie wartość podaną w parametrze LENGTH, to dane będą zapisywane do następnego pliku drugorzędnego. Parametr LENGTH ostatniego pliku jest ignorowany. Oznacza to, że *InterBase* będzie umieszczał w nim dane aż do zapełnienia całej przestrzeni dysku lub do osiągnięcia rozmiaru 4GB. Wynika stąd, że parametr LENGTH powinien być używany tylko wtedy, kiedy zostały zdefiniowane pliki drugorzędne. Nie ma żadnych mechanizmów, pozwalających określić, jakie dane będą zapisywane w poszczególnych plikach. Zadanie to jest realizowane automatycznie.

Do określenia plików drugorzędnych wykorzystujemy parametr FILE.

Przykład 5.6

Poniższe polecenie tworzy BD *Firma* przechowywaną w kilku plikach:

```
CREATE DATABASE 'Firma.gdb'
  FILE 'Firma 2.gdb' STARTING AT PAGE 1001 LENGTH 1000 PAGES
  FILE 'Firma 3.gdb' LENGTH 1000 PAGES
  FILE 'Firma 4.gdb';
```

Jeżeli nie zostanie zdefiniowany parametr LENGTH danego pliku, to musi zostać określony parametr STARTING AT PAGE pliku następującego po nim, który wskazuje numer strony, od której rozpoczyna się dany plik. Jeżeli zostaną określone oba parametry, tzn. LENGTH danego pliku oraz STARTING AT PAGE pliku następującego po nim, to będą wzięte pod uwagę ustawienia parametru LENGTH. Słówko PAGES jest całkowicie opcjonalne.

Ostatnim parametrem polecenia CREATE DATABASE jest DEFAULT CHARACTER SET. Pozwala on zdefiniować domyślny zestaw znaków dla danej bazy danych, który określa:

- jaki zestaw znaków będzie domyślnie używany dla pól tekstowych (CHAR, VARCHAR, BLOB text);
- kolejność znaków wykorzystywaną podczas sortowania danych.

Przykład 5.7
Poniższe polecenie tworzy bazę BD, używającą domyślnego zestawu znaków WIN1250, który jest używany przez polską wersję systemu Windows:

```
CREATE DATABASE 'Firma.gdb' DEFAULT CHARACTER SET WIN1250;
```

Jeżeli nie zostanie podany zestaw znaków, to zostanie on określony jako NONE. Oznacza to, że nie jest ustawiony żaden zestaw znaków. Dane będą odczytywane w oryginalnym formacie (czyli w takim, w jakim zostały tam wprowadzone).

Polecenie tworzenia tabeli – CREATE TABLE

Do tworzenie tabel wykorzystujemy polecenie `CREATE TABLE`, którego składnia jest następująca:

```
CREATE TABLE NazwaTabeli [EXTERNAL [FILE] 'NazwaPliku']

...

(<definicja_kolumny>[,<definicja_kolumny>|<warunek_integralności>...])
;
```

Listing 5.2

Pierwszym parametrem powyższego polecenia jest nazwa tabeli, która jest obligatoryjna. Musimy być ona unikalna w obrębie wszystkich tabel i procedur. Polecenie tworzenia tabeli powinno zawierać również przynajmniej jedną definicję kolumny.

Jednym z podstawowych elementów każdej tabeli są *kolumny* (pola). Kolumna to określenie praktyczne używane w kontekście tworzenia i wykorzystania tabel w konkretnym systemie baz danych. *Pole* natomiast pochodzi z teorii relacyjnych baz danych i jest pojęciem teoretycznym. Oba pojęcia są używane wymiennie w języku technicznym (w tym w niniejszej książce).

Definicja *kolumny* pozwala określić parametry, jakim będzie ona podlegała. Oto jej składnia:

```
<definicja_kolumny> = NazwaKolumny { typ_danych | COMPUTED [BY]
(<wyrażenie>) | NazwaDomeny}

[DEFAULT { literał | NULL | USER}]

[NOT NULL] [ <warunek_integralności>]

[COLLATE NazwaPorządkuSortowania]
```

Omówimy teraz parametry powyższego polecenia.
Parametry obowiązkowe to:

- nazwa kolumny – musi być unikalna w obrębie kolumn tej samej tabeli;
- jeden z poniższych parametrów:
  - typ danych,
  - wyrażenie – dla kolumn obliczanych,
  - nazwa domeny.

Natomiast do parametrów opcjonalnych zaliczamy:

- wartość domyślną;
- warunek integralności – może odwoływać się do konkretnej kolumny lub do zbioru kolumn danej tabeli. Do warunków integralności zaliczamy:

  - PRIMARY KEY – klucz główny,
  - UNIQUE – kolumna unikalna – jej wartości nie mogą być duplikowane oraz nie mogą zawierać wartości NULL,
  - FOREIGN KEY – klucz obcy – posiada dwa mechanizmy (ON UPDATE, ON DELETE), pozwalające na określenie, jakie działanie ma być podjęte w stosunku do kluczy obcych, gdy wartości odpowiadających im kluczy głównych ulegną zmianie;
- ograniczenie NOT NULL – jest wymagane dla kolumn zawierających atrybuty PRIMARY KEY i UNIQUE;
- ograniczenie CHECK;
- zestaw znaków CHARACTER SET.

Omówimy teraz kolejno wymienione powyżej parametry. Pierwszy parametr to nazwa kolumny, który jednoznacznie definiuje kolumnę w danej tabeli. Drugim obowiązkowym parametrem jest jeden z trzech parametrów, tj.: typ danych, wyrażenie (kolumny obliczane) lub nazwa *domeny* (patrz niżej). Typ danych definiuje zbiór wartości, jakie może przyjmować kolumna. Określa on również zestaw operacji, jakie mogą być wykonywane na jej wartościach, oraz rozmiar, jaki ona zajmuje. Parametr ten nie będzie omawiany, ponieważ został już szczegółowo opisany w poprzednich rozdziałach.

Serwer *InterBase* pozwala na definiowanie kolumn obliczanych. Są to tzw. kolumny wirtualne. Ich zawartość nie jest fizycznie przechowywana w bazie danych, lecz jest obliczana na podstawie określonego wyrażenia wraz z każdym dostępem do wartości tych kolumn. Składnia kolumny obliczanej jest następująca:

```
Nazwa_kolumny COMPUTED [BY] (<wyrażenie>);
```

Wyrażenie obliczane musi zwracać pojedynczą wartość (nie może to być tablica). Kolumna wymieniona w wyrażeniu musi być wcześniej zdefiniowana.

Przykład 5.8
Ten przykład pokazuje sposób użycia wyrażenia w polach kalkulowanych:

```
CREATE TABLE Towary
(
Id INT NOT NULL PRIMARY KEY,
Cena_Netto NUMERIC(9, 2),
VAT SMALLINT,
Cena_Brutto COMPUTED BY (Cena_Netto+Cena_Netto*VAT/100)
);
```

Przykład ten tworzy tabelę *Towary* z kolumną obliczaną *Cena_Brutto*, której wartość obliczana jest na podstawie dwóch pozostałych kolumn (*Cena_Netto*, *VAT*). Oczywiście, składnia wyrażeń dla kolumn obliczanych nie ogranicza się do operacji arytmetycznych.

Przykład 5.9
Poniższy przykład pokazuje sposób wykorzystania wyrażenia dla kolumn typów znakowych:

```
CREATE TABLE Pracownicy
(
Imię VARCHAR(20) NOT NULL,
Nazwisko VARCHAR(30) NOT NULL,
Pracownik COMPUTED BY (Imię ||‘ ’|| Nazwisko)
);
```

W wyniku realizacji tego polecenia zostanie stworzona tabela *Pracownicy*, składająca się z pól *Imię*, *Nazwisko*, *Pracownik*. Przy czym ostatnie pole tabeli definiuje się z zastosowaniem wyrażenia na podstawie operatora *konkatenacji*.

W definicji kolumny możemy użyć nazwy *domeny*, której definiowanie przeanalizujemy niżej. Taka kolumna dziedziczy wszystkie cechy wyspecyfikowane w definicji danej domeny. Niektóre z nich można zmienić w definicji kolumny. Może ona uaktualnić wartość domyślną, dodać nowy warunek (`CHECK`), zmienić porządek sortowania lub dodać ograniczenie `NOT NULL` (jeżeli nie wystąpiło w definicji domeny).

Przykład 5.10

Poniższe zdanie SQL tworzy tabelę *Studenci*, wykorzystując (utworzoną wcześniej) domenę *Dnazwisko*:

```
CREATE TABLE Studenci
(
Nr_Albumu INTEGER NOT NULL PRIMARY KEY,
Imię VARCHAR(20) NOT NULL,
Nazwisko DNazwisko NOT NULL
);
```

Przejdźmy teraz do parametrów opcjonalnych. Wartość domyślna została już opisana przy okazji omawiania domen. Dlatego też przejdziemy od razu do drugiego parametru, jakim są warunki integralności (ang. *integrity constraints*). Zdefiniowanie warunków integralności pozwala na sprawowanie kontroli nad relacjami pomiędzy kolumną a tabelą oraz pomiędzy tabelami. Są one bardzo ważnym mechanizmem wspierającym teorię relacyjnych baz danych. Zabezpieczają przed wprowadzaniem błędnych danych lub przed modyfikacją danych, która mogłaby wpłynąć na niepoprawność określonych relacji. Warunki integralności mogą być definiowane dla całej tabeli lub dla pojedynczej kolumny. Serwer *InterBase* sprawuje pełną kontrolę oraz w pełni wspiera wszystkie mechanizmy związane ze zdefiniowanymi warunkami integralności (patrz podrozdz. 2.1).

Pierwsze dwa warunki integralności to `PRIMARY KEY` i `UNIQUE`. Zapewniają one, że wartości w danej kolumnie są unikalne (nie posiadają duplikatów). Podczas próby wstawienia już istniejącej wartości do kolumny z powyższymi warunkami *InterBase* zwróci błąd informujący o naruszonym ograniczeniu.

Odwołując się do teorii relacyjnych baz danych, możemy zauważyć, że `PRIMARY KEY` jest *kluczem głównym* tabeli, natomiast `UNIQUE` jest *kluczem kandydującym*. Powyższe warunki integralności posiadają także te same właściwości co ich teoretyczne odpowiedniki. Mianowicie, klucz tabeli może być kluczem prostym (składającym się dokładnie z jednej kolumny) lub kluczem złożonym (składającym się z więcej niż jednej kolumny). Wartości klucza nie mogą również zawierać wartości NULL. Oznacza to, że definiując warunki integralności `PRIMARY KEY` lub `UNIQUE`, kolumny wchodzące w ich skład muszą być opatrzone atrybutem `NOT NULL`. Powyższe warunki integralności mogą być definiowane na poziomie danej kolumny lub na poziomie tabeli. Pierwszy sposób znajduje zastosowanie w przypadku tworzenia kluczy prostych, drugi zaś w odniesieniu zarówno do kluczy prostych, jak i złożonych.

Zdefiniowanie warunków integralności na poziomie kolumn uniemożliwia ich powtórne definiowanie na poziomie tabel. Tabela może posiadać tylko jeden klucz główny. Zatem warunek integralności `PRIMARY KEY` może wystąpić tylko raz w definicji całej tabeli. Ograniczenia tego nie posiada warunek integralności `UNIQUE`. Zasady te są w pełni zgodne z teorią, która zakłada posiadanie przez daną tabelę tylko jednego *klucza głównego*, dopuszczając jednocześnie istnienie wielu *kluczy kandydujących*. Klucze tabeli

mogą pełnić rolę kluczy obcych w innych tabelach, co oznacza, że dwa powyższe warunki integralności można wykorzystać do tworzenia relacji pomiędzy tabelami.

Trzecim warunkiem integralności jest `FOREIGN KEY` (*klucz obcy*). Jest on odpowiednikiem klucza obcego z teorii relacyjnych baz danych, który jest kolumną lub zbiorem kolumn odnoszącym się do klucza głównego innej tabeli. Relację pomiędzy tabelami możemy również tworzyć na podstawie *klucza kandydującego*. Możliwe jest to dzięki temu, że klucz kandydujący spełnia te same warunki co *klucz główny* tabeli.

Przedstawmy teraz najważniejsze cechy warunku integralności `FOREIGN KEY`:

- przed utworzeniem kolumny, będącej *kluczem obcym* (warunek integralności `FOREIGN KEY`), muszą zostać utworzone odpowiadające mu klucze (warunki integralności `PRIMARY KEY` lub `UNIQUE`) *tabeli nadrzędnej*;
- warunek integralności `FOREIGN KEY` posiada dwie opcje: `ON UPDATE` oraz `ON DELETE`. Określają one, jakie działanie ma być podjęte w odpowiedzi na *modyfikację* (lub *usunięcie*) wartości klucza *tabeli nadrzędnej*, odpowiadającego danemu *kluczowi obcemu*, tak aby zachować poprawność relacji; dodajmy, że działanie to może uniemożliwić modyfikację wartości klucza głównego albo odpowiednio zmodyfikować wartości *klucza obcego*. Powyższe dwie opcje mogą posiadać cztery parametry (`NO ACTION`, `CASCADE`, `SET DEFAULT`, `SET NULL`) (opisane w tab. 5.6):

Tabela 5.6. Opis parametrów opcji `ON UPDATE` oraz `ON DELETE`

Parametr	*Działanie*
`NO ACTION`	Wartości klucza obcego nie ulegają zmianie. Jeżeli modyfikacja lub usunięcie wartości klucza tabeli nadrzędnej spowodowałoby naruszenie integralności relacji, to taka operacja nie powiedzie się i zostanie zgłoszony błąd (ustawienie domyślne)
`CASCADE`	Wartości klucza obcego zostaną usunięte (w przypadku usunięcia odpowiadającej im wartości klucza tabeli nadrzędnej) lub zastąpione (w przypadku modyfikacji odpowiadającej im wartości klucza tabeli nadrzędnej) nową wartością klucza tabeli nadrzędnej
`SET DEFAULT`	Wartości klucza obcego zostaną zastąpione domyślną wartością kolumny (w przypadku modyfikacji czy usunięcia odpowiadającej im wartości klucza tabeli nadrzędnej). W przypadku braku zdefiniowanej wartości domyślnej zostanie użyta wartość NULL
`SET NULL`	Wartości klucza obcego zostaną zastąpione wartością NULL (w przypadku modyfikacji czy usunięcia odpowiadającej im wartości klucza tabeli nadrzędnej)

- jeżeli podczas definiowana *klucza obcego* nie zostanie określona żadna z jego opcji (tzn. `ON UPDATE` lub `ON DELETE`), to zostanie przyjęty domyślny parametr `NO ACTION`. Oznacza to, że w przypadku modyfikacji lub usuwania wartości klucza *tabeli nadrzędnej* nie będzie podejmowana żadna operacja, mająca na celu zmianę korespondujących z nią wartości *klucza obcego tabeli podrzędnej*. Mechanizm integralności relacji nie pozwoli jedynie na modyfikację czy usunięcie wartości klucza *tabeli nadrzędnej*, jeżeli istnieją odwołujące się do niego wartości *klucza obcego tabeli podrzędnej*. Jednakże w takim przypadku mechanizm kontroli może zostać jawnie określony przez użytkownika. Mianowicie, może on utworzyć odpowiedni *trigger*, którego zadaniem będzie realizowanie jednej z operacji przedstawionej w powyższej tabeli, jak również realizacja całkowicie nowej, określonej przez użytkownika operacji;

- nie jest możliwe dodanie wartości do *klucza obcego tabeli podrzędnej*, która nie ma swojego odpowiednika w wartości klucza *tabeli nadrzędnej*.

Przykład 5.11

Stworzymy tabelę *Projekty*, która przechowuje dane o projektach zrealizowanych przez poszczególnych studentów. W tak określonej relacji tabela *Studenci* jest *tabelą nadrzędną*, a tabela *Projekty* podrzędną. Posiada ona zatem *klucz obcy Nr_Albumu* (jest kluczem podstawowym tabeli *Studenci*), dzięki któremu łączy ona dany projekt ze studentem, który jest jego autorem. Chcemy również zabezpieczyć się przed utratą integralności relacji na skutek usunięcia lub zmodyfikowania informacji o studencie. Dlatego też z tym działaniem powinny być związane odpowiednie mechanizmy *kontroli poprawności relacji*. W naszym przykładzie żądamy następujących operacji:

- nie można zmodyfikować wartości *Nr_Albumu* w tabeli *Studenci*, dopóki istnieją rekordy z tabeli *Projekty* z nią powiązane (tzn. wartość ta występuje przynajmniej w jednym wierszu tabeli *Projekty*);
- usunięcie wartości *Nr_Albumu* z tabeli *Studenci* powoduje automatyczne usunięcie wszystkich powiązanych z nią wierszy tabeli *Projekty* (tzn. tych, w których występuje usuwana wartość).

Przy tak określonych warunkach definicja tabeli *Projekty* mogłaby wyglądać następująco:

```
CREATE TABLE PROJEKTY
(
Id_Projektu INTEGER NOT NULL PRIMARY KEY,
Nazwa VARCHAR(10),
Nr_Albumu INTEGER REFERENCES STUDENCI(Nr_Albumu)
ON UPDATE NO ACTION
ON DELETE CASCADE
);
```

Parametr `NO ACTION` jest parametrem domyślnym, dlatego też w powyższym przykładzie można opuścić opcję `ON UPDATE`. W przypadku gdy użytkownik definiuje *klucz obcy*, odwołujący się do klucza *tabeli nadrzędnej*, której właścicielem jest inny użytkownik, musi on mieć przyznane uprawnienie `REFERENCES`. W przeciwnym wypadku (przy braku uprawnień) *InterBase* poinformuje o braku stosownych uprawnień.

Powiedzmy teraz, jak definiować omówione warunki integralności. Podczas definiowania tych warunków możemy użyć parametru `CONSTRAINT`, który pozwala nam nadać nazwę danego warunku integralności. Parametr ten jest opcjonalny. Jeżeli zostanie on pominięty, to *InterBase* wygeneruje unikalną nazwę, która będzie zapisana w tabeli systemowej RDB$RELATION_CONSTRAINTS.

Składnia warunku integralności jest uzależniona od poziomu, na którym będzie definiowana. Na poziomie kolumny będzie ona następująca:

```
<kol_warunek_integralności> = [CONSTRAINT NazwaWarunku]
<kol_definicja_warunku>
[<kol_definicja_warunku> ...]
<kol_definicja_warunku> = {UNIQUE | PRIMARY KEY
| CHECK ( <warunek>)
| REFERENCES tabela_nadrzędna [(kolumna1 [, kolumna2 …])]
```

```
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
}
```

Składnia warunku integralności na poziomie tabeli jest następująca:

```
<tab_warunek_integralności>= [CONSTRAINT NazwaWarunku]
<tab_definicja_warunku>
[<tab_definicja_warunku> ...]
<tab_definicja_warunku> = {{PRIMARY KEY | UNIQUE} (kolumna1 [,
kolumna2 …])
| FOREIGN KEY (kolumna1 [, kolumna2 …]) REFERENCES tabela_nadrzędna
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<warunek>)}
```

Pokażmy teraz dwa przykłady, prezentujące definiowanie warunków integralności (Prz. 5.12 i 5.13).

Przykład 5.12

Prz. 5.12 przedstawia polecenia tworzenia tabeli wraz z definicją *klucza głównego* (PRIMARY KEY – definicja na poziomie kolumny):

```
CREATE TABLE TOWARY
(
Id_Towaru INTEGER NOT NULL PRIMARY KEY,
Nazwa VARCHAR(10)
);
```

Przykład 5.13

Ten przykład pokazuje polecenie tworzenia tabeli z dwoma *kluczami kandydującymi* (`UNIQUE` – definicja na poziomie kolumny oraz na poziomie tabeli):

```
CREATE TABLE FAKTURY
(
Id_Faktury INTEGER NOT NULL UNIQUE,
Nr_Faktury INTEGER NOT NULL,
Rok DATE NOT NULL,
CONSTRAINT Con_Fak UNIQUE(Nr_Faktury, Rok)
);
```

Pozostaje nam jeszcze do omówienia definiowanie warunków poprawności – `CHECK`. Pozwalają one określić, jakie wartości są poprawne dla danej kolumny. Sprawdzanie poprawności odbywa się na podstawie określonego warunku. Jeżeli jest on poprawny, dana wartość zapisywana jest do bazy. Jeżeli nie, to zostaje wyświetlony komunikat o naruszonym ograniczeniu, a dane nie są zapisywane do bazy danych. Warunek poprawności pozwala na porównywanie wartości wprowadzanych do różnych kolumn.

Składnia warunku poprawności `CHECK` jest następująca:

```
CHECK (<warunek>);
<warunek> = <war> <operator> {<war> | (<select_jedna>)}
```

```
| <war> [NOT] BETWEEN <war> AND <war>
| <war> [NOT] LIKE <war> [ESCAPE <war>]
| <war> [NOT] IN (<war> [, <war> …] | <select_lista>)
| <war> IS [NOT] NULL
| <war> {>= | <=}
| <war> [NOT] {= | < | >}
| {ALL | SOME | ANY} (<select_lista>)
| EXISTS (<select_wyrażenie>)
| SINGULAR (<select_wyrażenie>)
| <war> [NOT] CONTAINING <war>
| <war> [NOT] STARTING [WITH] <war>
| (<warunek>)
| NOT <warunek>
| <warunek> OR <warunek>
| <warunek> AND <warunek>

<war> = { kolumna [<wymiar_tablicy>] | :zmienna
| <stała> | <wyrażenie> | <funkcja>
| udf ([<war> [, <war> …]])
| NULL | USER | RDB$DB_KEY }
[COLLATE NazwaPorządkuSortowania]

<stała> = liczba | literał | NazwaZestawuZnaków literał

<funkcja> = COUNT (* | [ALL] <war> | DISTINCT <war>)
| SUM ([ALL] <war> | DISTINCT <war>)
| AVG ([ALL] <war> | DISTINCT <war>)
| MAX ([ALL] <war> | DISTINCT <war>)
| MIN ([ALL] <war> | DISTINCT <war>)
| CAST (<war> AS <typ_danych>)
| UPPER (<war>)
| GEN_ID (generator, <war>)

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

<select_jedna> = zapytanie, wybierające (SELECT) jedną kolumnę; zwraca dokładnie jedną wartość,

<select_lista> = zapytanie, wybierające (SELECT) jedną kolumnę; zwraca zero lub wiele wartości,

<select_expr> = zapytanie, wybierające (SELECT) listę wartości; zwraca zero lub wiele wartości.

Przykład 5.14

Pokażemy teraz przykład wykorzystania warunku poprawności zdefiniowanego na poziomie tabeli:

```
CREATE TABLE Towary
(
Id_Towaru INTEGER NOT NULL PRIMARY KEY,
Cena_Hurtowa NUMERIC (7, 2) NOT NULL,
Cena_Detaliczna NUMERIC (7, 2) NOT NULL,
CHECK (Cena_Detaliczna >= Cena_Hurtowa)
);
```

W tym przykładzie warunkiem jest *Cena_Detaliczna* >= *Cena_Hurtowa*.

Ostatnią opcją polecenia tworzenia tabel jest `EXTERNAL FILE`. Pozwala ona na utworzenie tabeli, której dane przechowywane są w zewnętrznej tabeli lub w pliku. Użycie opcji `EXTERNAL FILE` umożliwia:

- importowanie danych z pliku zawierającego pola o stałym rozmiarze do nowej lub istniejącej tabeli bazodanowej;
- użycie zapytania wybierającego (`SELECT`) w odniesieniu do pliku, tak jakby był on tabelą serwera *InterBase*;
- eksportowanie danych z istniejącej tabeli bazodanowej do pliku.

Polecenie tworzenia domeny – CREATE DOMAIN

Definicja 5-3

Domena jest globalną definicją typu kolumny.

Wykorzystuje się ją, gdy kilka kolumn posiada identyczne (przynajmniej w pewnej części) definicje. Pozwala to zaoszczędzić czas, jak również ułatwia zarządzanie definicjami kolumn. Pola bazujące na domenie dziedziczą wszystkie zawarte w niej cechy, jednocześnie mogąc niektóre z nich zmienić.

Domeny tworzymy za pomocą polecenia `CREATE DOMAIN`, którego składnia jest następująca.

```
CREATE DOMAIN NazwaDomeny [AS] <typ_danych>
  [DEFAULT {literał | NULL | USER}]
  [NOT NULL] [CHECK (<warunek>)]
  [COLLATE NazwaPorządkuSortowania];
```

Listing 5.3

Tworząc *domenę*, musimy podać jej nazwę, która będzie unikalna w obrębie wszystkich domen, oraz jej typ. Oprócz tego mamy do dyspozycji takie parametry, jak:

- wartość domyślna lub `NULL`;
- warunki poprawności (`CHECK`);
- porządek sortowania (`COLLATE`).

Typ danych definiuje zbiór wartości, jakie może przyjmować pole. Określa on również zestaw operacji, jakie mogą być wykonywane na wartościach pola, oraz rozmiar, jaki

ono zajmuje. Typu danych zdefiniowanego dla danej domeny nie można zastąpić innym typem w definicji kolumny.

Przykład 5.15

Przykładowa definicja domeny *Data_Urodzenia* typu `DATE` mogłaby wyglądać następująco:

```
CRAETE DOMAIN Data_Urodzenia DATE;
```

W definicji *domeny* możemy również użyć wartości domyślnych. Stanowią one źródło danych dla pól, do których nie została wprowadzona żadna wartość. W takim przypadku pola te przyjmą wartości domyślne (oczywiście, o ile zostały one zdefiniowane). Wartości domyślne zabezpieczają kolumnę przed wprowadzeniem błędnych danych związanych z niepodaniem odpowiedniej wartości danego pola. Przykładowo: dla pól przechowujących wartości typu `DATE` wartością domyślną może być dzisiejsza data, natomiast dla pól przechowujących tylko wartości (*TAK*/*NIE*) wartością domyślną może być *TAK*.

Domyślne wartości mogą należeć do następujących kategorii:

- literały – wartości podane w sposób jawny (łańcuchy znaków, liczby, daty);
- `NULL` – wartością domyślną jest NULL;
- `USER` – nazwa bieżącego użytkownika.

Przykład 5.16

Oto przykładowe definicje *domeny* z wartościami domyślnymi:

```
CREATE DOMAIN DNazwaUzytkownika VARCHAR(20) DEFAULT USER;
CREATE DOMAIN DIlosc SMALLINT DEFAULT 5;
```

Wartości domyślne zdefiniowane dla danej domeny mogą zostać zmienione w lokalnej definicji kolumny.

Wartości pola danej *domeny* możemy zabezpieczyć przed wprowadzaniem wartości NULL za pomocą parametru `NOT NULL`. Jeżeli ten parametr nie zostanie określony, to domyślnie wartość NULL jest poprawną wartością pola. Parametr `NOT NULL` zdefiniowany w domenie nie może zostać zmieniony w lokalnej definicji kolumny. Należy pamiętać o tym, aby nie tworzyć warunków wzajemnie się wykluczających, związanych z ograniczeniem `NOT NULL` i wartością domyślną NULL, np.:

```
CREATE DOMAIN NazwaDomeny INTEGER DEFAULT NULL NOT NULL;
```

W obrębie *domeny* możemy również zdefiniować ograniczenia, jakim będą podlegały wprowadzane wartości. Ograniczenia definiujemy, wykorzystując konstrukcję `CHECK()`. Pozwala ona określić zestaw warunków, który sprawdza, czy dana wartość jest poprawna. Jeżeli dla danej wartości warunki są prawdziwe (`true`), to jest ona poprawna i zostanie wstawiona do pola. W przeciwnym wypadku wartość jest niepoprawna i nie zostanie dodana do pola. Zostanie za to zgłoszony wyjątek informujący o naruszonym ograniczeniu.

Składnia konstrukcji `CHECK` jest następująca:

```
<warunek_domeny> = {
VALUE <operator> <wartość>
| VALUE [NOT] BETWEEN <wartość> AND <wartość>
| VALUE [NOT] LIKE <wartość> [ESCAPE <wartość>]
| VALUE [NOT] IN (<wartość> [ , <wartość> …])
| VALUE IS [NOT] NULL
```

```
| VALUE [NOT] CONTAINING <wartość>
| VALUE [NOT] STARTING [WITH] <wartość>
| (<warunek_domeny>)
| NOT <warunek_domeny>
| <warunek_domeny>OR <warunek_domeny>
| <warunek_domeny> AND <warunek_domeny>
}
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Definiując warunki, należy pamiętać o kilku ograniczeniach:

- warunki nie mogą odwoływać się do innej domeny czy kolumny;
- domena może posiadać tylko jedną definicję warunku (CHECK);
- nie można zmienić warunków w lokalnej definicji kolumny, można natomiast dodać nowe.

Konstrukcja CHECK posiada słowo VALUE, które oznacza bieżąca wartość wprowadzaną do pola. Zatem VALUE można rozpatrywać jako parametr formalny funkcji, poprzez który odnosimy się do danej wartości. Należy pamiętać o tym, że jeżeli definicja domeny pozwala na wprowadzenie wartości NULL, to należy wziąć to pod uwagę w definiowanych warunkach.

Przykład 5.17

Oto przykłady ilustrujące powyższe zalecenie:

```
CHECK ((VALUE IS NULL) OR (VALUE > 5));
```

lub definicja *domeny* wykorzystującej operator IN:

```
CREATE DOMAIN DPlec AS VARCHAR(9)
CHECK (VALUE IN ('Kobieta', 'Mężczyzna'));
```

Domena *DPlec* to oczywiście odpowiednik polskiego słowa *płeć* (z pierwszą literą *D*, która jest pewną konwencją nazewnictwa). Musieliśmy użyć odpowiedniego zastępnika, gdyż nazwy obiektów mogą zawierać tylko litery alfabetu angielskiego.

Parametr COLLATE pozwala określić porządek sortowania dla typów znakowych. Może on zostać wybrany z listy dostępnych wartości dla danego zestawu znaków. Domyślnie zestaw znaków jest tożsamy z zestawem znaków bazy danych, jednak może on zostać zmieniony w definicji domeny.

Przykład 5.18

Poniższy przykład pokazuje definicję domeny *DNazwisko* dla języka polskiego:

```
CREATE DOMAIN DNazwisko AS
VARCHAR(30) CHARACTER SET WIN1250 COLLATE PXW_PLK;
```

Parametr COLLATE może zostać zmieniony w lokalnej definicji kolumny.

5.2.2. Zdanie ALTER

Polecenie modyfikacji bazy danych – ALTER DATABASE

Polecenia `ALTER DATABASE` używa się w celu dodania jednego lub więcej drugorzędnych plików do istniejącej bazy danych. Polecenie to może być wykonane przez użytkownika, który utworzył bazę danych, użytkownika SYSDBA lub dowolnego użytkownika z uprawnieniami administratora BD.

Składnia tego polecenia jest następująca:

```
ALTER {DATABASE | SCHEMA}
  ADD <add_klauzula>;
  <add_klauzula> = FILE 'NazwaPliku' <info_pliku> [<add_klauzula>]
  <info_pliku> = {LENGTH [=] liczba [PAGE[S]] | STARTING [AT [PAGE]]
liczba }
  [<info_pliku>];
```

Listing 5.4

Podczas dodawania nowych plików należy określić rozmiar pliku (parametr `LENGTH`) lub numer strony, od której będzie rozpoczynał się następny plik (parametr `STARTING [AT [PAGE]]`).

Przykład 5.19
Ten przykład dodaje do istniejącej bazy *Firma*, z którą jesteśmy aktualnie połączeni, dwa nowe pliki:

```
ALTER DATABASE Firma
  ADD FILE 'Firma2.gdb' STARTING AT PAGE 1001
  ADD FILE 'Firma3.gdb' STARTING AT PAGE 2001;
```

Polecenie modyfikacji bazy tabeli – ALTER TABLE

Do zmiany struktury istniejącej tabeli służy polecenie `ALTER TABLE`. Pozwala ono na:

- dodanie nowej kolumny;
- usunięcie istniejącej kolumny;
- usunięcie warunku integralności kolumny lub tabeli;
- zmianę nazwy, typu lub pozycji kolumny.

Tabela może zostać zmodyfikowana przez użytkownika, który ją utworzył, użytkownika SYSDBA lub przez dowolnego użytkownika z uprawnieniami administratora systemu.

Przed usunięciem lub modyfikacją istniejącej kolumny należy:

- sprawdzić, czy posiadane przywileje pozwalają na wykonanie danej operacji;
- zapisać istniejące dane;
- usunąć wszelkie ograniczenie nałożone na daną kolumnę.

Przed modyfikacją kolumny dane w niej zawarte powinny zostać zapisane. W przeciwnym wypadku zostaną one utracone. Proces zachowywania danych może zostać ujęty w pięciu punktach:

- dodać do modyfikowanej tabeli nową kolumnę, której typ jest tożsamy z typem modyfikowanej kolumny;

- przekopiować dane z modyfikowanej kolumny do kolumny tymczasowej (nowo utworzonej);
- zmodyfikować daną kolumnę;
- przekopiować dane z kolumny tymczasowej do zmodyfikowanej kolumny;
- usunąć kolumnę tymczasową.

Omówimy teraz szczegółowo polecenie ALTER TABLE. Jedną z jego możliwości jest dodanie (do istniejącej tabeli) nowej kolumny. Składnia polecenia dodającego nową kolumnę jest następująca:

```
ALTER TABLE NazwaTabeli ADD <definicja_kolumny>;


<definicja_kolumny> = kolumna { <typ_danych> | [COMPUTED [BY]
(<wyrażenie>) | domaena}
[DEFAULT {literał | NULL | USER}]
[NOT NULL] [<kol_warunek_integralności>]
[COLLATE NazwaPorządkuSortowania]


<kol_warunek_integralności> = [CONSTRAINT NazwaWarunku]
<kol_definicja_warunku>
[ <kol_warunek_integralności>]


<kol_definicja_warunku> = {PRIMARY KEY | UNIQUE
| CHECK (<warunek>)
| REFERENCES TabelaNadrzędna [( kolumna1 [, kolumna2 …])]
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]}
```

Listing 5.5

Przykład 5.20

Poniższy przykład dodaje nową kolumnę *Wiek* do tabeli *Pracownicy*:

```
ALTER TABLE Pracownicy ADD Wiek INTEGER NOT NULL;
```

Jednocześnie (w obrębie tego samego polecenia) można dodać do istniejącej tabeli kilka kolumn, oddzielając je przecinkami.

Polecenia ALTER TABLE możemy użyć do dodania nowego warunku integralności na poziomie tabeli. Składnia tego polecenia jest następująca:

```
ALTER TABLE Name ADD [CONSTRAINT constraint] <tab_definicja_warunku>;
<tab_definicja_warunku> = {{PRIMARY KEY | UNIQUE} ( kolumna1 [,
kolumna2 …])
| FOREIGN KEY ( kolumna1 [, kolumna2 …]) REFERENCES tabela_nadrzędna
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK ( <warunek>)};
```

Listing 5.6

Rozważmy przykład.

Przykład 5.21

Poniższy przykład nakłada na kolumnę *Nazwa* w tabeli *Towary* warunek unikalności (UNIQUE).

```
ALTER TABLE Towary
ADD CONSTRAINT CON_Nazwa UNIQUE(Nazwa);
```

Polecenie ALTER TABLE możemy również wykorzystać do modyfikacji kolumny danej tabeli. Składnia powyższego polecenia jest następująca:

```
ALTER TABLE NazwaTabeli ALTER [COLUMN] NazwaKolumny <typ_modyfikacji>
<typ_modyfikacji>=nowa_nazwa_kolumny|nowy_typ_kolumny|
nowa_pozycja_kolumny
nowa_nazwa_kolumny = TO NowaNazwaKolumny
nowy_typ_kolumny = TYPE {<typ_danych> | NazwaDomeny}
nowa_pozycja_kolumny = POSITION liczba;
```

Listing 5.7

Pokażemy teraz trzy przykłady wykorzystania powyższego polecenia.

Przykład 5.22

Pierwszy przykład zmienia nazwę kolumny:

```
ALTER TABLE Towary ALTER Nazwa TO Nazwa_Towaru;
```

Drugi przykład zmienia typ kolumny:

```
ALTER TABLE Studenci ALTER Imię TYPE VARCHAR(25);
```

Trzeci przykład zmienia pozycję pola *Imię*:

```
ALTER TABLE Studenci ALTER Imię POSITION 3;
```

Zmiana typu kolumny z nieznakowej na znakową podlega następującym regułom:

- typ BLOB oraz typy tablicowe nie podlegają konwersji;
- typy pól nie mogą zostać skrócone;
- nowy typ pola musi być zdolny do przechowywania całej zawartości danego pola.

Nie jest możliwa konwersja pól znakowych na inne typy (nieznakowe).

W celu usunięcia istniejącej kolumny wykorzystujemy polecenie ALTER TABLE. Uprawnienia do tej operacji posiada jedynie użytkownik, który utworzył tabelę. Składnia tego polecenia jest następująca:

```
ALTER TABLE NazwaTabeli DROP kolumna1 [, kolumna2 ...];
```

Przykład 5.23

Polecenie usuwa kolumnę *Wiek* z tabeli *Pracownicy*.

```
ALTER TABLE Pracownicy DROP Wiek;
```

Nie jest możliwe usunięcie lub modyfikacja kolumny, która jest częścią klucza głównego (PRIMARY KEY), częścią *klucza kandydującego* (UNIQUE) lub częścią *klucza obcego* (FOREIGN KEY). Również nie można zmodyfikować lub usunąć kolumny, która występuje w jakimkolwiek kontekście (np. warunki integralności, widoki, procedury składowane, wyzwalacze itp.).

Polecenie ALTER TABLE można też wykorzystać do usunięcia warunków nałożonych na kolumnę. Składnia tego polecenia jest następująca:

```
ALTER TABLE NazwaTabeli
DROP CONSTRAINT NazwaWarunku;
```

Listing 5.8

Przykład 5.24

Poniższy przykład usuwa warunek integralności *Con_Fak* nałożony na tabelę *Faktury*:

```
ALTER TABLE Faktury
DROP CONSTRAINT Con_Fak;
```

Nazwy warunków integralności są przechowywane w tabeli systemowej RDB$RELATION_CONSTRAINTS. Nie jest możliwe usunięcie kolumny, która jest wykorzystywana przez warunki `CHECK`. Dlatego też, aby usunąć taką kolumnę, należy najpierw usunąć wykorzystujący ją warunek.

Polecenie modyfikacji domeny – ALTER DOMAIN

Polecenie modyfikacji domeny pozwala zmienić wszystkie parametry z wyjątkiem parametru `NOT NULL`. Zmiany dokonane w definicji domeny znajdują swoje odzwierciedlenie we wszystkich kolumnach z niej korzystających (oczywiście, jeżeli parametry te nie zostały zmienione w danej kolumnie). Domena może zostać zmodyfikowana przez użytkownika, który ją utworzył, użytkownika SYSDBA lub przez dowolnego użytkownika z uprawnieniami administratora systemu operacyjnego. Do modyfikacji domeny służy polecenie `ALTER DOMAIN`, którego składnia jest następująca:

```
ALTER DOMAIN NazwaDomeny {
[SET DEFAULT {literał | NULL | USER}]
| [DROP DEFAULT]
| [ADD [CONSTRAINT] CHECK (<warunek>)]
| [DROP CONSTRAINT]
| Nowa_NazwaDomeny
| TYPE typ_danych
};
```

Listing 5.9

Polecenie modyfikacji domeny pozwala:

- usunąć zdefiniowane wartości domyślne;
- zdefiniować nową wartość domyślną;
- usunąć istniejące warunki;
- dodać nowe warunki;
- zmienić nazwę lub typ domeny.

Przykład 5.25

Pokażemy teraz trzy przykłady zastosowania polecenia `ALTER DOMAIN`:

```
ALTER DOMAIN Ilosc SET DEFAULT 3;
ALTER DOMAIN Nazwisko TO Imię;
ALTER DOMAIN Nazwisko TYPE CHAR(50);
```

Pierwszy przykład modyfikuje wartość domyślną, drugi nazwę domeny, a trzeci jej typ.

5.2.3. Zdanie DROP

Polecenie usuwania bazy danych – DROP DATABASE

Polecenie `DROP DATABASE` usuwa całą bazę danych, do której jesteśmy aktualnie podłączeni, wraz ze związanymi z nią cieniami (ang. *shadows*) i plikami dzienników (ang. *log files*). Polecenie to może być wykonane przez użytkownika, który utworzył bazę, użytkownika SYSDBA lub dowolnego użytkownika z uprawnieniami administratora systemu.

Ogólny wygląd polecenia:

```
DROP DATABASE [NazwaBD];
```

Listing 5.10

Przykład 5.26

Następujący przykład usuwa bieżącą bazę danych:

```
DROP DATABASE;
```

polecenie:

```
DROP DATABASE Towary;
```

usuwa bazę *Towary*.

Polecenie usuwania tabeli – DROP TABLE

Polecenie `DROP TABLE` usuwa całą tabelę. Jego składnia jest następująca:

```
DROP TABLE NazwaTabeli;
```

Listing 5.11

Przykład 5.27

Przykład usuwa tabelę *Towary*:

```
DROP TABLE Towary;
```

Polecenie `DROP TABLE` usuwa dane, metadane oraz indeksy danej tabeli. Tabela może być usunięta przez użytkownika, który ją utworzył, użytkownika SYSDBA lub dowolnego użytkownika z uprawnieniami administratora systemu operacyjnego. Nie jest możliwe usunięcie tabeli, która występuje w jakimkolwiek kontekście (warunki integralności, widoki, procedury składowane, wyzwalacze itp.).

Polecenie usuwania domeny – DROP DOMAIN

Polecenie `DROP DOMAIN` służy do usunięcia domeny z bazy danych. Jeżeli jednak domena jest wykorzystywana przez którąkolwiek z kolumn, to takiej domeny nie można usunąć. Należy zatem najpierw zmodyfikować tabelę, której kolumna korzysta z danej domeny, poprzez usunięcie nazwy domeny z definicji kolumny. Domena może zostać usunięta przez użytkownika, który ją utworzył, użytkownika SYSDBA lub przez dowolnego użytkownika z uprawnieniami administratora systemu operacyjnego.

Składnia polecenia `DROP DOMAIN` jest następująca:

```
DROP DOMAIN NazwaDomeny;
```

Listing 5.12

Przykład 5.28

Następujący przykład usuwa domenę *Nazwisko*:

```
DROP DOMAIN Nazwisko;
```

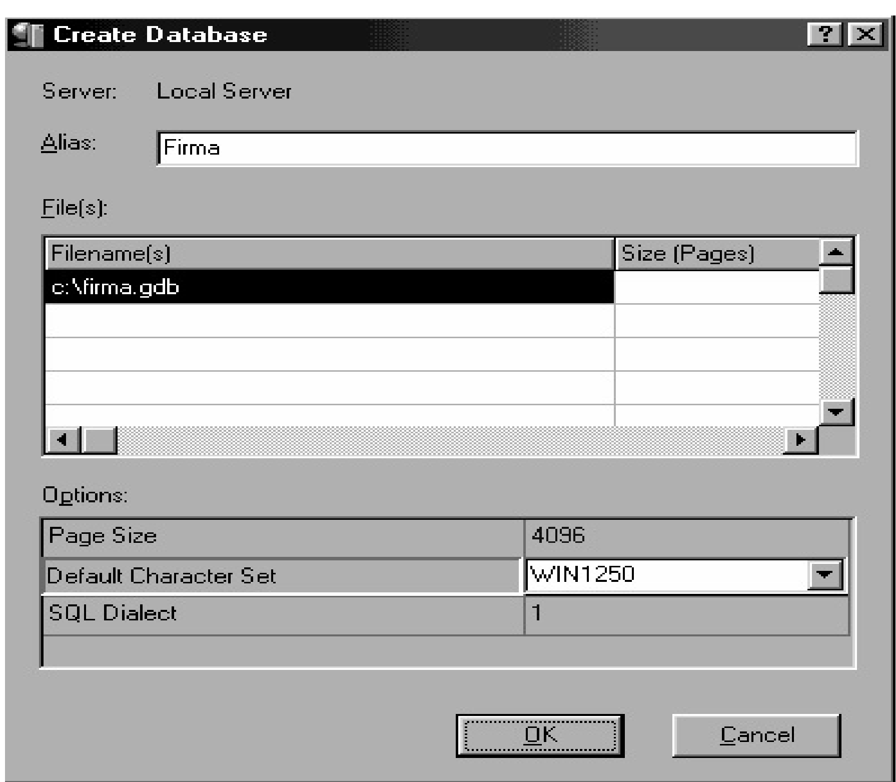
5.2.4. Operacje nad BD w IBConsole

Powyższe trzy polecenia nad BD (`CREATE DATABASE`, `ALTER DATABASE`, `DROP DATABASE`) mogą zostać wykonane poprzez narzędzie *IBConsole*. Dostarcza ono okienkowego interfejsu do wykonywania powyższych operacji. *IBConsole* uruchamiamy poprzez menu *Start/Programy/InterBase/IBConsole*.

Po zalogowaniu do odpowiedniego serwera klikamy w gałąź *Databases* prawym klawiszem myszy. Z podręcznego menu wybieramy opcję *Create Database...* (rys. 5.11).

Rys.5.11. Okno dialogowe *Databases*

Tworzenie BD możemy również rozpocząć poprzez menu *Database/Create Database...*. Następnie dostajemy okno, służące do wybrania odpowiednich parametrów nowo tworzonej bazy (rys. 5.12).



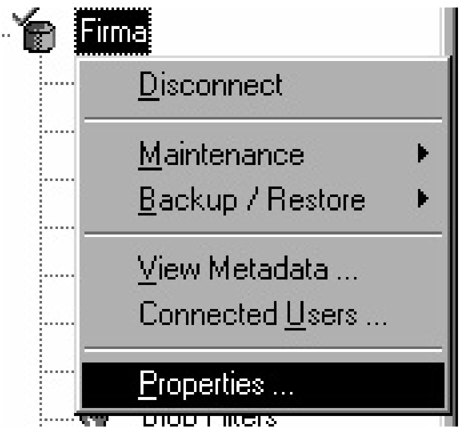
Rys. 5.12. Okno dialogowe *Create Database*

W pierwszym polu edycyjnym *Alias* wpisujemy *alias* (w naszym przykładzie *Firma*). Jest on unikalną nazwą, odnoszącą się do tworzonej bazy danych. Siatka danych *File(s)* składa się z dwóch kolumn. Pierwsza (*Filename(s)*) przechowuje listę plików, w których będzie przechowywana baza danych. Jeżeli zostanie podana tylko nazwa pliku (bez

ścieżki dostępu), to będzie on utworzony w katalogu roboczym. Druga kolumna (*Size (Pages)*) zawiera informacje o rozmiarach poszczególnych plików.

W dolnej części okna mamy do dyspozycji trzy opcje. Dwie pierwsze opcje zostały już omówione. Trzecia opcja (SQL *Dialect*) pozwala na zdefiniowanie dialektu języka SQL, jakiego będzie używała BD. Dialekt określa, jak będą interpretowane: znak cudzysłowu, dokładność dużych liczb oraz pewne typy, takie jak: `DATE`, `TIME`, `TIMESTAMP`. W większości przypadków powinno wybierać się *dialekt 3*. Umożliwia on korzystanie ze wszystkich funkcji serwera *InterBase*. Wciskając przycisk *OK*, tworzymy bazę danych o wprowadzonych parametrach.

Po utworzeniu bazy możemy dokonać kilku modyfikacji. Zmian dokonujemy, klikając prawym klawiszem myszy w odpowiedni *alias* bazy. Z menu podręcznego wybieramy pozycję *Properties* (rys. 5.13).



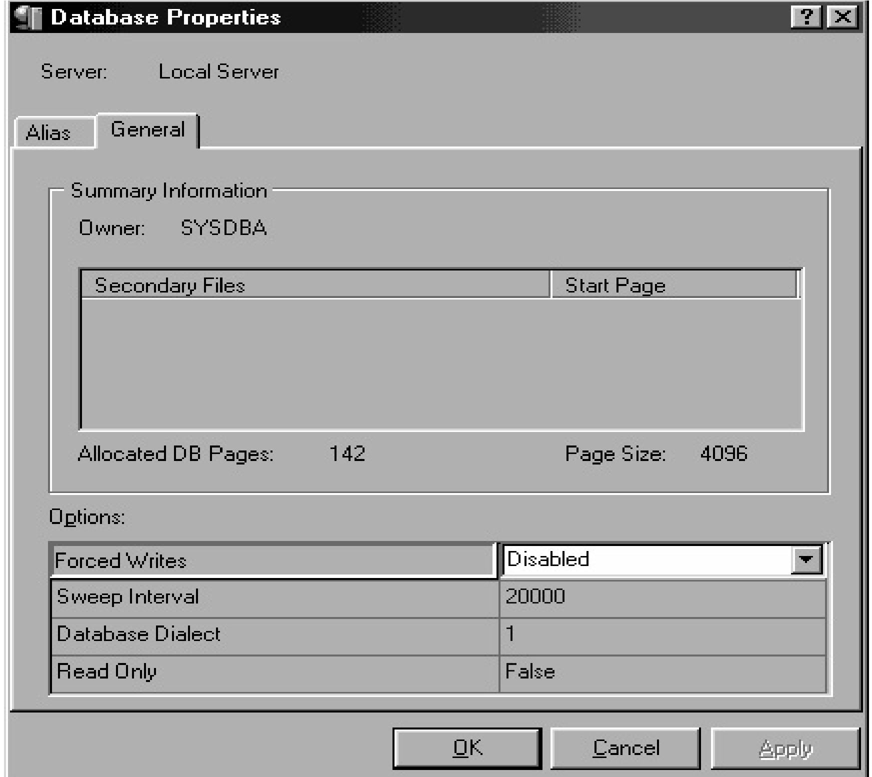
Rys. 5.13. Okno dialogowe do modyfikacji BD *Firma*

Wyboru tego możemy również dokonać poprzez menu *Database/Properties* (rys. 5.14). Po wybraniu powyższej opcji dostajemy okienko z dwoma zakładkami. Pierwsza *Alias* pozwala na zmianę *aliasu* oraz pliku, w którym przechowywana jest baza danych (aby zmienić plik, musimy rozłączyć się z bazą danych). Druga zakładka *General* zawiera cztery opcję, za pomocą których możemy zmodyfikować bieżącą bazę danych.

Pierwsza opcja (*Forced Writes*) określa, czy dane zapisywane do bazy są bezzwłocznie zapisywane do pliku. Jeżeli opcja jest aktywna (tzn. *Forced Writes = Enabled*), to dane zapisywane do bazy są bezpośrednio zapisywane w pliku. Natomiast jeżeli opcja jest nieaktywna (tzn. *Forced Writes = Disabled*), to dane zapisywane do BD są najpierw umieszczane w buforze. Dopiero kiedy bufor zostanie wypełniony, dane zostają zapisane do pliku bazy danych. Jeżeli jednak wystąpi poważny błąd przed tym, jak dane zostaną zapisane, to mogą one zostać utracone. Ustawienie opcji *Forced Writes* na *Enabled* zwiększa *bezpieczeństwo* danych, jednakże odbija się to na efektywności operacji zapisu.

Kolejna opcja to *Sweep Interval*. Pozwala ona określić liczbę zaistniałych transakcji, po których nastąpi automatyczne czyszczenie bazy danych. Operacja ta ma na celu opróżnianie pliku bazy danych ze starych rekordów. Działanie takie zapobiega nadmiernemu wzrostowi rozmiaru pliku, co może jednak powodować spadek efektywności. Ustawienie wartości na 0 wyłącza automatyczne czyszczenie pliku bazy.

Opcja *Database Dialect* została opisana poprzednio. Natomiast opcja *Read Only* określa, czy baza danych ma być tylko do odczytu. Ustawienie wartości na `true` zablokuje możliwość wprowadzania jakichkolwiek zmian.



Rys. 5.14. Okno dialogowe zakładki *Database/Properties*

Bazę danych możemy usunąć za pomocą polecenia z menu *Database/Drop Database*. Baza danych może być usunięta tylko przez użytkownika, który utworzył bazę lub przez użytkownika *SYSDBA*.

5.2.5. Typy danych serwera InterBase

Podczas definiowania kolumny w poleceniu tworzenia tabeli (patrz list. 5.2; Prz. 5.8 – 5.14) pierwszym atrybutem jest typ danych danego pola. Oprócz określenia zbioru danych, jakie mogą być zapisywane w danym polu, definiuje on również rodzaj możliwych do wykonania operacji na jego wartościach. Przykładowo: operacje arytmetyczne można wykonywać na polach numerycznych, co nie jest możliwe dla pól znakowych. Typ danych ma bezpośredni związek z ilością zajmowanego miejsca, dlatego też istotną kwestią jest ich odpowiedni wybór.

Warto również przypomnieć, że na podobne szczegóły należy zwracać uwagę również przy definiowaniu domeny (patrz list. 5.3; Prz. 5.16 – 5.17) i przy modyfikacji tabel (list. 5.5, 5.6).

W ten sposób typy danych wykorzystywane są w następujących sytuacjach:

- tworzenia tabel;
- modyfikacji kolumn tabeli;
- tworzenia domen;
- modyfikacji domen.

O podstawowych typach danych, omówionych w standardach ANSI SQL, wspomnieliśmy w podrozdz. 4.1.1. Serwer *InterBase* używa nieco różniących się w detalach typów danych niż podane w tab. 4.1 i 4.2.

Serwer *InterBase* wspiera następujące typy danych:

- `INTEGER` i `SMALLINT` – typy całkowitoliczbowe;
- `FLOAT` i `DOUBLE PRECISION` – typy zmiennoprzecinkowe;
- `NUMERIC` i `DECIMAL` – typy stałoprzecinkowe;
- `DATE`, `TIME` i `TIMESTAMP` – typy daty i czasu;
- `CHARACTER` i `CHARACTER VARYING` – typy znakowe;
- `BLOB` – typ dużych danych binarnych.

InterBase dostarcza typ `BLOB`, który służy do przechowywania danych niedających się zakwalifikować do żadnego ze standardowych typów danych. Jest on używany do przechowywania dużych obiektów o zmiennych rozmiarach, takich jak: pliki graficzne, pliki dźwiękowe, pliki wideo, duże ilości tekstu oraz wszystkie inne dane multimedialne.

InterBase również wspiera typ tablicowy. Jest on wielowymiarowym (1-16) wektorem złożonym z elementów dowolnego typu (wyłączając typ `BLOB`).

Typ `TIMESTAMP` przechowuje informacje o roku, miesiącu, dniu oraz czasie. `DATE` przechowuje informacje o roku, miesiącu i dniu. Natomiast typ `TIME` zawiera informacje o godzinie, minucie, sekundzie, dziesiątej, setnej oraz tysięcznej części sekundy.

Tabela 5.7 przedstawia typy danych serwera *InterBase*.

Tabela 5.7. Typy danych serwera *InterBase*

Nazwa typu	*Rozmiar*	*Zakres/Precyzja*	*Opis*
`BLOB`	Zmienny	Brak. Rozmiar segmentu jest ograniczony do 64KB	Rozmiar pola zmienia się dynamicznie. Jednostką pola jest segment. Podtyp (*subtype*) określa zawartość pola
`CHAR (n)`	`n` znaków	Od 1 do 32 767 bajtów	Stała liczba znaków Alternatywne słowo: `CHAR`
`DATE`	64 bity	od 1-1-100 do 29-02-32 768	
`DECIMAL(p,s)`	Zmienny (16, 32 lub 64 bity)	p (precyzja – liczba wszystkich cyfr) – od 1 do 18 S (skala – liczba cyfr po przecinku) – od 0 do 18	Przykładowo: typ `DECIMAL(7,3)` przechowuje liczbę w formacie `pppp.sss`
`DOUBLE PRECISION`	64 bity	Od 2.225x10-308 do 1797x10308	Przechowuje 15 cyfr
`FLOAT`	32 bity	Od 1.175x10-38 do 3.402x1038	Przechowuje 7 cyfr
`INTEGER`	32 bity	Od -2 147 483 648 do 2 147 483 647	Alternatywne słowo: `INT`
`NUMERIC(p,s)`	Zmienny (16, 32 lub 64 bity)	p (precyzja – ilość wszystkich cyfr) – od 1 do 18 S (skala – ilość cyfr po przecinku) – od 0 do 18	Przykładowo: typ `NUMERIC(7,3)` przechowuje liczbę w formacie `pppp.sss`
`SMALLINT`	16 bitów	Od -32 768 do 32 767	

Nazwa typu	*Rozmiar*	*Zakres/Precyzja*	*Opis*
`TIME`	64 bity	Od 0:00 do 23:59.9999	
`TIMESTAMP`	64 bity	Od 1-1-100 do 29-02-32 768	Przechowuje również informacje o czasie
`VARCHAR (n)`	n znaków	Od 1 do 32 765 bajtów	Zmienna liczba znaków. Alternatywne słowa: `CHAR VARYING`, `CHARACTER VARYING`

Składnia SQL, opisująca wykorzystanie typów danych, jest następująca:

```
<typ_danych> =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[ <wymiar_tablicy>]
| {DATE | TIME | TIMESTAMP} [ <wymiar_tablicy>]
| {DECIMAL | NUMERIC} [( precyzja [, skala])] [ <wymiar_tablicy>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(liczba)]
[ <wymiar_tablicy>] [CHARACTER SET NazwaZestawuZnaków]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(liczba)] [ <wymiar_tablicy>]
| BLOB [SUB_TYPE { liczba | nazwa_podtypu}] [SEGMENT SIZE liczba]
[CHARACTER SET NazwaZestawuZnaków]
| BLOB [(wielkość_segmentu [,nazwa_podtypu])]
<wymiar_tablicy> = [x:y [, x1:y1 …]].
```

Listing 5.13

Omówimy pokrótce podstawowe typy danych *InterBase*.

Typy numeryczne

Do typów numerycznych zaliczamy: typy całkowitoliczbowe (`INTEGER`, `SMALLINT`), typy stałoprzecinkowe (`DECIMAL`, `NUMERIC`) oraz typy zmiennoprzecinkowe (`FLOAT`, `DOUBLE PRECISION`).

Na wartościach należących do tych typów można wykonywać następujące operacje:

- operacje porównywania, wykorzystujące standardowe operatory relacji (`<`, `>`, `=`, `<=`, `>=`); możliwe jest też wykorzystanie operatorów `CONTAINING`, `STARTING WITH` oraz `LIKE`;
- działania arytmetyczne (`+`, `-`, `*`, `/`);
- konwersje – podczas wykonywania operacji arytmetycznych *InterBase* automatycznie konwertuje dane pomiędzy typami `INTEGER`, `FLOAT`, `CHAR`. W trakcie wykonywania operacji porównania danych numerycznych z danymi innym typów *InterBase* dokonuje konwersji tych drugich do typów numerycznych;
- na wartościach typów numerycznych można wykonywać operacje sortowania.

InterBase wspiera dwa typy liczb stałoprzecinkowych: `DECIMAL` oraz `NUMERIC`. Typy te wykorzystuje się do przechowywania wartości ze stałą liczbą miejsc po przecinku (np. przechowywanie wartości monetarnych).

Podczas definiowania typów stałoprzecinkowych można użyć dwóch parametrów precyzji i skali. Precyzja określa całkowitą liczbę wszystkich cyfr (od 1 do 18), skala nato-

miast określa liczbę cyfr po przecinku (od 0 do 18). Parametr skala nie może być większy niż precyzja. Składnia deklaracji typów jest następująca:

- `NUMERIC [( precyzja [, skala])];`
- `DECIMAL [(precyzja [, skala])].`

Deklaracje oraz cechy obu tych typów są bardzo podobne. Powiedzmy teraz o różnicach w ich wykorzystaniu. Deklaracja `NUMERIC (p, s)` oznacza, że taki typ jest zdolny do przechowywania liczby o dokładnie `p` cyfrach, zawierającej `s` cyfr po przecinku. Natomiast deklaracja `DECIMAL (p, s)` określa, że taki typ jest zdolny do przechowywania liczby o co najmniej `p` cyfrach (tzn. może przechowywać liczby o większej liczbie cyfr), zawierającej `s` cyfr po przecinku.

Podczas tworzenia kolumn typu `NUMERIC` lub `DECIMAL` *InterBase* określa typ, w jakim będą wewnętrznie przechowywane dane. Określenie typu bazuje na wartościach precyzji i skali (istotną rolę odgrywa również dialekt bazy danych). Jeżeli deklarujemy powyższe typy bez parametrów precyzja i skala, to są one wewnętrznie przechowywane jako liczby typu `INTEGER`. Jeżeli deklarujemy typy `NUMERIC` lub `DECIMAL` z parametrem precyzja, ale bez podania parametru skala, to będą one przechowywane wewnętrznie jako typy: `SMALLINT`, `INTEGER`, `DOUBLE PRECISION` lub `INT64` (64-bitowa liczba całkowita). Wpływ parametru precyzja i dialektu bazy danych na typy wykorzystywane do wewnętrznego przechowywania wartości przedstawia tab. 5.8.

Tabela 5.8. Typy danych serwera *InterBase* wykorzystywane do wewnętrznego przechowywania wartości

Precyzja	*Dialekt 1*	*Dialekt 3*
Od 1 do 4	Dla typu `NUMERIC` – `SMALLINT` dla typu `DECIMAL` – `INTEGER`	`SMALLINT`
Od 5 do 9	`INTEGER`	`INTEGER`
Od 10 do 18	`DOUBLE PRECISION`	`INT64`

Tabela 5.9 przedstawia w sposób bardziej dokładny wpływ obu parametrów precyzji i skali na typy wykorzystywane do wewnętrznego przechowywania wartości.

Tabela 5.9. Wpływ parametrów precyzji i skali na typy wykorzystywane do wewnętrznego przechowywania wartości

Deklarowany typ danych	*Wewnętrzny typ danych*
`NUMERIC`	`INTEGER`
`NUMERIC (4)`	`SMALLINT`
`NUMERIC (9)`	`INTEGER`
`NUMERIC (10)`	`DOUBLE PRECISION` – *dialekt 1* `INT64` – *dialekt 3*
`NUMERIC (4, 2)`	`SMALLINT`
`NUMERIC (9, 3)`	`INTEGER`
`NUMERIC (10, 4)`	`DOUBLE PRECISION` – *dialekt 1* `INT64` – *dialekt 3*
`DECIMAL`	`INTEGER`
`DECIMAL (4)`	`INTEGER`
`DECIMAL (9)`	`INTEGER`

Deklarowany typ danych	*Wewnętrzny typ danych*
DECIMAL (10)	DOUBLE PRECISION – *dialekt 1* INT64 – *dialekt 3*
DECIMAL (4, 2)	INTEGER
DECIMAL (9, 3)	INTEGER
DECIMAL (10, 4)	DOUBLE PRECISION – *dialekt 1* INT64 – *dialekt 3*

Typy daty i czasu

InterBase wspiera trzy typy związane z datą i czasem: DATE, TIME oraz TIMESTAMP. Typ DATE umożliwia przechowywanie dat z zakresu od 1 stycznia 100 roku do 29 lutego 32768 roku. Pole tego typu zajmuje rozmiar 32 bitów. Typ TIME służy do przechowywania wartości czasu z zakresu od 00:00 do 23:59.9999. Pole tego typu zajmuje również 32 bity. Typ TIMESTAMP umożliwia przechowywanie jednocześnie informacji o dacie i czasie. Pole tego typu zajmuje rozmiar 64 bitów (2x32bity).

Wiele języków nie używa typów DATE, TIME i TIMESTAMP. Zamiast nich używają one łańcuchów znaków lub struktur do przechowywania wartości związanych z datą i czasem. W takim przypadku wymagane są mechanizmy konwersji pomiędzy typami *InterBase* a typami używanymi w danej implementacji języka. Dla przykładu pokażemy sposoby dokonywania takich konwersji dla pól typu DATE:

- daty w postaci łańcuchów w formacie DD-MMM-YY (jest to standardowy format daty używany w języku SQL), konwertowane są automatycznie do typu DATE. Trzyliterowy skrót miesiąca jest anglojęzyczny. Oprócz tego *InterBase* interpretuje również daty w dwóch innych formatach. Pierwszy z nich to format amerykański: MM/DD/YY lub MM/DD/YYYY. Drugi to format europejski: DD.MM.YY lub DD.MM.YYYY;
- *InterBase* dostarcza dwóch funkcji do konwersji z i do typu DATE. Pierwsza z nich – isc_decode_date() – konwertuje dane z wewnętrznego formatu daty serwera *InterBase* do struktury time języka C. Druga (isc_encode_date()) wykonuje zadanie odwrotne, czyli konwertuje datę ze struktury time języka C do formatu DATE;
- do konwersji danych typu DATE z i do typów znakowych używamy funkcji cast(). Data przekazywana do tej funkcji musi być w formacie YYYY-MM-DD. W tym samym formacie będzie również łańcuch reprezentujący datę zwrócony przez tę funkcję.

Typy znakowe

InterBase wspiera cztery typy znakowe:

- CHAR(n) lub CHARACTER(n) – typy o stałym rozmiarze pola, które przechowują zawsze n znaków;
- VARCHAR(n) lub CHARACTER VARYING(n) – typy o zmiennym rozmiarze pola, które przechowują co najwyżej n znaków;
- NCHAR(n), NATIONAL CHAR(n) lub NATIONAL CHARACTER(n) – typy o stałym rozmiarze pola, które używają zestawu znaków ISO8859_1;
- NCHAR VARYING(n), NATIONAL CHAR VARYING(n) lub NATIONAL CHARACTER VARYING(n) – typy o zmiennym rozmiarze pola, które używają zestawu znaków ISO8859_1.

Podczas tworzenia pól typów znakowych można użyć parametru `CHARACTER SET`. Pozwala on określić inny niż domyślny (zdefiniowany podczas tworzenia BD) zestaw znaków danego pola. Od zdefiniowanego zestawu znaków zależy liczba bajtów wykorzystywanych do przechowywania jednego znaku (1, 2 lub 3 bajty). Maksymalny rozmiar przeznaczony dla pól typów znakowych to 32 767 bajtów. Tak więc maksymalna liczba znaków możliwych do przechowywania w polu danego typu zależy od zestawu znaków używanych przez to pole.

Omówimy teraz poszczególne typy pól znakowych. Pierwsze z nich to pola o stałym rozmiarze (`CHAR(n)` lub `CHARACTER(n)`). Jeżeli do pola tego typu zostanie wprowadzony łańcuch znaków o rozmiarze mniejszym niż `n`, to „brakujące" znaki zostaną uzupełnione spacjami. Jeżeli natomiast rozmiar wprowadzanego łańcucha znaków jest większy niż `n`, to zostanie on ucięty. Jeżeli argument `n` nie zostanie podany, to przyjmie on wartość domyślną równą 1 (tzn. `CHAR = CHAR (1)`). *InterBase* kompresuje spacje, które wypełniają brakujące znaki. Podczas odczytywania wartości pola *InterBase* uzupełnia łańcuch odpowiednią liczbą spacji. Oznacza to, że pola tego typu będą zajmowały tyle samo miejsca co ich odpowiedniki o zmiennym rozmiarze. Typy `NCHAR(n)`, `NATIONAL CHAR(n)` lub `NATIONAL CHARACTER(n)` są polami o stałym rozmiarze z ustawionym domyślnie zestawem znaków ISO8859_1. Tak więc wszystkie powyższe uwagi odnoszą się również do tych pól.

Pola o zmiennym rozmiarze (`VARCHAR(n)` lub `CHARACTER VARYING(n)`) wykorzystuje się do przechowywania łańcuchów znaków o zmiennej długości nieprzekraczającej `n`. Parametr `n` jest obligatoryjny. Podczas konwersji danych tych typów do typów o stałym rozmiarze brakujące znaki uzupełniane są spacjami. `NCHAR VARYING(n)`, `NATIONAL CHAR VARYING(n)` lub `NATIONAL CHARACTER VARYING(n)` są polami o zmiennym rozmiarze z ustawionym domyślnie zestawem znaków ISO8859_1.

Typ BLOB

Typ `BLOB` służy do przechowywania danych niedających się zakwalifikować do żadnego ze standardowych typów SQL. Wykorzystuje się go do przechowywania dużych ilości danych, tj. plików graficznych, plików dźwiękowych, plików wideo czy plików sformatowanego tekstu. Dane typu `BLOB` mogą wykorzystywać wszystkie bazodanowe mechanizmy (np. *transakcje*). Zatem użycie pól tego typu niesie ze sobą wiele korzyści bez konieczności przechowywania wskaźników do plików zawierających odpowiednie dane. Podczas zapisywania wartości do pól typu `BLOB` *InterBase* zapisuje w nim tylko wartość zwaną BLOB ID, która jest unikalną wartością numeryczną odnoszącą się do właściwych danych. Dane te zapisywane jako ciąg segmentów są jednostkami pól typu `BLOB`. Domyślny rozmiar segmentu to 80 bajtów. Wartość tę można zmienić za pomocą polecenia `SEGMENT SIZE` (rozmiar podajemy w bajtach), mogącego wystąpić w definicji pola. Maksymalna wielkość segmentu to 32 KB (32 767 bajtów).

Podczas definiowania pól typu `BLOB` można określić podtyp za pomocą parametru `SUB_TYPE`. Jest on liczbą, która precyzuje naturę danych pola `BLOB`. *InterBase* dostarcza dwóch predefiniowanych podtypów: 0 – dane binarne i 1 – dane tekstowe. Użytkownik może określić własny podtyp. W tym celu musi on użyć liczb ujemnych, gdyż liczby dodatnie zarezerwowane są przez serwer *InterBase*. Domyślną wartością podtypu jest 0. Natomiast podczas definicji podtypu danych tekstowych zamiast stałej 1 możemy użyć słówka `TEXT`. *InterBase* nie sprawdza formatu danych zapisanych do pól typu `BLOB`. Zadanie to powinno być realizowane przez aplikację.

Konwersje pomiędzy typami danych

Serwer *InterBase* dokonuje konwersji w celu wykonania operacji na danych różnych typów. Operacje takie to, przykładowo, działania arytmetyczne czy porównania. Konwer-

sje wykorzystywane są również, gdy dany język programowania używa typów, które nie są obsługiwane przez serwer. Operacje konwersji mogą zachodzić automatycznie (konwersje domyślne) albo z wykorzystaniem funkcji `CAST()` (konwersje jawne).

Konwersja automatyczna zachodzi wówczas, gdy dana wartość jest nieadekwatna do typu wykonywanej operacji. Oczywiście nie każda konwersja jest możliwa.

Przykład 5.28

Oto przykłady wykonania konwersji:

`2 + '3' = 5` – konwersja automatyczna łańcucha '3' na liczbę 3;

`2 + 'a'` – błąd konwersji.

Możemy wykorzystać funkcję `CAST()`, która pozwala na sprawowanie większej kontroli nad operacją konwersji. Zazwyczaj funkcja ta jest używana w warunkach `WHERE` do porównania danych różnych typów. Składnia funkcji `CAST()` jest następująca:

```
CAST (wartosc | NULL as TypDanych);
```

Dzięki wykorzystaniu funkcji `CAST()` możemy dokonywać następujących konwersji:

- typy daty i czasu na typy znakowe;
- typy znakowe na typy daty i czasu;
- typ `TIMESTAMP` na typy `DATE` lub `TIME`;
- typy `DATE` lub `TIME` na typ `TIMESTAMP`.

Zadania do samokontroli

1. Scharakteryzować narzędzia tworzenia obiektów BD.
2. Zapisać ogólną postać poleceń SQL:
   - *CREATE DATABASE;*
   - *CREATE TABLE;*
   - *CREATE DOMAIN;*
   - *ALTER TABLE;*
   - *ALTER DOMAIN;*
   - *DROP DATABASE,*
   - *DROP TABLE;*
   - *DROP DATABASE,*

   a następnie wyjaśnić semantykę i składnię poleceń.

6. Kontrola dostępu do danych

Kontrola dostępu do BD odnosi się bezpośrednio do tak ważnego aspektu bazy danych, jakim jest *poziom bezpieczeństwa*.

Bezpieczeństwo jest ściśle związane z mechanizmami zabezpieczającymi bazę przed przypadkowym lub celowym zagrożeniem. W związku z tym kwestie dotyczące ochrony oraz bezpieczeństwa nie zawsze dotyczą bezpośrednio danych przechowywanych w bazie. Jest to związek z całym systemem, którego narażenie może obciążyć w skutkach bazę danych. Zatem zakres bezpieczeństwa bazy danych dotyczy oprogramowania, sprzętu, ludzi oraz danych.

W tym podrozdziale omówiony zostanie problem kontroli dostępu do BD od strony języka SQL. Za pomocą SQL możemy nie tylko operować danymi, lecz także kontrolować dostęp do nich.

Utrzymywanie bezpieczeństwa dostępu do informacji w bazie danych wymaga na początku opracowania pewnych zasad, reguł i planu przyznawania praw dostępu. Wszystkie aspekty dotyczące bezpieczeństwa powinny być zawarte w projekcie bazy danych.

6.1. Mechanizmy zabezpieczeń

6.1.1. Podstawowe pojęcia

Mówiąc o ograniczeniu dostępu, musimy wyjaśnić, do czego ograniczamy dostęp oraz komu lub czemu ograniczamy dostęp do danych. Otóż ograniczyć dostęp do obiektów możemy *użytkownikom*, *rolom*, *wyzwalaczom*[1] oraz *procedurom wbudowanym*. Natomiast obiekty, do których chcemy kontrolować dostęp, to: tabele, perspektywy, role i procedury. Aby dokładniej zgłębić metody nadawania praw dostępu, omówione zostaną następujące nowe terminy:

- *role*;
- *przywileje*;
- *perspektywy*.

Trzy wymienione pojęcia dotyczą kontroli wewnątrz bazy i mają znaczenie podczas działania aplikacji bazodanowej na pliku bazy. Zanim jednak do tego dojdzie, użytkownik musi poprawnie przejść próbę *identyfikacji*, inaczej *uwierzytelniania*. Potocznie

---

[1] Patrz podrozdz. 6.3. Wyzwalacze. Wyzwalacz (ang. *trigger*) jest to pewien rodzaj procedury wywoływanej automatycznie w momencie wystąpienia pewnego zdarzenia. Wyzwalacze mają za zadanie wykonanie określonej akcji w określonej chwili.

proces ten nazywany jest *logowaniem* do bazy. Dlatego w dalszej części tego podrozdziału opisany został właśnie temat *uwierzytelnienia użytkowników*.

Użytkownicy BD byli już wcześniej omówieni w podrozdz. 1.1.3.

W zasadzie polityka bezpieczeństwa i ochrony informacji w bazach danych opiera się na czterech podstawowych komponentach:

- ***użytkownicy*;**
- ***role*;**
- ***przywileje*;**
- ***perspektywy*.**

W celu przeprowadzenia dalszej analizy zdefiniujemy na wstępie kilka podstawowych pojęć.

Definicja 6-1

Uprawnienie *– posiadanie praw lub przywilejów uprawniających do legalnego dostępu do systemu i obiektów systemowych.*

Jak już wspomniano, część zabezpieczeń dotyczy zarówno dostępu do samej bazy, jak też dostępu do konkretnych obiektów tej bazy. Ponadto, jeśli chodzi o dostęp do obiektów BD, uprawnienie to dotyczy również zakresu tego dostępu. Dlatego też *kontrola uprawnień* nazywa się także ***kontrolą dostępu***.

Moment nadania lub uzyskania uprawnień do obiektów musi być poprzedzony rozpoznaniem *podmiotu* żądającego dostępu.

Definicja 6-2

Podmiotem (***subiektem***) jest zazwyczaj użytkownik, ale może to być również program, natomiast ***obiektem*** może być tabela, perspektywa, procedura, wyzwalacz lub inny obiekt utworzony w systemie.

Mechanizmem rozpoznawania podmiotów jest *identyfikacja*.

Definicja 6-3

Identyfikacja (ang. *identification*) polega na stwierdzeniu tożsamości podmiotu, inaczej jest to metoda rozpoznania, czy podmiot jest tym, za kogo się podaje.

W przypadku systemu komputerowego bardzo często nadawania uprawnień dokonuje administrator systemu, który tworzy *konta użytkowników*.

Użytkownik posiada *unikalny identyfikator,* za pomocą którego jednoznacznie jest rozpoznawany w systemie, oraz *hasło*, które system wykorzystuje do potwierdzenia jego tożsamości. Taka procedura niekoniecznie pozwala na dostęp do SZBD. Udzielenie prawa dostępu do bazy danych w SZBD może wyglądać analogicznie, lecz za pomocą odrębnych procedur.

Zazwyczaj za kontrolę dostępu odpowiedzialny jest administrator bazy danych, który tworzy konta użytkowników, przypisując im odpowiednie identyfikatory oraz hasła za pomocą mechanizmów SZBD.

Identyfikatory oraz hasła mogą być przechowywane w bazie BD (najczęściej). Inne rozwiązanie to wykorzystanie listy zawierającej użytkowników systemu operacyjnego oraz

identyfikatora aktualnego użytkownika. Taki sposób identyfikacji zabezpiecza przed zarejestrowaniem się użytkownika, którego nazwa nie jest aktualnie używana przez system operacyjny.

Definicja 6-4

Przywileje (ang. *privilege*) determinują prawa dostępu do obiektów BD, takich jak: tabele, perspektywy i indeksy. Dotyczą one także praw tworzenia, usuwania lub modyfikacji obiektów, jak również korzystania z funkcji, jakie zawiera SZBD.

Po otrzymaniu dostępu do SZBD użytkownik automatycznie może uzyskać szereg przywilejów. Odpowiednie przywileje zostaną mu przydzielone, w zależności od funkcji pełnionej przez użytkownika w danej instytucji.

Definicja 6-5

Perspektywą (***widokiem***) (ang. *view*) nazywamy wirtualną tabelę obliczaną dynamicznie na podstawie jednej lub kilku tabel bazowych, jak również na podstawie innej perspektywy lub perspektyw.

Perspektywa nie istnieje w rzeczywistości w bazie, gdyż jest tworzona na żądanie użytkownika w momencie wydania takiego polecenia.

Perspektywy pozwalają na elastyczne zabezpieczanie danych poprzez ukrywanie informacji przed określonymi użytkownikami. Użytkownik widzi tylko te dane z tabeli, które znajdują się w perspektywie, nie wiedząc o istnieniu innych danych w tej relacji. Oczywiście, perspektywa może korzystać z wielu tabel, a użytkownik posiadający dostęp do tej perspektywy nie musi posiadać dostępu do tabel bazowych. Jest to ważna cecha perspektyw w świetle ochrony danych.

Perspektywy są również pomocne, gdy użytkownik posiada pełne prawo do tabeli, lecz w danym momencie nie potrzebuje widoku całej tabeli. Można w takiej sytuacji posłużyć się perspektywą w celu pokazania tylko tych informacji, których w danej chwili potrzebuje użytkownik.

Definicja 6-6

Rola (ang. *role*) to uprawnienie lub zestaw uprawnień, które pozwalają użytkownikom wykonywać określone funkcje i operacje w bazie danych.

Role stanowią bardzo pomocny mechanizm w kontroli dostępu użytkowników do bazy danych. Istota działania ról polega na tym, że użytkownicy nie posiadają bezpośredniego dostępu do obiektów, lecz dostęp ten uzyskują poprzez role, co w rezultacie daje dużo większe możliwości kontroli dostępu do danych.

Dzięki rolom administrator BD nie nadaje każdemu użytkownikowi prawa dostępu do obiektów. Zamiast tego tworzy role reprezentujące odpowiednie funkcje w danej organizacji, a użytkownicy przypisywani są do tych ról, które odpowiadają ich kwalifikacjom i obowiązkom.

Każdego użytkownika możemy z łatwością przenieść z jednej roli do drugiej. W zależności od potrzeb w poszczególnych rolach mogą być usuwane lub dodawane przywileje. Dzięki temu kontrolę nad użytkownikami można łatwo dostosować do zmieniających się wymagań organizacji.

Bardzo ważną rolę w kontekście bezpieczeństwa BD odgrywa *składowanie BD* i *szyfrowanie* danych z wykorzystaniem różnych metod kryptograficznych.

Definicja 6-7

Składowanie jest procesem okresowego tworzenia kopii bazy danych na dodatkowych nośnikach pamięci.

Składowany może być również stan programów. Każdy SZBD powinien zawierać mechanizmy składowania danych, pomagające w ich *odtwarzaniu* w razie awarii systemu czy nawet awarii SZBD. Zaleca się również tworzenie w regularnych odstępach czasu zapasowych kopii BD i pliku dziennika. Zapisywanie ich w bezpiecznym miejscu ma także istotne znaczenie. W razie awarii, po której baza danych stała się niedostępna lub całkowicie została usunięta, kopia archiwalna i informacje zapisane w dzienniku zostaną wykorzystane do odtworzenia spójnego stanu bazy.

Zapis dziennika jest regularnym procesem zapisywania informacji o wszystkich zmianach, jakie zaszły w bazie danych. Każdy SZBD powinien posiadać funkcje tworzenia dziennika. Za pomocą tych funkcji śledzony jest aktualny stan *transakcji* i zapisywane są zmiany dokonywane w bazie. **Główną zaletą tworzenia dziennika jest efektywna rekonstrukcja bazy po jej awarii**. W rezultacie za pomocą kopii archiwalnej oraz zapisów dziennika otrzymujemy ostatni, znany, spójny stan bazy danych. W przypadku braku zapisów dziennika wszystkie zmiany dokonane po utworzeniu ostatniej kopii zostałyby utracone.

Definicja 6-8

Szyfrowanie jest czynnością, polegającą na kodowaniu danych specjalnym algorytmem, który przekształca te dane w postać niemożliwą do odczytania bez specjalnego klucza deszyfrującego.

Przeważająca większość instytucji posiada dane o znaczeniu szczególnym. Aby zwiększyć ochronę oraz tajność tego typu danych, możemy zdecydować się na ich zaszyfrowanie. Istnieją SZBD (np. *Oracle*), które posiadają odpowiednie funkcje szyfrujące, przeznaczone właśnie do tego celu. Takie operacje jednak zmniejszają wydajność systemu, ponieważ na odszyfrowanie danych przed ich pokazaniem potrzeba dodatkowego czasu. Niewątpliwie najczęściej szyfrowania używa się do przesyłania danych przez linie komunikacyjne, gdzie informacje nie mogą podlegać innej ochronie.

Więzy integralności (patrz podrozdz. 2.1.4) mają również znaczenie w zachowaniu bezpieczeństwa bazy danych, gdyż zapobiegają występowaniu niepoprawnych danych w bazie.

6.1.2. Uwierzytelnianie użytkowników

Użytkownicy to najczęściej nazwy kont uprawnionych do łączenia z bazą danych.

Uwierzytelnianie, inaczej *autentykacja* do BD, jest pierwszym etapem, gdzie użytkownik w jawny sposób spotyka się z kontrolą dostępu do danych. Uwierzytelnianie użytkowników polega na analizie identyfikatora (nazwy) i hasła każdego użytkownika, które powinny zostać wcześniej zdefiniowane.

Podane w Prz. 5.3 polecenie SQL:

```
CREATE DATABASE 'Firma.gdb' USER 'Jan_Kowalski' PASSWORD 'Ajh80%#_';
```

jednocześnie wykonuje trzy operacje: 1) tworzy BD (*Firma*); 2) tworzy (definiuje) użytkownika (*Jan Kowalski*); 3) tworzy hasło tego użytkownika (*Ajh80%#_*).

Istnieje inna możliwość definiowania użytkowników (jest to jedna z funkcji administratora BD). Polega ona na zastosowaniu polecenia `CREATE USER`:

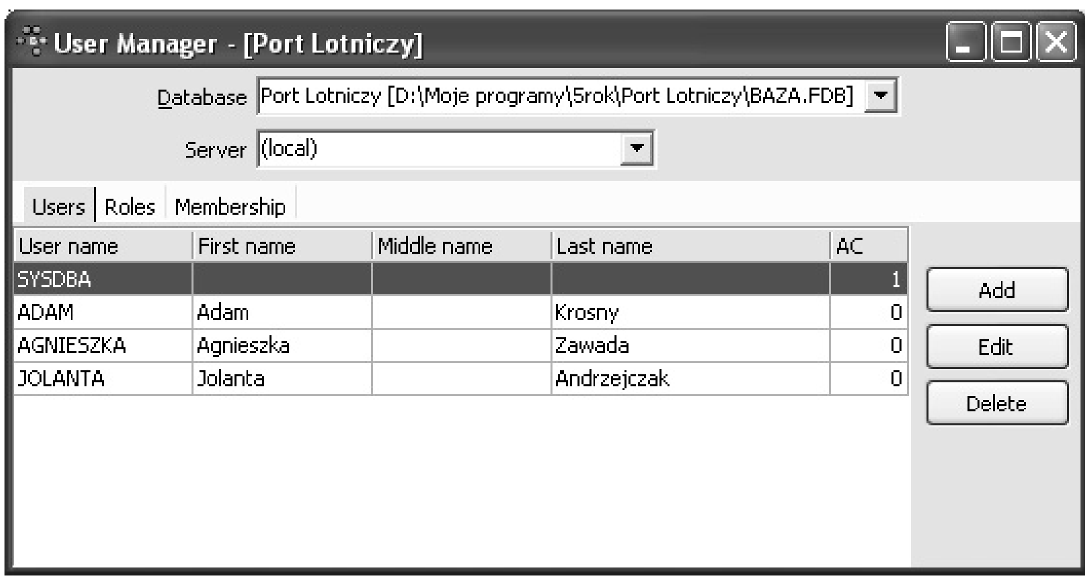
```
CREATE USER Nazwa_Użytkownika IDENTIFIED BY hasło;
```

Przykład 6.1

Przykład polecenia, które tworzy tego samego użytkownika w SZBD *Oracle*:

```
CREATE USER Jan_Kowalski IDENTIFIED BY Ajh80%#_;
```

Opisywane elementy można również stworzyć za pomocą specjalnych programów użytkowych będących częściami SZBD. Poniższy rysunek (rys. 6.1) przestawia okno, w którym mamy możliwość zarządzania kontami użytkowników BD *Port Lotniczy* w SZBD *FireBird*.



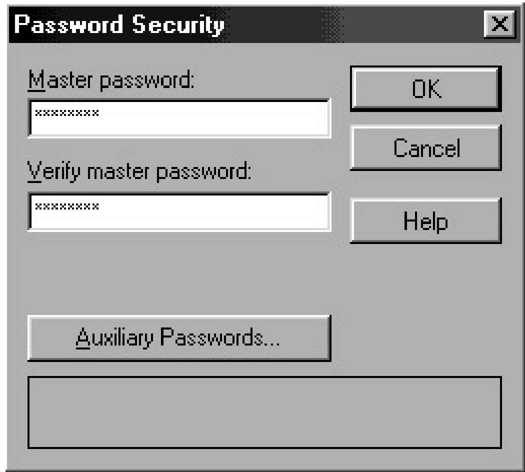
Rys. 6.1. Definiowanie użytkowników BD *Port Lotniczy* w SZBD *FireBird*

Podczas logowania zawsze występuje etap podania hasła odpowiadającego właściwemu *loginowi* (nazwie użytkownika). Nazwa użytkownika często jest ograniczona przez liczbę symboli (np. w SZBD *FireBird* nazwa użytkownika może posiadać maksymalnie 31 znaków).

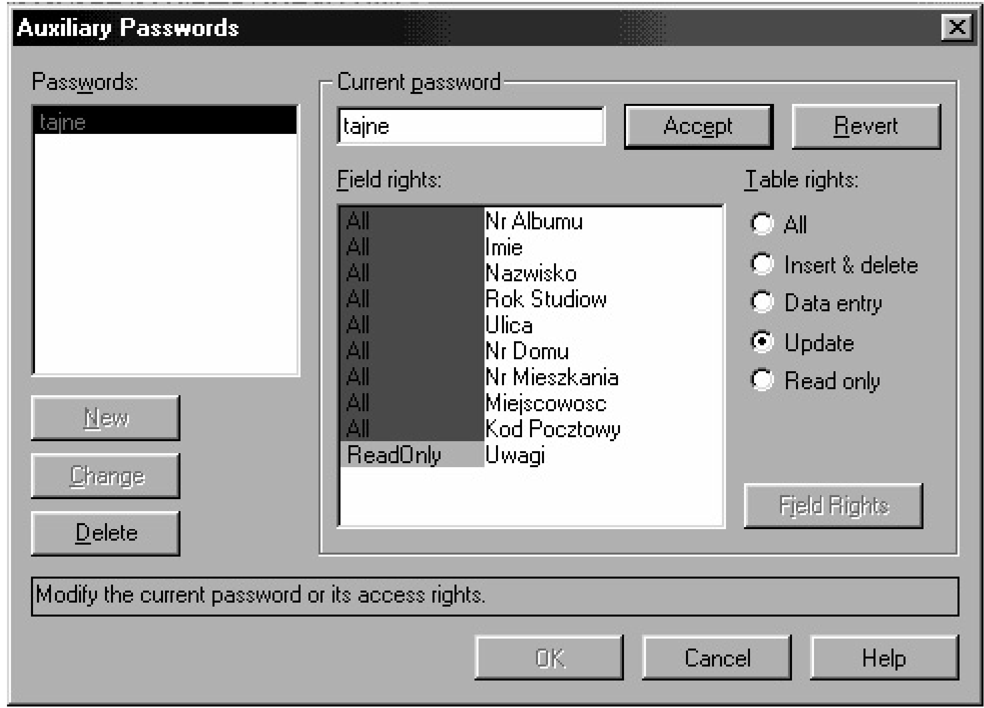
Nazwa użytkownika nie jest elementem typu case-sensitive, czyli wielkość liter nie jest istotna. Hasło natomiast jest elementem typu case-sensitive, zatem w haśle brana jest pod uwagę wielkość liter.

Na rys. 6.2 i 6.3 podano przykłady okien dialogowych w SZBD *Paradox* wykorzystywanych w operacjach logowania.

Lista *Passwords* na rys. 6.3. zawiera wykaz zdefiniowanych haseł pomocniczych. Trzy przyciski poniżej umożliwiają dodanie (*New*), zmianę (*Change*) lub usunięcie (*Delete*) hasła, które wpisujemy w polu *Current password*. Panel z przyciskami (`RadioGroup`) *Table rights* umożliwia zdefiniowanie poziomu bezpieczeństwa dla danego hasła.



Rys. 6.2. Okno dialogowe *Password Security* w SZBD *Paradox*



Rys. 6.3. Okno dialogowe *Auxiliary Password* w SZBD *Paradox*

Za pomocą przycisku *Accept* akceptujemy wprowadzone zmiany, natomiast za pomocą przycisku *Revert* wycofujemy się z nich. Hasła do tabeli przechowywane są w jej nagłówku w zaszyfrowanej postaci. Oczywiście, zawartość tabeli jest również zaszyfrowana.

Jeżeli zostanie dodane nowe pole do tabeli, w której są już ustawione hasła dodatkowe, to należy określić dostęp do tego pola. Działanie to jest konieczne, dlatego że BDE (*Borland Database Engine*) przypisuje prawo *None* wszystkim hasłom, których prawa dostępu są na poziomach *Read Only*, *Update* lub *Data Entry*. Należy również pamiętać o możliwych konfliktach z innym właściwościami tabeli. Na przykład, jeżeli otworzymy tabelę z uprawnieniami *Data Entry* i istnieje w niej pole wymagane (*Required*) z prawem *Read Only*, to nie będzie możliwe zatwierdzenie rekordu takiej tabeli.

W momencie otwarcia tabeli BDE tworzy sesję (*session*). Sprawuje ona kontrolę nad połączeniem z bazą. Sesje są tworzone, aby rozróżnić użytkowników korzystających

z bazy. BDE posiada specjalny bufor, w którym przechowuje wszystkie hasła (maksymalnie 25 haseł) użyte przez tabele typu *Paradox* w obrębie danej sesji. Jeżeli użytkownik wprowadzi hasło (w odpowiednim okienku dialogowym) lub zostanie ono dodane w kodzie, to zostaje ono wprowadzone do bufora. W momencie gdy aplikacja uzyskuje dostęp do tabeli i BDE rozpozna, że tabela jest zaszyfrowana, to będzie on, korzystając z listy haseł znajdujących się w buforze, testował każde z nich. Dla wszystkich haseł pasujących do tabeli zostanie ustanowiona suma wszystkich uprawnień uzyskanych dzięki nim.

Z takim działaniem BDE wiążą się pewne niebezpieczeństwa. Mianowicie, załóżmy, że hasło *x1y2* daje pełny dostęp do pierwszej tabeli, a hasło *a3b4* umożliwia dostęp na poziomie *Read Only* do tej samej tabeli. Załóżmy również, że hasło *x1y2* zapewnia dostęp *Read Only* do drugiej tabeli. Niech pewien użytkownik posiada uprawnienia dostępu do obu tabel na poziomie *Read Only*, czyli zna hasła *a3b4* i *x1y2* (nie wiedząc oczywiście, że to hasło pasuje do pierwszej tabeli, dając do niej pełny dostęp). Jeżeli użytkownik uzyska dostęp do dwóch tabel, to w buforze będą przechowywane oba hasła. W takim przypadku podczas powtórnego otwarcia pierwszej tabeli w obrębie tej samej sesji zostanie użyte hasło *x1y2* dające użytkownikowi pełen dostęp.

Szyfrowanie tabel powoduje 10-15% spadek szybkości dostępu do danych tabeli. Dzieje się tak dlatego, że BDE musi na bieżąco deszyfrować dane odczytywane z tabeli i szyfrować dane zapisywane w niej. Szyfrowane tabele dają zazwyczaj gorszy współczynnik kompresji, gdyż nie zawierają, w porównaniu z tabelami niezaszyfrowanymi, tak dużej liczby spacji. Współczynnik kompresji tabel niezaszyfrowanych wynosi średnio około 85-90%, podczas gdy tabel szyfrowanych tylko 10-15%.

Wcześniej zdefiniowanego użytkownika można usunąć albo zmienić odpowiednie identyfikatory za pomocą poleceń SQL. Przykładowo w *Oracle* pierwsza z tych operacji polega na zastosowaniu zdania `DROP USER`:

```
DROP USER Nazwa_Użytkownika;
```

dodawanie nowego słowa kluczowego powoduje usunięcie wszystkich obiektów z bazy stworzonych przez tego użytkownika:

```
DROP USER Nazwa_Użytkownika CASCADE;
```

Zmiana identyfikatora (hasła) wymaga zastosowania zdania `ALTER USER`:

```
ALTER USER Nazwa_Użytkownika IDENTIFIED BY nowe_hasło;
```

6.1.3. Role w bazach danych

Każda rola (posiadająca unikatową nazwę) dysponuje odpowiednimi uprawnieniami.

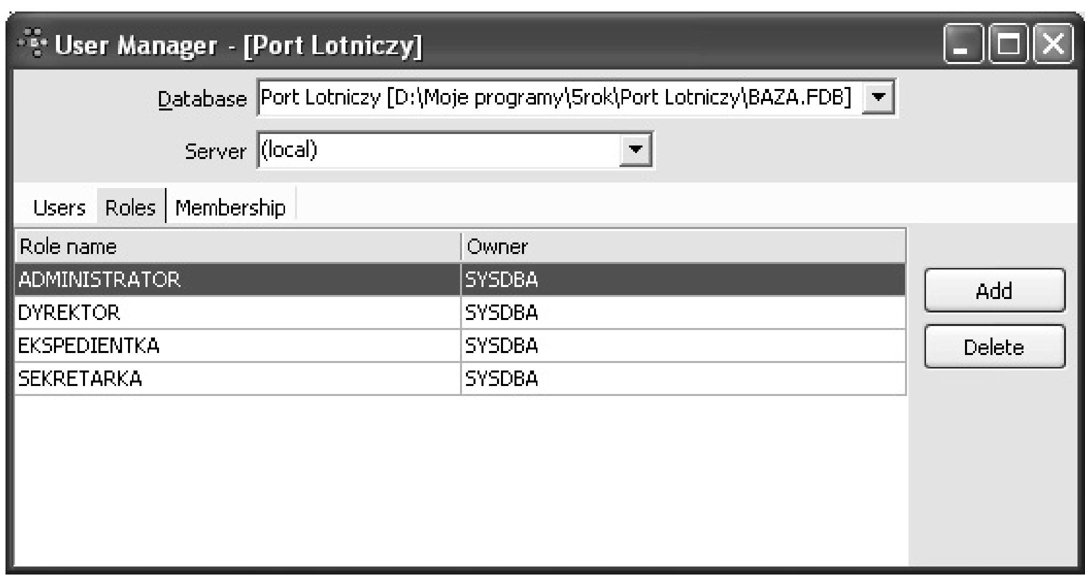
Rola może być zdefiniowana poprzez odpowiednie narzędzia lub za pomocą poleceń SQL.

We wspomnianej bazie *Port Lotniczy* zostały stworzone cztery przykładowe role:

- **administrator** – osoba upoważniona do administrowania i opieki nad bazą; posiada większość dostępnych uprawnień;
- **dyrektor** – osoba nadzorująca działanie innych pracowników;
- **ekspedientka** – osoba upoważniona do sprzedaży biletów pasażerom; posiada okrojone możliwości dostępu do danych;

- **sekretarka** – osoba wspomagająca pracę w biurze; podobnie jak ekspedientka posiada niewielki dostęp do danych w bazie.

Rys. 6.4 przedstawia wygląd odpowiedniego okna dialogowego.



Rys. 6.4. Definicja ról w SZBD *FireBird*

Mając zdefiniowanych użytkowników oraz role, możemy przystąpić do przyznawania użytkownikom odpowiednich ról. Każdy użytkownik powinien otrzymać dostęp tylko do tych danych, które są mu niezbędne do poprawnego wykonywania swojej pracy. Oczywiście, każdy użytkownik może posiadać kilka ról, jeśli tylko zachodzi taka potrzeba. Jak widać na rys. 6.5, użytkownikowi o nazwie *AGNIESZKA* przypisano zarówno prawa dostępu ekspedientki, jak i sekretarki.

Większość komercyjnych SZBD oferuje już zdefiniowane role. Przykładowo: w *Oracle* najbardziej znanymi rolami są:

```
CONNECT,
RESOURCE,
DBA.
```

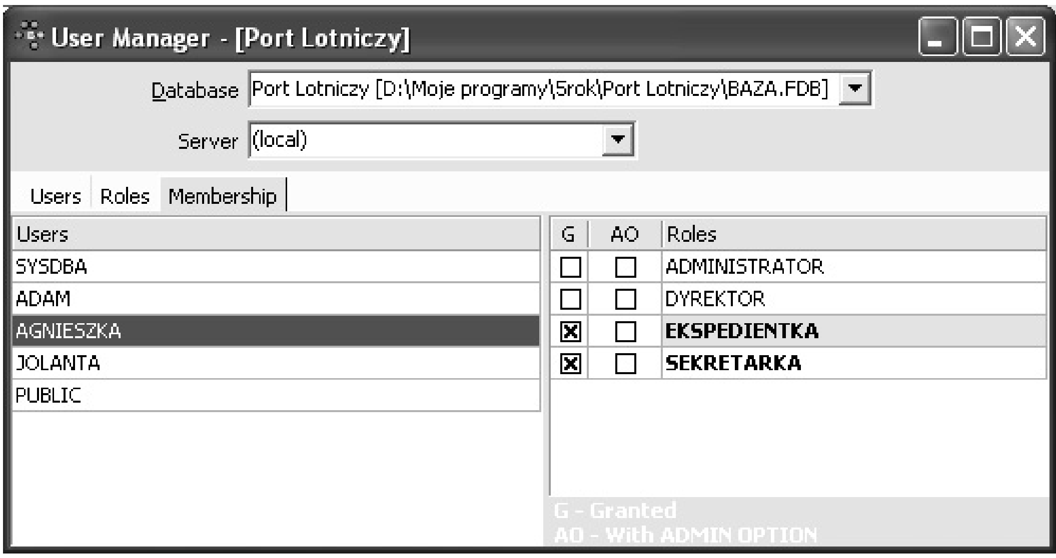
Rola `CONNECT` posiada początkowy (wstępny) poziom uprawnień, m.in. pozwala wykonywać polecenia `SELECT`, `UPDATE`, `INSERT`, `DELETE`. Użytkownik z rolą `CONNECT` może także tworzyć tabele i perspektywy (widoki).

Rola `RESOURCE` zwiększa uprawnienia użytkownika, w tym daje możliwość tworzenia indeksów i procedur.

Rola `DBA` obejmuje wszystkie możliwe uprawnienia, w tym przypisywanie ról innym użytkownikom.

W SQL dowolną rolę możemy przypisać użytkownikowi za pomocą polecenia `GRANT`:

```
GRANT Nazwa_Roli TO Nazwa_Użytkownika;
```



Rys. 6.5. Przykład przypisania użytkowników do ról w BD *Port Lotniczy*

Przykład 6.2

Poniższe zdanie przypisuje użytkownikowi *Adamczak* rolę `RESOURCE`:

```
GRANT RESOURCE TO Adamczak;
```

Aby usunąć rolę, zastosujemy frazę:

```
REVOKE Nazwa_Roli FROM Nazwa_Użytkownika;
```

Wykorzystanie ról w BD pozwala na wygodne i łatwe kontrolowanie dostępem do danych. Jest to duża zaleta ról, szczególnie gdy w bazie istnieje kilkudziesięciu, a nawet kilkuset użytkowników. Zmiana obowiązków pracownika czy też jego awans zmusza w konsekwencji administratora bazy do zmiany roli dla tego użytkownika, a nie do ponownego definiowania jego przywilejów.

6.1.4. Przywileje w bazach danych

Przywileje dzielimy na takie, które dotyczą obiektów (bazy danych, tabele, perspektywy, procedury wbudowane), oraz takie, które dotyczą zawartości obiektów (zawartość tabel lub perspektyw). Pierwsze nazywamy *przywilejami obiektowymi*, drugie – *przywilejami systemowymi*.

Różne SZBD posiadają różne przywilejów obu typów. Przykładowo w *FireBird* przywileje dotyczące obiektów to:

`CREATE` – pozwala na utworzenie obiektu określonego typu;

`DROP` – pozwala na usunięcie określonego obiektu;

`ALTER` – pozwala na modyfikację definicji danego obiektu,

zaś przywileje, dotyczące zawartości obiektów:

`ALL` – pozwala na dokonanie wszystkich poniższych;

`SELECT` – pozwala na przeglądanie zawartości danego obiektu;

`UPDATE`	–	pozwala na modyfikowanie istniejących danych w danym obiekcie;
`DELETE`	–	pozwala na usunięcie wierszy z określonego obiektu;
`INSERT`	–	pozwala na utworzenie dodatkowych wierszy w danym obiekcie;
`EXECUTE`	–	możliwość wywołania procedury zapamiętanej lub programu;
`REFERENCE`	–	możliwość określenia klucza obcego w danym obiekcie odnoszącego się do innego obiektu.

Przywileje możemy nadawać różnym obiektom, takim jak: tabele, perspektywy, procedury wbudowane oraz triggery (wyzwalacze). Przyznawanie przywilejów w SQL dokonuje się poleceniem `GRANT`, natomiast odbieranie poleceniem `REVOKE`.

Polecenie `GRANT` w *FireBird* może być użyte w następujących sytuacjach:

- przyznawanie przywilejów `SELECT`, `INSERT`, `UPDATE`, `DELETE` oraz `REFERENCES` na **tabelach** przez użytkowników, triggery, procedury wbudowane oraz widoki (opcjonalnie `WITH GRANT OPTION`);
- przyznawanie przywilejów `SELECT`, `INSERT`, `UPDATE`, `DELETE` na **perspektywach** przez użytkowników, triggery, procedury wbudowane oraz widoki (opcjonalnie `WITH GRANT OPTION`);
- przyznawanie przywilejów `SELECT`, `INSERT`, `UPDATE`, `DELETE` oraz `REFERENCES` na tabelach przez role;
- przyznawanie przywilejów `SELECT`, `INSERT`, `UPDATE`, `DELETE` na **perspektywach** przez role;
- przyznawanie ról użytkownikom (opcjonalnie `WITH ADMIN OPTION`);
- przyznawanie pozwolenia `EXECUTE` na **procedurach wbudowanych** przez użytkowników, triggery, procedury wbudowane oraz perspektywy (opcjonalnie `WITH GRANT OPTION`).

Użytkownik tworzący obiekt w bazie automatycznie staje się właścicielem tego obiektu oraz ma prawo do wykonywania wszystkich możliwych operacji na tym obiekcie. Domyślnie żaden inny użytkownik nie ma prawa do tego obiektu, chyba że właściciel tego obiektu nada mu do niego określone przywileje. W ramach wyjątku *FireBird* wspiera użytkownika `SYSDBA`, który ma dostęp do wszystkich obiektów bazy.

Właściciel obiektu może nadać jednemu lub wielu użytkownikom prawa do tego obiektu właśnie za pomocą polecenia `GRANT`. Jeżeli użyjemy tego polecenia z klauzulą `(WITH GRANT OPTION)`, to użytkownik otrzymujący pewne przywileje jest uprawniony do nadawania tych przywilejów innym użytkownikom.

Składnia polecenia `GRANT` ma postać:

```
GRANT Lista_przywilejów
ON Obiekt
TO Lista_użytkowników [WITH GRANT OPTION];
```

Listing 6.1

Przykład 6.3

Nadanie prawa odczytu, modyfikacji oraz usuwania do tabeli *Pracownik* użytkownikowi o nazwie *Bogdan* do całej tabeli możemy zrealizować następująco:

```
GRANT SELECT, UPDATE, DELETE
ON Pracownik
TO Bogdan;
```

Przykład 6.4

Następnym przykładem będzie nadanie temu samemu użytkownikowi prawa `REFERENCES`, pozwalającego na utworzenie klucza obcego odnoszącego się do klucza podstawowego *Id_Prac* tabeli *Pracownik*:

```
GRANT REFERENCES
ON Pracownik(Id_Prac)
TO Bogdan;
```

Przykład 6.5

Nadanie prawa wszystkim użytkownikom do modyfikacji kolumn *Telefon* oraz *Stanowisko* w tabeli *Pracownik*:

```
GRANT UPDATE (Telefon, Stanowisko)
ON Pracownik
TO Public;
```

Przykład 6.6

Nadanie prawa *procedurze wbudowanej* o nazwie *Przelicznik_Walut* do wstawiania danych w tabeli *Pracownik*:

```
GRANT INSERT
ON Pracownik
TO PROCEDURE Przelicznik_Walut;
```

Przykład 6.7

Jeżeli chcemy dodać wszystkie prawa dostępu do jakiegoś obiektu, można to zrobić za pomocą opcji `ALL`. Przypuśćmy, że chcemy nadać wszystkie prawa dostępu do tabeli *Pracownik* użytkownikowi o nazwie *Admin* z możliwością przekazywania praw. Polecenie, które wykona to zadanie, ma następującą postać:

```
GRANT ALL
ON Pracownik
TO Admin WITH GRANT OPTION;
```

W przypadku odbierania przywilejów możemy mieć kilka różnych sytuacji. Odbieranie przywilejów może być wykonane z klauzulą `CASCADE` lub `RESTRICT`. W przypadku użycia klauzuli `CASCADE` przywileje zostaną odebrane wszystkim użytkownikom, którzy otrzymali je bezpośrednio lub pośrednio od danego użytkownika. Zastosowanie klauzuli `RESTRICT` nie doprowadzi do usunięcia określonych przywilejów, jeżeli ten przekazał je innym użytkownikom.

Do usuwania przywilejów używamy polecenia `REVOKE`. Razem z nim możemy użyć również klauzuli `GRANT OPTION FOR`. Jej wystąpienie spowoduje odebranie użytkow-

nikowi prawa do nadawania przywilejów innym użytkownikom. Składnia polecenia `REVOKE` ma postać:

```
REVOKE [GRANT OPTION FOR]
ON Obiekt
FROM Lista_użytkowników [RESTRICT, CASCADE];
```

Listing 6.2

W niektórych SZBD (np. w *Oracle*) nadawanie użytkownikom przywilejów może polegać wyłącznie na przypisaniu ich do odpowiednich ról. Inaczej mówiąc, użytkownik lub grupa użytkowników posiada te przywileje, które są właściwe rolom, do których są przypisani.

Przywileje mogą być określone również za pomocą odpowiednich narzędzi SZBD. Zwróćmy ponowną uwagę na rys. 6.3. W prawej części okna dialogowego w *Paradoxe* użytkownik może zdefiniować prawa dostępu do pól danej tabeli. Na tym rysunku zostało więc pokazane, że do wszystkich pól (oprócz ostatniego) zostało nadane prawo dostępu `ALL`.

Tabela 6.1 zawiera wykaz poziomów bezpieczeństwa BD *Paradox* w porządku rosnącym od najmniej rygorystycznych do najbardziej (patrz rys. 6.3). Każdy poziom ochrony zawiera oprócz własnych uprawnień także wszystkie uprawnienia określone na niższych szczeblach.

Tabela 6.1. Wykaz poziomów bezpieczeństwa BD *Paradox*

Uprawnienie	*Opis*
All	Daje użytkownikowi pełne prawa do wszystkich funkcji tabeli, włączając w to możliwość przebudowy struktury tabeli lub jej usunięcia oraz zmiany lub usunięcia haseł
Insert & Delete	Daje użytkownikowi prawa do dodawania, zmiany bądź usuwania rekordów. Nie daje możliwości zmiany struktury tabeli lub jej usunięcia
Data Entry	Daje użytkownikowi prawa do wstawiania i modyfikowania niekluczowych pól. Nie daje możliwości usuwania rekordów i modyfikacji pól kluczowych
Update	Daje użytkownikowi prawa do odczytywania tabeli i modyfikowania pól niekluczowych. Nie daje możliwości dodawania i usuwania rekordów oraz modyfikacji pól kluczowych
Read Only	Daje użytkownikowi prawa do odczytywania tabeli. Nie daje możliwości jakichkolwiek zmian

W okienku *Auxiliary Passwords* na rys. 6.3 znajduje się również lista *Field rights*. Umożliwia ona zmianę poziomu praw dostępu do poszczególnego pola. Podwójne kliknięcie lub wciśnięcie przycisku *Field Rights* powoduje cykliczną zmianę trzech dostępnych wartości. Zostały one opisane w tab. 6.2.

Tabela 6.2. Opis zmian poziomu praw dostępu w liście *Field rights*

Prawo dostępu	*Opis*
All	Daje użytkownikowi pełne prawa dostępu do pola w zakresie ograniczeń nałożonych na tabelę
Read Only	Daje użytkownikowi prawo do odczytywania zawartości pola. Nie daje możliwości zmiany jego wartości
None	Pole jest niedostępne

6.2. Perspektywy

6.2.1. Typy perspektyw

Poprzez przywileje mogliśmy jedynie określić typy dostępu do obiektów. Chcąc określić prawa dostępu do poszczególnych fragmentów tabel, musimy skorzystać z perspektyw.

Perspektywa, jak podkreśliliśmy wyżej, jest tabelą wirtualną, która nie musi fizycznie istnieć w bazie danych, ale na żądanie użytkownika może w każdej chwili być wyliczona.

Użytkownik widzi perspektywę jako rzeczywistą tabelę posiadającą kolumny o ustalonych nazwach i wiersze danych. Jednak w odróżnieniu od zwykłych tabel bazowych perspektywa nie musi istnieć w bazie w postaci zbioru wartości danych.

Perspektywę definiuje się jako zapytanie dotyczące jednej lub kilku tabel bazowych lub też innych perspektyw. **Innymi słowy, perspektywa to zapytanie SQL opatrzone nazwą.**

Definicja perspektywy umieszczona jest w bazie danych. Kiedy następuje odwołanie do niej, SZBD generuje jej zawartość na podstawie tabel bazowych. Proces przekształcania zapytania nazywamy *rozkładem perspektywy*. Istnieje również możliwość przechowywania perspektywy w bazie danych w postaci *tabeli tymczasowej*. Proces ten nazywany jest *materializacją perspektywy*.

Ze względu na postać zapytania definiującego perspektywę perspektywy możemy podzielić na ***proste*** i ***złożone***.

Definicja 6-1

Perspektywa jest ***prosta***, jeżeli zapytanie ją definiujące:

- odwołuje się wyłącznie do jednej tabeli bazowej;
- nie wykorzystuje funkcji języka SQL w celu przetwarzania wartości pól udostępnionych przez perspektywę;
- nie wykorzystuje wartości wyliczanych;
- nie wykorzystuje funkcji grupowych;
- nie wykorzystuje operatorów zbiorowych.

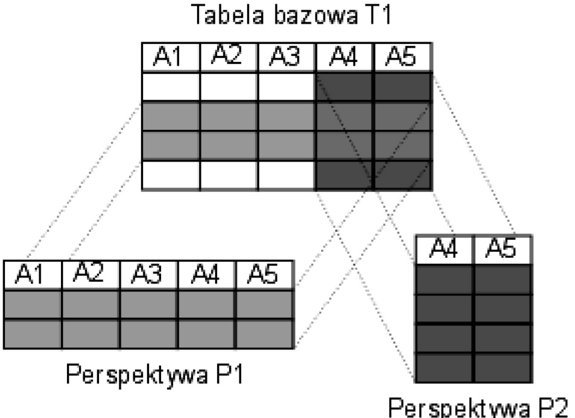
Definicja 6-2

Perspektywę nazywamy ***złożoną***, jeżeli zapytanie definiujące tej perspektywy nie spełnia przynajmniej jednego warunku przedstawionego dla perspektywy prostej.

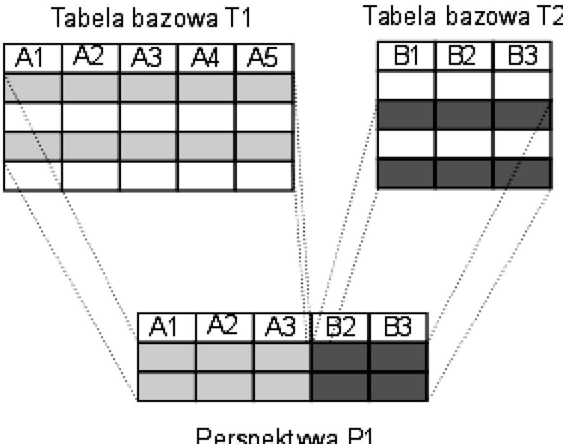
W praktyce różne rodzaje perspektyw wykorzystujemy do różnych celów. Oto inny podział perspektyw. Wyróżniamy w nim trzy rodzaje perspektyw:

- **perspektywy poziome** – składające się z wybranych rekordów tabeli;
- **perspektywy pionowe** – składające się z wybranych pól tabeli;
- **perspektywy grupujące –** polegające na grupowaniu i złączeniu danych.

Na rys. 6.6 schematycznie przedstawiono technologię tworzenia perspektyw prostych (poziomej, P1, i pionowej, P2), a na rys. 6.7 perspektywy złożonej.



Rys. 6.6. Perspektywy proste (bazujące na jednej tabeli)



Rys. 6.7. Perspektywa złożona

6.2.2. Tworzenie perspektyw

Do tworzenia perspektyw używamy instrukcji `CREATE VIEW`. Postać tego polecenia jest następująca:

```
CREATE VIEW NazwaPerspektywy [(NowaNazwaKolumny[,…])]
AS ZapytanieSELECT [WITH[CASCADED|LOCAL]CHECK OPTION];
```

Listing 6.3

Perspektywę tworzymy, wykorzystując odpowiednie zapytanie `SELECT` języka SQL. Każdej kolumnie można nadać inną nazwę. Jeśli w poleceniu podamy listę nazw kolumn, to musi ona mieć tyle elementów, ile jest wszystkich kolumn w wyniku zapytania

SELECT. Listę z nazwami kolumn musimy tworzyć zawsze wtedy, kiedy istnieją niejasności co do ich nazwy, np. w przypadku kolumn wyliczanych lub gdy powstają dwie kolumny o takich samych nazwach. Klauzula WITH GRANT OPTION gwarantuje, że rekordy, które nie spełniają warunku we frazie WHERE zapytania definiującego, nie mogą być dodane poprzez perspektywę do tabeli bazowej, na podstawie której jest zdefiniowana perspektywa.

Warunkiem utworzenia perspektywy jest posiadanie przez użytkownika dostępu typu SELECT do wszystkich tabel wchodzących w skład perspektywy.

Aby lepiej zobrazować tworzenie perspektyw, przedstawimy je na podstawie przykładowej tabeli *Pracownicy*.

Tabela 6.3. Tabela *Pracownicy*

Id	*Nazwisko*	*Imię*	*Telefon*	*NIP*	*Pensja*	*Dział*	*Stanowisko*
1	Nowacki	Kazimierz	501377465	1772635425	2500	FINANSÓW	Wicedyrektor
2	Cichosz	Marzena	697376482	2663546353	2800	KADR	Dyrektor
3	Margasiński	Piotr	509266544	9958574747	1500	TECHNICZNY	Mechanik
4	Głowacka	Iwona	600387756	6464644463	1900	KADR	Księgowa
5	Siwiec	Marlena	601192387	9483636373	1750	FINANSÓW	Sekretarka
6	Kot	Katarzyna	504127364	1092930494	2800	FINANSÓW	Dyrektor
7	Szczepaniak	Lucjan	503192344	1009487883	1400	TECHNICZNY	Mechanik

6.2.2.1. Perspektywy poziome

Perspektywa pozioma ogranicza dostęp użytkownika jedynie do wybranych wierszy z tabel lub perspektyw bazowych.

Rozważmy przykład.

Przykład 6.8

Załóżmy, że chcemy utworzyć perspektywę dla dyrektora działu *Finansowego*, tak aby zawarte w niej informacje dotyczyły tylko i wyłącznie pracowników tego działu.

Perspektywę nazwiemy *D_Fin*, a utworzymy ją poniższym poleceniem:

```
CREATE VIEW D_Fin AS
SELECT *
FROM Pracownik
WHERE Dział = 'FINANSÓW';
```

W ten sposób powstanie perspektywa o nazwie *D_Fin* i nazwach kolumn takich samych jak nazwy kolumn w tabeli *Pracownik*. Jeżeli teraz wykonamy polecenie:

```
SELECT * FROM D_Fin;
```

to otrzymamy wynik pokazany w tab. 6.4.

Tabela 6.4. Perspektywa pozioma *D_Fin*

Id	*Nazwisko*	*Imię*	*Telefon*	*NIP*	*Pensja*	*Dział*	*Stanowisko*
1	Nowacki	Kazimierz	501377465	1772635425	2500	FINANSÓW	Wicedyrektor
5	Siwiec	Marlena	601192387	9483636373	1750	FINANSÓW	Sekretarka
6	Kot	Katarzyna	504127364	1092930494	2800	FINANSÓW	Dyrektor

6.2.2.2. Perspektywy pionowe

Perspektywa pionowa ogranicza dostęp użytkownika tylko do wybranych kolumn z tabel umieszczonych w jej zapytaniu `SELECT`.

Przykład 6.9
Utwórzmy perspektywę nazwaną *Prac_1*, przeznaczoną dla tych pracowników, którzy nie powinni znać pensji, numeru NIP oraz stanowiska. Posłużymy się do tego następującym poleceniem:

```
CREATE VIEW Prac_1 AS
SELECT Id, Nazwisko, Imię, Telefon, Dział
FROM Pracownik;
```

Wynikiem będzie tab. 6.5.

Tabela 6.5. Perspektywa pionowa *Prac_1*

Id	*Nazwisko*	*Imię*	*Telefon*	*Dział*
1	Nowacki	Kazimierz	501377465	FINANSÓW
2	Cichosz	Marzena	697376482	KADR
3	Margasiński	Piotr	509266544	TECHNICZNY
4	Głowacka	Iwona	600387756	KADR
5	Siwiec	Marlena	601192387	FINANSÓW
6	Kot	Katarzyna	504127364	FINANSÓW
7	Szczepaniak	Lucjan	503192344	TECHNICZNY

Perspektywy pionowe są często wykorzystywane w sytuacjach, gdy do danych zawartych w tabeli mają dostęp użytkownicy lub grupy użytkowników o różnych stanowiskach. Pozwalają stworzyć dla każdej grupy użytkowników odpowiednie tabele zawierające tylko niezbędne dla nich informacje.

6.2.2.3. Perspektywy grupujące

Jednym z najczęstszych powodów stosowania tego rodzaju perspektyw jest upraszczanie zapytań dotyczących wielu tabel. Po utworzeniu takiej perspektywy często można odwoływać się do niej, stosując proste zapytania do jednej tabeli (perspektywy), zamiast stosować złożone zapytania do wielu tabel. Perspektywy tego typu często wykorzystują *funkcje agregujące*, takie jak: `SUM, AVG, MIN, MAX, COUNT`.

Przykład 6.10

Jako przykład zdefiniujmy perspektywę, pozwalającą wyświetlić liczbę pracowników w każdym dziale. Definicja takiej perspektywy będzie miała następującą postać:

```
CREATE VIEW Liczba_Prac AS
SELECT Dzial, COUNT(*) Liczba
FROM Pracownik
GROUP BY Dział;
```

Wynikiem takiej perspektywy będzie tab. 6.6.

Tabela 6.6. Perspektywa grupująca

Dział	*Liczba*
FINANSÓW	3
KADR	2
TECHNICZNY	2

Zauważmy, że w przedstawionym przykładzie jednej z kolumn nadano nazwę, ponieważ zawierała ona wynik funkcji agregującej COUNT.

Perspektywę usuwa się z bazy poleceniem DROP VIEW:

```
DROP VIEW NazwaPerspektywy [RESTRICT|CASCADE];
```

Polecenie to powoduje całkowite usunięcie definicji perspektywy z BD. Jeżeli w poleceniu DROP VIEW użyto opcję CASCADE, to zostaną usunięte również wszystkie obiekty zależne od perspektywy, czyli takie, które zawierają pewne odwołania do niej. Jeżeli użyto opcji RESTRICT, to perspektywa zostanie usunięta, pod warunkiem że nie istnieją obiekty zależne od niej. Domyślną opcją jest CASCADE.

Niewątpliwie perspektywy stanowią bardzo rozbudowany mechanizm, pozwalający upraszczać wiele niewygodnych sytuacji dotyczących baz danych. Niestety, wiążą się z tym również pewne ograniczenia oraz problemy. Przepisem na zbudowanie dobrze działającej bazy danych jest umiejętne użycie dostępnych mechanizmów, jakie dają perspektywy. Mechanizmy te posiadają zarówno zalety, jak i wady. Wśród zalet możemy wymienić następujące cechy perspektyw:

- niezależność danych – pomimo zmian dokonywanych w tabelach bazowych perspektywa może reprezentować spójny i niezmienny obraz struktury bazy danych;
- poprawa bezpieczeństwa – każdy użytkownik może zostać zaopatrzony w pewien zbiór perspektyw, które udostępniają mu jedynie dane dla niego niezbędne;
- wielodostęp – zmiany dokonywane na tabelach bazowych automatycznie widać również w perspektywach;
- integralność danych – stosując klauzulę WITH GRANT OPTION w poleceniu CREATE VIEW, mamy gwarancję, że żaden wiersz, który nie spełnia warunku selekcji zapytania definiującego perspektywę, nie zostanie nigdy dodany do tabeli bazowej. W tym zakresie zapewnia to integralność danych.

Wśród wad niewątpliwie wymienimy takie cechy, jak:

- ograniczona możliwość modyfikacji – w pewnych sytuacjach perspektywa nie pozwala na modyfikację danych w tabeli bazowej. Standard ISO określa, że perspektywa jest modyfikowalna wtedy i tylko wtedy, kiedy:
  - żadna kolumna nie występuje więcej niż jeden raz oraz nie jest stałą, wyrażeniem czy funkcją agregującą;
  - w wyniku zapytania nie są usuwane wiersze podwójne (nie występuje opcja DISTINCT);
  - jest to perspektywa prosta, czyli oparta na jednej tabeli bazowej;
  - w klauzuli WHERE nie występuje zagnieżdżone podzapytanie SELECT, w której występuje odwołanie do tabeli znajdującej się na liście FROM;
  - w zapytaniu definiującym perspektywy nie występują klauzule GROUP BY i HAVING;
- ograniczenia struktury – budowa perspektywy jest ściśle określona w momencie jej definiowania. Jeżeli zapytanie definiujące wykorzystuje całą tabelę bazową, czyli ma postać `SELECT * FROM …`, to dodane do niej w późniejszym czasie kolumny nie zostaną dodane do perspektywy;
- wydajność – korzystanie z perspektyw prowadzi, niestety, do spadku wydajności systemu. Nie jest to zauważalne w przypadku perspektyw prostych, lecz jeśli perspektywa jest złożona z wielu tabel bazowych, to rozkład zapytania spowoduje odczuwalne obciążenie systemu. Ponieważ rozkład zapytania następuje przy każdym odwołaniu się do perspektywy, spowolnienie systemu może stać się nawet uciążliwe.

6.3. Wyzwalacze

6.3.1. Podstawowe informacje

Wyzwalacz (ang. *trigger*) jest to pewien rodzaj procedury, ale wywoływany automatycznie, gdy zachodzi odpowiednie zdarzenie. *Wyzwalacze* mają za zadanie wykonanie określonej akcji w określonej chwili.

Wyzwalacze możemy rozpatrywać też jako metody, za pomocą których można zabezpieczać *integralność* BD.

Definicja 6-3

Wyzwalacz jest to specjalny typ procedury powiązany z daną tabelą, którą automatycznie wykonuje się przy każdej próbie modyfikacji jej danych. Tym samym pozwala ona reagować na wprowadzane w danej tabeli zmiany.

Zakładamy, że BD posiada tabele połączone przez odpowiednie pola kluczowe (niech to będą tabele Pracownicy {**Id_Prac**, Nazw_Prac, Imię_Prac, Stanowisko, Data_Zatrudn} i Adresy_Prac {**NIP**, Kod_Poczt, Miejscowość, Telefon, Data_Zameldowania, Id_Prac}, połączone poprzez Id_Prac (typ uczestnictwa tabel w tej relacji jest obowiązkowy).

W takiej sytuacji dobrze jest zdefiniować trigger, który przy każdej próbie usuwania pracownika z tabeli *Pracownicy* sprawdzi, czy został już usunięty odpowiedni rekord tabeli *Adresy_Prac* i na odwrót. Taki problem można rozwiązać także za pomocą metody integralności danych. Jednakże wyzwalacz pozwala na stworzenie bardziej skomplikowanych reguł. Przykładowo: można stworzyć trigger, który troszczy się o to, aby w nowym, dodawanym do bazy pracowniku nie było nieokreślonej daty zatrudnienia – pola *Data_Zatrudn* tabeli *Pracownicy*. Jeżeli zdarzyłaby się taka sytuacja, to będzie jej przypisana data aktualna. Dzięki takiemu rozwiązaniu w aplikacji nie musimy podawać żadnej daty. Wystarczy zostawić ten atrybut pusty, a SZBD sam je uzupełni. Jest to o tyle lepsze rozwiązanie, że nie musimy się martwić, że pracownik (odpowiadający za aktualizację BD) wstawi dowolną datę w to miejsce. Również wstawianie aktualnej daty w aplikacji nie miałoby większego sensu, ponieważ czas na komputerze, na którym ona działałaby, nie musi być poprawny. Stosując ten trigger, musimy się tylko upewnić, czy na serwerze bazy jest ustawiona poprawna data, a wtedy daty zatrudnienia wstawiane do bazy będą na pewno odpowiadały rzeczywistym datom.

Wyzwalacze często nazywa się **regułami „zdarzenie-warunek-akcja**" (ang. *event-condition-action rules*, ECA *rules*). Generując i korzystając z wyzwalaczy, należy pamiętać o kilku podstawowych kwestiach:

- wyzwalacz uruchamia się tylko w kontekście zdarzenia, którego opis znajduje się w tekście wyzwalacza; do takich zdarzeń należą operacje *wstawiania* (`Insert`), *modyfikacji* (`Update`), *usuwania* (`Delete`); w niektórych SZBD trigger reaguje na zdarzenie, jakim jest *koniec transakcji* (ang. *end of transaction*);
- operacja, którą inicjalizuje *wyzwalacz*, będzie od początku zablokowana; *wyzwalacz* wykonuje kontrolę *warunku*; jeżeli wynikiem tej kontroli będzie wartość *false*, wyzwalacz skończy swoje działanie; w innej sytuacji (*`true`*) SZBD zrealizuje akcję odpowiadającą zdarzeniu;
- wyzwalaczy nie można zdefiniować dla tabel tymczasowych.

Składnia i semantyka zdania SQL z triggerem daje programistom różne możliwości, dotyczące każdej części konstrukcji *zdarzenie-warunek-akcja*. Oto ważniejsze cechy deklarowania triggera:

- akcja może być wykonywana albo *przed* (ang. *before*), albo *po* (ang. *after*) zdarzeniu triggera; zaczynając od SQL *Server 2000*, powstała jeszcze jedna grupa wyzwalaczy – *zamiast* (ang. *instead of*), które wykonują operację zamiast operatora mającego zmienić dane;
- podczas wykonania triggera może operować zarówno *poprzednimi* (ang. *old*), jak i *nowymi* (ang. *new*) wartościami rekordów, operacji modyfikacji, która rozpatrywana jest jako zdarzenie;
- warunek może być opisany za pomocą frazy `WHERE`;
- w triggerach nie można stosować następujących operatorów SQL: wszystkie operatory `CREATE`, wszystkie operatory `ALTER TABLE`, `ALTER DATABASE`, `DROP`, `GRANT`, `REVOKE`, `TRANCATE TABLE`, `LOAD DATABASE`, `TRANSACTION`, `RECONFIGURE`, `UPDATE STATISTICS`, `SELECT INTO`, wszystkie operatory `DISK`;
- trigger jest częścią transakcji; jeżeli trigger nie zostanie wykonany, to i cała transakcja nie zostanie wykonana;
- triggery wykorzystują tabele logiczne znajdujące się w pamięci operacyjnej i mają taką samą strukturę co tabele BD, dla których właśnie triggery zostały stworzone;

każdy rekord wstawiony do chronionej przez trigger tabeli BD będzie wstawiony również do tabeli logicznej;

- dowolna tabela może mieć dowolną liczbę dowolnych wyzwalaczy.

6.3.2. Definiowanie wyzwalacza

Wyzwalacz definiujemy za pomocą operatora `CREATE TRIGGER`. Składnia wyzwalacza zależy od SZBD, dla której jest on tworzony. Przeanalizujemy składnię polecenia na podstawie standardu SQL-99.

Przykład 6.10

Mamy tabelę *Pracownicy_Firmy* {***Id_Prac***, *Nazw_Prac, Imię_Prac, Pensja*}. Następnie chcemy stworzyć trigger, który reaguje na zdarzenie zmiany (`Update`) wartości *Pensja* dowolnego rekordu tabeli, posiadającego informacje o kierownikach różnych rang. Ponadto trigger musi zablokować wprowadzenie zmniejszonego rozmiaru pensji dla odpowiednich osób. Deklaracja triggera ma następującą postać (wiersze zostały ponumerowane, co ułatwi dalszą analizę):

```
1 CREATE TRIGGER PensjaTrigger
2 AFTER UPDATE OF Pensja ON Pracownicy_Firmy
3 REFERENCING
4 OLD ROW AS OldPens,
5 NEW ROW AS NewPens
6 FOR EACH ROW
7 WHEN (OldPens.Pensja > NewPens.Pensja)
8 UPDATE Pracownicy_Firmy
9 SET Pensja = OldPens.Pensja
10 WHERE Id_Prac = NewPens.Id_Prac;
```

Listing 6.4

Pierwszy wiersz list. 6.4 posiada właśnie nazwę całego zdania (`CREATE TRIGGER`) i nazwę triggera (*PensjaTrigger*).

Drugi wiersz opisuje zdarzenie triggera – fakt modyfikacji (`UPDATE`) wartości pola *Pensja* tabeli *Pracownicy_Firmy.*

Wiersze 3-5 podają sposób, za pośrednictwem którego fragmenty kodu triggera, „odpowiadające” za realizację warunku i akcji, mogą się odnosić do rekordu tabeli w poprzednim (tzn. do wypełnienia operacji modyfikacji) (*old*) i nowym (*new*) stanie; rekordy te oznaczono jako *OldPens* i *NewPens.*

Wiersz 6 sprawia, że kod triggera zostanie wykonany jeden raz dla każdego rekordu, który ma być zmodyfikowany. Jeżeli nie byłoby frazy `FOR EACH ROW` albo zamiast niej wpisana była fraza `FOR EACH STATEMENT` (jest frazą domyślną), to trigger zaczynałby działanie jednokrotnie dla polecenia SQL, niezależnie od tego, jaka liczba operacji modyfikacji danych powinna być wykonana. W takim wypadku zamiast pseudonimów rekordów (`Old Row` i `New Row`) można definiować pseudonimy tabel (`Old Table` i `New Table`).

Wiersz 7 przedstawia warunek triggera, który oznacza tutaj, że następna akcja powinna być wykonana tylko w przypadku, jeśli nowa wartość *Pensja* jest mniejsza od poprzedniej.

Wiersze 8-10 zawierają kody triggera, które opisują wykonywane akcje, jeśli wynikiem sprawdzenia warunku będzie wartość *true*. Taką akcją jest polecenie `UPDATE`, które powinno przywrócić poprzednią wartość pola *Pensja*.

Podany przykład nie może pokazać wszystkich aspektów realizacji mechanizmów SQL-triggerów. Poniżej krótko przeanalizujemy inne ważne aspekty:

- wiersz 2 określa, że akcja triggera wykonuje się po (`AFTER`) powstaniu zdarzenia, które go wywołało. Słowo kluczowe AFTER można zamienić słowem *przed* (`BEFORE`). Będzie to oznaczać, że warunek we frazie `WHERE` powinien być sprawdzony przed operacją modyfikacji;
- oprócz `UPDATE` mogą być wykorzystywane i inne dopuszczalne operatory (o tym była mowa wyżej). Słowo `OF` nie jest niezbędne w naszym wypadku. Słowo to nie może być wykorzystywane do definicji zdarzeń `INSERT` i `DELETE`;
- fraza `WHERE` nie jest niezbędna; jeżeli tej frazy nie ma, akcja będzie wykonywana za każdym razem po uruchomieniu triggera bez wykonywania poprzednich analiz;
- kod akcji triggera może posiadać dowolną liczbę komend SQL kończących się średnikiem; komendy te umieszczamy pomiędzy słowami kluczowymi `BEGIN` i `END`.

Przeanalizujmy nową sytuację.

Przykład 6.11

Zakładamy, że mamy uniemożliwić sytuację, gdy dowolny kierownik dostaje roczną pensje niższą od 50 000. Niech na wejściu będzie dana tabela *Pracownicy_Firmy*.

Do niepożądanego wyniku może dojść w sytuacjach: wstawiania nowego rekordu, modyfikacji istniejących wartości w polu *Pensja* lub usuwania rekordów.

W tej sytuacji ważne jest, że pojedyncze polecenie `INSERT` albo `UPDATE` może wstawić lub zmodyfikować kilka rekordów tabeli. W trybie wykonania tych komend średnia wartość pensji będzie się zmieniać. Trigger musi więc zareagować na wartość końcową.

W naszej sytuacji należy stworzyć osobne wyzwalacze dla trzech wymienionych wyżej przypadków (`INSERT`, `UPDATE`, `DELETE`).

Na list. 6.5 podano opis triggera, uruchamiającego się w odpowiedzi na zdarzenie modyfikacji rekordów.

```
1 CREATE TRIGGER AVGPesjaTrigger
2 AFTER UPDATE OF Pensja ON Pracownicy_Firmy
3 REFERENCING
4 OLD Table AS OldPens_śr,
5 NEW Table AS NewPens_śr
6 FOR EACH STATEMENT
7 WHEN (50000 > (SELECT AVG(Pensja) FROM Pracownicy_Firmy))
8 BEGIN
9 DELETE FROM Pracownicy_Firmy
10 WHERE (Id_Prac, Nazw_Prac, Imię_Prac, Pensja) IN NewPens_śr;
11 INSERT INTO Pracownicy_Firmy
12 (SELECT * FROM OldPens_śr)
13 END;
```

Listing 6.5

Kody dwóch innych wyzwalaczy są podobne i nieco prostsze.

Wiersze 3-5 w list. 6.5 posiadają deklaracje pseudonimów, pozwalających na odesłanie do tabel z poprzednimi i nowymi rekordami.

Wiersz 6 deklaruje, że trigger powinien być wykonywany jeden raz dla każdej operacji modyfikacji, niezależnie od liczby rekordów podlegających poprawieniu.

Wiersz 7 podaje warunek z zastosowaniem funkcji `AVG()`, która oblicza średnią wartość na podstawie danych po ich modyfikacji.

Wiersze 8-13 posiadają kody akcji wyzwalacza, jeżeli wynikiem analizy warunku będzie wartość *true*; przy tym pierwsza część tego kodu (wiersze 9-10) ma usunąć z tabeli nowe (zmodyfikowane) rekordy, druga zaś odwrotną operację (powtórne zmodyfikowanie) rekordów do stanu przed wykonaniem operacji `UPDATE`.

Dla porównania przeanalizujemy przykłady stworzenia triggegów dla tabel na platformie MS SQL *Server*.

Przykład 6.11

Mamy tabelę *Autorzy* {***Id_Aut***, *Nazw_Aut, Imię_Aut, Telefon, Miasto, Kontrakt*}. Zadaniem jest stworzenie triggera, uruchamiającego się przy operacjach wstawiania i modyfikacji rekordów. Domyślnie będzie to trigger `AFTER`. Trigger wygląda następująco:

```
USE pubs
GO
CREATE TRIGGER TriAutor
ON Autory
FOR INSERT, UPDATE
AS raiserror ('%d rows have been modified', 0, 1
@@rowcount)
RETURN
```

Ten trigger będzie zwracał informacje o liczbie zmodyfikowanych rekordów.

Rozważmy wynik działania triggera:

```
USE pubs
GO
INSERT INTO Autory
(Id_Aut, Nazw_Aut, Imię_Aut, Telefon, Miasto, Kontrakt)
VALUES
('33-12', 'Szeszko', 'Michał', '374-123-25', 'Lublin', 'A13')
```

Po wykonaniu tego polecenia na ekranie pojawi się informacja:

```
1 rows have been modified
(1 row(s) affected).
```

Zadania do samokontroli

1. Wyjaśnić znaczenie pojęcia „bezpieczeństwo BD”.
2. Podać definicje pojęć (wraz z przykładami):

- uprawnienie;
- podmiot;
- identyfikacja;
- rola;
- przywilej.

3. Podać przykłady poleceń SQL, umożliwiające:
   - tworzenie użytkownika;
   - zmianę identyfikatorów użytkownika;
   - definiowanie roli;
   - usuwanie roli;
   - przypisywanie uprawnień.
4. W odniesieniu do tabeli Pracownicy (tab. 6.3) napisać polecenia SQL, umożliwiające:
   - użytkownikowi *Admin_BCD* wykonanie operacji pobierania danych ze wszystkich pól oraz modyfikację danych w polach *Nazwisko* i *Imię*;
   - użytkownikowi *Nowacki* wykonanie wszystkich operacji nad danymi pracowników działu finansów.
5. Wymienić typy perspektyw.
6. W odniesieniu do tabel *Zamówienia* i *Towary* napisać polecenia SQL, których wynikiem będą wszystkie możliwe typy perspektyw. Podać wyniki tych operacji.

Tabela *Zamówienia*

Id_Zam	*Id_Tow*	*Iłość_Tow*
Z1	54	10
Z1	23	1
Z2	23	2
Z3	10	10
Z3	54	20

Tabela *Towary*

Id_Tow	*Id_Produc*	*Nazwa_Tow*	*Cena_Tow*
54	B1	Chleb	1.50
24	B7	Cukierki	10.00
10	B1	Mąka	2.20
11	B2	Mąka	2.30
76	B5	Cukier	3.50
25	B8	Cukierki	15.50
23	B7	Cukierki	19.20

7. Zalety i wady stosowania perspektyw.
8. Wyjaśnić rolę i działanie wyzwalaczy. Podać definicję wyzwalacza.

9. Na podstawie Prz. 6.10 zapisać kod triggera, blokującego wprowadzenie zmian w polu Pensja dla pracowników, których rozmiar pensji jest nie wyższy od 2500.
10. Sformułować w formie wyzwalaczy podane niżej warunki dla tabeli Komputery {Model, Szybkość, RAM, HDD, Cena}:
    - przy modyfikacji wartości *Cena* zagwarantować, że nie istnieją komputery z niższą ceną i taką samą szybkością;
    - przy modyfikacji wartości RAM i HDD zagwarantować, że wartość HDD będzie przynajmniej o 100 razy większa od wartości RAM.

Część III
Organizacja dostępu do danych

7. Aplikacje korzystające z baz danych

7.1. Organizacja połączenia z bazami danych w C++ Builder

7.1.1. Ogólne sposoby dostępu do baz danych

Zanim zaczniemy opis zastosowania *Database Desktop* (opcja *Database Desktop* znajduje się w menu *Tools* systemu C++ Builder), warto zwrócić uwagę na organizację połączeń użytkownika z bazami danych. Dostęp do BD zasadniczo może być realizowany dwoma sposobami: *bezpośrednio* oraz *poprzez ujednolicony interfejs*.

Pierwszy sposób oznacza *bezpośredni dostęp* do konkretnego systemu zarządzania bazą danych. Taki sposób dostępu jest bardzo wydajny, gdyż rozwiązania dedykowane wybierają najkrótszą drogę dostępu do bazy, wywołując bezpośrednio jej funkcje. Mają one również bardzo szerokie możliwości dostępu do bazy rodzimej. Zazwyczaj mogą wykorzystywać wszystkie funkcje zaimplementowane przez producenta systemu BD. Są czasami bardzo specyficzne, ale przede wszystkim bardzo wygodne. Z punktu widzenia programisty pozwalają na proste wykonywanie operacji, które w innych systemach mogłyby wymagać dużego nakładu pracy.

Rozwiązania te posiadają jedną zasadniczą wadę: uniemożliwiają wykorzystanie aplikacji napisanej z wykorzystaniem dostępu dedykowanego do pracy z innymi SZBD. Przystosowanie aplikacji do pracy z inną bazą wymaga wprowadzenia wielu zmian, a nawet może prowadzić do całkowitego przebudowania aplikacji. Tak więc rozwiązania dedykowane, choć bardzo efektywne, mogą okazać się całkowicie nieprzydatne w przypadku aplikacji, od których wymaga się częstej zmiany serwera bazy danych lub też współpracy z wieloma serwerami jednocześnie. Tworzenie takiego oprogramowania wynika z potrzeb agregowania danych z różnych serwerów baz danych (np. w dużej firmie) bądź z tworzenia aplikacji przeznaczonych dla wielu odbiorców.

W takich przypadkach lepiej sprawdzają się rozwiązania poprzez *ujednolicony interfejs*. Umożliwiają one dostęp do praktycznie dowolnej bazy (o ile producent dostarczył odpowiedni sterownik), jednakże taki sposób dostępu do baz danych musi uwzględniać różnice w działaniu różnych SZBD. Poszczególni producenci prześcigają się w dodawaniu nowej funkcjonalności do swoich produktów. W przypadku rozwiązań dedykowanych funkcjonalność jest niewątpliwą zaletą, gdyż podnosi efektywność i wydajność aplikacji. Natomiast w przypadku tworzenia ujednoliconego interfejsu nie może być wykorzystana, gdyż inne serwery mogą nie implementować tych rozwiązań. Oznacza to, że metody ogólnego dostępu mogą używać tylko tych rozwiązań, które znajdują się w większości serwerów baz danych, całkowicie ignorując nowatorskie rozwiązania poszczególnych producentów.

Omówimy teraz najważniejsze metody dostępu do systemów zarządzania BD, stosując następujący podział:

- *bezpośredni* – dostęp realizowany poprzez komponenty rodzime (natywne); ogólny schemat połączenia podany jest na rys. 7.1;
- poprzez *ujednolicony interfejs* (np. BDE, ODBC, JDBC, ADO, DBExpress).

Rys. 7.1. Schemat bezpośredniego połączenia aplikacji z BD

Dostęp poprzez rozwiązania dedykowane realizuje bezpośredni dostęp do funkcji systemu zarządzania bazą danych. Aplikacja musi sama zadbać o konfigurację dostępu do bazy.

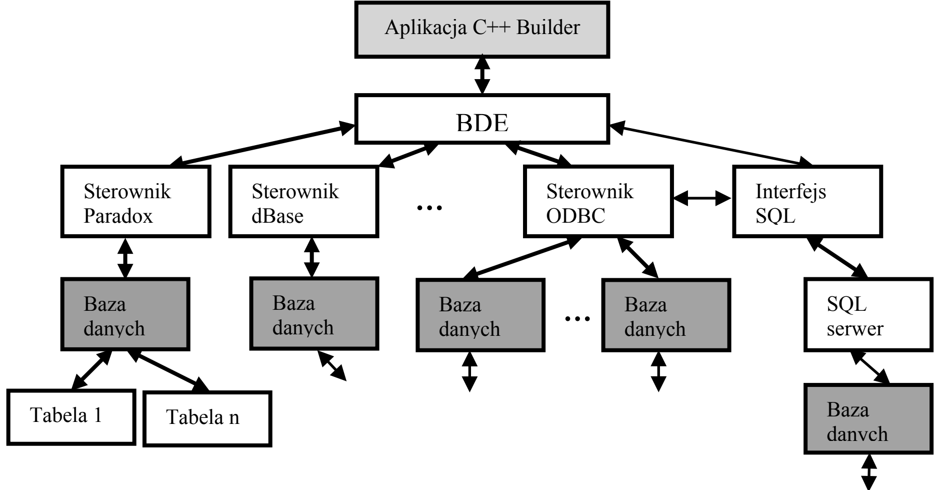
Zalety oraz wady tego modelu zostały opisane wyżej. Możemy dodać do tego, że zadanie to realizują w narzędziach Borland C++ Builder, a również w Delphi takie komponenty, jak:

- *InterBase Express* – dostęp do serwera *InterBase*;
- *Oracle Direct* – dostęp do serwera *Oracle*.

7.1.2. Dostęp do baz danych przez Borland Database Engine

Najprostszym narzędziem i podstawą pracy C++ Buildera z bazami danych jest BDE – procesor BD firmy Borland. Zakładki BDE i *Data Access* palety komponentów C++ Builder opierają się właśnie na BD BDE.

BDE jest pośrednikiem pomiędzy aplikacją a BD. Daje on użytkownikowi uniwersalny interfejs, dzięki czemu użytkownik nie musi modyfikować aplikacji przy zmianie realizacji BD. Aplikacja C++ Builder nigdy nie zwraca się bezpośrednio do BD, a wyłącznie do BDE poprzez *pseudonim* bazy danych i nazwę danej tabeli. Połączenie aplikacji z BD odpowiada schematowi podanemu na rys. 7.2.



Rys. 7.2. Schemat połączenia aplikacji z BD poprzez BDE

BDE jest realizowany w postaci dołączanych ***bibliotek DLL*** (pliki IDAPI01.dll, IDAPI32.dll). Te biblioteki, jak również inne posiadają API (ang. *Application Program Interface*), interfejs oprogramowania aplikacji – lista procedur i funkcji do pracy z BD, z których korzysta aplikacja.

Za pomocą *aliasu (pseudonimu)* BDE wybiera pasujący do danej bazy ***sterownik*** – program, który posiada mechanizm współdziałania z BD odpowiedniego typu (z odpowiednim SZBD). Jeżeli taki sterownik będzie znaleziony, to DBE łączy się za jego pośrednictwem z daną tabelą, przetwarza zapytanie użytkownika i zwraca do aplikacji wynik zapytania. BDE wspiera bezpośredni dostęp do takich SZBD, jak *Paradox*, *dBase*, MS *Access*, *FoxPro*.

Jeżeli BDE nie posiada wymaganego sterownika, to można wykorzystać sterownik ODBC (ang. *Open DataBase Connectivity*) – biblioteka DLL podobna do wyżej wspomnianej, ale stworzona przez firmę Microsoft. Biblioteka ta zawarta jest w pliku ODBC.dll. Połączenie z tabelą poprzez ODBC wymaga więcej czasu niż połączenie poprzez BDE, ale dzięki tej możliwości C++ Builder może pracować z większością nowoczesnych BD.

BDE wspiera język SQL, co umożliwia wymianę danych z SQL serwerami: *Sybase*, MS SQL, *Oracle*, *InterBase*.

BDE dostarcza ujednolicony interfejs dostępu do różnych formatów BD. Dzięki temu możliwe jest wykorzystanie wielu różnych formatów baz danych bez konieczności poznawania czy zakupu dodatkowych komponentów. Drugą ważną zaletą, wynikającą ze stosowania BDE, jest łatwy sposób zmiany wykorzystywanej bazy, bez wprowadzania dodatkowych zmian do aplikacji. Trzecią istotną zaletą jest dostępność takich mechanizmów, jak: transakcje, dwukierunkowe kursory, uaktualnienia danych wykorzystujące pamięć podręczną, a także obsługa XML. Mechanizmy te mogą być bardzo wymagające pod względem wykorzystywanej pamięci RAM czy przestrzeni dyskowej.

Interfejs BDE można wykorzystać do tworzenia zarówno aplikacji wykorzystujących lokalne bazy danych, jak i aplikacji *klient/serwer*. Pierwszy model obsługi baz danych nazywamy *architekturą jednowarstwową*. W modelu tym dane oraz aplikacja zlokalizowane są na tym samym komputerze. Drugi model to *architektura dwuwarstwowa*, w której dane są zlokalizowane na serwerze, podczas gdy aplikacje rezydują na dowolnej liczbie komputerów połączonych z owym serwerem.

Architektura jednowarstwowa znajduje zastosowanie w prostych jednostanowiskowych aplikacjach, w których najważniejszymi cechami są cena oraz wydajność, a wielodostępność, bezpieczeństwo oraz integralność schodzą na drugi plan. Przedstawimy teraz zalety architektury jednowarstwowej:

- bardzo duża wydajność w przypadku korzystania z biblioteki VCL;
- wiele z wykorzystywanych formatów baz danych jest bezpłatna;
- można wykorzystać format *dBase*, który jest rozpoznawany przez większość aplikacji pozwalających na importowanie danych;
- zmiana formatu bazy BD jest stosunkowo prosta;
- zazwyczaj wykorzystywane są tylko standardowe komponenty BD, dzięki czemu nie jest konieczne nabywanie dodatkowych bibliotek.

Do wad *architektury jednowarstwowej* należy zaliczyć brak zaawansowanych mechanizmów bezpieczeństwa oraz integralności danych.

Architektura dwuwarstwowa zapewnia wielodostępowość do danych zgromadzonych na serwerze, przy zaawansowanych mechanizmach bezpieczeństwa oraz zachowaniu inte-

gralności danych. Do obsługi tej technologii BDE używa specjalnej warstwy, zwanej łączami SQL (SQL *Links*). Warstwa ta może być traktowana jako warstwa tłumacząca pomiędzy interfejsem API BDE a interfejsem API bazy danych. Wersja Enterprise zawiera łącza SQL do takich serwerów, jak: *Oracle*, MS SQL *Server*, DB2 oraz *InterBase*, podczas gdy wersja *Professional* zawiera łącza tylko do serwera *InterBase*.

Przedstawimy teraz zalety architektury dwuwarstwowej:

- wszystkie systemy zarządzania bazą danych obsługiwane przez SQL *Links* stosują technologię klient/serwer;
- wszystkie SZBD obsługiwane przez SQL *Links* są w pełni relacyjnymi bazami danych;
- przejście z jednego sytemu SZBD na inny jest stosunkowo proste;
- łącza SQL *Links* są dużo bardziej wydajne niż ODBC.

Oto największe wady architektury dwuwarstwowej:

- większość serwerów baz danych to serwery komercyjne, co wiąże się z obowiązkiem uiszczenia stosownych opłat licencyjnych;
- sterowniki SQL *Links* są dedykowane do konkretnego SZBD, co ogranicza ich stosowalność jedynie do baz: *Oracle*, *Sybase*, MS SQL *Server*, *Informix*, DB2, *Access* oraz *InterBase*;
- interfejs BDE oraz SQL *Links* wprowadzają dwie dodatkowe warstwy pomiędzy aplikację a bazę danych, co może niekorzystnie wpłynąć na wydajność.

ODBC to ogólny, programowy interfejs API, umożliwiający programistom tworzenie aplikacji dla każdej bazy danych posiadającej sterownik ODBC. Wymaga to wprawdzie znajomości API ODBC, ale nie trzeba już poznawać interfejsu każdej bazy, dla której tworzono aplikacje. API ODBC nie był łatwy w użyciu, jednakże był pierwszym krokiem na drodze do rozwiązania problemu istnienia tak wielu API. Jak wiadomo, ODBC szybko stał się standardową metodą dostępu do danych.

O niektórych dodatkowych cechach ODBC będzie mowa niżej, jak też o innych najnowszych technologiach (ADO i dbExpress). Teraz zwrócimy uwagę na organizację dostępu do BD na podstawie komponentów środowiska C++ Bulder.

7.2. Dostęp do baz danych na podstawie komponentów środowiska Borland C++ Builder

7.2.1. Komponenty BDE

W rozdziale tym omówimy metody dostępu do baz danych, wykorzystujące komponenty dostarczane wraz ze środowiskiem Borland C++ Builder. Zaczniemy od komponentów BDE.

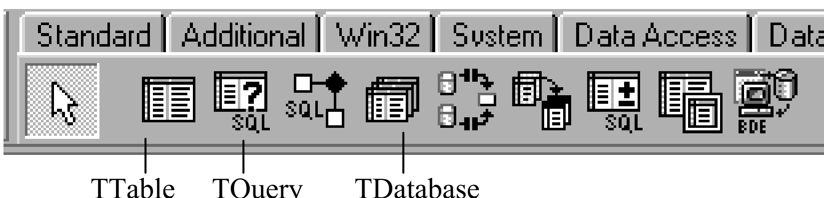
Zanim przejdziemy do omawiania komponentów, zdefiniujemy kilka podstawowych pojęć, których będziemy używać tylko w tych konkretnych znaczeniach. Pozwoli to nam uniknąć niejednoznaczności. Oto one:

- **zbiór danych** może być utożsamiany z dwuwymiarową strukturą danych (tabelą). Jest on uporządkowanym zbiorem danych zorganizowanym w kolumny oraz wiersze. Pojęcie zbioru danych jest pojęciem abstrakcyjnym, którego odpowiednikiem w bibliotece VCL jest klasa `TDataSet`;
- **tabela** stanowi konkretyzację pojęcia zbioru danych. Odnosi się ona do konkretnego fizycznego pliku, posiadającego unikalną nazwę oraz lokalizację. Odpowiednikiem tabeli w bibliotece VCL jest klasa `TTable`;
- **zapytanie** stanowi konkretyzację pojęcia zbioru danych, jednakże nie odnosi się ona do żadnego konkretnego pliku. Stanowi ono niejako tabelę tymczasową będącą wynikiem zapytania. Zapytanie jest reprezentowane w bibliotece VCL poprzez klasę `TQuery`;
- **baza danych** jest zorganizowanym zbiorem tabel, który jest identyfikowany poprzez nazwę katalogu (bazy lokalne *Paradox*, *dBase*) lub nazwę pliku na serwerze SQL. Odpowiednikiem bazy danych w bibliotece VCL jest klasa `TDatabase`;
- **indeks** określa logiczną kolejność rekordów w tabeli. Kolejność ta opiera się na rosnącej lub malejącej wartości wyrażenia stanowiącego konkatenację wybranych pól bazy danych.

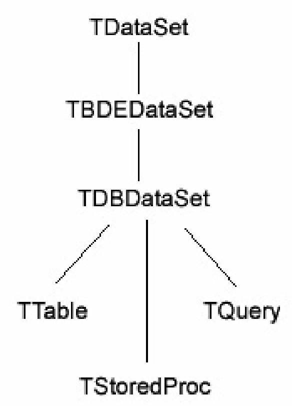
Komponenty związane z obsługą baz danych zależnych od BDE znajdują się na zakładce BDE palety komponentów (rys. 7.3).

Najważniejszymi komponentami powyższej zakładki są: `TTable`, `TQuery`, `TStoredProc`. Wszystkie one bezpośrednio wywodzą się od klas `TDBDataSet`, `TBDEDataSet` oraz abstrakcyjnej klasy `TDataSet`. Hierarchię tych klas przedstawia rys. 7.4.

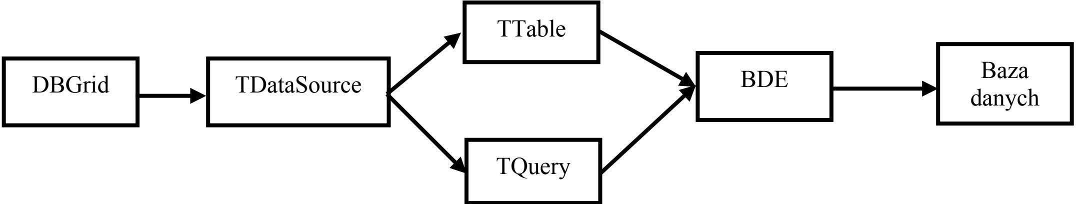
Dalszy schemat, tj. rys. 7.5, przedstawia współdziałanie komponentów odwzorowania-edytowania (np. `DBGrid`) danych i BDE.



Rys. 7.3. Komponenty zakładki BDE



Rys. 7.4. Hierarchia klas



Rys. 7.5. Współdziałanie komponentów odwzorowania-edytowania (DBGrid) danych i BDE

Klasa TDataSet reprezentuje abstrakcyjny zbiór danych z możliwościami nawigacji oraz niezależnymi od fizycznej reprezentacji elementami dostępu do danych. Pełni ona rolę klasy bazowej dla wszystkich komponentów zbiorów danych. Wprowadza ona wiele właściwości, metod oraz zdarzeń, z których większość jest abstrakcyjna.

Klasa TBDEDataSet spełnia rolę zbioru danych zależnego od BDE. Kolejna klasa to TDBDataSet, która umożliwia łączenie zbiorów danych w bazy danych oraz w sesje. Żadna z tych klas nie jest bezpośrednio wykorzystywana w aplikacji. Zamiast tego używane są klasy potomne klasy TDBDataSet, takie jak: TTable, TQuery, TStoredProc. Aby jednak zrozumieć i poznać ich działanie, należy najpierw omówić klasę TDataSet, która zawiera bardzo wiele właściwości, metod oraz zdarzeń wykorzystywanych w tych komponentach.

7.2.2. Klasa TDataSet

Jak już wcześniej powiedzieliśmy, klasa TDataSet jest klasą bazową dla wszystkich komponentów zbiorów danych. Jest ona niezależna od jakiejkolwiek technologii BD, a w szczególności od BDE. Przedstawimy teraz jej właściwości, metody oraz zdarzenia. Większość z nich jest abstrakcyjna, a zatem będą one zaimplementowane w klasach potomnych.

7.2.2.1. Otwieranie i zamykanie zbioru danych

Otwarcie zbioru powoduje, że zostaje on „wypełniony” danymi. Oznacza to, że możemy z nich korzystać, wykonując wszystkie dostępne operacje. Zatem jakakolwiek operacja na zbiorze danych musi być poprzedzona jego otwarciem. Do otwierania zbiorów danych wykorzystujemy właściwość Active. Przypisanie jej wartości true powoduje otwarcie zbioru danych, natomiast przypisanie wartości false spowoduje jego zamknięcie. Z właściwości Active możemy korzystać z poziomu inspektora obiektów (*Object Inspector*) lub w kodzie aplikacji. Dodatkowo w trakcie wykonania programu mogą (do *otwarcia/zamknięcia* zbiorów danych) zostać wykorzystane dwie metody: Open() oraz Close(). Działanie tych metod jest równoważne z odpowiednim ustawieniem właściwości Active. Poniższe fragmenty kodu źródłowego (list. 7.1) powinny rozwiać wszelkie wątpliwości.

```
procedure TDataSet.Open;
begin
  Active := true;
end;

procedure TDataSet.Close;
begin
```

```
  Active := false;
end;
```

Listing 7.1

Zmiana właściwości `Active` powoduje cały szereg działań, z których przedstawimy tylko najważniejsze.

Ustawienie właściwości `Active` na `true` powoduje następującą sekwencję operacji:

- wywołanie zdarzenia `BeforeOpen`;
- ustawienie stanu zbioru danych na `dsBrowse`;
- otwarcie kursora;
- wywołanie zdarzenia `AfterOpen`.

Natomiast ustawienie właściwości `Active` na `false` powoduje następującą sekwencję operacji:

- wywołanie zdarzenia `BeforeClose`;
- ustawienie stanu zbioru danych na `dsInactive`;
- zamknięcie kursora;
- wywołanie zdarzenia `AfterClose`.

7.2.2.2. Przemieszczanie się po rekordach zbioru danych

Klasa `TDataSet` dostarcza kilku metod, pozwalających w łatwy sposób poruszać się po zbiorze danych (przesuwać pozycję kursora z jednego rekordu na inny). Aktualny rekord, na który wskazuje kursor, nazywamy rekordem bieżącym. To właśnie tego rekordu będą dotyczyły wszystkie operacje wykonywane na zbiorze danych (np. edycja, usuwanie). Pierwsze dwie metody służące do nawigowania po zbiorze danych to `First()` i `Last()`. Pierwsza z nich przesuwa kursor na pierwszy rekord, druga zaś na ostatni rekord w danym zbiorze danych.

Kolejne dwie metody `Next()` i `Prior()` przesuwają odpowiednio kursor o jeden rekord w przód lub o jeden rekord wstecz. Ostatnią metodą jest funkcja `MoveBy(int Distance)`, która przesuwa aktualną pozycję kursora o zadaną liczbę rekordów. Liczba dodatnia przesuwa w przód, ujemna zaś w kierunku przeciwnym. Funkcja ta poza tym zwraca liczbę rekordów, o jaką faktycznie udało się przesunąć pozycję kursora. Wartość ta z reguły równa jest wartości parametru wejściowego funkcji. W przypadku gdy kursor znajduje się w pozycji przed końcem zbioru danych, a wartość parametru jest zbyt duża, zwrócona wartość będzie mniejsza niż wartość parametru. W takim przypadku kursor ustawi się na ostatnim rekordzie zbioru danych.

Klasa `TDataSet` posiada dwie właściwości typu logicznego (`bool`), pozwalające kontrolować położenie kursora względem pierwszego i ostatniego rekordu. Pierwsza z nich, `Bof`, przyjmuje wartość `true`, jeżeli bieżący rekord jest aktualnie pierwszym rekordem zbioru danych, w przeciwnym przypadku przyjmuje wartość `false`. Druga właściwość, `Eof`, przyjmuje wartość `true`, jeżeli bieżący rekord jest jednocześnie ostatnim rekordem zbioru danych, w przeciwnym przypadku przyjmuje wartość `false`.

Opisane powyżej działanie tych dwóch właściwości różni się nieco w praktyce. Problem dotyczy przypadku, gdy stojąc na ostatnim rekordzie (właściwość `Eof` przyjmuje wartość `true`), cofniemy się na rekord poprzedni (właściwość `Eof` przyjmuje wartość `false`)), a następnie znów powrócimy na ostatni. W takim przypadku właściwość `Eof` nie

przyjmie wartości `true`. Działanie takie nie jest jednak błędem, lecz celową konwencją funkcjonowania BDE, który w warunkach potencjalnego wielodostępu nie może być pewny, że rekord, który przed chwilą był ostatnim rekordem zbioru danych, jest nim nadal (np. inna aplikacja mogła dodać na końcu zbioru danych nowy rekord). Analogiczny problem dotyczy właściwości `Bof`. Tak więc BDE ustawia właściwość `Bof` w następujących sytuacjach:

- otwarcie zbioru danych;
- wywołanie metody `First()`;
- próba przejścia przed pierwszy rekord zbioru danych (za pomocą metody `Prior()`).

BDE ustawia natomiast właściwość `Eof` na `true` w następujących przypadkach:

- otwarcie pustego zbioru danych;
- wywołanie metody `Last()`;
- próba przejścia za ostatni rekord zbioru danych (za pomocą metody `Next()`).

Oprócz tego powyższe dwie właściwości przyjmują wartość `true`, w przypadku gdy zbiór danych lub ustawiony zakres jest pusty. We wszystkich pozostałych przypadkach przyjmują wartość `false`.

Jeszcze jednym ważnym elementem, związanym z nawigacją po rekordach zbioru danych, są *zakładki*. Umożliwiają one zapamiętanie bieżącej pozycji kursora oraz późniejsze jej odtworzenie. Zakładka reprezentowana jest w klasie `TDataSet` poprzez właściwość `Bookmark` typu `AnsiString`. Przechowuje ona informacje o danej pozycji kursora w zbiorze danych. W celu zapamiętania bieżącej pozycji kursora należy właściwość `Bookmark` przypisać tymczasowej zmiennej typu `AnsiString`. Późniejsze przypisanie właściwości `Bookmark` wartość tymczasowej zmiennej spowoduje ustawienie kursora na zapamiętaną pozycję. Poniższy fragment kodu (list. 7.2) pokazuje przykład użycia zakładki:

```
AnsiString zakladka;
zakladka = Table1->Bookmark; // zapamiętuje aktualną pozycję
/* tutaj wykonujemy operacje na zbiorze danych, mogące zmienić
aktualną pozycję kursora (np. Next(), Locate(), itp...) */
Table1->Bookmark = zakladka; // odtwarza zapamiętaną pozycję
```

Listing 7.2

7.2.2.3. Dostęp do pól zbioru danych

Dostęp do poszczególnych pól zbioru danych uzyskujemy dzięki klasie `TField` oraz klasom potomnym. Oprócz standardowych operacji związanych z odczytywaniem i zapisywaniem danych umożliwia ona także, dzięki wykorzystaniu jej właściwości, takie operacje, jak: tworzenie pól oraz modyfikację już istniejących, zmianę ich kolejności, usuwanie istniejących pól, a także tworzenie pól obliczanych i pól lookupowych.

Klasa `TDataSet` dostarcza trzech sposobów dostępu do pól zbioru danych:

- właściwość tablicowa `System::Variant FieldValues[AnsiString FieldName];`
- funkcja `TField* _fastcall FieldByName(const AnsiString FieldName);`
- właściwość `TFields* Fields`.

Właściwość tablicowa `FieldValues` umożliwia dostęp (odczyt lub zapis) do wartości poszczególnych pól bieżącego rekordu. Argument *FieldName* jest nazwą pola, z którego chcemy odczytać lub zapisać daną wartość.

Łańcuch znaków *FieldName* może być nazwą pojedynczego pola lub nazwami kilku pól oddzielonych średnikami. Jeżeli pole o danej nazwie nie istnieje, to zostanie zgłoszony wyjątek klasy `EDatabaseError`. Dodatkowo właściwość ta jest domyślną właściwością tablicową, co oznacza, że można opuścić jej nazwę.

Poniższe przykłady pokazują odczyt i zapis wartości do poszczególnych pól bieżącego rekordu.

Przykład 7.1

```
AnsiString s,
s = Table1->FieldValues["Imię"]; // obie instrukcje,
s = (*Table1) ["Imię"]; // są równoważne,
...
Table1->FieldValues["nazwisko"] = s; // obie instrukcje
(*Table1) ["Nazwisko"] = s; // są równoważne
```

Przykłady 7.1 pokazują również, że wielkość liter w nazwie pola jest nieistotna.

Pokażemy teraz, w jaki sposób uzyskać jednocześnie dostęp do kilku pól rekordu:

```
const AnsiString Napis = „Imię: %s \nNazwisko: %s";
Variant TabArr;
AnsiString s;
TabArr = Table1->FieldValues[„Imię;Nazwisko"];
s = TabArr.GetElement(0);
ShowMessage(Format(Napis, ARRAYOFCONST ((s, TabArr.GetElement(1)))));
```

Listing 7.3

Kolejne dwa sposoby dostępu do pól nie zwracają konkretnych wartości danego pola, ale wskaźnik na obiekt klasy `TField`. Oprócz metod dostępu do wartości pola zawiera ona także wiele innych metod. Tutaj ograniczymy się tylko do tych pierwszych, natomiast pozostałe omówimy w następnym rozdziale.

Funkcja `FieldByName()` przyjmuje jako parametr nazwę pola, a zwraca wskaźnik klasy `TField`, który odpowiada nazwie danego pola. Natomiast właściwość `Fields` zwraca wskaźnik klasy `TFields`. Klasa ta posiada właściwość tablicową `Fields` indeksowaną od 0, a zwracająca wskaźnik klasy `TField`, reprezentujący pole o danym indeksie.

Posiadając już konkretny wskaźnik na obiekt klasy `TField`, możemy odczytać lub zapisywać wartość reprezentowanego przez niego pola. Do tego celu wykorzystujemy zestaw właściwości związanych z różnymi typami pola (zależnymi od typów przechowywanych wartości). Właściwości te zostaną przedstawione w tab. 7.1.

Tabela 7.1. Zestaw właściwości związanych z różnymi typami pola

Właściwość	*Typ wartości*
AsBCD	TBcd
AsBoolean	Boolean
AsCurrency	Currency
AsDateTime	TDateTime
AsFloat	Double
AsInteger	Integer
AsSQLTimeStamp	TSQLTimeStamp
AsString	String
AsVariant	Variant

Przykład 7.2 pokazuje użycie powyższych właściwości.

Przykład 7.2

```
AnsiString s;
Currency c;
s = Table1->FieldByName(„Imię")->AsString;
//…
Table1->Fields->Fields[2]->AsCurrency = c;
```

Porównajmy teraz pokrótce trzy przedstawione sposoby dostępu do wartości pól danego rekordu. Pierwsza właściwość `FieldValues` jest wygodnym sposobem dostępu do danych. Wystarczy znać tylko nazwę pola. Dzięki zwracanemu typowi wariantowemu (`Variant`) nie musimy wykorzystywać żadnych dodatkowych mechanizmów konwersji.

Możemy również uzyskać dostęp do kilku wartości pól, wykorzystując zmienną wariantową (tablicową). Ponadto właściwość ta jest domyślną właściwością tablicową, co pozwala nam opuścić jej nazwę. Z drugiej strony właściwość ta pracuje zawsze na wartościach typu `Variant`, co jest nieco wolniejsze od dostępu za pomocą właściwości klasy `TField`.

Drugi sposób dostępu do wartości pól umożliwia nam funkcja `FieldByName()`. Do danego pola odwołujemy się za pomocą jego nazwy. Natomiast, aby uzyskać dostęp do jego wartości, musimy skorzystać z jednej z właściwości dostępu do danych (`AsXxx`).

Ostatnim sposobem dostępu do pól jest wykorzystanie właściwości `Fields`. Aby uzyskać dostęp do danego pola, musimy znać jego indeks, czyli położenie w danym zbiorze danych. Jeżeli pola tworzone są dynamicznie, to kolejność pól we właściwości `Fields` odpowiada fizycznej kolejności pól w zbiorze danych. Jeżeli natomiast pola tworzone są za pomocą edytora pól, to ustalona w nim kolejność będzie kolejnością pól we właściwości `Fields`. Tak więc właściwość ta znajduje zastosowanie w dwóch przypadkach. W pierwszym, gdy chcemy iterować za pomocą pętli wszystkie pola zbioru danych. W drugim natomiast, gdy zamierzamy pracować ze zbiorem danych, którego struktura jest nieznana na poziomie projektowania.

Wszystkie powyższe właściwości umożliwiają dostęp tylko do wartości pól bieżącego rekordu. Aby więc uzyskać dostęp do pól innego rekordu, należy skorzystać z jednej z metod nawigacyjnych.

7.2.2.4. Modyfikowanie danych

Edycja danych może odbywać się tylko wtedy, kiedy zbiór danych znajduje się w trybie edycji. Tryb ten uzyskujemy dzięki wywołaniu metody `Edit()`. Wprowadzone zmiany możemy utrwalić za pomocą metody `Post()` lub wycofać się z nich, wywołując metodę `Cancel()`.

Edycja danych przebiega według następującego schematu:

- wywołanie metody `Edit()`;
- przypisanie nowych wartości odpowiednim polom rekordu;
- utrwalenie zmian poprzez metodę `Post()` lub wycofanie się z wprowadzonych zamian za pomocą metody `Cancel()`.

Oto prosty przykład edycji danych.

Przykład 7.3

```
Table1->Edit();
Table1->FieldValues[„Nazwisko"] = „Kowalski";
Table1->Post();
```

Utrwalenie zmian następuje także automatycznie przy zmianie aktualnej pozycji kursora (podczas wywołania metod nawigacyjnych). Jeżeli zbiór danych jest pusty, to wywołanie metody `Edit()` spowoduje wywołanie metody `Insert()`.

Wywołanie metody `Edit()` pociąga za sobą wiele operacji, które przedstawimy w kolejności, w jakiej zostaną wykonane:

- sprawdzenie właściwości `CanModify`. Jeżeli jej wartością jest `false`, to zbiór danych nie może być modyfikowany i zostaje wyrzucony wyjątek;
- wywołanie zdarzenia `BeforeEdit`;
- ustawienie stanu zbioru danych na `dsEdit`;
- wywołanie zdarzenia `AfterEdit`.

Niektóre zbiory danych są zbiorami tylko do odczytu. Przykładem mogą być tabele, będące plikami tylko do odczytu (np. zapisane na CD-ROM-ie), jak również wirtualne zbiory danych będące wynikami zapytań SQL. W takich przypadkach właściwość `CanModify` przejmie wartość `false`.

Dodawanie nowych rekordów może być realizowane dwoma sposobami. Pierwszy z nich polega na wykorzystaniu metody `Insert()`, która wstawia (przed rekordem bieżącym) nowy pusty rekord do zbioru danych. Po jej wywołaniu użytkownik może, w sposób analogiczny do edytowania danych, wprowadzać dane do poszczególnych pól, a następnie bądź to zatwierdzać, bądź to wycofywać się z wprowadzonych zmian. Drugi sposób polega na wykorzystaniu metody `Append()`, która wstawia (za ostatnim rekordem) nowy pusty rekord do zbioru danych.

Tak więc różnica w działaniu obu metod dotyczy miejsca wstawienia nowego rekordu. Działanie metod `Insert()` czy `Append()` jest bardzo podobne do działania metody `Edit()`. Mianowicie, po ich wywołaniu sprawdzana jest właściwość `CanModify`.

Wstawienie nowego rekordu pociąga za sobą zmianę pozycji kursora, co wywołuje parę zdarzeń, związaną ze zmianą bieżącego rekordu zbioru danych. Zdarzenia te to `BeforeScroll` oraz `AfterScroll`. A zatem po wywołaniu metod `Insert()` lub `Append()` nastąpi następująca sekwencja działań:

- sprawdzenie właściwości CanModify. Jeżeli jej wartością jest false, to zbiór danych nie może być modyfikowany i zostaje wyrzucony wyjątek;
- wywołanie zdarzenia BeforeInsert;
- wywołanie zdarzenia BeforeScroll;
- ustawienie stanu zbioru danych na dsInsert;
- wywołanie zdarzenia AfterInsert;
- wywołanie zdarzenia AfterScroll.

Różnice pomiędzy metodami Insert() i Append() będą widoczne tylko w przypadku tabel typu *Paradox* czy *dBase*, gdyż w przypadku baz danych SQL fizyczne położenie rekordu w bazie zależne jest od implementacji serwera SQL.

Przykład 7.4
Oto prosty przykład pokazujący dodanie nowego rekordu:

```
Table1->Insert();
Table1->FieldByName("Imię")->AsString = "Jan";
Table1->Post();
```

Klasa TDataSet dostarcza dwóch dodatkowych metod, pozwalających wstawić nowy rekord wraz z wartościami poszczególnych pól. Pierwsza z nich to InsertRecord(), druga zaś to AppendRecord(). Oto ich deklaracje:

```
void __fastcall InsertRecord(const System::TVarRec * Values,
                     const int Values_Size);
void __fastcall AppendRecord(const System::TVarRec * Values,
                     const int Values_Size);
```

Obie powyższe metody wstawiają nowy, wypełniony (wartościami przekazanymi poprzez argument Values) rekord do zbioru danych. Jednocześnie zostaje on zatwierdzony w bazie danych. Nowo dodany rekord staje się rekordem bieżącym.

Działanie metody AppendRecord() jest analogiczne do działania metody InsertRecord(). Różnica pomiędzy nimi związana jest z miejscem, w którym zostanie wstawiony nowy rekord. Została ona przedstawiona podczas opisu metod Insert() oraz Append().

7.2.2.5. Usuwanie danych

Do usuwania rekordów służy metoda Delete(). Wywołanie jej skutkuje usunięciem bieżącego rekordu i upozycjonowaniem się zbioru na następnym rekordzie. Jeżeli zbiór danych jest pusty, to zostanie wygenerowany wyjątek klasy EDatabaseError.

Wywołanie metody Delete() powoduje:

- sprawdzenie, czy zbiór danych nie jest pusty;
- wywołanie metody CheckBrowseMode() (zatwierdza ona wprowadzone wcześniej zmiany);
- wywołanie zdarzenia BeforeDelete;
- usunięcie bieżącego rekordu;
- usunięcie wszystkich buforów zaalokowanych przez usunięty rekord;

- ustawienie stanu zbioru na `dsBrowse`;
- ustawienie kursora na następny rekord zbioru danych (jeżeli usunięty rekord był ostatni, to kursor zostanie ustawiony na rekordzie poprzednim);
- wywołanie zdarzenia `AfterDelete`.

7.2.2.6. Tryby pracy zbioru danych

Tryby pracy (stany) zbioru danych reprezentowane są przez właściwość `State`, która jest właściwością tylko do odczytu. Stan zbioru danych określa, jakie operacje mogą być wykonane na jego danych.

Wywołanie wielu metod klasy `TDataSet` powoduje zmianę stanu zbioru danych, a tym samym właściwości `State`. Przykładowo: wywołanie funkcji `Insert()` powoduje przejście zbioru danych w tryb wstawiania, a tym samym zmianę właściwości `State` na `dsInsert`. Właściwość `State` jest typu `TDataSetState`, który jest zdefiniowany następująco:

```
enum TDataSetState { dsInactive, dsBrowse, dsEdit, dsInsert,
dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,
dsCurValue, dsBlockRead, dsInternalCalc, dsOpening };
```

Listing 7.4

Poniższa tabela (tab. 7.2) przedstawia tryby pracy zbioru danych.

Tabela 7.2. Tryby pracy zbioru danych

Wartość	*Opis*
`dsInactive`	Zbiór danych jest zamknięty, więc jego dane są niedostępne
`dsBrowse`	Dane mogą być przeglądane, ale nie mogą być modyfikowane. Jest to domyślny stan po otwarciu zbioru danych
`dsEdit`	Bieżący rekord może być modyfikowany
`dsInsert`	Bieżący rekord jest nowo dodawanym, ale jeszcze niezatwierdzonym rekordem zbioru danych
`dsSetKey`	Dotyczy tylko zbiorów danych klasy `TTable` i `TClientDataSet`. Stan umożliwia wyszukiwanie lub ustawianie zakresów rekordów. Dane mogą być przeglądane, ale nie mogą być modyfikowane
`dsCalcFields`	Wykonywana jest metoda związana ze zdarzeniem `OnCalcFields`. Pola niebędące polami obliczanymi nie mogą być modyfikowane
`dsFilter`	Wykonywana jest metoda związana ze zdarzeniem `OnFilterRecord`. Dane nie mogą być modyfikowane
`dsNewValue`	Stan krótkotrwały wykorzystywany tylko wewnętrznie przez metody klasy `TDataSet`. Ustawiany jest w momencie dostępu do właściwości `NewValue`
`dsOldValue`	Stan krótkotrwały wykorzystywany tylko wewnętrznie przez metody klasy `TDataSet`. Ustawiany jest w momencie dostępu do właściwości `OldValue`
`dsCurValue`	Stan krótkotrwały wykorzystywany tylko wewnętrznie przez metody klasy `TDataSet`. Ustawiany jest w momencie dostępu do właściwości `CurValue`
`dsBlockRead`	Kontrolki wrażliwe na dane (*data-aware*) nie są uaktualnione oraz nie są jeszcze wykonane zdarzenia związane z przejściem do następnego rekordu
`dsInternalCalc`	Stan krótkotrwały wykorzystywany tylko wewnętrznie przez metody klasy `TDataSet`. Występuje, gdy wartości pól, których właściwość `FileKind` przyjmuje wartość `fkInternalCalc`, powinny zostać obliczone
`dsOpening`	Zbiór danych jest w trakcie operacji otwarcia

Otwarcie zbioru danych zmienia stan z dsInactive na dsBrowse. Metody Edit() oraz Insert() powodują odpowiednio przejście w tryby pracy dsEdit oraz dsInsert.

Jeżeli zbiór danych jest komponentem klasy TTable lub TClientDataSet, to wywołanie metod SetKey() oraz SetRange() ustawia stan zbioru na dsSetKey.

Zatwierdzanie (Post()) lub anulowanie wprowadzonych danych (Cancel()), dodawanie rekordu (InsertRecord(), AppenRecord()), a także usuwanie (Delete()) powoduje przejście do stanu dsBrowse. Zamknięcie zbioru danych (Close()) ustawia stan dsInactive.

Niektóre tryby pracy, takie jak dsCalcFields, dsFilter, dsNewValue, dsOldValue oraz dsCurValue, nie są widoczne oraz nie mogą być ustawiane przez aplikację. Stany te są zmieniane automatycznie, kiedy wywoływane są zdarzenia OnCalcFields, OnFilterRecord lub następuje dostęp do niektórych właściwości komponentów klasy TField.

Zmiana stanu zbioru danych wywołuje zdarzenie OnStateChange klasy TDataSource, z którym powiązany jest dany zbiór danych.

7.2.2.7. Filtrowanie danych

Filtrowanie umożliwia ograniczenie zbioru dostępnych rekordów. Do tego celu możemy użyć właściwości Filter lub zdarzenia OnFilterRecord. Aby uaktywnić filtrowanie, należy ustawić właściwość Filtered na true. Ustawienie jej na false sprawi, że warunki filtrowania nie będą brane pod uwagę. Właściwość Filter typu AnsiString służy do określenia warunku filtra. Tylko dane spełniające ten warunek zostaną udostępnione.

Przykład 7.5

Przykładowy warunek mógłby wyglądać następująco:

```
... wiek >= 18 ...
```

Wybiera on wszystkie osoby, które ukończyły 18. rok życia.

Przykład 7.6

Następny przykład wybiera wszystkie osoby mieszkające w Lublinie:

```
... miasto = 'Lublin'...
```

Jak widzimy, w powyższym przykładzie łańcuchy znaków muszą być ujmowane w pojedyncze apostrofy. Jeżeli nazwa pola zawiera spacje, to musi być ujęta w nawiasy kwadratowe:

```
... [Nr Albumu] = 1 ...
```

Powyższe warunki filtrowania nazywamy elementarnymi. Składają się one tylko z nazwy pola oraz wymaganej wartości.

Warunki elementarne można negować (not) lub łączyć w warunki złożone za pomocą operatów logicznych (or, and).

Przykład 7.7

Przykładowo: warunek złożony mógłby wglądać następująco:

```
... (miasto = 'Lublin' or miasto = 'Warszawa') and wiek >= 18 ...
```

Jeżeli chcemy uzyskać rekordy z niewypełnionymi polami, to musimy użyć słówka NULL.

Przykład 7.8

Poniższy przykład wybiera osoby mieszkające w Lublinie oraz te, dla których miasto zamieszkania jest nieokreślone:

```
... miasto = 'Lublin' or miasto = NULL lub
miasto = 'Lublin' or miasto is NULL ...
```

Oba powyższe przykłady są równoważne. Jednakże w warunkach `WHERE` poleceń języka SQL poprawna jest tylko składnia użyta w drugim warunku filtrowania.

Tabela 7.3 prezentuje operatory możliwe do wykorzystania w warunkach filtrowania.

Tabela 7.3. Opis operatorów do wykorzystania w warunkach filtrowania

Operator	*Opis*
<	Mniejszy
>	Większy
<=	Mniejszy lub równy
>=	Większy lub równy
=	Równy
<>	Różny
`AND`	Logiczne "i"
`NOT`	Logiczne "nie"
`OR`	Logiczne "lub"
+	Dodaje liczby, łączy łańcuchy znaków, dodaje liczby do dat (tylko niektóre sterowniki)
-	Odejmuje liczby, odejmuje daty lub odejmuje liczby od dat (tylko niektóre sterowniki)
*	Mnoży dwie liczby (tylko niektóre sterowniki)
/	Dzieli dwie liczby (tylko niektóre sterowniki)
*	Jeżeli występuje na końcu, to zastępuje dowolny ciąg znaków (`FilterOptions` nie może zawierać opcji `foNoPartial-Compare`)

Klasa `TDataSet` zawiera właściwość zbiorową `FilterOptions`. Określa ona, jak będą traktowane wartości łańcuchowe występujące w warunkach filtrowania. Definicja typu właściwości jest następująca:

```
enum TFilterOption {foCaseInsensitive, foNoPartialCompare};
typedef Set<TFilterOption, foCaseInsensitive,
foNoPartialCompare>TFilterOptions;
```

Listing 7.5

Znaczenie dwóch powyższych wartości podane jest w tab. 7.4.

Tabela 7.4. Znaczenie wartości `foCaseInsensitive` i `foNoPartialCompare`

Wartość	*Opis*
`foCaseInsensitive`	Porównania wartości będących łańcuchami znaków są niewrażliwe na wielkość liter
`foNoPartialCompare`	Znak gwiazdki (`*`) w zakończeniu łańcuchów znaków jest traktowany literalnie. Brak tej wartości we właściwości `FilterOptions` sprawia, że znak gwiazdki (`*`) na końcu łańcuchów znakowych jest traktowany jako dowolny ciąg znaków

Oprócz właściwości `Filter` możemy wykorzystać zdarzenia `OnFilterRecord`. Zdarzenie to jest wywoływane, gdy dany rekord staje się rekordem bieżącym oraz właściwość `Filtered` przyjmuje wartość `true`. Metoda obsługi tego zdarzenia posiada parametr `Accept` przekazywany przez zmienną. Jeżeli przyjmuje on wartość `true`, to dany rekord spełnia warunek filtrowania i rekord zostanie udostępniony. W przeciwnym wypadku (`Accept == false`) rekord zostanie pominięty.

Przykład 7.9

Poniższy przykład pokazuje warunek filtrowania, wybierający tylko studentów 3. roku:

```
void __fastcall TDM1::CustomerFilterRecord(TDataSet *DataSet, bool
&Accept)
{
  Accept = DataSet->FieldByName("ROK")->AsInteger == 3;
}
```

Zdarzenia `OnFilterRecord` używamy wtedy, kiedy kryteria warunku wyszukiwania nie mogą być zaimplementowane we względnie prostej konstrukcji warunku, mogącej wystąpić we właściwości `Filter`. Przykładem może być chociażby porównywanie wartości pól, które nie jest wspierane przez lokalne typy baz danych (*Paradox*, *dBASE*, *Access*, *FoxPro*). Natomiast warunek filtrowania definiowany w zdarzeniu może przybierać dowolne formy, ograniczone tylko możliwościami języka programowania.

Należy pamiętać o tym, że gdy określimy warunek filtrowania we właściwości `Filter` oraz zdefiniujemy zdarzenie `OnFilterRecord`, może okazać się, że oba warunki filtrowania wzajemnie się wykluczają.

Obok metod zbioru danych umożliwiających poruszanie się po rekordach tego zbioru jako całości (`First()`, `Last()`, `Next()`, `Prior()`), dostępne są analogiczne metody nawigacyjne symulujące taką wędrówkę po przefiltrowanym zbiorze rekordów (`FindFirst()`, `FindLast()`, `FindNext()`, `FindPrior()`). Umożliwiają one znalezienie pierwszego, ostatniego, następnego oraz poprzedniego rekordu spełniającego zdefiniowany warunek filtrowania. Metody te są bezparametrowymi funkcjami typu `bool`. Zawracają one wartość `true`, gdy została zmieniona pozycja bieżącego rekordu (w przeciwnym przypadku zwracają `false`).

Ostatnia uwaga dotyczy zbiorów jednokierunkowych, dla których zdefiniowanie warunków filtrowania generuje wyjątek. Zatem filtrowanie dostępne jest tylko dla zbiorów dwukierunkowych. Dodajmy tylko, że wszystkie zbiory oparte na interfejsie BDE są dwukierunkowe.

Zbiory jednokierunkowe umożliwiają poruszanie się tylko do przodu, zbiory dwukierunkowe są nawigowalne w obydwu kierunkach. Zbiory jednokierunkowe wywodzą się z klasy `TCustomSQLDataSet`. Wywołują one wyjątki dla wszystkich metod nawigacyj-

nych oprócz `First()` i `Next()`. Nie wspierają one również filtrowania, zakładek, pól odnośników oraz wszystkich innych mechanizmów wykorzystujących wewnętrzny bufor. BDE buforuje rekordy pozyskane z bazy danych, dlatego też wszystkie zbiory danych korzystające z BDE są dwukierunkowe.

7.2.2.8. Metody wyszukiwania

Dwa najważniejsze sposoby wyszukiwania zdefiniowane w klasie `TDataSet` implementowane są przez metodę `Locate()` oraz `Lookup()`. Pierwsza z nich po znalezieniu szukanego rekordu pozycjonuje się na nim. Druga natomiast nie zmienia pozycji bieżącego rekordu, zamiast tego zwraca wartości znalezionego rekordu. Oto definicja metody `Locate()`:

```
enum TLocateOption { loCaseInsensitive, loPartialKey };
typedef Set<TLocateOption, loCaseInsensitive,
loPartialKey>  TLocateOptions;
virtual bool __fastcall Locate(const AnsiString KeyFields,
const System::Variant &KeyValues, TLocateOptions Options);
```

Listing 7.6

Funkcja `Locate()` wyrzuca wyjątek, gdy zbiór danych jest jednokierunkowy. Zwraca natomiast wartość `true`, gdy szukany rekord został znaleziony, w przeciwnym przypadku zwraca `false`. Pierwszy parametr `KeyFields` jest nazwą pola lub łańcuchem nazw pól rozdzielonych średnikami. Drugi parametr określa szukane wartości pól. Jest on typu `Variant`, a zatem może być zarówno pojedynczą wartością, jak i tablicą zawierającą szukane wartości. Trzeci parametr określa opcje wyszukiwania.

Element `loCaseInsensitive` oznacza, że w trakcie przeszukiwania nie będą rozróżniane duże i małe litery. Drugi element `loPartialKey` oznacza, że porównanie wartości dopuszcza występowanie częściowego dopasowania. Jeżeli wartość przekazana w parametrze `KeyValues` jest początkiem wartości pola danego rekordu, to mówimy o częściowym dopasowaniu tychże wartości. Metoda `Locate()` jest nadpisywana w klasach potomnych, które reprezentują dwukierunkowe zbiory danych.

Przykład 7.10

Ten przykład pokazuje wykorzystanie metody `Locate()`:

```
TLocateOptions opcje;
opcje << loCaseInsensitive;
if (!Table1->Locate("Nazwisko", "Kowalski", opcje))
  ShowMessage("Szukany rekord nie został znaleziony");
```

Jeżeli chcemy przeszukiwać zbiór danych ze względu na większą liczbę pól, to musimy zastosować funkcję `VarArrayOf()`. Jej deklaracja jest następująca:

```
Variant __fastcall VarArrayOf(const Variant * Values,
const int Values_Size);
```

Tworzy ona i wypełnia jednowymiarową tablicę wariantową.

Przykład 7.11

Oto kolejny przykład, wyszukujący tym razem osoby na podstawie imienia i nazwiska:

```
TLocateOptions opcje;
opcje << loCaseInsensitive;
```

```
Variant dane[] = {"Jan", "Kowalski"};
if (!Table1->Locate("Imię;Nazwisko", VarArrayOf(dane, 2), opcje))
  ShowMessage("Szukany rekord nie został znaleziony");
```

Drugą metodą wyszukiwania jest funkcja `Lookup()`. Wyrzuca ona wyjątek, gdy zbiór danych jest jednokierunkowy. Zwraca zaś zmienną wariantową zawierającą wartości pól znalezionego rekordu. Jeżeli rekord nie zostanie znaleziony, to zwraca ona zmienną wariantową z wartością `varNull` (typ ten odczytujemy za pomocą funkcji `VarType()`).

Pierwszy parametr `KeyFields` jest nazwą pola lub łańcuchem nazw pól rozdzielonych średnikami. Drugi parametr określa szukane wartości pól. Jest on typu `Variant`, a zatem może być zarówno pojedynczą wartością, jak i tablicą zawierającą szukane wartości. Trzeci parametr jest nazwą pola lub łańcuchem nazw pól rozdzielonych średnikami, których wartości mają być zwracane przez funkcję (oczywiście, jeżeli rekord zostanie znaleziony).

Metoda `Lookup()` jest nadpisywana w klasach potomnych, które reprezentują dwukierunkowe zbiory danych.

Przykład 7.12
Oto przykład wykorzystania metody `Lookup()`:

```
const AnsiString Info = "Student mieszka w mieście %s i ma %s lat";
Variant v;
Variant dane[] = {"Jan", "Kowalski"};
  v = Table1->Lookup("Imię;Nazwisko",
VarArrayOf(dane, 2),"Miasto;Wiek");
  if (!VarIsNull(v))
    ShowMessage(Format(Info,            ARRAYOFCONST((v.GetElement(0),
v.GetElement(1)) ) ));
```

7.2.2.9. Zdarzenia klasy TDataSet

Klasa `TDataSet` udostępnia pokaźny zasób zdarzeń, który może zostać wykorzystany przez programistę w wielu sytuacjach. Mianowicie, pozwalają one sprawdzać poprawność danych, dokonywać obliczeń na wartościach pól, a także reagować na zdarzenia związane z dodawaniem, edycją oraz usuwaniem rekordów. Tabela 7.4 prezentuje wszystkie zdarzenia zbioru danych.

Tabela 7.4. Zdarzenia zbioru danych

Zdarzenie	*Opis*
`AfterCancel`	Wywoływane jest po anulowaniu zmian dokonanych w bieżącym rekordzie. Zdarzenie to wywoływane jest przez metodę `Cancel()`
`AfterClose`	Wywoływane jest po zamknięciu zbioru danych
`AfterDelete`	Wywoływane jest po usunięciu bieżącego rekordu. Zdarzenie to wywoływane jest przez metodę `Delete()`
`AfterEdit`	Wywoływane jest po przejściu zbioru danych w tryb edycji (`dsEdit`). Zdarzenie to wywoływane jest przez metodę `Edit()`
`AfterInsert`	Wywoływane jest po przejściu zbioru danych w tryb wstawiania (`dsInsert`). Zdarzenie to wywoływane jest przez metodę `Insert()` lub `Append()`
`AfterOpen`	Wywoływane jest po otwarciu zbioru danych

`AfterPost`	Wywoływane jest po zatwierdzeniu zmian dokonanych w bieżącym rekordzie. Zdarzenie to wywoływane jest przez metodę `Post()`
`AfterRefresh`	Wywoływane jest po odświeżeniu zbioru danych. Zdarzenie to wywoływane jest przez metodę `Refresh()`
`AfterScroll`	Wywoływane jest po zmianie pozycji bieżącego rekordu. Zdarzenie to wywoływane jest przez metody nawigacyjne
`BeforeCancel`	Wywoływane jest przed anulowaniem zmian dokonanych w bieżącym rekordzie. Zdarzenie to wywoływane jest przez metodę `Cancel()`
`BeforeClose`	Wywoływane jest przed zamknięciem zbioru danych
`BeforeDelete`	Wywoływane jest przed usunięciem bieżącego rekordu. Zdarzenie to wywoływane jest przez metodę `Delete()`
`BeforeEdit`	Wywoływane jest przed przejściem zbioru danych w tryb edycji (`dsEdit`). Zdarzenie to wywoływane jest przez metodę `Edit()`
`BeforeInsert`	Wywoływane jest przed przejściem zbioru danych w tryb wstawiania (`dsInsert`). Zdarzenie to wywoływane jest przez metodę `Insert()` lub `Append()`
`BeforeOpen`	Wywoływane jest przed otwarciem zbioru danych
`BeforePost`	Wywoływane jest przed zatwierdzeniem zmian dokonanych w bieżącym rekordzie. Zdarzenie to wywoływane jest przez metodę `Post()`
`BeforeScroll`	Wywoływane jest przed zmianą pozycji bieżącego rekordu. Zdarzenie to wywoływane jest przez metody nawigacyjne
`OnCalcFields`	Wywoływane jest, gdy obliczana zostaje wartość pól obliczanych (kalkulowanych). Zdarzenie to opisane będzie w dalszej części książki
`OnDeleteError`	Wywoływane jest podczas wystąpienia błędu przy usuwaniu bieżącego rekordu
`OnEditError`	Wywoływane jest podczas wystąpienia błędu przy przejściu zbioru danych w tryb edycji
`OnFilterRecord`	Wywoływane jest, gdy dany rekord staje się rekordem bieżącym oraz filtrowanie jest aktywne. Zdarzenie to opisane będzie w dalszej części książki
`OnNewRecord`	Wywoływane jest, gdy zbiór danych przechodzi w tryb wstawiania (`dsInsert`)
`OnPostError`	Wywoływane jest podczas wystąpienia błędu przy zatwierdzaniu zmian dokonanych w bieżącym rekordzie

7.2.3. Klasa TTable

Klasa `TTable`, dziedzicząca pośrednio po klasie `TDataSet`, umożliwia dostęp do tabeli bazy danych. Dostęp ten korzysta z interfejsu BDE.

7.2.3.1. Wizualizacja dostępu do tabeli

Zanim przejdziemy do szczegółowego omawiania komponentu `Table`, pokażemy, w jaki sposób uzyskać wizualny dostęp do danych zawartych w tabeli.

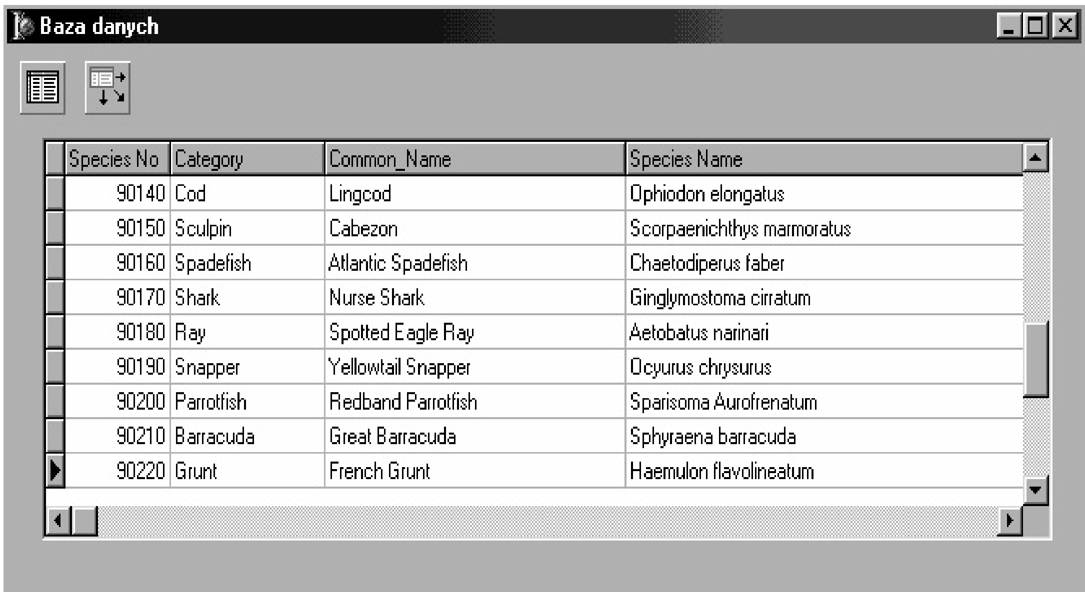
Na formatce układamy trzy komponenty:

- `Table` – zakładka BDE;
- `DataSource` – zakładka *Data Access*;
- `DBGrid` – zakładka *DataControls*.

W komponencie `Table` ustawmy następujące właściwości:

- `DatabaseName` = DBDEMOS*;*
- `TableName` = *biolife.db;*
- `Active = true`.

W komponencie `DataSource` ustawiamy właściwość `DataSet`, odpowiadającą nazwie komponentu `Table` (*Table1*). Natomiast w komponencie `DBGrid` ustawiamy właściwość `DataSource` równą nazwie komponentu `DataSource` (*DataSource1*). Jeżeli wszystkie właściwości zostały ustawione prawidłowe, to formatka powinna wyglądać jak w przykładzie na rys. 7.4.



Rys. 7.4. Przykładowy wygląd formatki BD

Jak widzimy, na powyższym zrzucie dane są dostępne również na etapie projektowania. Po uruchomieniu aplikacji zrzut wyglądałby podobnie, z wyjątkiem komponentów niewizualnych.

Przedstawimy teraz właściwości komponentu `Table`. Pierwszą z nich jest `DatabaseName`. Określa ona nazwę bazy danych, z którą będzie połączony zbiór danych klasy `TDBDataSet`. Właściwość ta może przyjmować: nazwę bazy danych (zdefiniowaną we właściwości `DatabaseName` komponentu `DatabaseName`), nazwę *aliasu* skojarzonego z daną bazą danych lub ścieżkę dostępu do katalogu, w którym znajdują się pliki tabel BD.

Nazwę tabeli określamy we właściwości `TableName`. Jeżeli nazwa bazy została określona prawidłowo, to nazwę tabeli możemy wybrać z listy rozwijanej, zawierającej wszystkie dostępne tabele. Właściwość `Active` została omówiona poprzednio. Zmiana wartości jednej z właściwości `DatabaseName` czy `TableName` pociąga za sobą zmianę wartości właściwości `Active` na `false`.

7.2.3.2. Kontrola dostępu do tabeli

Dostęp do tabeli można kontrolować za pomocą trzech właściwości: `CanModify`, `Exclusive`, `ReadOnly`. Pierwsza z właściwości, `CanModify`, jest właściwością tylko do odczytu. Została wprowadzona już w klasie `TDataSet`, jednakże zawsze zwraca wartość `true`. Natomiast w klasach reprezentujących zbiory jednokierunkowe przyjmuje

ona zawsze wartość `false`. W pozostałych klasach dziedziczących po klasie `TDataSet` właściwość ta jest ustawiana w zależności od rzeczywistych możliwości danego obiektu do modyfikacji danych zbioru danych.

Właściwość `CanModify` komponentu `Table` informuje, czy operacje modyfikacji danych tabeli (dodawanie, edycja, usuwanie rekordów) są możliwe. Jeżeli `CanModify` przyjmuje wartość `true`, to dane tabeli mogą być modyfikowane. W przeciwnym wypadku (`CanModify == false`) tabela jest tylko do odczytu. Właściwość `CanModify` jest ustawiana automatycznie w momencie otwarcia tabeli. Najczęściej właściwość `CanModify` przyjmuje wartość `false`, gdy:

- właściwość `ReadOnly` przyjmuje wartość `true`;
- inna aplikacja uzyskała wyłączny dostęp do tabeli;
- tabela bazy danych została zaprojektowana tylko do odczytu.

Nawet gdy właściwość `CanModify` przyjmuje wartość `true`, nie gwarantuje to możliwości modyfikacji danych tabeli. Jednym z czynników wpływających na taką sytuację jest brak odpowiednich uprawnień danego użytkownika do modyfikacji danych tabeli.

Druga właściwość `Exclusive` umożliwia wyłączny dostęp do danych tabeli typu *Paradox* oraz *dBase*. Ustawienie właściwości `Exclusive` na `true` przed otwarciem tabeli spowoduje, że otwarcie jej nastąpi w trybie wyłączności. Oznacza to, że nie będzie mogła z niej korzystać żadna inna aplikacja. Jeżeli dana aplikacja otworzy tabelę, która została otwarta przez inną aplikację w trybie wyłączności, to zostanie zgłoszony wyjątek.

Ustawienie właściwości `Exclusive` oraz właściwości `Active` na `true` w trybie projektowania (w *Object Inspectorze*) będzie skutkowało wyrzuceniem wyjątku podczas uruchomienia aplikacji, gdyż tabela została już otwarta w trybie wyłączności przez IDE. Zmiana wartości właściwości `Exclusive` wymaga zamknięcia tabeli.

Komponent `Table` może próbować uzyskać wyłączny dostęp do tabel baz danych serwerów SQL. Jednakże niektóre z nich nie wspierają otwierania tabel w trybie wyłączności. Inne natomiast mogą umożliwiać blokowanie tabel, ale pozwalają innym aplikacjom na dostęp do nich tylko w trybie odczytu.

Ostatnia właściwość `ReadOnly` określa, czy dana tabela jest tylko w trybie do odczytu. Ustawienie tej właściwości na `true` zabezpieczy tabelę przed operacjami dodawania, edycji oraz usuwania rekordów. Domyślnie właściwość `ReadOnly` przyjmuje wartość `false`, co oznacza, że dane tabeli mogą być modyfikowane. Jednakże nawet gdy właściwość `ReadOnly` przyjmuje wartość `true`, nie gwarantuje to możliwości modyfikacji danych tabeli. Jednym z czynników wpływających na taką sytuację jest brak odpowiednich uprawnień danego użytkownika do modyfikacji danych tabeli. Ustawienie właściwości `ReadOnly` na `true` powoduje, że właściwość `CanModify` przyjmuje wartość `false`.

7.2.3.3. Sortowanie tabeli

Tabelę sortujemy poprzez wykorzystanie właściwości `IndexName` oraz `IndexFieldName` typu `AnsiString`. Pierwsza z nich określa nazwę indeksu, według którego tabela jest aktualnie posortowana. Jeżeli nazwa ta jest pusta, to porządek rekordów określony jest poprzez domyślny indeks tabeli. Na przykład: dla tabel typu *Paradox* będzie to porządek ustalony na podstawie klucza głównego (indeks główny), a dla tabel typu *dBase* porządek zgodny z fizycznym położeniem rekordów. Jeżeli wartość właściwości jest poprawną nazwą indeksu, to porządek sortowania rekordów tabeli będzie ustalony na podstawie tego indeksu.

Druga właściwość `IndexFieldName` jest nazwą pola lub łańcuchem nazw pól oddzielonych średnikami, względem których określony zostanie porządek sortowania. Dla tabel typu *dBase* oraz *Paradox* pole będące wartością właściwości `IndexFieldName` musi być indeksowane. W przypadku podania kilku nazw pól oddzielonych średnikami, wymagane jest, aby istniał indeks oparty na tych samych polach (również w tej samej kolejności). Ograniczenia te nie dotyczą tabel serwerów SQL. Właściwości `IndexName` oraz `IndexFieldName` wykluczają się wzajemnie. Ustawienie jednej z nich powoduje wyczyszczenie drugiej.

Przykład 7.13

Oto przykłady użycia właściwości `IndexName` oraz `IndexFieldName`, które to zmieniają porządek rekordów tabeli:

```
Table1->IndexName = "Ind_Nazwisko"; {porządek ustalony według indeksu
Ind_Nazwisko}

Table1->IndexName = ""; //porządek ustalony według klucza głównego

Table1->IndexFieldName = "Imię;Nazwisko"; {porządek ustalony według
pól Imię i Nazwisko}

Table1->IndexFieldName = "Nr_Albumu"; {porządek ustalony według klucza
głównego(, którym jest pole Nr_Albumu)}
```

Listę wszystkich dostępnych indeksów możemy uzyskać dzięki metodzie `GetIndexNames()`. Oto jej deklaracja:

```
void __fastcall GetIndexNames(Classes::TStrings* List);
```

Przykład wypełnienia komponentu `ComboBox` nazwami indeksów tabeli:

```
Table1->GetIndexNames(ComboBox1->Items);
```

Lista przekazana jako parametr `List` procedury musi być uprzednio utworzona. Po wykonaniu tej funkcji obiekt `List` zostaje wypełniony nazwami indeksów, jakimi dysponuje ta tabela, z wyjątkiem klucza głównego tabeli *Paradox*, który nie posiada żadnej nazwy. Jeżeli chcemy otrzymać listę wszystkich indeksów, wraz z informacjami o ich strukturze, możemy skorzystać z właściwości `IndexDefs`. Jest ona kolekcją definicji indeksów, z której każdy z nich opisuje indeks danej tabeli. Bieżący stan właściwości `IndexDefs` może nie odzwierciedlać aktualnych wartości indeksów tabeli. Dlatego też przed użyciem tej właściwości należy wywołać metodę `Update()`, która uaktualni kolekcję indeksów.

Przykład 7.14

Poniższy przykład pokazuje sposób użycia właściwości `IndexDefs` do odczytu wszystkich indeksów (łącznie z indeksem głównym tabeli *Paradox*):

```
Table1->IndexDefs->Update();

ComboBox1->Clear();

  for (int i = 0;i < Table1->IndexDefs->Count; i++)

    if (Table1->IndexDefs->Items[i]->Name != "")

      ComboBox1->Items->Add(Table1->IndexDefs->Items[i]->Name);

    else

      ComboBox1->Items->Add("Indeks główny");
```

7.2.3.4. Zakresy tabeli

Zakresy pozwalają ograniczyć zakres widocznych rekordów. Tak więc ich działanie podobne jest do filtrów, z tą jednak różnicą, że działają na podstawie indeksów. Ustawienie

zakresu udostępnia tylko rekordy znajdujące się pomiędzy wartością początkową i końcową zakresu w kontekście bieżącego indeksu.

Zakresy ustawiamy dzięki metodzie `SetRange()`. Oto jej deklaracja:

```
void __fastcall SetRange(const System::TVarRec * StartValues, const
int StartValues_Size, const System::TVarRec * EndValues, const int
EndValues_Size);
```

Posiada ona dwa parametry: pierwszy jest tablicą wartości początkowych, drugi zaś końcowych definiowanego zakresu. Wywołanie tej metody ustawia nowy zakres na podstawie przekazanych wartości, kasując jednocześnie poprzedni zakres (oczywiście, jeżeli istniał).

Parametr `StartValues` jest otwartą tablicą, określającą wartości pól (wchodzących w skład bieżącego indeksu), które będą wartościami pierwszego rekordu. Natomiast wartości parametru `EndValues` będą wartościami ostatniego rekordu zakresu. Metoda `SetRange()` jest sekwencją trzech metod `SetRangeStart()`, `SetRangeEnd()` oraz `ApplyRange()`. Ustawienie zakresów skutkuje upozycjonowaniem się zbioru danych na pierwszym rekordzie. Wywołanie metody `SetRange()` powoduje następującą sekwencję zdarzeń:

- przejście zbioru danych w tryb `dsSetKey`;
- usunięcie poprzednich zakresów (oczywiście, o ile istniały);
- ustawienie początkowych i końcowych wartości zakresów;
- uaktywnienie zakresu tabeli.

Przykład 7.15

Poniższy przykład ilustruje użycie zakresów dla tabeli *Student*, której właściwość `IndexName` jest pustym łańcuchem, co oznacza, że aktualny indeks jest kluczem głównym:

```
Table1->SetRange(ARRAYOFCONST((100)),ARRAYOFCONST((300)));
```

Przykład ten wybiera wszystkie rekordy, których wartość klucza głównego (*Nr_Albumu*) zawiera się w przedziale [100, 300]. Drugi przykład wybiera wszystkich studentów, których imię i nazwisko znajduje się pomiędzy osobami *Anna Nowak* i *Jan Kowalski*:

```
Table1->IndexName = "Ind_ImięNazwisko";

Table1->SetRange(ARRAYOFCONST( ("Anna", "Nowak") ),

ARRAYOFCONST( ("Jan", "Kowalski") ) );
```

Jeżeli parametr `StartValues` lub `EndValues` zawiera mniej elementów niż liczba pól w indeksie, to pozostałe brakujące elementy będą pasowały do dowolnych wartości.

Metoda `SetRange()` wymaga posortowanego zbioru rekordów, dlatego też dla tabel typu *dBase* oraz *Paradox* wykorzystywane pola muszą być zaindeksowane. Do kasowania zakresu tabeli wykorzystujemy metodę `CancelRange()`.

Jak powiedzieliśmy powyżej, metoda `SetRange()` może być zastąpiona sekwencją trzech metod: `SetRangeStart()`, `SetRangeEnd()` oraz `ApplyRange()`. Użycie jednak zakresów w drugim ze sposobów pozwala ustalić, czy zdefiniowany zakres jest zbiorem otwartym, czy zamkniętym. W przypadku metody `SetRange()` jest ona zawsze zbiorem domkniętym.

Wywołanie pierwszej z metod `SetRangeStart()` ustawia stan zbioru danych na `dsSetKey`. Po jej wywołaniu przypisujemy już początkowe wartości zakresu. W tym celu możemy wykorzystać jeden z poznanych wcześniej sposobów. Podobne zadanie wykonuje metoda `SetRangeEnd()`, ale w odniesieniu do wartości końcowych zakresu.

Wywołanie metody `ApplyRange()` powoduje ustawienie zdefiniowanego zakresu. Jak już wspomnieliśmy, dzięki wykorzystaniu omawianej sekwencji metod możemy kontrolować „otwartość" oraz „domkniętość" zakresów. Dokonujemy tego za pomocą właściwości `KeyExclusive` typu `bool`. Ustawienie jej na `false` (ustawienie domyślne) spowoduje, że zakresy będą domknięte, natomiast ustawienie na `true` spowoduje, że będą otwarte.

Przykład 7.16

Prześledźmy następujący przykład:

```
Table1->SetRangeStart();
  Table1->KeyExclusive = true;
  Table1->FieldByName("Nr_Albumu")->AsInteger = 100;
Table1->SetRangeEnd();
  Table1->FieldByName("Nr_ Albumu")->AsInteger = 300;
Table1->ApplyRange();
```

Ustawiony zakres wybierze wszystkich studentów, których numer albumu jest większy niż 100 i mniejszy lub równy niż 300. Wyjaśnijmy jeszcze tylko, dlaczego instrukcja:

```
Table1->FieldByName("Nr_Albumu")->AsInteger = 100;
```

nie powoduje fizycznego przypisania wartości 100 polu *Nr_Albumu*. Jest to spowodowane tym, że wywołanie metody `SetRangeStart()` powoduje zmianę stanu zbioru na `dsSetKey`. Dzięki temu powyższa instrukcja nie powoduje zapisania wartości 100 polu *Nr_Albumu* (jak to zdarzyłoby się, gdyby stan zbioru danych byłby ustawiony na `dsEdit` czy `dsInsert`), ale dokonuje zapisania tej wartości jako początkowej wartości zakresu dla pola *Nr_Albumu*.

7.2.3.5. Wyszukiwanie rekordów

Oprócz poznanych wcześniej metod wprowadzonych w klasie `TDataSet` klasa `TTable` udostępnia własne metody korzystające z uporządkowania tabeli (właściwość `IndexName` czy `IndexFieldName`). Dla tabel typu *dBase* i *Paradox* można wyszukiwać tylko względem pól, które zostały zaindeksowane. Jak powiedzieliśmy wcześniej, tabele serwerów SQL nie stawiają aż tak ultymatywnego żądania, lecz brak takowego indeksu odbija się niekorzystnie na efektywności wyszukiwania.

Pierwszą z omówionych metod będzie metoda `FindKey()`. Oto jej deklaracja:

```
bool __fastcall FindKey(const System::TVarRec * KeyValues,
const int KeyValues_Size);
```

Parametr `KeyValues` jest tablicą zawierającą wartości wyszukiwania określone w kontekście bieżącego indeksu. Jeżeli zostanie znaleziony rekord, którego wartości są równe wartościom parametru funkcji, to zbiór danych pozycjonuje się na znalezionym rekordzie, a funkcja zwraca wartość `true`. W przeciwnym przypadku pozycja bieżącego rekordu nie zmienia się, funkcja natomiast zwraca wartość `false`. Przedstawmy teraz przykłady wykorzystania tej funkcji (Prz. 7.17).

Przykład 7.17

```
Table1->FindKey(ARRAYOFCONST( (200) )); // szuka studenta, którego
numer albumu jest 200
Table1->IndexName = "Ind_ImięNazwisko"; //szuka studenta o Imieniu
Table1->FindKey(ARRAYOFCONST( ("Jan", "Kowalski") )); // Jan
o nazwisku Kowalski
```

Jeżeli liczba elementów tablicy `KeyValues` jest mniejsza niż liczba pól zaindeksowanych, to pozostałe brakujące elementy będą pasowały do dowolnych wartości.

Użycie funkcji `FindKey()` wymaga od programisty pamiętania kolejności pól występujących w bieżącym indeksie. Możemy jednak skorzystać z pary metod `SetKey()` i `GotoKey()`, które rozwiązują powyższy problem. Wywołanie metody `SetKey()` wprowadza tabelę w stan `dsSetKey`. Następnie poszczególnym polom wchodzącym w skład indeksu przypisywane są wartości, które będą stanowiły kryterium wyszukiwania.

Wywołanie funkcji `GotoKey()` rozpoczyna wyszukiwanie. Jeżeli szukany rekord zostanie znaleziony, staje się rekordem bieżącym, a funkcja zwraca wartość `true`. W przeciwnym przypadku pozycja bieżącego rekordu nie zmienia się, funkcja zaś zwraca wartość `false`.

Przykład 7.18

Oto przykłady pokazujące użycie pary metod `SetKey()` i `GotoKey()`:

```
Table1->IndexName = "Ind_ImięNazwisko";
Table1->SetKey();
  Table1->Fields->Fields[1]->AsString = "Jan";
  Table1->Fields->Fields[2]->AsString = "Kowalski";
Table1->GotoKey();


Table1->IndexName = "Ind_ImięNazwisko";
Table1->SetKey();
  Table1->FieldByName("Nazwisko")->AsString = "Kowalski";
  Table1->FieldByName("Imię")->AsString = "Jan";
Table1->GotoKey();
```

Jak widzimy (w drugim przykładzie), dzięki odwołaniu się do metody `FieldByName()` nie musimy pamiętać kolejności pól występujących w indeksie, a jedynie, że wolno odwoływać się tylko do pól wchodzących w skład bieżącego indeksu.

Omówione powyżej metody realizują wyszukiwanie rekordu, którego wartości pól równe są wartościom stanowiącym kryterium wyszukiwania.

Klasa `TTable` dostarcza również metody umożliwiające realizację tzw. *wyszukiwania przybliżonego*. Polega ono na tym, że kryterium wyszukiwania spełnia ten rekord, którego wartości są równe wartościom zadanym (jako kryterium wyszukiwania), lub w przypadku jego braku ten rekord, którego wartości są bezpośrednio większe od poszukiwanych wartości. Tak jak w przypadku *wyszukiwania dokładnego* istnieją dwa sposoby realizacji *wyszukiwania przybliżonego*. Pierwszy z nich polega na wywołaniu następującej funkcji:

```
void __fastcall FindNearest(const System::TVarRec * KeyValues, const
int KeyValues_Size);
```

drugi natomiast polega na wywołaniu pary metod `SetKey()` oraz `GotoNearest()`.

Przykład 7.19

Poniższe przykłady wyjaśniają dokładnie sposoby wykorzystania powyższych metod *wyszukiwania przybliżonego*:

```
Table1->IndexName = "Ind_ImięNazwisko";
Table1->FindNearest(ARRAYOFCONST( ("Jan", "Kowalski") ));
Table1->IndexName = "Ind_ImięNazwisko";
```

```
Table1->SetKey();
  Table1->Fields->Fields[1]->AsString = "Jan";
  Table1->Fields->Fields[2]->AsString = "Kowalski";
Table1->GotoNearest();
```

Uwagi dotyczące metod wyszukiwania przybliżonego są analogiczne jak w przypadku metod realizujących *wyszukiwanie dokładne* z dwoma wyjątkami. W przypadku nieistnienia rekordu spełniającego kryteria *wyszukiwania przybliżonego* ostatni rekord staje się rekordem bieżącym. Metoda `FindNearest()` nie zwraca żadnej wartości (`void`), dlatego też w żaden sposób nie informuje użytkownika o efektach poszukiwań (uwaga ta dotyczy również metody `GotoNearest()`). Druga uwaga dotyczy możliwości użycia właściwości `KeyExclusive` (tylko w odniesieniu do metody `GotoNearest()`). Określa ona pozycję bieżącego rekordu w przypadku znalezienia rekordu spełniającego kryterium wyszukiwania. Jeżeli przyjmuje ona wartość `false`, to bieżącym rekordem stanie się rekord spełniający kryteria wyszukiwania. Jeżeli natomiast `KeyExclusive` przyjmie wartość `true`, to bieżącym rekordem stanie się rekord następny (oczywiście, w kolejności przyjętej przez bieżący indeks).

7.2.3.6. Relacje Master/Details

Relacje *Master/Details* (*Nadrzędny/Podrzędny*) stanowią wizualizację relacji typu *jeden-do-wielu* pomiędzy tabelami bazy danych. Dzięki nim możemy unaocznić fakt istnienia takich związków. Relacja ta polega na tym, że w *tabeli podrzędnej* (*Detail*) dostępne są tylko te dane, które odnoszą się (wartość *klucza obcego tabeli podrzędnej* równa jest wartości *klucza głównego tabeli nadrzędnej*) do rekordu bieżącego *tabeli nadrzędnej* (*Master*). Specyfika relacji *Master/Details* wymaga, aby *klucz obcy tabeli podrzędnej* był zaindeksowany.

Przytoczmy następujący przykład. Mamy dane dwie tabele: *Klienci* i *Zamówienia* (typowa relacja *jeden-do-wielu*). Pierwsza z nich jest *tabelą nadrzędną*, druga zaś *podrzędną*. Dzięki relacji *Master/Details* możemy stworzyć reprezentację graficzną, polegającą na tym, że w jednej tabeli będą wyświetlone dane klientów, natomiast w drugiej dane dotyczące zamówień dokonanych tylko przez klienta, który jest aktualnie bieżącym klientem w pierwszej tabeli. W celu realizacji tego przykładu wykorzystamy tabele zgrupowane w bazie DB*Demos*, która dostarczana jest wraz z interfejsem BDE.

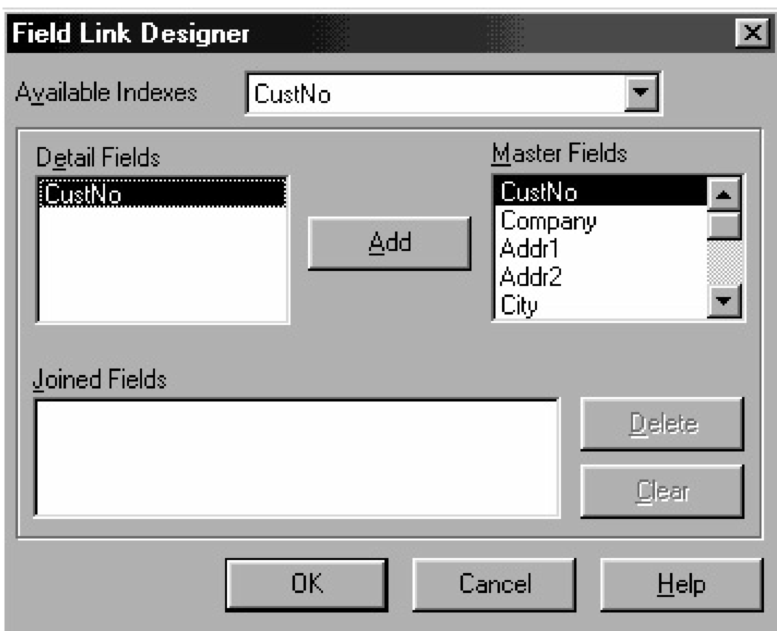
Relację *Master/Details* tworzymy, wykorzystując dwie właściwości: `MasterSource` oraz `MasterFields`. Pierwsza z nich określa komponent `DataSource` powiązany z *tabelą nadrzędną* (może to być dowolny zbiór danych). Połączenie pomiędzy obiema tabelami oparte jest na indeksie nałożonym na *klucz obcy tabeli podrzędnej*. Dlatego też należy odpowiednio ustawić właściwość `IndexName` lub `IndexFieldName`. Właściwość `MasterFields` określa *klucz główny tabeli nadrzędnej*. Ustawienia dwóch ostatnich właściwości możemy dokonać na etapie projektowania, wykorzystując edytor właściwości `MasterFields` *Field Link Designer*.

Przykład 7.20

Prześledźmy teraz przykład utworzenia relacji *Master/Details*:

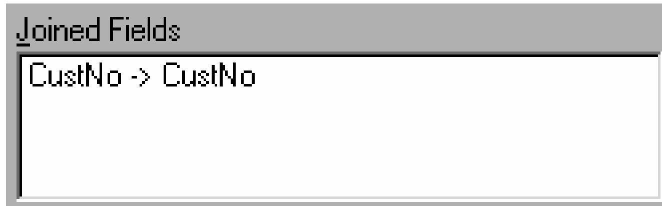
- kładziemy na formatce dwa komponenty `Table` oraz dwa komponenty `DataSource`;
- ustawiamy właściwości pierwszej tabeli (*Table1*):
  - `DatabaseName` = *DBDemos*,
  - `TableName` = *Customer*,
  - `Name` = *TKlienci*;

- ustawiamy właściwości drugiej tabeli (*Table2*):
  - `DatabaseName` = *DBDemos*,
  - `TableName` = *Orders*,
  - `Name` = *TZamowienia*;
- ustawiamy właściwości pierwszego źródła danych (*DataSource1*):
  - `DataSet` = *Tklienci*,
  - `Name` = *DSKlienci*;
- ustawiamy właściwości pierwszego źródła danych (*DataSource1*):
  - `DataSet` = *TZamowienia*,
  - `Name` = *DSZamowienia*;
- kładziemy na formatce dwa komponenty `DBGrid`;
- łączymy oba `DBGridy` z odpowiednimi komponentami `DataSource`:
  - `DBGrid1->DataSource` = *DSKlienci*,
  - `DBGrid2->DataSource` = *DSZamowienia*;
- ustawiamy właściwość `MasterSource` tabeli `TZamowienia` na *DSKlienci*;
- uruchamiamy edytor właściwości `MasterFields`. Okienko dialogowe o nazwie *Field Link Designer* pozwala nam na zdefiniowanie połączenia pomiędzy tabelami:
  - wybieramy z `ComboBoxa` *Available Indexes* indeks *CustNo* nałożony na klucz obcy o tej samej nazwie,
  - w `ListBoxie` *Detail Fields* dostępne są pola wchodzące w skład wybranego indeksu. W drugim `ListBoxie` *Master Fields* wyświetlane są wszystkie pola *tabeli nadrzędnej*. W obu komponentach zaznaczamy pola tworzące związek między tabelami. W naszym przypadku będą to: *klucz obcy* tabeli *Orders* oraz *klucz główny* tabeli *Customer*, oba o tej samej nazwie – *CustNo* (patrz rys. 7.5),



Rys. 7.5. Okno dialogowe *Field Link Designer*

– klikami przycisk *Add*, który ustanawia połączenie pomiędzy obiema tabelami, co objawia się odpowiednim wpisem w `ListBoxie` *Joined Fields* (rys. 7.6),

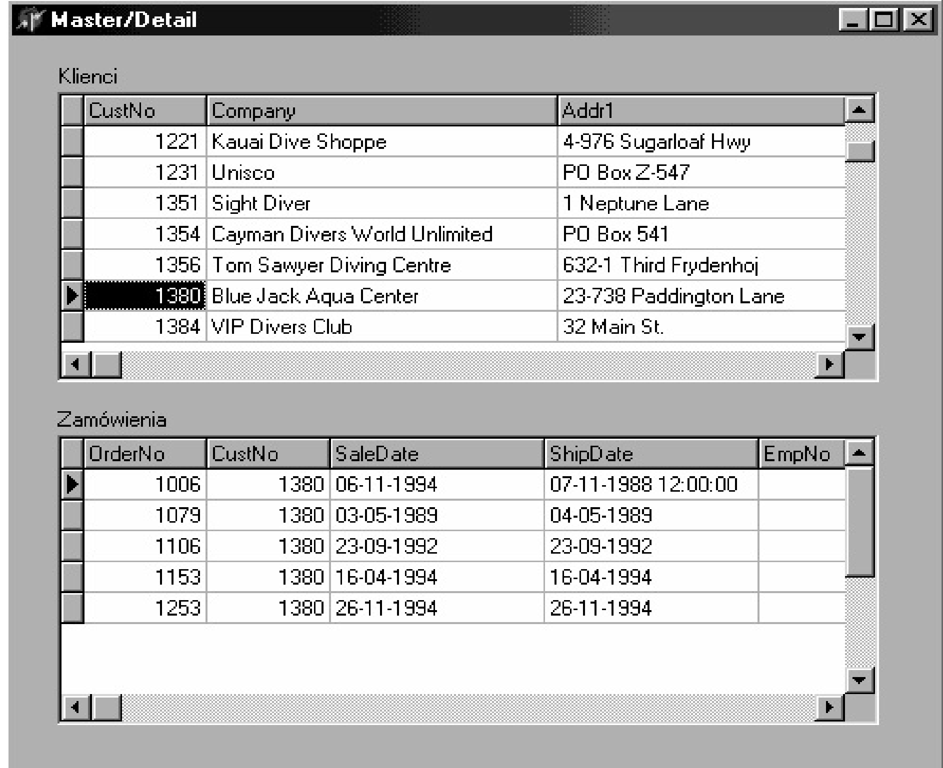


Rys. 7.6. Okno dialogowe *Joined Fields*

– na zakończenie klikami przycisk *OK*, aby zatwierdzić wprowadzone operacje;

- ustawiamy właściwość `Active` obu tabel na `true`;
- kompilujemy i uruchamiamy aplikację.

Aplikacja nasza działa już w założony sposób. Przesuwając się po rekordach *tabeli nadrzędnej*, zmienia się zawartość *tabeli podrzędnej*. Mianowicie, są w niej dostępne tylko te rekordy, które są powiązane z rekordem bieżącym w *tabeli nadrzędnej*. Dodajmy jeszcze, że aplikacja ta została utworzona bez konieczności wpisania chociażby jednej linijki kodu. Na rys. 7.7 widzimy naszą aplikację.



Rys. 7.7. Okno dialogowe po wypelnieniu połączenia pomiędzy tabelami *Klienci* i *Zamówienia*

7.2.3.7. Dynamiczne tworzenie tabel

Oprócz poznanego wcześniej sposobu projektowania tabel (przed utworzeniem aplikacji) możliwe jest także dynamiczne tworzenie tabel (z poziomu działającej aplikacji). Działanie to realizuje metoda `CreateTable()`, wykorzystując przy tym bieżącą definicję ta-

beli. Tak więc metoda ta musi zostać poprzedzona szeregiem czynności wstępnych. Oto one:

- utworzenie obiektu klasy `Ttable`;
- ustawienie właściwości `DatabaseName`;
- przypisanie właściwości `TableName` unikalnej nazwy nowo tworzonej tabeli;
- określenie typu tabeli: `ttDefault`, `ttParadox`, `ttDBase`, `ttASCII`. Ostatnie trzy wartości oznaczają odpowiednio: tabelę typu *Paradox*, *dBase* oraz plik tekstowy. Ustawienie wartości `ttDefault` sprawi, że określenie typu tabeli nastąpi na podstawie rozszerzenia nazwy pliku. Rozszerzenie *.db* lub jego brak oznacza tabelę typu *Paradox*, rozszerzenie *.dfb* określa tabelę typu *dBase*, natomiast rozszerzenie *.txt* plik tekstowy;
- zdefiniowanie struktury kolejnych pól za pomocą właściwości `FieldDefs`, a dokładniej za pomocą metody `Add()`, która zawiera cztery parametry:
  - nazwę pola,
  - typ pola (`TFieldType`),
  - rozmiar pola (istotny tylko dla pól typu `String` i `Memo`),
  - wymagane wartości pola (wartość `true` oznacza, że pole jest wymagane, wartość `false` dopuszcza wartość NULL);
- określenie struktury indeksów za pomocą metody `Add()` właściwości `IndexDefs`. Metoda ta zawiera następujące parametry:
  - nazwę indeksu,
  - nazwy pól oddzielone średnikami wchodzące w skład indeksu,
  - opcje indeksu typu `TIndexOptions`. Elementami tego zbioru mogą być następujące właściwości: `ixPrimary`, `ixUnique`, `ixDescending`, `ixExpression`, `ixCaseInsensitive`, `ixNonMaintained`. Ich znaczenie przedstawia tab. 7.5.

Tabela 7.5. Opcje indeksu typu `TIndexOptions`

Wartość	*Opis*
`ixPrimary`	Indeks będzie kluczem głównym tabeli (nie dotyczy tabel typu dBase)
`ixUnique`	Każda wartość indeksu musi być unikalna
`ixDescending`	Porządek indeksu będzie malejący
`ixExpression`	Indeks na bazie wyrażenia (dotyczy tabel typu dBase)
`ixCaseInsensitive`	Indeks będzie niewrażliwy na wielkość liter
`ixNonMaintained`	Indeks nie będzie automatycznie uaktualniany

Po określeniu powyższych czynności wstępnych możemy wywołać metodę `CreateTable()`, która utworzy tabelę o zdefiniowanej strukturze.

Przykład 7.21

Poniższy przykład tworzy tabelę *Studenci*:

```
Table1->Active = false;
Table1->DatabaseName = "Uniwersytet";
```

```
    Table1->TableType = ttParadox;
    Table1->TableName = "Studenci";

   if (!Table1->Exists)
    {
      Table1->FieldDefs->Clear();
      TFieldDef *pNewDef = Table1->FieldDefs->AddFieldDef();
        pNewDef->Name = "Nr_Albumu";
        pNewDef->DataType = ftInteger;
        pNewDef->Required = true;

      pNewDef = Table1->FieldDefs->AddFieldDef();
        pNewDef->Name = "Imię";
        pNewDef->DataType = ftString;
        pNewDef->Size = 20;

      pNewDef = Table1->FieldDefs->AddFieldDef();
        pNewDef->Name = "Nazwisko";
        pNewDef->DataType = ftString;
        pNewDef->Size = 30;

     Table1->IndexDefs->Clear();
     Table1->IndexDefs->Add("","Nr_Albumu",
TIndexOptions() <<ixPrimary);
     Table1->IndexDefs->Add("Ind_ImięNazwisko","Imię;Nazwisko",
TIndexOptions() << ixCaseInsensitive);
     Table1->CreateTable();
    }
```

7.2.4. Klasa TField

7.2.4.1. Klasa TField a typy danych

Klasa `TField` reprezentuje pole (kolumnę) tabeli. Jest ona klasą abstrakcyjną, która definiuje szereg metod i właściwości wykorzystywanych w klasach potomnych. Realizują one następujące funkcje:

- dostęp do wartości pól zbioru danych;
- konwersję pomiędzy wartościami różnych typów;
- kontrolę wprowadzanych przez użytkownika wartości;
- sposób wyświetlania danych;
- obliczanie pól kalkulowanych;
- obsługę pól odnośników (*lookup*).

Jako komponenty pól nie wykorzystuje się bezpośrednio klasy `TField`. Do tego celu służy cały szereg klas potomnych. Klasy dziedziczące po klasie `TField` są nierozerwalnie związane z typem pola, do jakiego odnoszą się w fizycznej tabeli (związek ten odzwierciedlony jest także w ich nazwach). Reprezentują one większość znanych typów pól różnych BD. Zostaną one przedstawione w tab. 7.6.

Tabela 7.6. Klasy i odpowiadające im typy danych

Klasa	*Opis*
`TADTField`	Reprezentuje abstrakcyjny typ danych – *Abstract Data Type* (*ADT*). Są to typy definiowane przez użytkownika, podobne do rekordów
`TaggregateField`	Reprezentuje pole agregowane, wykorzystywane w komponencie `ClientDataSet`
`TarrayField`	Reprezentuje pole, będące tablicą pól zawierających wartości skalarne lub abstrakcyjne
`TautoIncField`	Reprezentuje pole typu całkowitego z zakresu od 2,147,483,648 do 2,147,483,647, którego wartość jest unikatowa dla kolejnych rekordów (jest ona automatycznie zwiększana o 1)
`TBCDField`	Reprezentuje pole, którego wartości są liczbami rzeczywistymi o stałej liczbie cyfr po przecinku. Wykorzystywane często do przechowywania wartości monetarnych
`TbinaryField`	Reprezentuje pole binarne. Zazwyczaj niewykorzystywana bezpośrednio, będąca jednak klasą bazową dla dwóch kolejnych klas
`TBlobField`	Reprezentuje pole typu `BLOB` (duże obiekty binarne). Teoretyczne maksymalne ograniczenie to 2 GB
`TbooleanField`	Reprezentuje pole typu logicznego
`TbytesField`	Reprezentuje pole binarne o stałym rozmiarze
`TcurrencyField`	Reprezentuje pole typu walutowego z zakresu od (-/+)$5.0*10^{-324}$ do $1.7*10^{308}$
`TdataSetField`	Reprezentuje tabelę w obiektowo-relacyjnej bazie danych
`TdateField`	Reprezentuje datę
`TdateTimeField`	Reprezentuje datę/czas
`TfloatField`	Reprezentuje liczby zmiennoprzecinkowe
`TFMTBCDField`	Reprezentuje pole, którego wartości są liczbami rzeczywistymi o stałej liczbie cyfr po przecinku. Zapewniają one większą precyzję niż pola typu `TBCDField`, co odbija się nieco na ich efektywności
`TgraphicField`	Reprezentuje pole graficzne
`TguidField`	Reprezentuje pole, którego wartości są identyfikatorami GUID
`TIDispatchField`	Reprezentuje pole, którego wartości są wskaźnikami do interfejsów `IDispatch`
`TintegerField`	Reprezentuje 32-bitowe liczby całkowite
`TinterfaceField`	Reprezentuje pole, którego wartości są wskaźnikami do interfejsów `IUnknown`
`TlargeintField`	Reprezentuje 64-bitowe liczby całkowite
`TmemoField`	Reprezentuje pole tekstowe
`TreferenceField`	Reprezentuje pole, którego wartości są wskaźnikami lub referencjami do obiektów
`TsmallIntField`	Reprezentuje 16-bitowe liczby całkowite
`TSQLTimeStampField`	Reprezentuje datę/czas w zbiorach danych *DBExpress*

Klasa	*Opis*
TstringField	Reprezentuje pole łańcuchowe
TtimeField	Reprezentuje czas
TvarBytesField	Reprezentuje pole binarne o zmiennej długości
TvariantField	Reprezentuje pole wariantowe
TwideStringField	Reprezentuje pole łańcuchowe Unicode
TwordField	Reprezentuje 16-bitowe dodatnie liczby całkowite

Komponenty `Field` określają przede wszystkim typ danych pola. Ponadto umożliwiają ustalenie całego szeregu właściwości charakteryzujących dane pole.

7.2.4.2. Właściwości klasy TField

Klasa `TField` wprowadza właściwości umożliwiające dostęp do danych pola. Umożliwiają one konwersję typu pola na tym określonej właściwości. Oto te właściwości: `AsBCD`, `AsBoolean`, `AsCurrency`, `AsDateTime`, `AsFloat`, `AsInteger`, `AsSQLTimeStamp`, `AsString`, `AsVariant`. Wszystkie one zostały już poprzednio opisane. Z tego też powodu powiemy tylko o właściwości `Value`, która umożliwia bezpośredni dostęp do danych pola.

Klasa `TField` udostępnia również właściwość `IsNull`, która testuje, czy pole jest puste (`IsNull == true`), czy też zawiera jakąś wartość (`IsNull == false`).

Każda z powyższych właściwości może zostać nadpisana (poprzez nadpisanie metod obsługujących tę właściwość) w klasach potomnych. Ma to na celu określanie specyficznych mechanizmów konwersji pomiędzy typem pola a typem odpowiedniej właściwości.

Pole tabeli może być fizycznym polem tabeli lub polem wirtualnym (obliczeniowym, lookupowym, agregowanym). Właściwością określającą typ pola jest `FieldKind`. Właściwość ta jest wypełniana automatycznie podczas tworzenia komponentu pola. Jest ona wartością wyliczeniową, mogącą przyjmować opcje przedstawione w tab. 7.7.

Tabela 7.7. Opis wartości wyliczeniowych

Wartość	*Opis*
fkData	Pole jest fizyczną kolumną w tabeli danej bazy danych
fkCalculated	Pole jest polem obliczanym (w zdarzeniu `OnCalcFields`)
fkLookup	Pole jest polem lookupowym (nieobsługiwane w przypadku zbiorów jednokierunkowych)
fkInternalCalc	Pole jest polem obliczanym, ale wartości są przechowywane w zbiorze danych
fkAggregate	Pole jest polem agregowanym

Pola obliczane przez serwery SQL posiadają właściwość `FieldKind` ustawioną na wartość `fkInternalCalc` zamiast `fkCalculated`.

Kolejną grupą są właściwości reprezentujące pole zbioru danych. Jedną z podstawowych właściwości jest `FieldName`. Wskazuje ona nazwę pola zbioru danych. Dla pól kalkulowanych czy pól odnośników właściwość ta oznacza pole wirtualne, które nie ma swojego odpowiednika w fizycznej tabeli. Natomiast dla pól danych nazwa musi być tożsama z nazwą pola w fizycznej tabeli lub z nazwą kolumny w zapytaniu SQL. Podczas tworzenia pól danych właściwość ta jest wypełniana automatycznie.

Właściwość DisplayLabel zawiera tekst wyświetlany w nagłówku danej kolumny w komponencie siatki DBGrid. Jest również źródłem właściwości Caption komponentu Label tworzonego przy wykorzystaniu mechanizmu *przeciągania pól*.

W rzeczywistości komponent DBGrid korzysta z właściwości DisplayName (a nie z właściwości DisplayLabel). Właściwość ta jest tylko do odczytu. Jeżeli właściwość DisplayLabel jest ustawiona, to DisplayName zwraca właśnie tę wartość. Jeżeli jest pusta, to DisplayName zwraca wartość właściwości FieldName. Analiza poniższego kodu źródłowego powinna wyjaśnić przedstawione działanie tychże właściwości (list. 7.7).

```
function TField.GetDisplayName: string;
begin
  if FDisplayLabel <> '' then
    Result := FDisplayLabel else
    Result := FFieldName;
end;
```

Listing 7.7

Właściwość DisplayText jest tylko do odczytu. Reprezentuje ona tekst wyświetlany w kontrolce danych, związany z wartością danego pola niebędącego w trybie edycji. Jeżeli pole obsługuje zdarzenie OnGetText, to właściwość DisplayText udostępnia wartość zwróconą przez parametr Text.

Wartość pola, która będzie wyświetlona w kontrolce danych w trybie edycji, dostępna jest dzięki właściwości Text. Właściwość ta udostępnia wartość w formacie charakterystycznym dla trybu edycji. Przykładowo dla pól typu walutowego Currency właściwość Text udostępnia wartość pola bez separatorów tysięcznych oraz bez symbolu walutowego.

Każde pole ma przypisany unikatowy indeks, który określa jego miejsce w zbiorze danych. Pozycja ta jest dostępna dzięki właściwości Index. Za jej pomocą możemy również modyfikować tę kolejność (właściwość ta jest indeksowana od 0). Kolejność ta oczywiście może być różna od fizycznej kolejności pól w tabeli. Fizyczny porządek pól w tabeli możemy odczytać dzięki właściwości FieldNo (indeksowanej od 1), która jest tyko do odczytu.

Na poziomie komponentu Field możemy określić widoczność pola w komponencie DBGrid. Jeżeli właściwość Visible przyjmie wartość true, to pole będzie widoczne w komponencie siatki, w przeciwnym przypadku pole zostanie pominięte.

Komponent Field posiada właściwość Required, która określa, czy dane pole jest polem wymaganym. Podczas tworzenia komponentu pola właściwość ta jest ustawiana na podstawie definicji pola w tabeli. Jeżeli właściwość Required przyjmuje wartość true, co jest zgodne z definicją pola w tabeli (posiada ono atrybut not null), to w przypadku zatwierdzenia pola z wartością NULL zostanie wygenerowany wyjątek klasy EDatabaseError. Jeżeli natomiast pole w tabeli może przyjmować wartość NULL, a nastąpi ustawienie właściwości Required na true, to komponent pola sam musi generować wyjątek w przypadku próby wstawienia wartości NULL. Wyjątek ten należy wyrzucić w zdarzeniu OnValidate.

Tabela 7.8 przedstawia w porządku alfabetycznym ważniejsze właściwości klasy Tfield.

Tabela 7.8. Ważniejsze właściwości klasy `Tfield`

Właściwość	*Opis*
`Alignment`	Wyrównuje tekst w obiekcie kontrolnym. (`aLeftJustify` – wyrównuje do lewej, `taCenter` wyrównuje względem środka, `taRightJustify` wyrównuje do prawej)
`ConstraintErrorMessage`	Określa tekst, który pojawi się, gdy użytkownik wprowadzi wartość naruszającą istniejące reguły integralności
`Currency`	Określa, kiedy wartości pola będą traktowane jako wartości monetarne (tylko pola numeryczne)
`CustomConstraint`	Umożliwia zdefiniowanie lokalnego ograniczenia nałożonego na wartości pola. Musi ono podlegać regułom składniowym języka SQL
`DisplayFormat`	Określa formatowanie, któremu podlegać będą wartości pola wyświetlane w kontrolkach danych
`DisplayLabel`	Określa nazwę kolumny związanej z danym polem, która użyta będzie m.in. w komponencie `DBGrid`
`DisplayWidth`	Określa liczbę znaków, jaka będzie wyświetlana w kontrolce danych
`EditFormat`	Określa formatowanie, któremu podlegać będą wartości pola wyświetlane w kontrolkach danych w trybie edycji
`EditMask`	Określa maskę wprowadzania danych
`FieldKind`	Określa typ pola
`FieldName`	Określa nazwę kolumny w tabeli, do której odnosi się komponent pola
`HasConstraints`	Określa, czy na pole nałożone są warunki integralności
`ImportedConstraint`	Określa warunki integralności zaimportowane ze słownika danych lub serwera SQL
`Index`	Określa indeks komponentu pola w zbiorze danych
`KeyFields`	Określa pole (lub pola), które musi pasować do pola w tabeli lookupowej
`LookupDataSet`	Określa tabelę, z której czerpane będą wartości pola lookupowego
`LookupKeyFields`	Określa pole (lub pola) lookupowe, które musi pasować do pola wskazanego przez właściwość `KeyFields`
`LookupResultField`	Określa pole, z którego czerpane będą dane stanowiące wartości danego pola
`MaxValue`	Określa maksymalną wartość (tylko pola numeryczne)
`MinValue`	Określa minimalną wartość (tylko pola numeryczne)
`Name`	Określa nazwę komponentu pola
`Origin`	Określa oryginalną nazwę kolumny w danej tabeli (dotyczy tylko pól komponentu `TQuery`)
`Precision`	Określa liczbę cyfr po przecinku
`ReadOnly`	Określa, czy dane pole jest tylko do odczytu (wartości tego pola nie będą mogły być modyfikowane)
`Size`	Określa maksymalną liczbę znaków, jaka może być wprowadzona do tego pola
`Transliterate`	Określa, czy dane transferowane pomiędzy zbiorem danych a tabelą bazy danych będą tłumaczone, uwzględniając ustawienia lokalne systemu operacyjnego
`Visible`	Określa, czy dane pole będzie wyświetlane w komponencie `DBGrid`

7.2.4.3. Tworzenie komponentów pól

Ze względu na sposób tworzenia komponentów potomnych względem klasy `TField` wyróżniamy dwa ich typy: dynamiczne i stałe (komponenty pól). Domyślnie komponenty pól tworzone są w sposób dynamiczny w momencie otwarcia zbioru danych na podstawie fizycznej struktury tabeli. Zbiór danych tworzy komponent pola (zdeterminowany przez typ kolumny) dla każdej kolumny tabeli. Komponenty te istnieją tak długo, jak długo otwarty jest zbiór danych.

Za każdym razem, gdy zbiór jest ponownie otwierany, komponenty te tworzone są na nowo, odzwierciedlając strukturę tabeli. Zatem gdy zostanie ona zmieniona, to po ponownym otwarciu zbioru zostanie ona uwzględniona. Wynika stąd, że taki sposób tworzenia komponentów sprawdza się w aplikacjach, w których w momencie ich tworzenia nie jest znana struktura tabeli bądź też będzie się ona zmieniać. Przykładowo aplikacja służąca do przeglądania różnych tabel za pomocą tego samego zbioru danych będzie wykorzystywać dynamiczny sposób tworzenia komponentów pól.

Komponenty tworzone dynamicznie nie pozwalają na zmianę ich kolejności, jak i na zablokowanie dostępu do niektórych pól. Nie można również tworzyć pól dodatkowych, takich jak pola obliczane czy pola lookupowe. Pola te nie są dostępne na etapie projektowania, a zatem każda zmiana właściwości wymaga odpowiedniej linii w kodzie programu.

Wszystkich tych ograniczeń możemy pozbyć się, wykorzystując stałe komponenty pól. Umożliwiają one zatem większą kontrolę nad polami danej tabeli, jednakże zmiany struktury tabeli nie będą odzwierciedlone w definicjach pól. Tak więc dokonanie niezbędnych zmian leży w gestii programisty. Jeżeli zmiany dotyczą typu pola, to należy usunąć i ponownie utworzyć komponent pola. W innych przypadkach zmian dokonujemy w *Object Inspectorze*.

Użycie stałych komponentów pól pozwala:

- modyfikować ich właściwości zarówno w *Object Inspectorze*, jak i w kodzie aplikacji;
- tworzyć pola lookupowe, pola kalkulowane oraz pola zagregowane;
- kontrolować poprawność wprowadzanych danych;
- usunąć komponenty z listy stałych komponentów pól (usunięte pola będą niedostępne z poziomu aplikacji).

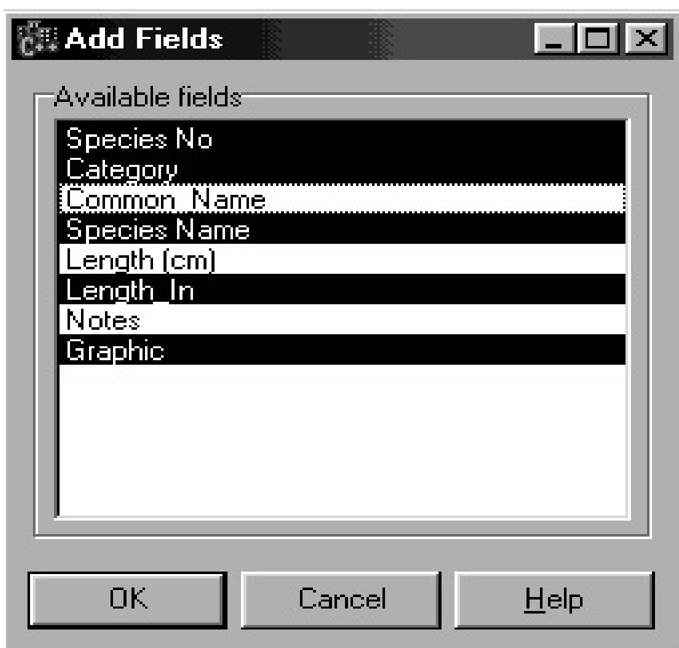
Stałe komponenty pól tworzymy, wykorzystując edytor pól – *Fields Editor*. Tak utworzone komponenty przechowywane są w module aplikacji (w tym, w którym znajduje się komponent klasy `TDataSet`, w kontekście którego utworzyliśmy stałe komponenty pól). Stałe komponenty pól posiadają wszystkie cechy „tradycyjnych” komponentów (wybieranych z palety komponentów). Oznacza to, że ich właściwości modyfikujemy, wykorzystując *Object Inspector*. Możliwe jest również zdefiniowanie zdarzeń dla poszczególnych komponentów pól.

Wszystkie komponenty pól danego zbioru danych są jednocześnie albo stałymi, albo dynamicznymi komponentami pól. Jeżeli utworzymy stałe komponenty pól, po czym będziemy chcieli wrócić do ich dynamicznych odpowiedników, to należy usunąć z edytora pól wszystkie jego pozycje (odpowiadające stałym komponentom pól).

Tworzenie stałych komponentów pól przedstawimy w kilku punktach:

- umieszczamy komponent klasy `TDataSet` na formularzu lub w module danych (*Data Module*);

- łączymy komponent klasy `TDataSet` z odpowiednim zbiorem danych. Przykładowo: komponent `Table` łączymy z tabelą za pomocą właściwości `TableName`, komponent `Query` wymaga podania odpowiedniego zapytania we właściwości SQL, natomiast komponent `StoredProc` posiada właściwość `StoredProcName`, która określa procedurę zapamiętaną;
- uruchamiamy edytor pól (*Fields Editor*) za pomocą podwójnego kliknięcia lewym przyciskiem myszki na komponent klasy `TDataSet` lub wybierając z jego menu kontekstowego pozycję *Fields Editor…*. Okno *Fields Editor* zawiera pasek tytułowy, przyciski nawigacyjne oraz `ListBox`. Pasek tytułowy wyświetla nazwę modułu danych lub formatki zawierającej zbiór danych oraz nazwę samego zbioru danych (nazwy te połączone są za pomocą strzałki (->)). Poniżej znajdują się przyciski nawigacyjne, która służą do poruszania się po zbiorze danych. Jeżeli zbiór jest zamknięty lub pusty, to przyciski te są nieaktywne; `ListBox` natomiast wyświetla stałe komponenty pól. Przy pierwszym uruchomieniu edytora pól `ListBox` jest pusty. Odpowiada to sytuacji, że wszystkie komponenty pól są tworzone dynamicznie;
- za pomocą menu podręcznego dodajemy wybrane pola, korzystając z pozycji *Add Fields…* (rys. 7.9). Po wybraniu interesujących nas pól z okna *Add Fields* klikamy w przycisk *OK*. Komponenty pól usuwamy za pomocą podręcznego menu lub za pomocą klawisza *Delete*;



Rys. 7.9. Przykładowy wygląd okna *Add Fields*

- zaznaczając teraz odpowiedni komponent pola (lub kilka komponentów pól), możemy modyfikować jego właściwości, a także przypisywać mu zdarzenia.

Podczas otwarcia zbioru danych dla każdego komponentu pola, który nie jest polem kalkulowanym, dokonywane jest sprawdzenie, czy istnieje kolumna w danym zbiorze danych odpowiadająca temu polu (oznacza to, że nazwa pola – `FieldName` jest tożsama z nazwą danej kolumny zbioru danych). Jeżeli dla któregokolwiek z pól taka kolumna nie istnieje, to generowany jest wyjątek, a zbiór danych nie zostanie otwarty.

Kolejność pól w *Fields Editor* możemy modyfikować, przeciągając odpowiednie komponenty pól na nowe pozycje. Kolejność ta będzie widoczna w komponencie `DBGrid`,

jak również we właściwości `Fields` klasy `TFields`. Kolejność pól możemy również zmieniać za pomocą właściwości `Index`.

Oprócz pól posiadających swoje fizyczne odpowiedniki w zbiorze danych możliwe jest również tworzenie dodatkowych komponentów pól (nieistniejących fizycznie w zbiorze danych). Te nowe komponenty pól umożliwiają tylko wyświetlanie danych, nie dają natomiast możliwości ich edytowania. Aby utworzyć nowy komponent pola, należy z menu podręcznego *Fields Editor* wybrać pozycję *New Field* (rys. 7.10). Dzięki temu ukaże się nam okno *New Field*, zawierające trzy grupy opcji: *Field properties*, *Field type* oraz *Lookup definition*.

Rys. 7.10. Ogólny wygląd okna *New Field*

Pierwsza grupa *Field properties* zawiera cztery komponenty edycyjne. Pierwszy z nich, *Name*, określa nazwę pola, która odpowiada właściwości `FieldName`. Nazwa ta jest używana do utworzenia nazwy komponentu pola – właściwość `Name` – i jest wyświetlana jako *konkatenacja* nazwy zbioru danych i nazwy pola w okienku edycyjnym *Component*.

Komponent `ComboBox` *Type* pozwala na wybór typu danych nowo tworzonego pola. Ostatni komponent należący do tej grupy – *Size* określa maksymalną liczbę znaków, jaka będzie wyświetlana oraz jaka będzie mogła być wprowadzona do komponentu bazującego na typie łańcuchowym lub bajtowym. Oznacza to, że podany rozmiar będzie mieć wpływ tylko dla komponentów następujących klas: `TStringField`, `TBytesField`, `TVarBytesField`.

Druga grupa opcji *Field type* określa typ nowo tworzonego komponentu pola. Domyślnym typem jest *Data*. Druga opcja umożliwia utworzenie pola obliczanego, trzecia zaś pola lookupowego. Ostatni wybór uaktywnia trzecią grupę opcji *Lookup definition*, odnoszącą się tylko do pól lookupowych.

Pola danych (*Data*) tworzymy w celu zastąpienia standardowych komponentów pól tworzonych automatycznie za pomocą edytora pól. Jednym z ważniejszych powodów tworzenia takich pól jest chęć zmiany klasy danego komponentu pola, która jest dobierana automatycznie i nie może być zmieniona. Załóżmy, że chcemy zmienić typ pola

`TSmallIntField` na typ `TIntegerField`. Musimy zatem usunąć komponent pola, a następnie utworzyć nowy (w oknie *New Field*) z podanym typem `TIntegerField`.

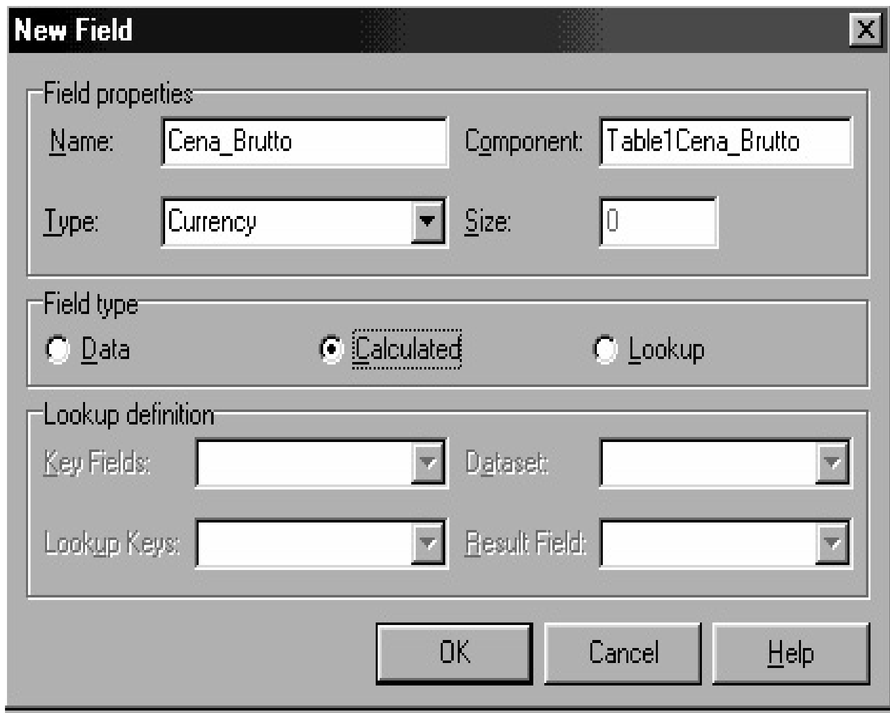
Pola obliczane (*Calculated*) są polami wirtualnymi, tzn. nieistniejącymi fizycznie w zbiorze danych. Wartość tych pól obliczana jest w procedurze obsługującej zdarzenie `OnCalcFields` i jest dostępna tylko do odczytu. Procedura tworzenia tego typu pól jest analogiczna do procedury tworzenia pól danych (*Data*). Różnica polega na wybraniu z grupy *Field type* opcji *Calculated* (zamiast *Data*).

Aby pole mogło udostępniać obliczoną wartość, musimy zaimplementować procedurę obsługi zdarzenia `OnCalcFields`. Jeżeli tego nie uczynimy, to pole zawsze będzie zwracało wartość NULL. Procedurę zdarzeniową definiujemy w kontekście komponentu `DataSet`, do którego należy utworzone uprzednio pole obliczane. W ciele procedury umieszczamy kod, którego zadaniem jest obliczenie odpowiedniej wartości.

Przykład 7.22

Prześledźmy następujący przykład.

Załóżmy, że mamy tabelę *Towary*, z polami *Id_Towaru*, *Cena_Netto* oraz *VAT*. W takiej tabeli potrzebujemy oczywiście jeszcze ceny brutto. Umieszczenie takiej kolumny w definicji tabeli byłoby złamaniem trzeciej postaci normalnej. Problem ten możemy rozwiązać dzięki polu obliczanemu *Cena_Brutto*. Pole to będzie polem wirtualnym, udostępniającym wartość pola *Cena_Brutto* na podstawie pól *Cena_Netto* oraz *VAT*. Jednakże wartość tego pola nie będzie fizycznie przechowywana w tabeli, a jedynie udostępniana jako wartość tylko do odczytu. Dzięki temu nasza tabela dalej spełnia kryteria trzeciej postaci normalnej.



Rys. 7.11. Utworzenie pola wyliczeniowego *Cena_Brutto*

Po utworzeniu pola obliczanego *Cena_Brutto* (rys. 7.11) pozostaje nam jeszcze zdefiniowanie procedury obsługującej zdarzenie `OnCalcFields`:

```
void __fastcall TForm1::TowaryCalcFields(TDataSet *DataSet)
{
```

```
  TowaryCena_Brutto->Value =
    TowaryCena_Netto->Value * (1 + TowaryVAT->Value / 100.0);
}
```

Przykład 7.23

Pola obliczane nie ograniczają się bynajmniej tylko do pól liczbowych. Drugi przykład pokazuje, jak utworzyć pole obliczane, którego wartość jest *konkatenacją* dwóch łańcuchów:

```
void __fastcall TForm1::OsobyCalcFields(TDataSet *DataSet)
{
  DataSet->FieldByName("Osoba")->AsString =
    DataSet->FieldByName("Imię")->AsString + " " +
      DataSet->FieldByName("Nazwisko")->AsString;
}
```

Analizując powyższe przykłady, widzimy, że do wartości pól możemy odwołać się albo poprzez nazwę komponentu pola (pierwszy przykład), albo poprzez metody lub właściwości klasy `TDataSet` (drugi przykład). W drugim przykładzie dostęp do poszczególnych pól uzyskujemy dzięki wskaźnikowi `DataSet` przekazanemu do metody obsługującej zdarzenie `OnCalcFields`.

Drugim rodzajem pól wirtualnych są pola odnośników (*lookup fields*), zwane *polami lookupowymi*. Służą one do wyświetlania wartości pola z innego zbioru danych. Tak jak i pola obliczane są one również polami tylko do odczytu.

Pola lookupowe oprócz właściwości `FieldName` reprezentującej wartość pola posiadają jeszcze właściwość `LookupResultField`, która jest nazwą pola drugiej tabeli, z którego będą czerpane wartości, oraz właściwość `LookupDataSet`, która jest nazwą tejże tabeli. Pola lookupowe posiadają jeszcze dwie ważne właściwości. Pierwsza z nich `KeyFields` reprezentuje pole (lub pola) pierwszej tabeli, podczas gdy `LookupKeyFields` pole (lub pola) drugiej tabeli. Wartość pola `FieldName` jest równa wartości pobranej z pola wymienionego we właściwości `LookupResultField` z rekordu drugiej tabeli, dla którego wartość pola `LookupKeyFields` jest równa wartości pola `KeyFields` z pierwszej tabeli. Tak więc związek pomiędzy tymi dwiema tabeli ustanowiony zostaje dzięki polom: `KeyFields` z pierwszej tabeli oraz `LookupKeyFields` z drugiej tabeli. Powiązanie tych właściwości przypomina działanie metody `Lookup()` klasy `TDataSet`. Skojarzenie to potwierdza analiza poniższego kodu źródłowego:

```
procedure TField.CalcLookupValue;
begin
  if FLookupCache then
    Value := LookupList.ValueOfKey(FDataSet.FieldValues[FKeyFields])
  else if (FLookupDataSet <> nil) and FLookupDataSet.Active then
    Value := FLookupDataSet.Lookup(FLookupKeyFields,
      FDataSet.FieldValues[FKeyFields], FLookupResultField);
end;
```

Listing 7.8

Dodajmy tylko, że pole `FLookupCache` określa, czy dane mają być przechowywane w lokalnej pamięci (`LookupList`), czy pobierane za każdym razem za pomocą metody `Lookup()` z tabeli lookupowej. Do pola `FLookupCache` odwołujemy się, wykorzystując właściwość `LookupCache`.

Rozważmy następujący przykład.

Przykład 7.24

Załóżmy, że mamy tabelę *Odbiorcy* posiadającą dwie kolumny: *Id_Odbiorcy* oraz *Nazwa*. Mamy też drugą tabelę *Zamówienia* o trzech kolumnach: *Id_Zamówienia*, *Id_Odbiorcy*, *Data*. Obie tabele pozostają ze sobą w relacji za pomocą kolumn *Id_Odbiorcy*. Użycie w charakterze kluczy pól typu całkowitoliczbowego jest dobrym rozwiązaniem, gdyż zwiększa efektywność złączeń. Z drugiej jednak strony pole *Id_Odbiorcy* niewiele mówi użytkownikowi aplikacji, który odbiorców identyfikuje za pomocą nazw. Zatem dodanie do tabeli *Zamówienia pola lookupowego* ułatwi użytkownikowi korzystanie z tej tabeli, gdyż odbiorcy będą reprezentowani za pomocą ich nazw (zaczerpniętych z tabeli *Odbiorcy*). Natomiast złączenie obu tabel nadal będzie funkcjonowało na podstawie kolumny *Id_Odbiorcy*.

Pola lookupowe tworzymy analogicznie do pozostałych rodzajów pól. Tak więc po uruchomieniu okna *New Field* wypełniamy pierwszą grupę opcji oraz w drugiej wybieramy opcję *Lookup*. Po dokonaniu ostatniego wyboru uaktywni się trzecia grupa opcji *Lookup definiton*. Zawiera ona cztery komponenty `ComboBox`. W pierwszym z nich, *Key Fields*, wybieramy nazwę pola pierwszej tabeli (najczęściej jest to *klucz obcy*), na podstawie wartości którego będą wybrane odpowiednie wartości z drugiej tabeli. W drugim o nazwie *DataSet* wybieramy zbiór danych utożsamiany z drugą tabelą (tabelą lookupową). Po tym wyborze zostały uaktywnione pozostałe komponenty, które dotyczą ostatnio wybranego zbioru danych. W trzecim `ComboBoxie` *Lookup Keys* wybieramy nazwę pola drugiej tabeli (najczęściej *klucz główny*), która będzie odpowiadała polu z pierwszej tabeli wybranym w `ComboBoxie` *Key Fields*. Ostatni `ComboBox` o nazwie *Result Field* określa nazwę pola, z którego będą czerpane wartości z tabeli lookupowej (wyświetlane użytkownikowi). Jeżeli zamiast pojedynczych pól będziemy chcieli użyć kilku pól, to ich nazwy musimy rozdzielić średnikami.

Po utworzeniu *pola lookupowego* możemy dokonywać zmian w jego definicji, modyfikując opisane wyżej właściwości. Musimy pamiętać, ażeby w roli lookupowego zbioru danych nie wybrać komponentu, którego działanie jest zdeterminowane przez tabelę główną, gdyż nastąpi odwołanie cykliczne, co może skutkować trudnymi do wykrycia błędami. W szczególności nie możemy również wybrać jako lookupowy zbiór danych komponent klasy `TDataSet`, reprezentujący tabelę, w kontekście której tworzymy pole.

Na koniec dodajmy jeszcze, że edytor pól wykorzystujemy również do usuwania komponentów pól. W tym celu wykorzystujemy jego menu podręczne lub klawisz *Delete*.

7.2.4.4. Zdarzenia klasy TField

Klasa `TField` udostępnia również zdarzenia, które można wykorzystać do kontrolowania wartości zapisywanych i odczytywanych z komponentu pola. Zdarzenia te definiujemy, wykorzystując *Object Inspector*. Klasy potomne nie dodają dodatkowych zdarzeń, a zatem ich liczba jest stała dla wszystkich komponentów pól i jest równa 4. Tabela tab.7.9 prezentuje zdarzenia klasy `Tfield`.

Tabela 7.9. Zdarzenia klasy `Tfield`

Zdarzenie	*Opis*
`OnChange`	Wywoływane, gdy wartość pola ulega zmianie
`OnGetText`	Wywoływane, gdy odczytywana jest wartość pola
`OnSetText`	Wywoływane, gdy do pola zapisywana jest wartość
`OnValidate`	Wywoływane w celu sprawdzenia poprawności wprowadzanych danych

Zdarzenia `OnGetText` oraz `OnSetText` wykorzystywane są do formatowania danych w sposób zdefiniowany przez programistę.

Zdarzenie `OnGetText` używane jest w celu przygotowania wartości dla właściwości `DisplayText` lub `Text`. Oto jego deklaracja:

```
typedef void __fastcall (__closure *TFieldGetTextEvent)(TField*
Sender, AnsiString &Text, bool DisplayText);


__property TFieldGetTextEvent OnGetText = {read=FOnGetText,
write=FOnGetText};
```

Listing 7.9

Widzimy zatem, że mamy do dyspozycji trzy parametry. Pierwszy z nich, `Sender`, wskazuje na komponent pola, w kontekście którego zaistniało to zdarzenie. Parametr `Text` jest parametrem przekazywanym przez referencję. Wartość tego parametru będzie tekstem zwróconym przez właściwość `Text` lub `DisplayText`, a zatem będzie to wartość wyświetlana w komponentach edycyjnych. Ostatni parametr `DisplayText` określa, czy wartość pola jest tylko wyświetlana (tzn. nie jest edytowana), czy podlega aktualnie edycji.

Jeżeli nie utworzymy procedury obsługi tego zdarzenia, to właściwości `Text` oraz `DisplayText` będą zwracały wartość właściwości `AsString`. Poniższy przykład pokazuje, w jaki sposób możemy, wykorzystując zdarzenie `OnGetText`, spolonizować wartości logiczne pola *WartośćLogiczna*:

```
void __fastcall TForm1::Table1WartoscLogicznaGetText(TField *Sender,

      AnsiString &Text, bool DisplayText)

{

  if (Sender->AsInteger)

    Text = "Tak";

  else

    Text = "Nie";

}
```

Listing 7.10

Analogicznym zdarzeniem do `OnGetText` jest zdarzenie `OnSetText`, działające jednak „w kierunku przeciwnym". Mianowicie, wywoływane jest ono wtedy, kiedy przypisywana jest wartość właściwości `Text`. A zatem obsługując to zdarzenie, możemy zmienić wartość (wprowadzoną przez użytkownika w kontrolce danych), która zostanie zapisana w komponencie pola. Oto deklaracja zdarzenia `OnSetText`:

```
typedef void __fastcall (__closure *TFieldSetTextEvent)(TField*
Sender, const AnsiString Text);

__property TFieldSetTextEvent OnSetText = {read=FOnSetText,
write=FOnSetText};
```

Listing 7.11

Zdarzenie to posiada dwa parametry. Pierwszy, `Sender` (tak jak i w zdarzeniu `OnGetText`), wskazuje na komponent pola, w kontekście którego zaistniało to zdarzenie. Drugi parametr `Text` zwraca wartość właściwości `Text`. Na tej podstawie możemy bezpośrednio ustawić wartość pola, wykorzystując właściwość `AsString`.

Jeżeli nie utworzymy procedury obsługującej zdarzenie, to właściwości `AsString` zostanie przypisana zawartość właściwości `Text`.

Przykład 7.25

Ten przykład dodaje do tekstu wprowadzonego do pola *Uwagi* bieżącą datę:

```
void __fastcall TForm1::Table1UwagiSetText(TField *Sender,
      const AnsiString Text)
{
if (!Text.IsEmpty())
  Sender->AsString = Text + " - Uwagi sporządzono " +
DateToStr(Date());
else
  Sender->AsString = Text;
}
```

Zdarzenie `OnChange` wywoływane jest, gdy dana wartość zapisywana jest do bufora rekordu. Operacje tę możemy przedstawić w kilku punktach:

- wywoływane jest zdarzenie `OnValidate` w celu skontrolowania poprawności wprowadzanych danych;
- jeżeli zdarzenie `OnValidate` zaakceptuje wprowadzone wartości, to zostaną one zapisane do bufora rekordu;
- jeżeli zapis danych nie wywoła wyjątku, to zostanie wywołane zdarzenie `OnChange`.

Dzięki zdarzeniu `OnValidate` możemy kontrolować poprawność wprowadzanych danych. Jeżeli dane nie są poprawne, to należy wygenerować wyjątek. Poniższy przykład (7.26) pokazuje wykorzystanie tego zdarzenia.

Przykład 7.26

```
void __fastcall TForm1::Table1WiekValidate(TField *Sender)
{
  if (Sender->AsInteger < 0)
    throw Exception("Wartość tego pola nie może być ujemna");
}
```

7.2.5. Komponent Query

Komponent `Query` reprezentuje zbiór danych z rezultatem powstałym na bazie zapytania SQL. Umożliwia on dostęp do jednej lub kilku tabel serwerów baz danych SQL (*Sybase*, SQL *Server*, *Oracle*, *Informix*, DB2, *InterBase*), lokalnych baz danych (*Paradox*, *InterBase*, *dBASE*, *Access*, *FoxPro*) oraz baz danych korzystających z mechanizmu ODBC.

Komponent `Query` używamy ze względu na dwie ważne własności:

- umożliwia on dostęp do danych będących złączeniem kilku tabel;
- umożliwia ograniczenie zbioru wierszy i kolumn danej tabeli (lub kilku tabel).

7.2.5.1. Właściwość SQL

Zapytanie wykorzystywane przez komponent `Query` reprezentuje pojedyncze wyrażenie języka DDL (*Data Definition Language*) lub języka DML (*Data Manipulation Language*). Język używany w treści zapytań jest zależny od języka serwera SQL, pod adresem którego kierowane są zapytania. Jednakże najczęściej jest to język SQL zgodny ze standardem SQL-92. Komponent `Query` implementuje całą podstawową funkcjonalność wprowadzoną przez `DataSet`, jak również wszystkie specyficzne mechanizmy charakterystyczne dla komponentów korzystających z zapytań SQL.

Właściwością przechowującą treść zapytania SQL jest właściwość o tej samej nazwie. Oto jej deklaracja:

```
__property Classes::TStrings* SQL = {read=FSQL, write=SetQuery};
```

Typ właściwości `TStrings` pozwala na umieszczanie treści zapytań w kilku liniach, co ma na celu zwiększenie czytelności zapytania. Przypisanie właściwości SQL odpowiedniego zapytania nie powoduje jeszcze wykonania jego treści. Aby tego dokonać, należy wywołać metodę `Open()` lub `ExecSQL()`.

Na etapie projektowania do umieszczenia zapytania we właściwości SQL wykorzystujemy edytor linii *String List Editor*, który jest dostępny z poziomu *Object Inspectora*. Wykorzystanie właściwości `SQL` w kodzie aplikacji wiąże się z użyciem metod i właściwości klasy `TStrings`.

Przykład 7.27

Oto kilka przykładów:

wyszukujemy studentów 3. roku:

```
Query1->SQL->Clear();
  Query1->SQL->Add("select * from Studenci");
  Query1->SQL->Add("where Rok_Studiów = 3");
Query1->Open();
```

usuwamy studentów 3. roku:

```
Query1->SQL->Clear();
  Query1->SQL->Add("Delete from Studenci");
  Query1->SQL->Add("where Rok_Studiów = 3");
Query1->ExecSQL();
```

Możemy zauważyć, że powyższe przykłady wykorzystują do wykonania zapytań SQL dwie różne metody. Pierwszą z nich – `Open()` wykorzystujemy do wykonywania

zapytań SQL zawierających frazę `SELECT`, a tym samym zwracających w wyniku zbiór danych. Drugą metodę – `ExecSQL()` stosujemy do takich fraz, jak: `INSERT, UPDATE, DELETE, CREATE TABLE` itp., czyli niezwracających zbioru danych, a jedynie wykonujących pewne operacje w kontekście danej tabeli.

7.2.5.2. Zapytania parametryczne

Zapytania w obu Prz. 7.27 należą do grupy *zapytań statycznych*, tzn. ich treść jest całkowicie zdeterminowana już na etapie projektowania aplikacji. Załóżmy jednak, że chcemy wyszukać oprócz studentów 3. roku także tych z 5. roku. Istnieją dwa rozwiązania tego problemu. Pierwsze z nich polega na zmianie treści samego zapytania SQL. Drugie rozwiązanie wykorzystuje mechanizm *zapytań dynamicznych* lub inaczej *parametrycznych*.

Działanie to polega na zdefiniowaniu w treści zapytania parametru, którego wartość będzie mogła ulegać zmianie. Przy czym struktura zapytania SQL pozostaje niezmieniona. Nazwa parametru w zapytaniu SQL musi być poprzedzona znakiem „:".

Przykład 7.28
Przykład pokazuje sposób określenia parametru w treści zapytania SQL:

```
select * from Studenci where Rok_Szkolny = :P_Rok_Szkolny
```

Parametry danego zapytania reprezentowane są poprzez właściwość `Params` zadeklarowaną w następujący sposób:

```
__property TParams* Params = {read=FParams, write=SetParamsList};
```

Dostęp do parametrów, podobnie jak i do pól, może odbywać się na trzy sposoby:

- poprzez właściwość tablicową – `System::Variant ParamValues [AnsiString ParamName];`
- poprzez właściwość `TParam* Items[Word Index];`
- poprzez funkcję `TParam* __fastcall ParamByName(const AnsiString Value).`

Pierwszy sposób umożliwia dostęp tylko do wartości danego parametru. `ParamName` jest łańcuchem znaków będących nazwą parametru. Jeżeli chcemy uzyskać dostęp do kilku parametrów, to ich nazwy oddzielamy średnikami. Oczywiście `ParamName` musi reprezentować poprawne nazwy parametrów, w przeciwnym razie zostanie zgłoszony wyjątek.

Właściwość `Items` jest listą parametrów, do których odwołujemy się poprzez atrybut `Index`, który może przyjmować wartości od 0 do `Count` – 1. Właściwość `Items` zwraca obiekt klasy `TParam`, dzięki czemu możemy, oprócz wartości, uzyskać dostęp także do innych właściwości parametru (który reprezentowany jest przez klasę `TParam`). Odwołanie się do wartości może nastąpić poprzez właściwość `Value`, zwracająca wartość typu `Variant`, lub poprzez szereg właściwości typu `AsXxx` , automatycznie dokonujących konwersji na typ `Xxx`.

Funkcja `ParamByName()` umożliwia dostęp do parametru poprzez jego nazwę. Podobnie jak właściwość `Items` udostępnia ona obiekt klasy `TParam`, ale zwalnia programistę z konieczności pamiętania kolejności parametrów.

Przykład 7.29
W rzeczywistości teraz przeanalizujemy trzy przykłady prezentujące wykorzystanie parametrów w zapytaniach języka SQL:

- wstawianie wierszy do tabeli *Studenci* (*Nazwisko*, *Rok_Studiow*):

```
Query1->SQL->Clear();
  Query1->SQL->Add("insert into Studenci (Nazwisko, Rok_Studiów)");
  Query1->SQL->Add("values (:Nazwisko, :Rok_Studiów)");
  Query1->Params->Items[0]->AsString = "Kowalski";
  Query1->Params->Items[1]->AsInteger = 4;
Query1->ExecSQL();
```

- pobieranie danych o studencie Kowalski z tabeli *Studenci*:

```
Query1->SQL->Clear();
  Query1->SQL->Add("select * from studenci");
  Query1->SQL->Add("where Nazwisko = :P_Nazwisko");
  Query1->ParamByName("P_Nazwisko")->AsString = "Kowalski";
Query1->Open();
```

- pobieranie danych o studentach 2. roku z tabeli *Studenci*:

```
Query1->SQL->Clear();
  Query1->SQL->Add("select * from studenci");
  Query1->SQL->Add("where Rok_Studiów = :Rok_Studiów");
  Query1->Params->ParamValues["Rok_Studiów"] = 2;
Query1->Open();
```

Analizując powyższe przykłady, możemy zauważyć, że nazwa parametru może być tożsama z nazwą pola. Nie będzie tu jednak żadnej niejednoznaczności, gdyż nazwy parametrów są poprzedzane znakiem dwukropka (`:`). Druga uwaga dotyczy metody `ParamByName()`, która jest wywoływana na rzecz obiektu klasy `TQuery`, a nie obiektu klasy `TParam`. Oczywiście, możemy odwoływać się do niej poprzez właściwość `Params`, gdyż metoda `ParamByName()` jest również zdefiniowana w klasie `TParam`. Nie są to jednak dwie różne metody. Po prostu metoda klasy `TQuery` korzysta z metody klasy `TParam`.

Przykład pochodzący z kodu źródłowego powinien rozwiać wszelkie wątpliwości:

```
function TQuery.ParamByName(const Value: string): TParam;
begin
  Result := FParams.ParamByName(Value);
end;
```

Przykład 7.30

Prześledźmy jeszcze dwa przykłady:

```
select * from Studenci order by :kolumna
select count(*) from :tabela
```

Oba te przykłady, choć składniowo poprawne, nie mogą być jednak użyte w praktyce, gdyż nazwy kolumn i nazwy tabel nie mogą być parametryzowane.

Rozwiązaniem tego problemu może być zmiana właściwości SQL. Ponieważ jest ona klasy `Tstrings`, możemy do jej budowy wykorzystać standardowe operacje łańcuchowe. Jednakże wygodniejsze jest użycie funkcji `Format()`. Oto jej deklaracja:

```
extern PACKAGE AnsiString __fastcall Format(const AnsiString Format,
const System::TVarRec* Args, const int Args_Size);
```

Listing 7.12

Funkcja ta zastępuje symbole specjalne, występujące w łańcuchu znaków przekazanym jako pierwszy parametr, kolejnymi elementami tablicy przekazanej poprzez parametr `Args` o rozmiarze `Args_Size`. Wszystkie symbole specjalne rozpoczynają się znakiem procenta (`%`), po którym następują znaki charakteryzujące dany parametr. Oto kilka przykładów:

`%c` – pojedynczy znak;

`%d` – liczba całkowita;

`%f` – liczba rzeczywista;

`%p` – wskaźnik;

`%s` – łańcuch znaków.

Rozważmy przykład, który pokazuje sposób użycia funkcji `Format`.

Przykład 7.31

```
const AnsiString Szablon = "Student %s %s jest na %d roku";
AnsiString s;
s = Format(Szablon, ARRAYOFCONST(("Jan", "Kowalski", 3)));
```

Powyższy kod spowoduje, że łańcuch s zawierać będzie następujący napis:

Student Jan Kowalski jest na 3. roku.

Pokażemy teraz, jak wykorzystać funkcję `Format()` do dynamicznego tworzenia zapytań języka SQL:

```
const AnsiString Zapytanie = "select count(*) from %s";
  Query1->Close();
  Query1->SQL->Clear();
  Query1->SQL->Add(Format(Zapytanie, ARRAYOFCONST(("student"))));
  Query1->Open();
```

7.2.5.3. Przygotowywanie zapytań

Instrukcje SQL przekazywane są do serwera w postaci źródłowej, co oznacza, że każda taka instrukcja przed wykonaniem musi zostać przeanalizowana, zweryfikowana oraz skompilowana. Jeżeli wykonujemy to samo zapytanie wielokrotnie, to wszystkie powyższe czynności wstępne będą wykonywane za każdym razem, mimo że są one identyczne. Rozwiązaniem tego problemu jest użycie metody `Prepare()`. Powoduje ona, że BDE oraz serwer bazy danych alokuje pewną część zasobów potrzebną do wstępnego przygotowania zapytania. Dzięki temu każde następne jego wykonanie realizowane jest z pominięciem opisanych powyżej czynności przygotowawczych. Oczywiste jest, że taki sposób postępowania zwiększa efektywność przetwarzania zapytań języka SQL.

Komponent `Query` automatycznie przygotowuje zapytanie, wywołując metodę `Prepare()` (jeżeli nie było ono wcześniej przygotowane), oraz po jego zakończeniu wywołuje metodę `UnPrepare()` (ma ona na celu zwolnienie zasobów zaalokowanych poprzez metodę `Prepare()`). Każde wywołanie metody `Prepare()` powinno być zbilansowane odpowiednim wywołaniem metody `UnPrepare()`. Jest to podyktowane faktem, że przygotowanie zapytania pochłania pewną część zasobów, dlatego też musi być ona w odpowiednim miejscu zwolniona.

Podsumowując powyższe rozważania, przytoczmy przykład.

Przykład 7.32

```
Query1->Prepare();
try
{
  for (int i = 1; i <=5 ; i++)
  {
    Query1->ParamByName("Rok_Studiow")->AsInteger = i;
    Query1->Open();
    try
    {
      //Tutaj wykorzystujemy wynik zapytania
    }
    __finally
    {
      Query1->Close();
    }
  }
}
__finally
{
  Query1->UnPrepare();
}
```

7.2.5.4. Relacje typu Master/Detail

Komponent `Query`, podobnie jak `Table` umożliwia tworzenie związków pomiędzy tabelami, jednakże dzięki elastyczności języka SQL mogą być one o wiele bardziej skomplikowane. Relacje określamy, wykorzystując zapytanie parametryczne oraz właściwość `DataSource`. Właściwość ta wskazuje na zbiór danych (`DataSet`), z którego będą czerpane wartości parametrów zdefiniowanych w treści zapytania SQL.

Wypełniane są tylko te parametry, których nazwy odpowiadają nazwom pól w zbiorze nadrzędnym. Pozostałe zaś, jeżeli istnieją, muszą być ustawione w kodzie aplikacji. Zapytanie parametryczne zdefiniowane w zbiorze reprezentującym *relację podrzędną* wykonywane jest (z odpowiednio ustawionymi wartościami parametrów) za każdym razem, gdy zmieni się pozycja kursora w zbiorze reprezentującym *relację nadrzędną*.

Dzięki odpowiedniemu wykorzystaniu właściwości `DataSource` możemy tworzyć związki *Master/Detail* o wiele bardziej skomplikowane niż ich tradycyjne odpowiedniki (oparte na komponentach `Table`). Klasycznym przykładem jest tutaj relacja wielowartościowa pomiędzy dwiema tabelami.

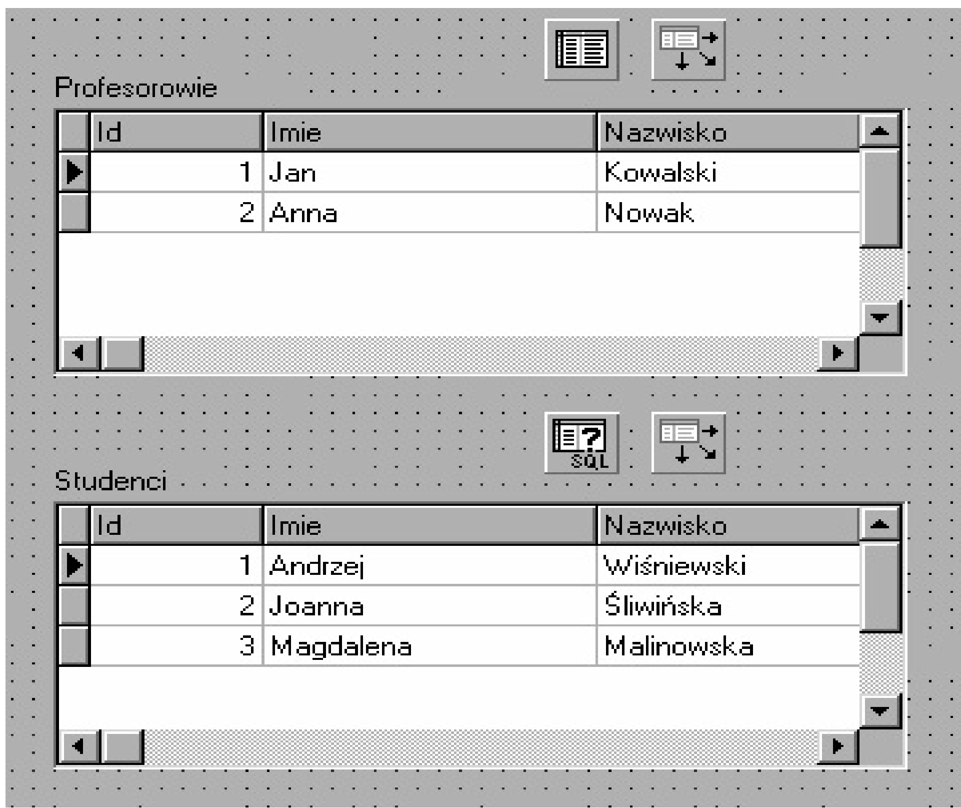
Przykład 7.33

Rozważmy dwie tabele: *Studenci*, *Profesorowie*, będące ze sobą w relacji *wiele-do-wielu*. Umieśćmy na formatce komponent `Table`, reprezentujący tabelę *Profesorowie*, oraz komponent `Query` z właściwością SQL, będącą złączeniem tabel *Stud_Prof* (tabela mieszająca) oraz *Studenci*. Zadanie to realizuje następujące polecenie języka SQL:

```
select s.* from Stud_Prof sp, Studenci s
where s.Id = sp.Id_Stud and sp.Id_Prof = :Id;
```

Oprócz tego umieszczamy na formatce dwa komponenty `DataSource` oraz dwa komponenty `DBGrid` odpowiednio ze sobą połączone (*DBGrid1 -> DataSource1 -> Table1* oraz (*DBGrid2 -> DataSource2 -> Query1*).

Ostatnim elementem będzie ustawienie właściwości `DataSource` komponentu *Query1* na *DataSource1*. Sprawi to, że przy zmianie pozycji bieżącego rekordu komponentu *Table1* zostanie ustawiony parametr `:Id` (zgodnie z wartością pola *Id* tabeli *Profesorowie*), a następnie zostanie wykonane zapytanie (przechowywane we właściwości SQL komponentu *Query1*). Dzięki temu komponent *Query1* będzie zawierał tych studentów, z którymi ma zajęcia profesor o przekazanym w parametrze numerze *Id*. Formatka z tabelami *Profesorowie* i *Studenci* może mieć wygląd pokazany na rys. 7.12.



Rys. 7.12. Wygląd formatki z tabelami

7.2.5.5. Edytowalność wyników zapytań SQL

Domyślnie wyniki zapytań SQL są zbiorami tylko do odczytu. Działanie takie jest podyktowane faktem, iż zapytanie SQL może być złączeniem wielu tabel (zawierającym nierzadko podzapytania, konstrukcje grupujące oraz funkcje agregujące), co w konsekwencji utrudniałoby, a nawet uniemożliwiałoby analizę pól i tabel, które należy uaktualnić. Jednakże komponent `Query` posiada właściwość `RequestLive` typu logicznego, pozwalającą czynić wyniki prostszych zapytań SQL edytowalnymi.

Jeżeli właściwość `RequestLive` przyjmuje wartość `false`, to wynik zapytania jest tylko do odczytu. Ustawienie tej właściwości na `true` nie gwarantuje, że wynik zapytania będzie mógł być edytowalny. Decyduje o tym system baz danych na podstawie wewnętrznych kryteriów oceny edytowalności wyników zapytania SQL. Ostatecznym rozstrzygnięciem możliwości edytowania wyników zapytania SQL jest właściwość `CanModify`, będąca właściwością tylko do odczytu.

Możemy jednak przedstawić ogólne reguły, które określają możliwość edycji wyników zapytań SQL.

Dla lokalnych typów baz danych wynik zapytania może być edytowalny, jeżeli nie zawiera żadnego z poniższych elementów:

- klauzuli `DISTINCT`;
- złączeń typu `INNER, OUTER, UNION`;
- funkcji agregujących;
- podzapytań;
- klauzuli *`ORDER BY`* niebazującej na indeksie.

Dla zdalnych serwerów BD wynik zapytania może być edytowalny, jeżeli nie zawiera żadnego z poniższych elementów:

- klauzuli `DISTINCT`;
- funkcji agregujących;
- podzapytań, które odwołują się do innych tabel.

Jeżeli zapytanie nie spełnia powyższych reguł, to nie może być wprost edytowane za pomocą komponentu `Query`. Istnieje jednak możliwość podłączenia zewnętrznego komponentu z odpowiednio przygotowanymi zapytaniami DML (`INSERT, UPDAT, DELETE`), czyniąc wynik zapytania (przynajmniej w pewnej części) edytowalnym.

W tym celu możemy wykorzystać komponent `UpdateSQL`, który umożliwia modyfikowalność wyników zwracanych przez komponenty `Query` oraz `StoredProc`, wykorzystujących *buforowane uaktualnienia* (ang. *cached updates*).

7.2.6. Komponent StoredProc

Komponent `StoredProc` służy do obsługi *procedur zapamiętanych* (*składowanych*).

Definicja 7-1

Procedura zapamiętana jest obiektem serwera baz danych, pozwalającym wykonywać określone zadania (zdefiniowane na serwerze baz danych) oraz zwracać ich wynik do aplikacji klienta.

Konstrukcja ta jest podobna do funkcji znanych z klasycznych języków programowania. Może ona przyjmować parametry wejściowe, jak i zwracać wartości poprzez odpowiednie parametry wyjściowe. Występująca w poprzednim zdaniu liczba mnoga, odnosząca się do parametrów wyjściowych, została użyta świadomie, gdyż *procedura zapamiętana* może zwracać kilka wartości. *Procedura zapamiętana* może również zwracać poszczególne rekordy będące wynikiem zapytania. W takim przypadku procedura udostępnia odpowiedni kursor. Rozróżnienie pomiędzy tymi dwiema procedurami realizowane jest nie tylko na poziomie samej procedury, lecz także na poziomie komponentu ją obsługującego. W pierwszym wypadku używamy komponentu `StoredProc` i jego metody `ExecProc()`. W drugim natomiast wykorzystujemy komponent `Query` i jego metodę `Open()`. *Procedura zapamiętana* zostaje użyta we właściwości SQL.

Komponent `StoredProc` posiada funkcjonalność komponentu `DataSet`, jak również wprowadza dodatkowe typowe dla reprezentowanych obiektów właściwości i metody.

Prześledźmy teraz w punktach kolejne etapy wykorzystania komponentu `StoredProc` do wywołania skojarzonej z nim procedury.

Ustawiamy właściwość `DatabaseName`, określającą serwer bazy danych, na którym zlokalizowana jest *procedura zapamiętana*. Możemy tego dokonać, wykorzystując właściwość `DatabaseName` komponentu `Database` lub wykorzystując zdefiniowany uprzednio *alias*. Następnie określamy nazwę *procedury zapamiętanej* za pomocą właściwości `StoredProcName`. Dostęp do parametrów wejściowych, jak i wyjściowych uzyskujemy dzięki właściwości `Params` klasy `TParams`. Oznacza to, iż dostęp do nich uzyskujemy w sposób analogiczny jak w przypadku parametrów komponentu `Query`. Mianowicie, mamy do wykorzystania trzy poniższe sposoby:

- poprzez właściwość tablicową `System::Variant ParamValues [AnsiString ParamName]`;
- poprzez właściwość `TParam* Items[Word Index]`;
- poprzez funkcję `TParam* __fastcall ParamByName(const AnsiString Value)`.

Wykonanie *procedury zapamiętanej* realizujemy poprzez wywołanie metody `ExecProc()`. Wyniki zwracane przez naszą procedurę dostępne są dzięki właściwości `Params`.

7.3. Dostęp do baz danych InterBase

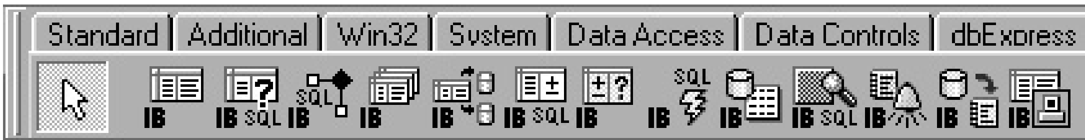
7.3.1. Ogólna charakterystyka komponentów InterBase Express

Komponenty *InterBase Express* (IBX) dostępne są na zakładce *InterBase*. Przeznaczone są one do współpracy z serwerem baz danych *InterBase*. Odwołują się do jego funkcji bezpośrednio, z pominięciem BDE.

Dzięki pominięciu warstwy pośredniej dostęp ten charakteryzuje się wysoką efektywnością oraz dużymi możliwościami, jeżeli chodzi o dostęp do zaawansowanych mechanizmów serwera *InterBase* (komunikaty POST_EVENT, tworzenie kopii zapasowych, zarządzanie użytkownikami itp.).

Wadą tych komponentów, jak każdych realizujących dostęp dedykowany, jest fakt, iż pozwalają one na dostęp tylko do konkretnych serwerów BD. W przypadku komponentów IBX są to serwery *InterBase* oraz jego klon *FireBird*. Należy jednak zaznaczyć, że oba te serwery rozwijane są niezależnie, a co za tym idzie, w każdym z nich pojawiają się nowe elementy. Może się zatem okazać, że komponenty IBX przeznaczone pierwotnie dla serwera *InterBase* nie będą obsługiwały niektórych funkcji serwer *FireBird* lub, co gorsza, będą produkowały „ukryte”, a zatem trudne do wychwycenia błędy. Z drugiej jednak strony grupa programistów rozwijająca serwer *FireBird* zapowiedziała, że jeżeli nastąpią tak duże rozbieżności, to napiszą dedykowaną wersję komponentów przeznaczonych do współpracy z tym właśnie serwerem.

Na zakładce *InterBase* (rys. 7.13) znajdują się następujące komponenty (od lewej):



Rys. 7.13. Komponenty zakładki *InterBase*

- `IBTable` jest odpowiednikiem komponentu `Table`, umożliwiającym dostęp do jednej tabeli lub widoku;
- `IBQuery` jest odpowiednikiem komponentu `Query`, pozwalającym na wykonywanie zapytań SQL. Zbiór wynikowy zwrócony przez komponent `Query` jest zbiorem tylko do odczytu. Komponent ten w połączeniu z `IBUpdateSQL` pozwala uczynić zbiór wynikowy modyfikowalnym;
- `IBStoredProc` jest odpowiednikiem komponentu `StoredProc`, umożliwiającym wykonanie *procedury zapamiętanej*;
- `IBDatabase` jest odpowiednikiem komponentu `Database`, realizującym połączenie z bazą danych;
- `IBTransaction` zapewnia całkowitą kontrolę nad transakcjami realizowanymi na podstawie jednego lub więcej połączeń;
- `IBUpdateSQL` jest odpowiednikiem komponentu `UpdateSQL`, umożliwiającym modyfikowanie zbiorów *tylko do odczytu*;
- `IBDataSet` reprezentuje zbiór danych otrzymywany za pośrednictwem zapytania `SELECT` języka SQL, czyniąc go jednocześnie modyfikowalnym. Jego działanie jest analogiczne do zachowania się połączonych komponentów `IBQuery` oraz `IBUpdateSQL`;
- `IBSQL` umożliwia wykonywanie zapytań SQL bez narzutu komponentu zbioru danych;
- `IBDatabaseInfo` umożliwia uzyskanie informacji o strukturze i stanie bazy danych;
- `IBSQLMonitor` służy do monitorowania poleceń języka SQL wysyłanych do serwera *InterBase*;
- `IBEvents` umożliwia aplikacji odbieranie zdarzeń wysyłanych przez serwer;
- `IBExtract` umożliwia pobranie metadanych, takich jak: tabele, widoki, role, indeksy itd. z serwera *InterBase*;
- `IBClientDataSet` jest odpowiednikiem komponentu `ClientDataSet`.

Jak już powiedzieliśmy we wstępie, komponenty *InterBase Express* posiadają dużo większe możliwości współpracy z serwerem *InterBase* niż komponenty realizujące dostęp za pośrednictwem BDE. Przykładowo: komponent `IBEvents` umożliwia odbieranie zdarzeń wysyłanych przez serwer, natomiast komponent `IBTransaction` może obsługiwać kilka współbieżnych transakcji w obrębie jednej lub kilku baz danych, jak również jedną transakcję obsługującą kilka baz danych.

7.3.2. Komponent IBDatabase

Komponent `IBDatabase` umożliwia aplikacji wykorzystującej technologię IBX dostęp do bazy danych. Mówiąc dokładniej, hermetyzuje połączenie wraz ze wszystkimi jego parametrami. Komponent `IBDatabase` udostępnia metody, właściwości oraz zdarzenia, pozwalające kontrolować różne aspekty połączenia z bazą danych.

7.3.2.1. Fizyczna lokalizacja bazy danych

Określenie fizycznej lokalizacji BD realizowane jest na podstawie właściwości `DatabaseName` (typu `AnsiString`). Format ścieżki dostępu, określający fizyczne położenie pliku z bazą danych, zależy od wykorzystywanego protokołu sieciowego:

- TCP/IP – `<nazwa_serwera>:<nazwa_pliku>`;
- NetBEUI – `\\<nazwa_serwera>\<nazwa_pliku>`;
- SPX – `<nazwa_serwera>@<nazwa_pliku>`.

W przypadku gdy serwer bazy danych zlokalizowany jest na tym samym komputerze co aplikacja, właściwość `DatabaseName` może zawierać tylko fizyczną lokalizację pliku (z BD).

7.3.2.2. Połączenie z serwerem baz danych

Połączenie z serwerem baz danych z wykorzystaniem komponentu `IBDatabase` może być realizowane na dwa sposoby:

- za pomocą metody `Open()`;
- wykorzystując właściwość `Connected` (`Connected = true`).

Oba powyższe sposoby powiązane są ze sobą na poziomie kodu. Poniższy listing prezentuje istotę sytuacji:

```
procedure TCustomConnection.Open;
begin
  SetConnected(True);
end;
```

Listing 7.13

Ustawienie właściwości `Connected` na `false` spowoduje rozłączenie z serwerem baz danych. Analogiczne działanie będzie rezultatem wywołania metody `Close()`.

Łączenie z bazą poprzedzone jest wyświetleniem okienka dialogowego, w którym użytkownik podaje nazwę użytkownika i hasło (oczywiście konto o takich parametrach musi być założone na serwerze bazy danych, z którym nawiązujemy połączenie).

Domyślne zachowanie możemy przedefiniować za pomocą właściwości `LoginPromt` oraz zdarzenia `OnLogin`. Właściwość `LoginPromt` (typu logicznego) określa, czy bezpośrednio przed nawiązaniem połączenia pojawi się okienko logowania. Jeżeli `LoginPromt` przyjmuje wartość `true`, to parametry logowania (nazwa użytkownika i hasło) pobierane są z okienka dialogowego. Jeżeli zaś `LoginPromt` przyjmie wartość `false`, to parametry logowania będą pobrane z właściwości `Params`. Oznacza to, że jej parametry USER_NAME oraz PASSWORD muszą zawierać odpowiednie wartości. Ustawienie ich wartości może odbywać się na etapie projektowania albo bezpośrednio w kodzie aplikacji.

Przypisanie parametrów logowania właściwości `Params` na etapie projektowania jest dobrym rozwiązaniem w trakcie pracy projektowej nad aplikacją. Mianowicie, zwalnia to programistę od konieczności wpisywania przy każdej próbie połączenia z serwerem odpowiedniego hasła. Sposób ten jednak w przy ostatecznej kompilacji (przeznaczonej do dystrybucji) powinien zostać zmieniony, gdyż stanowi poważne zagrożenie dla bezpieczeństwa systemu. Jeżeli decydujemy się na rozwiązanie pozbawione okienka logowania, to powinniśmy ustawiać parametry logowania w kodzie aplikacji, wykorzystując w tym celu mechanizmy kryptograficzne:

```
IBDatabase1->Params->Values["User_Name"] = "SYSDBA";

IBDatabase1->Params->Values["Password"]  =  Deszyfruj(IBDatabase1  -
>Params->Values["Password"]);
```

Listing 7.14

Użycie zdarzenia `OnLogin` pozwala zastąpić domyślne okienko logowania. Wartości parametrów przekazane przez użytkownika powinny być przypisane zmiennej `LoginParams` przekazanej do procedury zdarzeniowej:

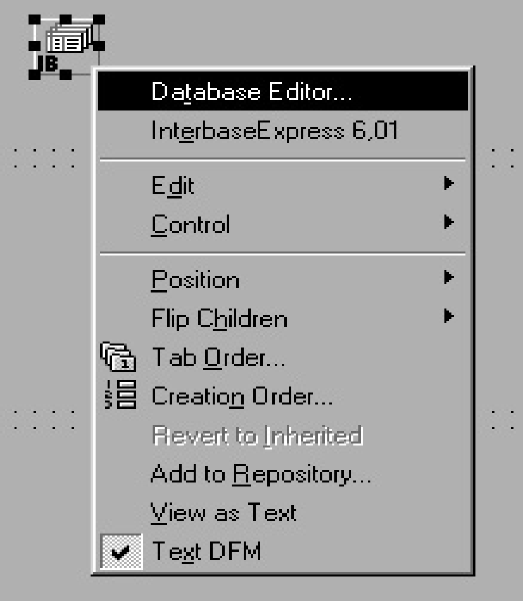
```
void __fastcall TForm1::IBDatabase1Login(TIBDatabase *Database,
      TStrings *LoginParams)
{
  TLoginForm *LoginForm = new TLoginForm(Application);
  try
  {
    if (LoginForm->ShowModal() == mrOk)
    {
      LoginParams->Values["USER_NAME"] = LoginForm->EUserName->Text;
      LoginParams->Values["PASSWORD"] = LoginForm->EPassWord->Text;
    }
  }
  __finally
  {
    delete LoginForm;
  }
}
```

Listing 7.15

Oczywiście, zdarzenie `OnLogin`, tak jak i standardowe okienko logowania wywoływane jest tylko wtedy, kiedy właściwość `LoginPromt` przyjmuje wartość `true`.

Połączenie z bazą danych jest powiązane z otwartością zbioru danych. Mianowicie, otwarcie pierwszego zbioru danych powoduje zainicjalizowanie połączenia z serwerem bazy danych. Rozłączenie natomiast zamyka wszystkie otwarte zbiory danych.

Parametry połączenia możemy ustawić także w edytorze komponentu `IBDatabase`. Uruchamiamy go, wybierając z menu podręcznego pozycję *Database Editor* (rys. 7.14).



Rys. 7.14. Okno edytora komponentów *Database Editor*

Okno edytora umożliwia zdefiniowanie najważniejszych parametrów połączenia. Mianowicie, możemy określić rodzaj serwera (lokalny, zdalny wraz z protokołem i adresem), nazwę bazy danych, nazwę użytkownika oraz hasło, rolę, zestaw znaków, a także właściwość `LoginPromt`.

Przykład 7.34
Oto przykład poprawnie określonych właściwości (patrz rys. 7.15).

Za pomocą przycisku *Test* możemy przetestować poprawność parametrów połączenia.

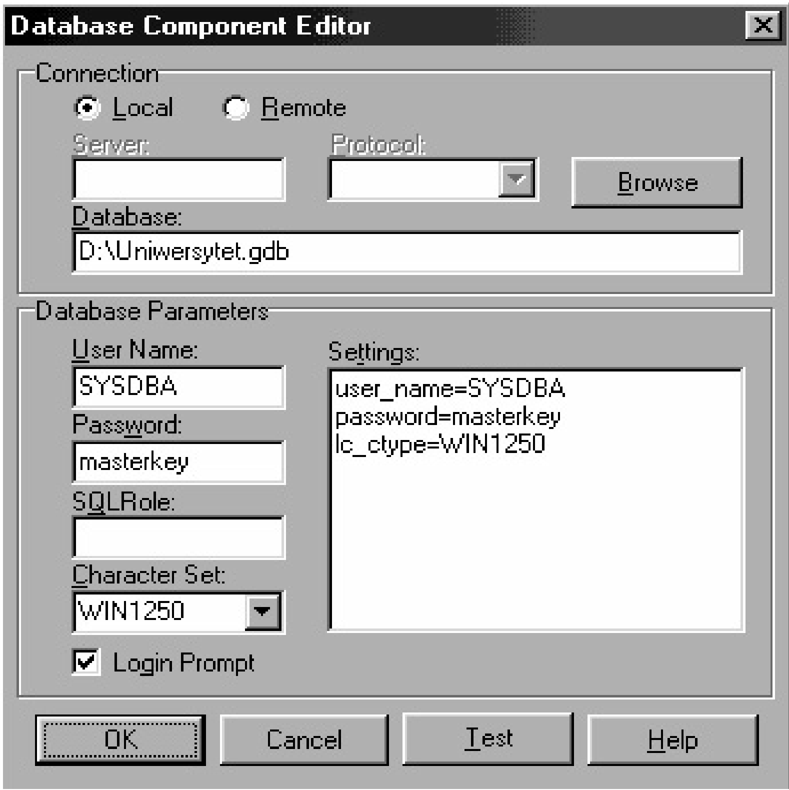
7.3.2.3. Iteracja po wszystkich zbiorach danych

Komponent `IBDatabase` dostarcza dwie właściwości, umożliwiające iterację po wszystkich zbiorach danych z nim skojarzonych. Pierwsza z nich, `DataSets`, jest indeksowaną właściwością tablicową, zawierającą zbiory danych związane z danym komponentem `IBDatabase`. Druga, `DataSetCount`, określa liczby tych zbiorów danych.

Dzięki tym właściwościom możemy kontrolować zachowanie się zbiorów danych. Listing 7.16 przedstawia odświeżenie zawartości wszystkich komponentów `TIBQuery`:

```
TIBQuery *IBQuery;
for (int i = 0; i < IBDatabase1->DataSetCount; i ++)
  {
    IBQuery = dynamic_cast<TIBQuery*>(IBDatabase1->DataSets[i]);
    if (IBQuery)
      {
        IBQuery->Close();
        IBQuery->Open();
      }
  }
```

Listing 7.16



Rys. 7.15. Okno edytora *Database Editor* ze zdefiniowaniem najważniejszych parametrów połączenia

7.3.3. Komponet IBTransaction

Komponent `IBTransaction` sprawuje kontrolę nad jednym lub kilkoma połączeniami z BD. Jak pamiętamy, połączenie to jest realizowane na podstawie komponentu `IBDatabase`. Każdy komponent dziedziczący po klasie `TIBCustomDataSet` i każdy obiekt klasy `TIBSQL` wymaga użycia komponentu transakcji (`IBTransaction`). Konsekwencją tego jest fakt, iż wszelkie operacje na danych realizowane są w kontekście transakcji.

Dane zapisywane w bazie mają charakter nietrwały. Rozstrzygnięcie należy do metod komponentu `IBTransaction`. Mogą one, mianowicie, zatwierdzić bądź wycofać wprowadzone zmiany.

Transakcja w kategoriach bazodanowych jest sekwencją zdarzeń, która musi być wykonana jako niepodzielna całość.

Tak więc, jeżeli wystąpi jakiś błąd, to możemy wycofać wszystkie wykonane dotychczas operacje. Jeżeli natomiast wszystko przebiegnie pomyślnie, to dane możemy zatwierdzić. Należy pamiętać, że dane zatwierdzone nie mogą być już anulowane, jak również nie ma powrotu do anulowanych danych.

Transakcje powinny trwać możliwie krótko, ponieważ zakładane przez nie blokady mogą przyczynić się do spadku efektywności serwera bazodanowego, a także do konfliktów uaktualnień.

7.3.3.1. Użycie komponentu IBTransaction

Jak powiedzieliśmy, komponent IBTransaction sprawuje kontrolę nad połączeniem z bazą danych (realizowanym poprzez komponent IBDatabase). Zatem konieczne jest przypisanie komponentu IBTransaction do odpowiedniego komponentu IBDatabase. Działanie to realizujemy poprzez metodę AddDatabase(). Natomiast związek pomiędzy komponentem zbioru danych (dziedziczącym po TIBCustomDataSet) a komponentem transakcji realizujemy na podstawie właściwości Transaction tego drugiego.

Transakcję otwieramy dzięki metodzie StartTransaction() lub za pomocą właściwości Active. Związek pomiędzy tymi właściwościami reprezentowany jest na poziomie kodu:

```
procedure TIBTransaction.SetActive(Value: Boolean);
begin
  if csReading in ComponentState then
    FStreamedActive := Value
  else
    if Value and not InTransaction then
      StartTransaction
    else
      if not Value and InTransaction then
        Rollback;
end;
```

Listing 7.17

Jeżeli jednak otworzymy komponent zbioru danych powiązany z zamkniętą transakcją, to nastąpi automatyczne jej otwarcie.

Ustawianie właściwości Transaction dla każdego komponentu zbioru danych może być uciążliwe. Dlatego też komponent IBDatabase posiada właściwość DefaultTransaction. Przypisanie jej odpowiedniego komponentu IBTransaction sprawi, iż będzie on domyślnym komponentem transakcji. Fakt ten pozwoli na automatyczne importowanie komponentu IBTransaction do właściwości Transaction komponentu zbioru danych skojarzonego z komponentem IBDatabase (właściwość Database).

Do zatwierdzenia bądź anulowania transakcji służą cztery metody: Commit() i ComitRetaining() oraz Rollback() i RollbackRetaining(). Metoda Commit() zatwierdza wszystkie wprowadzone zmiany, kończąc jednocześnie transakcję. Metoda Rollback() anuluje wprowadzone zmiany, również zamykając transakcję.

Zamknięcie transakcji powoduje zamknięcie wszystkich skojarzonych z nią zbiorów danych. Działanie to może być niepożądane. Dlatego też komponent IBTransaction udostępnia metody z „Retaining", które zachowują bieżący kontekst transakcji, dzięki czemu zbiory danych, tak jak i transakcja nie są zamykane.

Komponent IBTransaction posiada właściwość DefaultAction, której określa rodzaj podejmowanej akcji, jeżeli minie czas trwania transakcji określony we właściwości IdleTimer. Właściwości te pozwalają skrócić czas trwania transakcji. Oprócz tego właściwość DefaultAction określa sposób postępowania w przypadku niszczenia komponentu IBTransaction w odpowiedzi na zamknięcie połączenia z bazą danych

(komponent IBDatabase). Należy podkreślić, że zamknięcie transakcji za pomocą właściwości Active (=false) powoduje wywołanie metody Rollback().

7.3.4. Komponet IBTable

Komponent IBTable, tak jak i jego pierwowzór Table umożliwia dostęp do pojedynczej tabeli bazy danych. Pamiętamy, że głównym przeznaczeniem komponentu Table (zakładka BDE) jest łączenie go z tabelami lokalnych baz danych (*Paradox*, *dBase*). Jego rola we współpracy ze zdalnymi serwerami baz danych powinna ograniczać się do obsługi niewielkich tabel. Oprócz tego zaletą komponentu Table jest fakt, iż jego użytkownik nie musi znać języka SQL.

Klasa TIBTable dziedziczy bezpośrednio po klasie TIBCustomDataSet, a następnie po klasie TDataSet. Zatem związek komponentu IBTable z komponentem Table jest tylko powierzchowny. Jest on ustalony na podstawie analogicznych metod, właściwości i zdarzeń obu komponentów.

Wprowadzenie komponentu IBTable do technologii IBX miało na celu ułatwienie programistom przejście z zestawu komponentów BDE. Wykorzystanie komponentu IBTable powinno podlegać tym samym regułom i ograniczeniom, co wykorzystanie komponentu Table.

Komponent IBTable umożliwia dostęp do wszystkich wierszy i kolumn obsługiwanej tabeli w kontekście odczytywania, jak i modyfikacji jej danych. Możliwe jest także wykorzystanie mechanizmów zakresów oraz filtrów do ograniczenia (bazującego na określonych kryteriach) zestawu widocznych rekordów tabeli. Komponent IBTable pozwala również na tworzenie, zmianę nazwy oraz usuwanie tabeli. Za jego pomocą możemy również tworzyć relacje *Nadrzędny/Podrzędny* (ang. *Master/Detail*).

Opiszemy teraz w punktach proces tworzenia aplikacji, wykorzystującej komponent IBTable do obsługi wybranej tabeli bazy danych:

- układamy na formatce komponent IBTable (zakładka *InterBase*);
- ustawiamy właściwość Database na nazwę komponentu IBDatabase realizującego dostęp do bazy danych;
- ustawiamy właściwość Transaction na nazwę komponentu IBTransaction, realizującego kontrolę transakcji w kontekście komponentu IBDatabase. W tym celu należy ustawić również właściwość DefaultDatabase komponentu IBTransaction na nazwę odpowiedniego komponentu IBDatabase;
- w edytorze komponentu IBDatabase ustawiamy odpowiednie parametry połączenia;
- we właściwości TableName ustawiamy nazwę tabeli wybraną z rozwijanej listy;
- ustawienie właściwości Active na true powoduje otwarcie tabeli.

Wyświetlenie zawartości tabeli realizujemy na podstawie komponentu DataSource i dowolnego komponentu edycyjnego (np. DBGrid). Postępowanie to jest analogiczne jak w przypadku tradycyjnych komponentów bazodanowych (BDE).

Komponent IBTable jest odpowiednikiem komponentu Table. Oznacza to, iż posiada podobne właściwości, metody i zdarzenia. Oprócz tego charakteryzują go podobne zalety, jak i wady. Tak więc jego funkcje ograniczają się tylko do tabel lokalnych lub do

niedużych tabel umieszczonych na zdalnym serwerze. O wiele większą elastycznością charakteryzuje się komponent `IBQuery`, który jest odpowiednikiem komponentu `Query`.

7.3.5. Komponet IBQuery

Najważniejszą własnością komponentu `IBQuery` jest możliwość wysyłania zapytań w języku SQL pod adresem serwera baz danych, a także odbierania jego wyników. A więc działanie to jest analogiczne jak w przypadku jego pierwowzoru – komponentu `Query`.

Klasa `TIBQuery`, tak jak i klasa `TIBTable` dziedziczy bezpośrednio po klasie `TIBCustomDataSet`. Zatem związek komponentu `IBQuery` z komponentem `Query` odnosi się tylko do analogicznego interfejsu. Nie jest natomiast ustalony w relacjach dziedziczenia (poza wspólnym przodkiem – klasą `TDataSet`).

Opiszemy teraz w punktach proces tworzenia aplikacji wykorzystującej komponent `IBQuery`:

- układamy na formatce komponent `IBQuery` (zakładka *InterBase*);
- ustawiamy właściwość `Database` na nazwę komponentu `IBDatabase`, realizującego dostęp do bazy danych;
- ustawiamy właściwość `Transaction` na nazwę komponentu `IBTransaction`, realizującego kontrolę transakcji w kontekście komponentu `IBDatabase`. W tym celu należy ustawić również właściwość `DefaultDatabase` komponentu `IBTransaction` na nazwę odpowiedniego komponentu `IBDatabase`;
- w edytorze komponentu `IBDatabase` ustawiamy odpowiednie parametry połączenia;
- we właściwości SQL podajemy treść zapytania, które chcemy wykonać;
- ustawiamy właściwość `Active` na `true`, co powoduje wykonanie zapytania i zwrócenie wyników.

Wyświetlenie zawartości tabeli realizujemy na podstawie komponentu `DataSource` i dowolnego komponentu edycyjnego (np. `DBGrid`). Postępowanie to jest analogiczne jak w przypadku tradycyjnych komponentów bazodanowych (BDE).

Zawartość komponentu `IBQuery` jest ze swej natury tylko do odczytu i nie może być wprost edytowana. Taka sytuacja dyskwalifikowałaby go z większości ważnych zastosowań, gdyż zapewnienie modyfikowalności zbiorów wynikowych jest jednym z ważniejszych aspektów ich funkcjonalności. Rozwiązaniem tej sytuacji jest wykorzystanie dodatkowego komponentu – `IBUpdateSQL`.

Współdziałanie komponentów zapewnia właściwość `UpdateObject` komponentu `IBQuery`, którą ustawiamy na komponent `IBUpdateSQL`. Trzy jego właściwości: `DeleteSQL`, `InsertSQL`, `ModifySQL`, pozwalają na wprowadzenie odpowiednich treści zapytań języka SQL (DML), dzięki którym możliwa jest modyfikacja danych. Zapytania te mają naturę parametryczną. Mianowicie, nazwy parametrów muszą być tożsame z nazwami pól, które mają podlegać modyfikacji oraz dodatkowo muszą być poprzedzone znakiem dwukropka (`:`). Ponadto w treści zapytań może pojawić się parametr specjalny poprzedzony przedrostkiem OLD_. Oznacza on poprzednią wartość pola, a zatem może być wykorzystywany w zapytaniach `MODIFY` oraz `DELETE`. Parametr ten powinien

odpowiadać kluczowi głównemu, aby zapewnić jednoznaczną identyfikowalność modyfikowanego rekordu.

7.3.6. Komponet IBDataSet

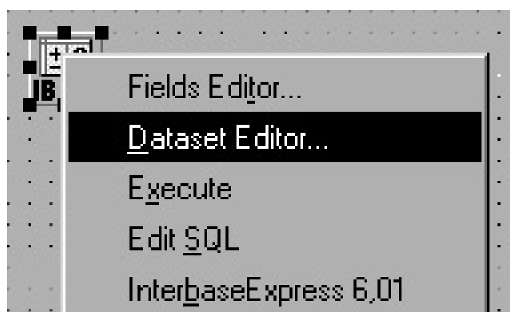
Komponent `IBDataSet` stanowi funkcjonalne połączenie komponentów `IBQuery` i `IBUpdateSQL`, dzięki czemu zapewnia prostą i efektywną edytowalność wyników zapytań.

Komponent `IBDataSet` posiada 5 właściwości, które służą do pobierania i obsługi wyników zapytań SQL. Pierwsza z nich, właściwość `SelectSQL`, pozwala na wprowadzenie treści zapytania języka SQL pobierającej zbiór danych. Kolejne trzy: `DeleteSQL`, `InsertSQL` i `ModifySQL`, służą do wprowadzania zapytań kodyfikacyjnych (DML), analogicznie jak w przypadku komponentu `IBUpdateSQL`. Ostatnia właściwość, `RefreshSQL`, wykorzystywana jest w celu odświeżenia wartości pól bieżącego rekordu. Dzięki temu w przypadku aktualizacji wartości pól na poziomie serwera nie jest konieczne pobranie na nowo całego zbioru, a tylko jednego rekordu.

Proces tworzenia aplikacji wykorzystującej komponent `IBDataSet` wygląda następująco:

- układamy na formatce komponent `IBDataSet` (zakładka *InterBase*);
- ustawiamy właściwość `Database` na nazwę komponentu `IBDatabase`, realizującego dostęp do BD;
- ustawiamy właściwość `Transaction` na nazwę komponentu `IBTransaction`, realizującego kontrolę transakcji w kontekście komponentu `IBDatabase`. W tym celu należy ustawić również właściwość `DefaultDatabase` komponentu `IBTransaction` na nazwę odpowiedniego komponentu `IBDatabase`;
- w edytorze komponentu `IBDatabase` ustawiamy odpowiednie parametry połączenia;
- we właściwości `SelectSQL` podajemy treść zapytania, które chcemy wykonać;
- ustawiamy właściwość `Active` na `true`, co powoduje wykonanie zapytania i zwrócenie wyników.

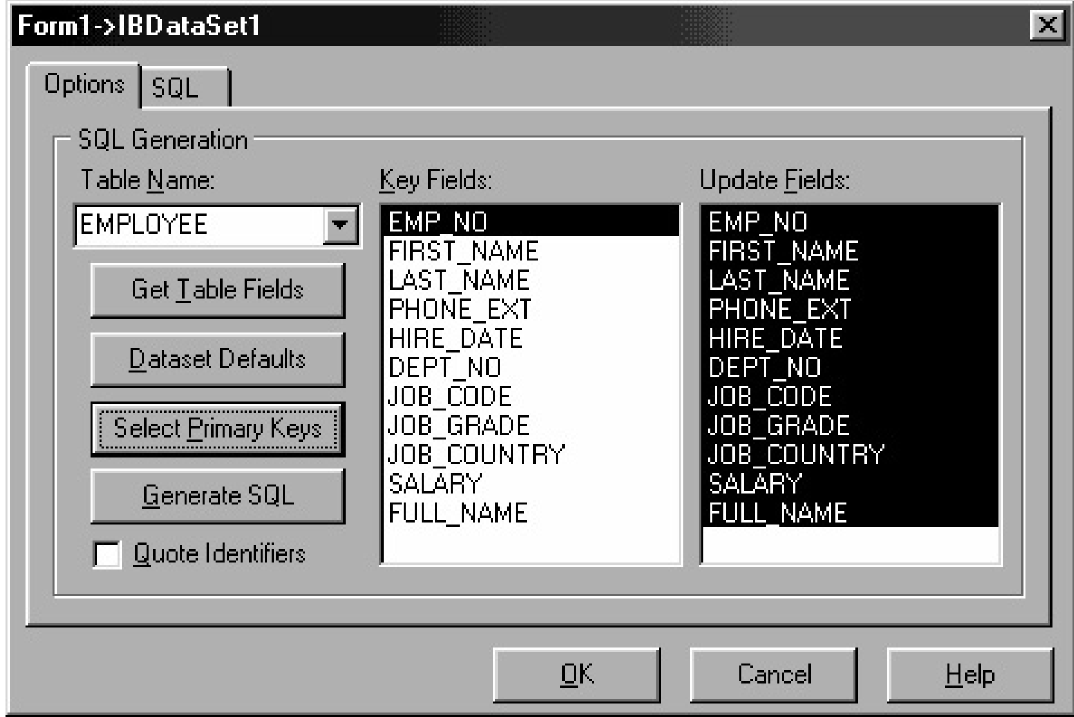
Stworzenie zbioru edytowalnego wymaga uzupełnienia trzech właściwości aktualizacyjnych. Dla ułatwienia możemy skorzystać z kreatora, który wygeneruje za nas odpowiednie zapytania. W tym celu klikamy prawym klawiszem myszy w komponent `IBDataSet` i z menu podręcznego wybieramy pozycję *Dataset Editor* (rys. 7.16).



Rys. 7.16. Okno komponentu `IBDataSet`

Po czym na ekranie pojawi się okienko *IBDataset*. Przykład 7.35 ulustruje wygląd takiego okienka dla tabeli *Employee*.

Przykład 7.35



W przedstawionym powyżej okienku ustalamy parametry, które będą podstawą przy generowaniu zapytań. Z `Comboboxa` *Table Name* wybieramy nazwę tabeli, która będzie podlegała modyfikacji. W lewym `Listboxie` wybieramy pole, które jest kluczem w wybranej tabeli. W prawym natomiast zaznaczamy pola, które będą podlegać modyfikacji. Wyboru *klucza głównego* możemy również dokonać dzięki przyciskowi *Select Primary Keys*. Następnie wciskamy przycisk *Generate SQL*, który wygeneruje odpowiednie treści zapytań. Po zatwierdzeniu wprowadzonych zmian przyciskiem *OK* komponent `IBDataSet` jest w pełni edytowalnym komponentem zbioru danych.

7.4. Dostęp do BD przez Microsoft ActiveX Data Objects

7.4.1. Ogólna charakterystyka ADO

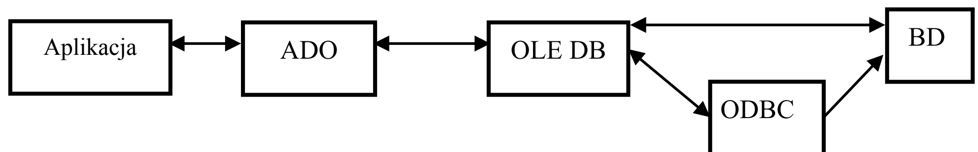
Mimo istnienia tak wielu metod dostępu do danych Microsoft postanowił poprzez *Universal Data Access* stworzyć nowy standard. Pierwszym etapem jego wprowadzania były bazy danych OLE – otwarta specyfikacja zaprojektowana jako kontynuacja sukcesu ODBC. Zaczynając od C++ Builder 5, powstała możliwość pracy z BD za pośrednictwem opracowanej przez Microsoft technologii *ActiveX Data Objects* (ADO).

ADO jest interfejsem wysokiego poziomu, który został zrealizowany na podstawie technologii OLE (*Object Linking and Embeding* – połączenie i wprowadzenie obiektów) DB dostępu do danych. Interfejs ADO musi być zainstalowany w systemie operacyjnym. Jest on dostarczany wraz z systemem Windows 2000 i nowszymi oraz jest dostępny bezpłatnie dla wcześniejszych wersji systemu. Dostępność ADO od kilku lat sprawia, że istnieje dość znaczna liczba sterowników realizujących dostęp do wielu baz. Dostarczanie interfejsu ADO wraz z systemem operacyjnym wyeliminuje kłopoty związane z instalowaniem wraz z każdą aplikacją motorów baz danych. ADO stanowi więc alternatywę dla BDE.

Bazy danych OLE to otwarty standard, umożliwiający dostęp do każdego rodzaju danych. Wspomnijmy, że ODBC stworzono jako metodę dostępu do danych w bazach relacyjnych, podczas gdy bazy danych OLE zaprojektowano z myślą zarówno o relacyjnych, jak i nierelacyjnych źródłach danych (skrzynki pocztowe, tekstowe i graficzne dane w sieci WWW, usługi katalogowe oraz przechowywane na komputerach klasy mainframe dane IMS i VSAM). ODBC używa sterowników ODBC do komunikowania się ze źródłami danych, natomiast bazy danych OLE wykorzystują do tego celu dostawców danych. Bazy danych OLE umożliwiają nauczenie się tylko jednego modelu obiektowego dla wszystkich baz, a nawet używanie tego modelu podczas pracy z nierelacyjnymi źródłami danych. Technologia OLE to znaczny postęp w porównaniu z ODBC, ponieważ potrafi porozumiewać się z wieloma różnymi źródłami danych. Aby zapewnić kompatybilność z poprzednimi wersjami, do baz danych OLE dołączono dostawcę dla ODBC. Dzisiaj istnieją również dostawcy dla *Jet*, SQL *Server*, *Oracle*, NT 5 *Active Directory* i wielu innych programów.

W ten sposób OLE DB to interfejs programowania na poziomie systemu i jest on następcą ODBC. OLE DB jest interfejsem dość złożonym. ADO ułatwia dostęp do ponad sześćdziesięciu interfejsów OLE DB, umieszczając je w 20 prostych obiektach.

Na rys. 7.17 pokazana jest organizacja dostępu do BD poprzez ADO (z wykorzystaniem OLE DB, a także z możliwością użycia ODBC):



Rys. 7.17. Dostęp do bazy danych poprzez ADO (z wykorzystaniem OLE DB, a także z możliwością użycia ODBC)

Obiekty *ActiveX Data* (ADO) opierają się na modelu obiektowym, którego możemy używać do pobierania danych z baz OLE. Technologia ta istniała już od pewnego czasu, a od wersji 2.0 stała się preferowaną przez Microsoft metodą dostępu do danych.

Warto podkreślić, że po wprowadzeniu na rynek pod koniec 1996 r. obiekty ADO (wersja ADO 1.0) używane były głównie jako metoda dostępu do danych poprzez skrypt z wykorzystaniem technologii *Active Server Pages*. Posiadały one jedynie podstawowy zestaw funkcji, służący jako dostęp do danych w architekturze typu *klient/serwer*. W tym czasie obiekty ADO służyły jako podzestaw dla popularnych modeli obiektów DAO i RDO. Programiści używali jedynego dostępnego dostawcy OLE DB – ogólnego dostawcy dla wszystkich źródeł danych ODBC.

Jesienią 1997 r. Microsoft wprowadził na rynek ADO 1.5 oraz *Microsoft Data Access Components* 1.0 (MDAC) i IIS 4.0. Obiekty ADO w wersji 1.5 zawierały m.in. *Remote Data Services* (RDS) oraz obsługę zestawów rekordów odłączonych od bazy. Pierwszym dostawcą baz danych OLE, który ujrzał światło dzienne, był dostawca dla aparatu *Jet* o nazwie JOLT.

Latem 1998 r. wraz z wypuszczeniem wersji 2.0 MDAC powstała trzecia wersja obiektów ADO – ADO 2.0. Był to prawie nadzestaw RDO 2.0 i DAO 3.5, zawierający dodatkowo kilka interesujących opcji. Do ADO 2.0 dołączono również dostawców baz danych OLE dla *Jet*, SQL *Server* i *Oracle*. Wiele z nowych opcji było już wcześniej wykorzystanych w obiektach RDO i DAO, włączając w to operacje asynchroniczne i powtórną synchronizację poprzez kursory klienta. Obiekty ADO zawierały też zupełnie nowe technologie, takie jak:

- *Microsoft Data Links*;
- stałe zestawy rekordów;
- zestawy rekordów tworzone przez użytkownika;
- zestawy rekordów OLAP.

Wydane wraz z SQL *Serverem* 7.0, pakietem *Office* 2000 i *Internet Explorerem* 5.0 obiekty ADO w wersji 2.1 zawierają wszystkie opcje umieszczone w wersji 2.0 oraz dwa nowe modele obiektu:

- ADO – dla operacji DDL i zabezpieczeń (ADOX);
- JRO (*Jet Replication Objects*) – dla opcji replikacji oraz funkcji zmniejszania i naprawiania baz danych w silniku Jet.

Jak wszystko w tej branży, obiekty ADO i BD OLE są przez cały czas udoskonalane. Trudno jest czasami nadążyć za zmianami, jako że Microsoft wydaje nowe wersje obiektów ADO, niezależnie od pakietu *Office* i naszej książki[1].

7.4.2. Komponenty BDE a komponenty ADO

Dostęp do technologii ADO w C++ Builderze realizowany jest za pomocą komponentów umieszczonych w zakładce ADO. Enkapsulują one takie obiekty, jak `Connection`, `Command` i `Recordset`. Ich odpowiednikami w C++ Builder są komponenty `ADOConnection`, `ADOCommand` i `ADORecordset`.

Połączenie z BD poprzez ADO realizuje się poprzez następujący łańcuch: zbiór danych → źródło danych (komponent `DataSourse`) → komponenty sterowania i odwzorowania danych (`DBGrid`, `DBEdit`). Jak widzimy, różnica występuje tylko w pierwszym komponencie łańcucha: zamiast komponentów ze strony *Data Access* biblioteki zastosowano komponenty ze strony ADO.

Większość komponentów ADO (w podstawowym ujęciu) nie różni się od komponentów BDE. Widać więc, że większość ma swoje odpowiedniki (tab. 7.10).

Tabela 7.10. Porównanie komponentów ADO z komponentami BDE

Komponent ADO	*Komponent BDE*
`ADOTable`	`Table`
`ADOQuery`	`Query`
`ADOStoredProc`	`StoredProc`
`ADODataSet`	`Table, Query, StoredProc`

[1] Więcej informacji i najnowsze szczegóły: http:// www.microsoft.com/data.

Komponent ADO	*Komponent BDE*
ADOConnection	Database
ADOCommand	-
RDSConnection	-

W tab. 7.11 podano krótkie charakterystyki podstawowych komponentów ADO.

Tabela 7.11. Charakterystyki podstawowych komponentów ADO

Komponent	*Charakterystyka*
ADOConnection	Pozwala na nawiązanie połączenia z bazą danych, które może następnie udostępniać innym komponentom
ADODataSet	Uniwersalny komponent połączenia ze zbiorami danych; odpowiada takim komponentom BDE, jak: Table, Query, StoredProc. Może się łączyć z jedną lub wieloma tabelami. Połączenie realizuje bezpośrednio lub przez ADOConnection
ADOTable	Przeznaczony do pracy z jedną tabelą. Połączenie realizuje się bezpośrednio lub przez ADOConnection
ADOQuery	Wykorzystywany do pracy ze zbiorem danych za pomocą poleceń SQL. Połączenie ze zbiorem danych realizuje się bezpośrednio lub przez ADOConnection
ADOStoredProc	Przeznaczony jest do wykonywania procedur przechowywanych na serwerze. Połączenie ze zbiorem danych realizuje się bezpośrednio lub przez ADOConnection
ADOCommand	Wykorzystywany jest głównie do wykonywania poleceń SQL niezwracających wielu wyników. Wspólnie z innymi komponentami może być wykorzystywany do pracy z tabelami. Połączenie ze zbiorem danych realizuje się bezpośrednio lub przez ADOConnection

Możemy dodatkowo podkreślić, że nie wszystkie źródła danych mogą pracować ze wszystkimi typami pól. Na przykład: źródło danych *Paradox ADO* nie współpracuje z polami typu graficznego.

W odróżnieniu od komponentów BDE Table, Query i innych w komponentach ADO nie ma właściwości DatabaseName wskazującego na BD. Dostęp do BD polega na zastosowaniu wiersza połączenia – właściwości ADOConnection, którego nazwę podajemy we właściwości innych komponentów. Przykładowo: połączenie z BD można realizować za pomocą właściwości ConnectionString komponentu ADOTable.

Połączenie komponentów z BD można realizować również poprzez właściwość Connection, która łączy używany komponent z komponentem ADOConnection. W komponencie ADOConnection parametry połączenia zadajemy poprzez właściwość ConnectionString. We wszystkich innych zbiorach danych wystarczy we właściwości Connection podać nazwę komponenta ADOConnection.

Połączenie z BD komponentów zbioru danych powiązanych z ADOConnection będzie możliwie nawet wtedy, kiedy w ADOConnection nie zostaną zrobione żadne akcje do uruchomienia BD. Wystarczy w komponencie zbioru danych wpisać: Active = true. Po tym właściwość Connected komponenta ADOConnection automatycznie będzie równa true.

Bardzo ważną cechą komponenta ADOConnection jest to, że umożliwia on sterowanie atrybutami, warunkami podłączenia odpowiednich zbiorów danych, definiuje schemat blokowania rekordów, typ kursora i inne. Dodatkowo metody ADOConnection pozwalają na pracę z transakcjami.

Podczas realizacji połączenia ADOConnection z BD wykorzystuje się metodę *Open*:

```
HIDESBASE void __fastcall Open (const WideString UserId,
                                const WideString Password);
```

Przykład 7.36

```
ADOConnection1->Open(„A”, „1”);
```

Parametry UserId i Password mogą być również podane w łańcuchu połączenia ConnectionString.

Zamykamy połączenia z BD poprzez metodę *Close*.

Teraz zwróćmy uwagę na komponent ADOTable. Jak widać z tab. 7.10, ten komponent pełni takie same funkcje, jak i komponent Table. Nazwę BD podajemy przez właściwości ConnectionString albo przez Connection: należy podać właściwość TableName. Tak jak w przypadku BDE, aby połączyć się z bazą, zastosujemy metodę *Open*: Active = true.

Podstawowe sposoby pracy z ADOTable nie różnią się od sposobów pracy z komponentem Table. Programowy dostęp do pól uzyskujemy tak samo, jak i w komponentach Table i Query: poprzez indeks na podstawie właściwości Fields[i], poprzez nazwę pola na podstawie metody *FieldByName* („<*nazwa_pola*>”) lub poprzez nazwę obiektu pola.

Ograniczenia na wprowadzane wartości w komponencie ADOTable można realizować wyłącznie na poziomie pól. Komponent ADOTable nie posiada właściwości Constraints, jak to ma miejsce w komponentach Table i Query.

Filtracja danych może być wykonana tak samo, jak i w BDE: za pomocą właściwości Filter, w której przechowywane są warunki wyboru danych. Istnieje jednak różnica. Polega ona na tym, że w komponentach ADO w wierszu Filter nazwy pól obowiązkowo powinny być oddzielone od operacji przez spacje. Również przez spacje oddzielamy operatorów AND i OR. Rozważmy przykład.

Przykład 7.37

W komponentach BDE filtr (wybieramy wszystkich studentów oprócz 1. i 5. roku studiów) może być zapisany następująco:

```
(Rok_Studiów<5)AND(Rok_Studiów>1);
```

ten sam warunek w ADO powinien wyglądać inaczej:

```
(Rok_Studiów < 5) AND (Rok_Studiów > 1).
```

Właściwość Filter działa, jeżeli Filtered = true.

I ostatnia istotna rzecz. Komponenty BDE nie posiadają możliwości właściwej komponentom ADO. Mianowicie, chodzi o wykorzystanie metod przechowywania zbioru danych w pliku i odczytywania zbioru danych z tego właśnie pliku. Przykładowo: zapisanie danych do pliku polega na zastosowaniu metody *SaveToFile*:

```
void __fastcall SaveToFile (const WideString FileName,
                            TPersistFormat Format);
```

Parametr `FileName` wskazuje na plik, w którym przechowuje się zbiór danych, parametr `Format` definiuje format pliku. Ostatni parametr może przyjmować jedną z dwóch wartości: `pfADTG` (ADTG, *Advanced Data Tablegram*) albo `pfXML` – format XML (dla ADO 2.1 i wyżej).

Komponenty `ADOQuery` i `ADOStoredProc` praktycznie niczym się nie różnią od komponentów odpowiednio `Query` i `StoredProc`.

7.5. Dostęp do BD przez komponenty dbExpress

W C++ Builder 6 w bibliotece komponentów pojawiła się zakładka *dbExpress*, która wspiera najnowszą technologię firmy Borland służącą do obsługi BD. Podstawą architektury *dbExpress* są sterowniki dla różnych typów baz danych. Każdy z tych sterowników implementuje zestaw interfejsów, umożliwiających dostęp do danych specyficznych dla danego systemu zarządzania bazą danych.

W tab. 7.12 podano krótką informację o sterownikach C++ Builder.

Tabela 7.12. Sterowniki C++ Builder

Sterownik	*Plik sterownika*	*Wersja*
InterBase driver	DBEXPINT.dll	InterBase 6.5
DB2 driver	DBEXPDB2.dll	DB2 7.x
Oracle driver	DBEXPORA.dll	Oracle 8.1.7
MySQL driver	DBEXPMYS.dll	MySQL 3.23.xx
Informix driver	DBEXPINF.dll	Informix 9.21

Sterowniki umożliwiają szybki dostęp do serwerów SQL na podstawie zunifikowanego interfejsu.

Zasadniczą cechą tej technologii jest ograniczenie roli warstwy pośredniej. Zadaniem sterownika jest przekazywanie zapytania do serwera BD bez żadnej ingerencji w treść zapytania.

bdExpress w przeciwieństwie do BDE nie zużywa zasobów serwera związanych z *metadanymi*. Nie zużywa on również tylu zasobów po stronie klienta co BDE, gdyż wykorzystuje tylko jednokierunkowe zbiory danych (nie ma więc potrzeby cacheowania danych).

Technologia *dbExpress* dostępna jest zarówno dla *Delphi*, jak i *Kylixa*. Dzięki temu aplikacje z wykorzystaniem tego interfejsu mogą być wykorzystywane nie tylko na platformie *Windows*, ale również dla platformy *Linux*.

Strona biblioteki *dbExpress* posiada komponenty przedstawione w tab. 7.13.

Tabela 7.13. Opis komponentów biblioteki *dbExpress*

Komponent	*Opis*
`SQLConnection`	Umożliwia połączenie dbExpress z serwerem BD
`SQLDataSet`	Jednokierunkowy zbiór danych posiadający wynik zapytania SQL do serwera
`SQLQuery`	Jednokierunkowy zbiór danych, który wykonuje polecenia SQL do serwera
`SQLStoredProc`	Jednokierunkowy zbiór danych, który wykonuje procedurę przechowującą się na serwerze
`SQLTable`	Jednokierunkowy zbiór danych w wyglądzie tabeli
`SQLMonitor`	Przechwytuje powiadomienie, przez które realizuje się wymiana informacjami pomiędzy serwerem a aplikacją
`SQLClientDataSET`	Zbiór danych klienta, wykorzystujący wewnętrzne `SQLDataSet` i `DataSetProvider` do definiowania i modyfikacji danych

Wspomniany w tab. 7.13 jednokierunkowy zbiór danych należy zbiorom, które nie przechowuje bufor. To zwiększa wydajność operacji nad tymi zbiorami, ale stanowi dodatkowe ograniczenia na ten zbiór. Między innymi takimi ograniczeniami są:

- system wspiera tylko dwie metody nawigacji (*First* i *Next*), chociaż istnieje możliwość bezpośredniej realizacji polecenia `UPDATE`;
- zbiór nie wspiera możliwości filtracji, bo filtracja danych nie jest możliwa bez przechowywania w pomięci zbioru rekordów, chociaż filtracja może być bezpośrednio wpisana w polecenie SQL.

Połączenie *dbExpress* z BD polega na zastosowaniu komponentu `SQLConnection`. Aby `SQLConnection` został podłączony do bazy, należy podać w jego właściwości `ConnectionName` nazwę konfiguracji połączenia. Tę nazwę wybieramy z listy w oknie *Object Inspector*.

Podstawowym parametrem jest nazwa BD – `Database`. Dla *InterBase* jest to nazwa pliku *.gdb*, dla *Oracle* – wejście do *TNSNames.ora*, dla DB2 – węzeł, dla MySQL – nazwa podana przez polecenie `CREATE DATABASE`. Oprócz tego musimy podać nazwę użytkownika (`User_Name`) i hasło (`Password`).

Komponenty `SQLTable`, `SQLQuery`, `SQLDataSet`, `SQLStoredProc` są to jednokierunkowe zbiory danych o pewnych cechach, o których wspomnieliśmy wcześniej. Wszystkie te komponenty łączą się z BD poprzez `SQLConnection`. Właściwość `SQLConnection` posiada każdy z czterech wymienionych tutaj komponentów.

Trudno jest jednoznacznie odpowiedzieć, które z przedstawionych w tym rozdziale rozwiązań jest najlepsze. W każdym przypadku programista powinien dokładnie rozważyć możliwości każdego rozwiązania i wybrać najodpowiedniejsze do tworzonej aplikacji. Powiedzmy teraz o kryteriach, jakimi powinien się on kierować w wyborze najlepszego rozwiązania:

- wydajność;
- koszt;
- dostępność;
- przenośność.

Zadania do samokontroli

1. Scharakteryzować i porównać podstawowe metody podłączenia klienta (użytkownika) do BD.
2. Scharakteryzować podstawowe elementy i podstawy działania BDE i ODBC.
3. Na czym polegają podobieństwa i różnice pomiędzy komponentami BDE, IBX, dbExpress i ADO?
4. Stworzyć i opisać aplikacje do połączenia z jedną z BD (patrz pkt 10 zadań do samokontroli do rozdz. 3):
   - *Biblioteka*;
   - *Apteka*;
   - *Restauracja*;
   - *Przychodnia*;
   - *Szpital*;
   - *Szkoła podstawowa*;
   - *Wydział uczelni wyższej*;
   - *Stacja paliw*;
   - *Wypożyczalnia samochodów*;
   - *Wydawnictwo*;
   - *Księgarnia*.

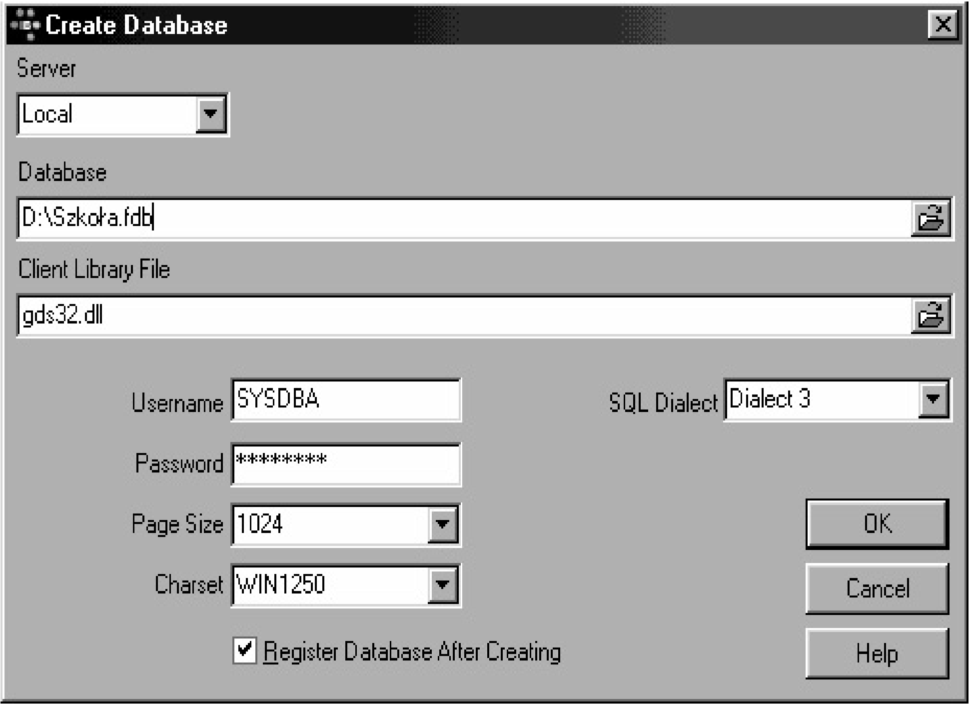
8. Przykładowe aplikacje BD

8.1. Aplikacja wykorzystująca komponenty InterBase Express

Pokażemy teraz prosty przykład, jak stworzyć aplikację bazodanową, wykorzystującą omówione techniki i komponenty. W pierwszej części zostanie utworzona baza danych dla serwera *FireBird*, w drugiej natomiast aplikacja ją obsługująca. Do utworzenia bazy danych wykorzystamy narzędzie *IBExpert*. Aplikacja zaś zostanie utworzona poprzez narzędzie Borland C++ Builder 6.0. Nasze oprogramowanie będzie bazą danych obsługującą szkołę w zakresie dydaktyki (nazwa BD + *Szkoła*). Mianowicie, będzie ona pozwalała przechowywać i wykorzystywać informacje o uczniach, klasach, nauczycielach i przedmiotach, a także o ich wzajemnych relacjach.

8.1.1. Tworzenie bazy danych

Tworzenie bazy danych rozpoczynamy od utworzenia fizycznego pliku przechowującego całą bazę danych. W tym celu z menu *Database* wybieramy pozycję *Create Database*. Okienko to wypełniamy w sposób przedstawiony na rys. 8.1.

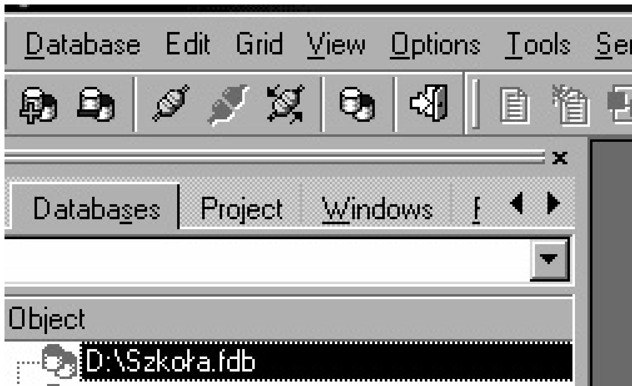


Rys. 8.1. Wypełnione okno *Create Database* dla BD *Szkoła*

Kontrolka *Server* (*ComboBox*) pozwala na wybór lokalizacji serwera. Pierwsza opcja *Local* oznacza serwer lokalny, druga opcja *Remote* – serwer zdalny. W tym drugim przypadku musimy określić w dwóch dodatkowych kontrolkach adres serwera i używany protokół. Kontrolka *Database* określa fizyczne położenie pliku bazy danych. Okienko edycyjne *Client Library File* zawiera nazwę biblioteki rezydującej po stronie *Klienta*.

Kontrolki *Username* i *Password* służą odpowiednio do podania nazwy i hasła użytkownika. Parametr *Page Size* określa rozmiar strony. Zestaw znaków wybieramy w kontrolce *Charset*. Ostatnia opcja SQL *Dialect* pozwala wybrać wersję języka SQL. W tym przypadku mamy dostępne dwie opcje: starszą wersję – *Dialekt 1* oraz nowszą *Dialekt 3*.

Po wciśnięciu przycisku *OK* (dzięki zaznaczeniu opcji *Register Database After Creating*) pojawi się okienko rejestracji bazy danych w narzędziu *IBExpert*. Dzięki zawartym w nim opcjom możemy określić cały zestaw parametrów, służących do sprawowania kontroli nad zachowaniem bazy danych w kategoriach narzędzia *IBExpert*. Jedyną opcją, którą musimy określić na tym etapie, jest wersja serwera – *Server Version*. W naszym przypadku będzie to *Firebird* 1.5. Po kliknięciu w przycisk *Register* nasza baza danych zostanie zarejestrowana, co objawi się wpisem w okienku *Object*, znajdującym się w lewym górnym rogu, poniżej paska narzędziowego (rys. 8.2).



Rys. 8.2. Wygląd okna *Object* z zarejestrowaną BD *Szkoła*

Podwójne kliknięcie w naszą bazę danych skutkować będzie nawiązaniem połączenia, co objawi się rozwinięciem listy obiektów, jak i zmianą kolorów ikonki reprezentującej BD.

Przypomnijmy sobie, że podstawową strukturą każdej bazy są tabele. Utworzenie tabeli rozpoczynamy od kliknięcia prawym klawiszem myszy w gałąź *Tables* i wybraniu z menu podręcznego pozycji *New Table.* Jako pierwszą utworzymy tabelę o uczniach (*Uczeń*) o następującej strukturze: *Uczeń* (*Id_Uczen* PK, *Imię*, *Nazwisko*, *Średnia*, *Id_Klasa* FK); PK tutaj oznacza Primary Key, FK – Foreign Key.

W okienku zlokalizowanym w prawym górnym rogu wprowadzamy nazwę tabeli. W naszym przypadku będzie to *Uczeń*. W siatce znajdującej się w środkowej części okna wprowadzamy kolejno nazwy pól wraz z parametrami określającymi ich typ oraz ewentualnie jakieś dodatkowe opcje.

Pierwsze pole *Id_Uczen* typu `INTEGER` będzie pełniło rolę klucza podstawowego (KP). W tym celu klikamy dwukrotnie w kratkę w kolumnie *PK*, co spowoduje pojawienie się ikonki klucza. Oprócz tego pole to przyjmie atrybut `not null`, aby zagwarantować zgodność modelu fizycznego z modelem teoretycznym. Nazwę pola wprowadzamy w kolumnie *Field Name*, typ natomiast w kolumnie *Field Type*.

Pola tekstowe zdefiniujemy na podstawie domen. W tym celu utworzymy kilka domen o różnych rozmiarach z ustalonym z góry zestawem znaków i porządkiem sortowania. Tworzenie domen rozpoczynamy od kliknięcia prawym klawiszem myszy w gałąź

Domains i wyboru z menu podręcznego opcji *New Domain*. Utworzymy dwie domeny o dwóch różnych rozmiarach. Krótszą, mogącą przechować maksymalnie 20 znaków, i dłuższą o długości 40 znaków.

W tym celu wypełniamy siatkę w sposób pokazany na rys. 8.3.

DVARCHAR40 : VARCHAR(40) CHARACTER SET WIN1250

Domains | Description | Used by | DDL

Name	Field Type	Size	Scale	Not Null	Subtype	Charset	Collate
DVARCHAR20	VARCHAR	20		☒		WIN1250	PXW_PLK
DVARCHAR40	VARCHAR	40		☒		WIN1250	PXW_PLK

Rys. 8.3. Okno dialogowe z opisem stworzonych domen

Za pomocą przycisku z ikonką żółtej błyskawicy (lub za pomocą kombinacji *Ctrl+F9*) kompilujemy nasze domeny. Jeżeli będziemy chcieli zmodyfikować opcje opisane „na szarym tle" (np. `not null`), to musimy wcisnąć przycisk z czerwonym wykrzyknikiem w kółku. Umożliwi to bezpośrednią modyfikację tabel systemowych, co w tym przypadku jest jedyną drogą do zmiany tych parametrów.

Następnie tworzymy dwa pola tekstowe: *Imię* i *Nazwisko*, wykorzystując utworzone uprzednio domeny. W tym celu po wpisaniu nazw pól wybieramy z rozwijanej listy w kolumnie *Domain* nazwę odpowiedniej domeny. W przypadku imienia będzie to DVARCHAR20, a nazwiska DVARCHAR40. Jak możemy zauważyć, opcja `not null` w obu tych polach pozostaje odznaczona, mimo że domeny, na których opierają się te pola, mają włączoną tę opcję. Rozbieżność ta wynika z faktu, iż pola tabeli mają własne wpisy na temat opcji `not null`, niezależne od opcji domeny. Jednakże pole dziedziczy po domenie zaznaczoną opcję `not null`, co sprawi, że będzie się ono zachowywało dokładnie tak, jakby miało tę opcję włączoną. Sytuacja odwrotna jednakże nie zachodzi. Mianowicie, jeżeli pole `not null` korzysta z definicji domeny bez tego parametru, to pozostaje ono nadal polem z opcją `not null`. A zatem nie jest możliwe odebranie opcji `not null` zdefiniowanej w domenie na poziomie pola z niej korzystającego. Należy być zatem świadomym tego faktu i nie nakładać ograniczenia `not null` w definicji domen, jeżeli zachodzi podejrzenie, że którekolwiek z pól ją wykorzystujących może być polem `null`.

Pole *Średnia* będzie przechowywało informacje o średniej ocen uzyskanych przez danego ucznia. Wykorzystamy tu pole stałoprzecinkowe typu `NUMERIC(3, 2)`. Oznacza to, iż pole będzie przechowywało liczby trzycyfrowe, z czego część ułamkowa reprezentowana będzie przez dwie cyfry, natomiast na część całkowitą pozostanie nam jedna cyfra. Nie oznacza to oczywiście, że nie można będzie przypisywać temu polu liczb o większej długości. Rzeczywisty rozmiar tego pola będzie typem najlepiej przystosowanym do przechowywania liczb o określonym przez nas rozmiarze. Szczegóły zostały opisane w rozdziale o serwerze *InterBase*.

Ostatnie pole *Id_Klasa* typu `INTEGER` będzie pełniło rolę klucza obcego typu `not null`. Definicjami *kluczy obcych* zajmiemy się po utworzeniu wszystkich tabel bazy danych.

Struktura i parametry tabeli *Uczeń* powinny wyglądać jak na rys. 8.4.

ID_UCZEN INTEGER NOT NULL

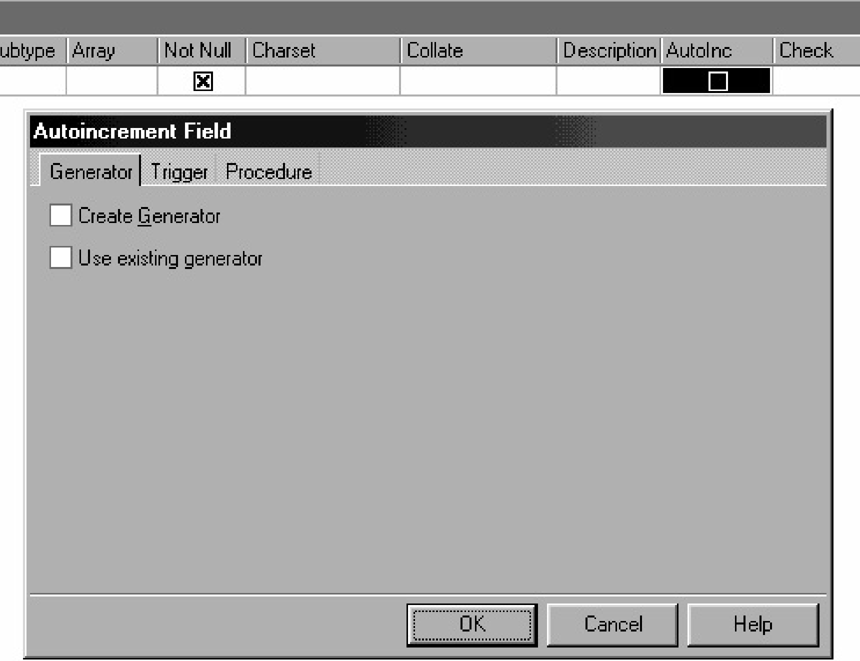
#	FK	PK	Field Name	Field Type	Domain	Size	Scale	Subtype	Array	Not Null	Charset	Collate
1		PK	ID_UCZEN	INTEGER						☒		
2			IMIE	VARCHAR	DVARCHAR20	20				☐	WIN1250	PXW_PLK
3			NAZWISKO	VARCHAR	DVARCHAR40	40				☐	WIN1250	PXW_PLK
4			SREDNIA	NUMERIC		3	2			☐		
5	FK		ID_KLASA	INTEGER						☒		

Rys. 8.4. Struktura i parametry tabeli *Uczeń*

W podobny sposób definiujemy pozostałe tabele:

- *Klasa* (*Id_Klasa* PK, *Nazwa*, *Liczba_Uczniów*);
- *Nauczyciel* (*Id_Nauczyciel* PK, *Imię*, *Nazwisko*, *Pesel*);
- *Przedmiot* (*Id_Przedmiot* Kk, *Nazwa*, *Opis*).

Pole klucza w powyższych tabelach będzie służyło jedynie celom teoretycznym (jednoznacznej identyfikowalności rekordu tabeli oraz definiowaniu relacji). Nie będzie ono nigdy dostępne wprost dla użytkownika, a zatem możemy utworzyć mechanizm automatycznego wypełniania tych pól unikatowymi wartościami. W tym celu klikamy dwukrotnie w kratkę w kolumnie *AutoInc*. Jeżeli chcemy tego dokonać po przekompilowaniu tabeli, to musimy wcisnąć przycisk z paska narzędziowego *Edit Table Structure* lub wcisnąć klawisz *F2* (patrz rys. 8.5).



Rys. 8.5. Okno dialogowe do utworzenia mechanizmu automatycznego wypełniania pól kluczowych

W okienku *Autoincrement Field* zaznaczamy opcję *Create Generator* i przechodzimy na zakładkę *Trigger*. Na niej również zaznaczamy opcję *Create Trigger* i klikamy przycisk *OK*. Dzięki temu wartość pola będzie automatycznie wypełniana – podczas dodawania nowego rekordu – unikatowymi liczbami czerpanymi z utworzonego generatora. W przypadku tabeli *Uczeń* tworzymy tylko generator, gdyż wartość tego pola będziemy pokazywali użytkownikowi i pozwalali ją modyfikować.

W przedstawionych powyżej definicjach tabel na uwagę zasługuje pole *Liczba_Uczniów*. Mianowicie, jego wartość powinna odpowiadać faktycznej liczbie uczniów zapisanych do danej klasy. W celu zagwarantowania poprawnej wartości temu polu możemy za każdym razem, gdy dodajemy bądź usuwamy ucznia, zwiększać bądź zmniejszać tę wartość. Jeżeli tego typu operacje umieścimy w odpowiednim wyzwalaczu, to zagwarantuje nam to automatyczność tego procesu i sprawi, że wartość tego pola będzie odpowiadała faktycznej liczbie uczniów w danej klasie. Możemy też powiązać to pole w sposób bezpośredni – za pomocą odpowiedniej formuły – z zawartością tabeli *Uczeń*. W tym celu przygotujemy pole kalkulowane. Mianowicie, jego wartość obliczana będzie za każdym razem, gdy będzie potrzebna. W naszym przypadku wykorzystamy drugi sposób. W tym celu w kolumnie *Computed Source* wpisujemy następujące zapytanie:

```
(SELECT COUNT(Id_Uczen) FROM Uczen
WHERE Id_Klasa = Klasa.Id_Klasa)
```

Musimy pamiętać, aby jego treść umieścić w nawiasie. Jeżeli nie wprowadzimy treści tego zapytania i skompilujemy tabelę, to nie będziemy mieli już możliwości wprowadzenia go. Jedynym rozwiązaniem będzie usunięcie pola i utworzenie go od nowa. W tym celu usuwamy pole *Liczba_Uczniow* albo za pomocą prawego klawisza myszy i opcji *Drop Field*, albo za pomocą ikonki *Drop/delete field* z paska narzędziowego.

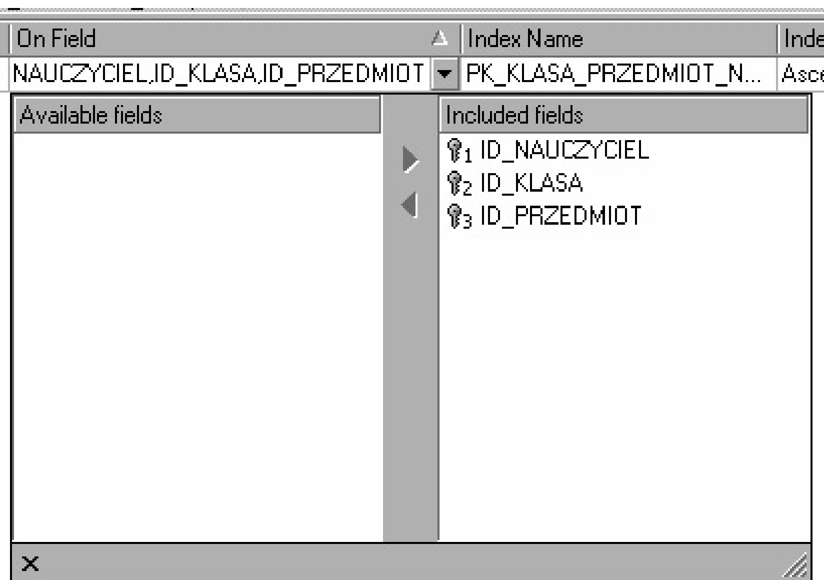
Następnie dodajemy nowe pole w analogiczny sposób. Mianowicie, albo z menu podręcznego wybieramy opcję *New Field*, albo za pomocą ikonki *Add field* z paska narzędziowego. W okienku *Adding New Field* wypełniamy kontrolkę *Field* nazwą pola. Na zakładce *Raw Datatype* wybieramy typ pola. Natomiast zakładka *Computed by* służy do wpisania odpowiedniej formuły.

Oprócz powyższych czterech tabel musimy zdefiniować także tabele łączące, określające relacje pomiędzy klasami, przedmiotami i nauczycielami. W naszym modelu będzie to układ następujący:

- *Klasa_Przedmiot* (*Id_Klasa* PK FK, *Id_Przedmiot* PK FK);
- *Nauczyciel_Klasa_Przedmiot* (*Id_Nauczyciel* PK FK, *Id_Klasa* PK FK, *Id_Przedmiot* PK FK).

Definiowanie klucza złożonego przebiega w sposób analogiczny jak klucza prostego. Mianowicie, wystarczy dwukrotnie kliknąć w kolumnie *PK* wszystkich pól wchodzących w skład klucza. Jeżeli tego nie zrobimy, to po przekompilowaniu tabeli podwójne klinięcie nie będzie skutkowało. Zatem aby zmodyfikować klucz, należy na zakładce *Constraints* i podzakładce *Primary key* dokonać stosownych zmian.

W zasadzie jedynym polem, które powinniśmy zmienić, jest wartość w kolumnie *On-Field*. Określa ona pole wchodzące w skład klucza. Za pomocą ikonki z czarnym trójkącikiem otwieramy okienko z dwoma listami. Po lewej stronie znajdują się pola tabeli, podczas gdy po prawej pola wchodzące w skład klucza. Przenosząc nazwy pól pomiędzy tymi listami, definiujemy właściwą strukturę klucza (rys. 8.6).



Rys. 8.6. Okno definiujące właściwą strukturę klucza

W modelu teoretycznym naszego projektu funkcjonuje pojęcie *klucza obcego* (ang. *Foreign Key,* FK). Oznaczenie wybranych kolumn jako *kluczy obcych* ma dwa zasadnicze cele:

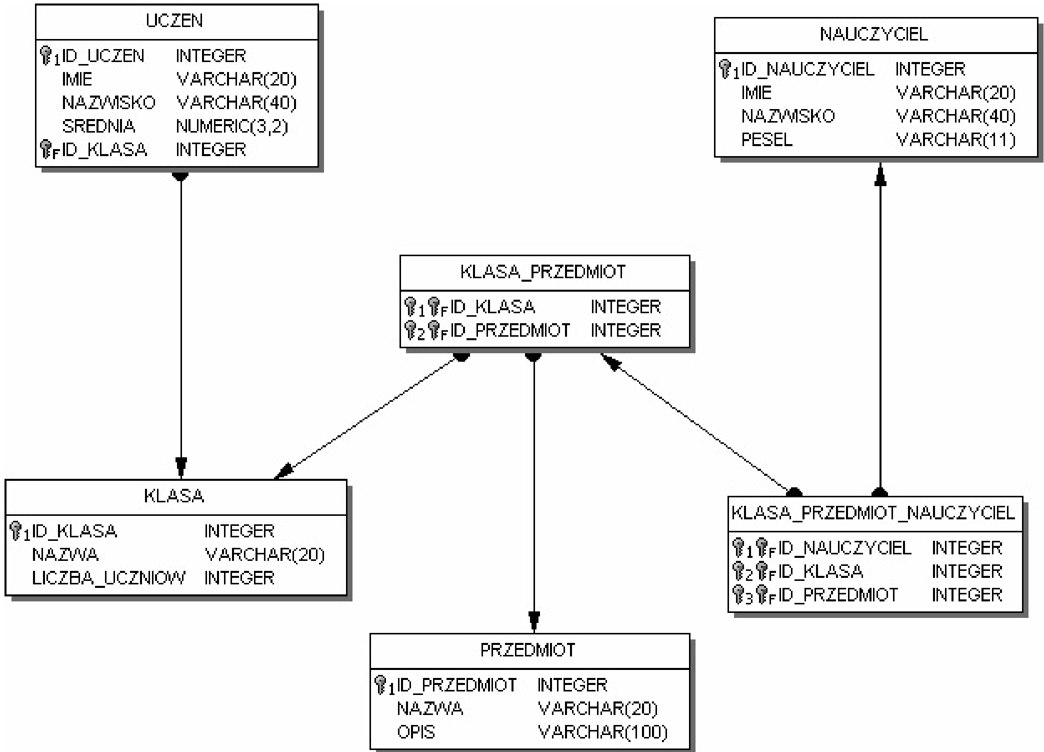
- utrzymanie więzów integralności;
- zoptymalizowanie czasu wykonywania złączeń.

Pierwszy cel służy utrzymaniu integralności na poziomie relacji. Mianowicie, zapobiega on sytuacji, w której występowałyby rekordy utrzymujące nieistniejące relacje. Problem integralności pojawiłby się, gdyby w *tabeli podrzędnej* istniał rekord odwołujący się do nieistniejącego rekordu w *tabeli nadrzędnej*. Mechanizm *kluczy obcych* ma zapobiegać tego typu błędom.

Drugi cel realizowany jest poprzez nałożenie indeksu na pole lub pola wchodzące w skład klucza. Dzięki temu operacje złączeń (bazujące na mechanizmach wyszukiwania) wykonywane są w sposób możliwie najefektywniejszy.

Tworzenie *kluczy obcych* odbywa się poprzez zakładkę *Constraints*, a następnie jej podzakładkę *Foreign keys*. Operacje te wykonujemy na tabeli, która posiada *klucze obce*, czyli na *tabeli podrzędnej*. Całą procedurę rozpoczynamy od wyboru z menu podręcznego pozycji *New foreign key*. Pierwsze pole siatki *Constraints name* określa nazwę ograniczenia, która musi być unikatowa. Nazwa ta będzie dostępna w komunikacie błędu, jaki pojawi się w przypadku naruszenia integralności *klucza obcego*. Druga kolumna *On Field* określa pole lub pola wchodzące w skład *klucza obcego*. Trzecia kolumna *FK Table* pozwala na wybór *tabeli nadrzędnej*, do której odnosi się tworzony *klucz obcy*.

Kolumna *FK Field* jest wypełniana automatycznie *kluczem głównym* tabeli *FK Table*. W tym momencie możemy już skompilować wprowadzoną definicję *klucza obcego*. W naszej bazie danych musimy utworzyć 6 *kluczy obcych*: trzy w tabeli *Klasa_Przedmiot_Nauczyciel*, dwa w tabeli *Klasa_Przedmiot* oraz jeden w tabeli *Uczeń*. Przedstawiony na rys. 8.7 schemat BD powinien ułatwić zdefiniowanie *kluczy obcych*.



Rys. 8.7. Ogólny schemat BD *Szkoła*

Przy tworzeniu *kluczy obcych* ominęliśmy dwie ważne opcje (pozostawiając ich ustawienia domyślne): *Update Rule* i *Delete Rule*. W obu przypadkach mamy do dyspozycji cztery opcje: *No Action*, *Cascade*, *Set Null*, *Set Default*. Działanie powyższych opcji zostało opisane w podrozdz. 5.1.6. Integralność referacyjna.

Stworzona w taki sposób baza danych pozwala już w tym momencie przejść do tworzenia aplikacji. Oczywiście, nasza baza jest bardzo prosta. Jednakże w miarę rozwoju aplikacji będziemy dodawać bardziej zaawansowane mechanizmy, czyniące cały projekt bardziej funkcjonalny. Do wykorzystania mamy takie mechanizmy, jak: widoki, procedury, wyzwalacze, generatory, wyjątki, udf-y, role, indeksy.

8.1.2. Tworzenie aplikacji

Rozpoczynając tworzenie aplikacji, musimy zdecydować o modelu, jaki będziemy wykorzystywać. Do dyspozycji mamy dwa główne typy interfejsu użytkownika: SDI oraz MDI. W pierwszym modelu każde okno aplikacji jest niezależnym oknem interfejsu użytkownika, w drugim natomiast wszystkie okna potomne są umieszczone wewnątrz głównego okna aplikacji.

W naszej aplikacji wykorzystamy technologię MDI, co zapewni nam pewien porządek i ład, którego rola będzie doceniona w bazach danych mogących otwierać wiele okien. Oczywiście, dobrze zaprojektowany interfejs SDI może zapewniać równie dobry sposób współpracy z dużą bazą danych.

8.1.2.1. Tworzenie projektu MDI

Pierwszym krokiem przy tworzeniu aplikacji MDI jest określenie głównej formatki jako okna MDI. W tym celu właściwość `FormStyle` ustawiamy na `fsMDIForm`. Poza tym ustawiamy właściwość `WindowState` na `wsMaximized`, co zapewni, że po uruchomieniu aplikacji główne jej okno będzie zmaksymalizowane. Tak utworzony formularz zapisujemy w pliku pod nazwą *UGłówny* (z rozszerzeniem *.pas*), natomiast formularz nazywamy `FGlowny` (właściwość `Name`). Projekt powinien przyjąć nazwę *PSzkola* (z rozszerzeniem *.bpr*).

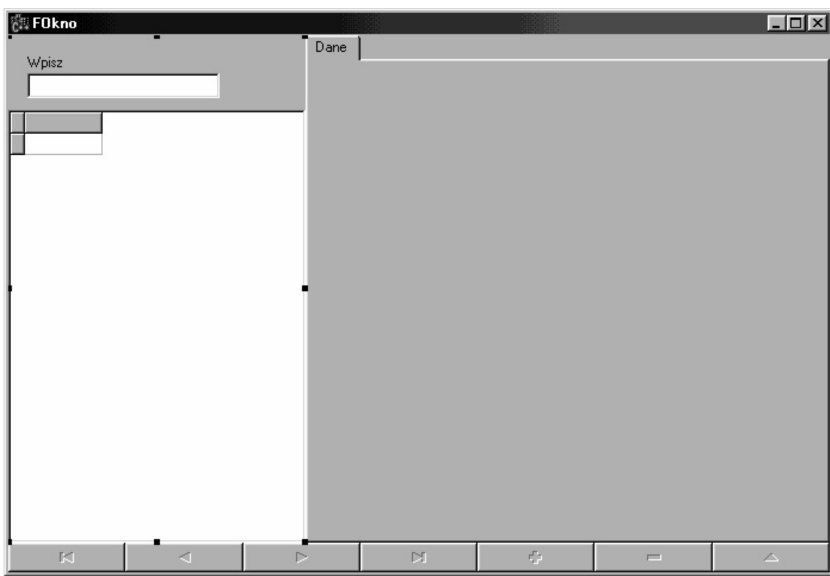
Okna potomne będą wykorzystywane do wyświetlania zawartości głównych tabel bazy danych. Wszystkie one będą miały identyczny interfejs, co można wykorzystać przy ich projektowaniu. Pierwszą najprostszą metodą będzie skopiowanie raz utworzonego formularza i wprowadzenie w poszczególnych oknach drobnych zmian (mających na celu skojarzenie elementów wizualnych z komponentami bazodanowymi). Druga metoda polega na stworzeniu ogólnego formularza, którego zmienne opcje będą parametryzowane. Trzecia i zarazem najlepsza metoda polegać będzie na wykorzystaniu technik programowania obiektowego, co pozwoli nam skorzystać ze wszystkich jego zalet.

Mianowicie, przy projektowaniu formularzy możemy wykorzystać mechanizm dziedziczenia. W tym celu tworzymy formularz bazowy, po którym będą dziedziczyć poszczególne formularze potomne. Formularz bazowy tworzymy tak jak każdy inny formularz. Mianowicie, z menu *File/New* wybieramy pozycję *Form* lub na pasku narzędziowym klikamy w przycisk *New Form*. Pierwszą i najważniejszą właściwością, jaką musimy ustawić, jest `FormStyle`. W naszym formularzu bazowym przyjmie ona wartość `fsMDIChild`. Formatka będzie posiadała kilka komponentów, które będą wspólne dla wszystkich formularzy potomnych. Będą to mianowicie:

- `DBGrid` (zakładka *Data Controls*) – wyświetlanie zawartości tabeli w odniesieniu do najważniejszych pól;

- `Panel` (zakładka Standard) – wykorzystywany do ułożenia na nim komponentów;
- PageControl (zakładka Win32) – wyświetlanie wszystkich danych danego rekordu, wraz z jedną zakładką TabSheet;
- Edit (zakładka Standard) – wpisywanie szukanego ciągu;
- Label (zakładka Standard) – opis elementu wyszukiwania;
- DBNavigator (zakładka Data Controls) – przyciski służące do wykonywania podstawowych operacji na zbiorze danych.

Komponenty na formatce powinny być położone w następujący sposób (rys. 8.8):



Rys. 8.8. Okno klasy bazowej *FOkno*

Aby zapewnić prawidłowe położenie komponentów podczas operacji zmiany rozmiaru okna, należy wykorzystać właściwości `Align` oraz `Anchors`. Komponenty `Label`, `Edit` oraz `DBGrid` układamy na komponencie `Panel`, którego właściwość `Align` przyjmuje wartość `alLeft`. Dodatkowo właściwość `BevelOuter` ustawiamy na `bvNone`, aby stał się on „niewidoczny".

Właściwość `Align` komponentu `DBGrid` ustawiamy na `alBottom` oraz `Anchors/akBottom` na `true`.

Komponent `PageContol` powinien wypełnić cały pozostały obszar, a zatem jego właściwość `Align` przyjmie wartość `alClient`.

W dolnej części okna umieszczamy komponent `DBNavigator`, którego właściwość `Align` ustawiamy na `alBottom`. Za pomocą właściwości `VisibleButtons` określamy, które przyciski będą widoczne. W naszym przypadku trzem ostatnim przyciskom (`nbPost`, `nbCancel`, `nbRefresh`) ustawiamy opcję widoczności na `false`, gdyż nie będziemy z nich korzystać. Dodatkowo modyfikujmy właściwość `Hints`, zamieniając anglojęzyczne opisy na ich polskie odpowiedniki.

Włączamy również właściwość `ShowHint` (`=true`). Ostatnią właściwością, jaką ustawimy, będzie `Position`. Określa ona początkowe położenie okna. W naszym przypadku wybieramy wartość `poMainFormCenter`, co spowoduje, że formatka zajmie centralną pozycję względem głównego okna aplikacji.

Tak utworzony formularz o nazwie FOkno zapisujemy w pliku pod nazwą *UOkno.cpp*.

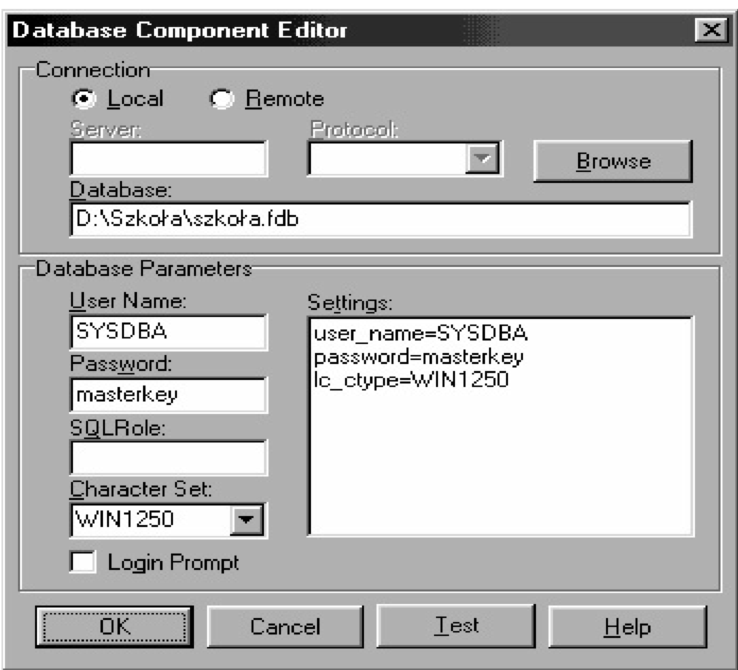
Przechodzimy teraz do utworzenia pierwszego formularza potomnego o nazwie FOkno-Uczen, służącego do wyświetlania informacji o uczniach danej szkoły. W tym celu z menu *File/New* wybieramy pozycję *Other*, która spowoduje pojawienie się okienka *New Items*.

Następnie przechodzimy na zakładkę *PSzkoła*, na której znajdują się utworzone wcześniej formatki, które mogą pełnić role klas bazowych. W naszym przypadku wybieramy formatkę FOkno i klikamy przycisk *OK*. Na wstępie zmienimy dwie etykiety. Mianowicie, właściwość Caption formatki ustawiamy na *Uczniowie* oraz właściwość Caption komponentu Label na wartość *Wpisz imię i nazwisko*.

Ostatnią i najważniejszą rzeczą, jaką musimy wykonać, jest połączenie komponentów kontrolnych ze zbiorem danych. Komponenty bazodanowe zostaną umieszczone w specjalnie do tego przeznaczonym komponencie – DataModule. Takie postępowanie ma na celu logiczne oddzielenie obiektów bazodanowych od interfejsu użytkownika, co znacznie ułatwia proces projektowania. Moduł danych tworzymy, wybierając z menu *File/New* pozycję *Data Module*, którego właściwość Name ustawiamy na DM. Następnie z zakładki *InterBase* układamy na nim trzy komponenty: IBDatabase, IBTransaction, IBDataSet, natomiast z zakładki *Data Access* komponent DataSource. Szczegóły zostaną opisane w następnym podrozdziale.

8.1.2.2. Moduł danych i komponenty bazodanowe

Poprawne skonfigurowanie połączenia z bazą rozpoczniemy od komponentu IBDatabase. W tym celu dwukrotnie klikamy w jego ikonkę. W okienku *Database Component Editor* wypełniamy poszczególne opcje będące parametrami połączenia. Pierwszy parametr *Database* określa lokalizację pliku bazy danych. Jeżeli BD umieszczona jest na tym samym komputerze co nasza aplikacja, to możemy ją zlokalizować za pomocą przycisku *Browse*. Kolejne parametry, jakie musimy wypełnić, to: *User Name*, *Password* i *Character Set*. W naszym przypadku przyjmą one odpowiednio następujące wartości: SYSDBA, *masterkey* i WIN1250. A zatem wypełnione okienko wygląda jak na rys. 8.9.



Rys. 8.9. Okno edytora komponentów BD *Szkoła*

Na rys. 8.9 widzimy, że opcja *Login Prompt* została odznaczona. Dzięki temu nie będzie pojawiało się okienko logowania, co znacznie ułatwi proces tworzenia aplikacji. W tym przypadku logowanie będzie przebiegało automatycznie, a parametry logowania będą pobierane wprost z właściwości `Params`. Oczywiście, ze względów bezpieczeństwa podczas ostatecznej kompilacji aplikacji należy zaznaczyć właściwość *Login Prompt*, a także wykasować hasło (parametr *Password*).

Drugi komponent `IBTransaction` jest odpowiedzialny za sprawowanie kontroli nad połączeniem w kontekście pojedynczej transakcji. Najważniejszym elementem jest ustawienie odpowiedniego poziomu izolacji transakcji. Najczęściej wykorzystywanym (również w naszym przypadku) i najbardziej naturalnym jest poziom *Read Commited*, który pozwala na dostęp do danych zatwierdzonych przez inne transakcje.

W celu ustawienia odpowiedniego poziomu klikamy dwukrotnie w komponent `IBTransaction` i w sekcji *Transaction Properties* zaznaczamy odpowiednią pozycję. W tym momencie powrócimy jeszcze na chwilę do komponentu `IBDatabase`. Mianowicie, ustawimy właściwość `DefaultTransaction` na `IBTransaction`. Dzięki temu komponent `IBTransaction` będzie domyślną i jedyną transakcją komponentu `IBDatabase`. Również komponent `IBTransaction` będzie przechowywał wskazanie na komponent `IBDatabase`.

Dzięki związkowi domyślności właściwość `Transaction` komponentów zbioru danych (np. `IBDataSet`) będzie automatycznie wypełniana (tylko wtedy, kiedy jej wartość jest pusta) domyślną transakcją w momencie powiązania tego komponentu z komponentem `IBDatabase` (za pomocą właściwości `Database`).

Kolejnym komponentem, jakiego użyjemy, będzie `IBDataSet`. Pierwszą ustawioną właściwością będzie `Database`, która wskazuje komponent klasy `TIBDatabase` realizujący połączenie z bazą danych. Po jej ustawieniu, zgodnie z tym, co zostało wcześniej napisane, właściwość `Transaction` automatycznie wypełni się nazwą transakcji domyślnej komponentu `IBDatabase`. W naszym wypadku będzie to `IBTransaction`.

Komponent `IBDataSet` do prawidłowego funkcjonowania wymaga jeszcze wypełnienia właściwości `SelectSQL` treścią odpowiedniego zapytania SQL. W tym celu przechodzimy wprost do edytora właściwości `SelectSQL` lub wybieramy z menu podręcznego komponentu pozycję *Edit SQL*.

W oknie *CommandText Editor*, a dokładniej w kontrolce zatytułowanej *SQL*, wpisujemy treść zapytania SQL. W naszym przypadku rozpoczynamy od najprostszego zapytania dotyczącego tabeli *Uczeń*, a mianowicie:

```
select * from UCZEN
```

Dwa `ListBoxy` znajdujące się po lewej stronie mogą ułatwić to zadanie (podwójne kliknięcie w daną pozycję wprowadza część frazy zapytania SQL). Po kliknięciu w przycisk *OK* następuje zapisanie treści zapytania we właściwości `SelectSQL`. Na zakończenie zmieniamy nazwę komponentu na `IBDSUczen` oraz wartość właściwości `Active` ustawiamy na `true`. Spowoduje to połączenie z bazą danych i wykonanie treści wprowadzonego uprzednio zapytania SQL. Oczywiście, jednocześnie nastąpi otwarcie transakcji.

Ostatnim komponentem, jaki wykorzystamy w tej części, będzie komponent źródła danych `DataSource`. Najważniejszą jego właściwością jest `DataSet`, która wskazuje na zbiór danych, do którego się on odnosi. W naszym wypadku będzie to `IBDSUczen`. Poza tym zmieniamy jego nazwę na `DSUczen`.

Aby zobaczyć zawartość tabeli *Uczeń*, musimy jeszcze skojarzyć komponent `DBGDane` znajdujący się na formatce `FOknoUczen` z komponentem źródła danych `DSUczen`. Nie będzie to jednak możliwe, dopóki nie dołączymy modułu `DM`. W tym celu w pliku *UOknoUczen.cpp* należy dodać wpis `#include "UDM.h"`.

Drugi sposób polega na wybraniu z menu *File* pozycji *Include Unit Hd.* i wybraniu nazwy *UDM*. Dzięki temu po rozwinięciu właściwości `DataSource` zobaczymy komponent `DSUczen` z modułu *UDM*. Nazwa modułu danych to `DM`, a zatem wartością właściwości będzie `DM->DSUczen`.

Po tych operacjach powinniśmy zobaczyć w komponencie `DBGDane` zawartość tabeli *Uczeń*. W podobny sposób łączymy komponent `DBNavigator` ze źródłem danych. W tym celu wykorzystujemy właściwość `DataSource`, której wartość ustawiamy na `DM->DSUczen`.

8.1.3. Pierwsze poprawki aplikacji

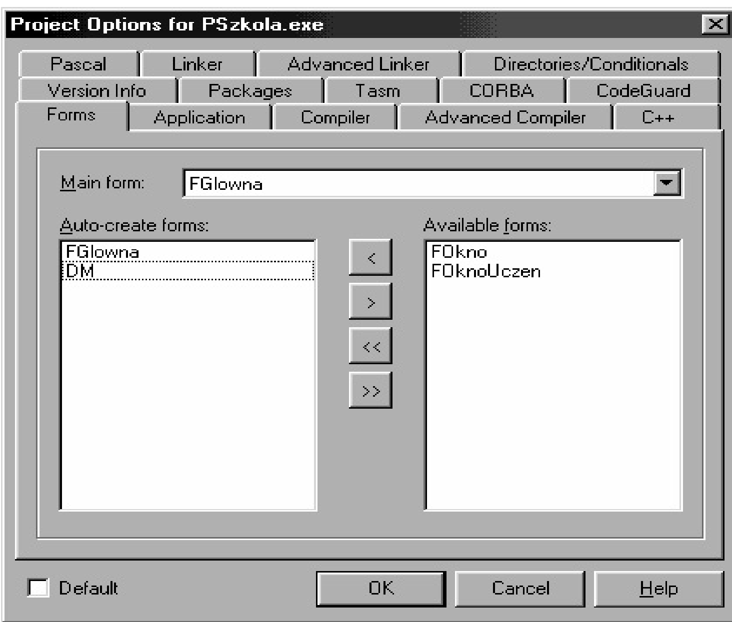
Tak utworzone okno ma kilka wad:

- okno *Uczniowie* pojawia się automatycznie przy starcie, oprócz tego pojawia się też okno klasy bazowej;
- użytkownik nie ma możliwości zamknięcia okna, a jedynie jego zminimalizowanie;
- komponent `DBGDane` jako nazwy kolumn wyświetla nieprzyjazne użytkownikowi nazwy pól;
- aby zobaczyć dalsze kolumny komponentu `DBGDane`, musimy użyć poziomego paska przewijania;
- nie można edytować danych tabeli.

Pierwszy problem związany jest z automatycznym tworzeniem formularzy. Każdy nowo utworzony formularz będzie tworzony automatycznie przy starcie aplikacji. Możemy jednak zadecydować, które formularze projektu powinny być tworzone automatycznie, a które dynamicznie. W tym celu wybieramy z menu *Project* pozycję *Options*.

W okienku *Project Options for PSzkola.exe* widzimy dwa `ListBoxy`. W pierwszym, zatytułowanym *Auto-create forms*, znajdują się formularze tworzone automatycznie. W drugim zaś, zatytułowanym *Available forms*, formularze, które nie są tworzone automatycznie, ale są dostępne i w każdym momencie mogą być utworzone dynamicznie. Wystarczy zatem przenieść formularze `FOkno` i `FOknoUCzen` z lewego do prawego `ListBoxa` (rys. 8.10).

Możemy teraz zobaczyć, że po uruchomieniu aplikacji nasze okna nie pokazują się automatycznie. Nie mamy jednak żadnej możliwości ich zobaczenia. W tym celu utworzymy elementy interfejsu użytkownika, takie jak: menu i pasek narzędziowy. Nie będziemy ich jednak tworzyli niezależnie, ale posłużymy się komponentem `ActionList`. Oprócz tego, że ułatwi nam projektowanie, to zapewni również logiczny związek pomiędzy elementami menu i paska narzędziowego. Mianowicie, jeżeli zmienimy jakąkolwiek opcję akcji, to będzie to widoczne w obu elementach interfejsu użytkownika. Z drugiej strony, jeżeli zmiana będzie dotyczyła któregokolwiek z elementów korzystających z akcji, to także inne elementy przyjmą tę nową opcję (oczywiście tylko w kontekście właściwości obsługiwanych przez akcję).



Rys. 8.10. Okno z definiowaniem opcji BD *Szkoła*

Proces tworzenia akcji rozpoczniemy od umieszczenia na formatce komponentu `ActionList` z zakładki *Standard*. Następnie dwukrotnym kliknięciem otwieramy okienko *Editing FGlowna->ActionList1*. Nową akcję dodajemy za pomocą żółtego przycisku *New Action* zlokalizowanego w lewym górnym rogu. Nowo utworzonej akcji nadajemy nazwę `AUczen` (właściwość `Name`) i przypisujemy ją do kategorii `Obiekty` (właściwość `Category`).

Następnie wybieramy z właściwości `ShortCut` skrót *Ctrl+U* oraz wprowadzamy do właściwości `Hint` tekst *Uczniowie|Wyświetla listę uczniów*. Pionowa kreska umieszczona wewnątrz napisu rozdziela podpowiedź na dwie części. Pierwsza jest podpowiedzią krótką i pojawia się automatycznie przy danym elemencie kontrolnym. Podpowiedź długa może być wykorzystana przez programistę do bardziej szczegółowego wyjaśnienia danej opcji, np. poprzez wyświetlenie jej na pasku statusu.

Kolejną właściwością, jaką ustawimy, będzie `Caption`. Wartość ta będzie tekstem opisującym daną pozycję menu. Mianowicie, przyjmie ona wartość *Uczniowie*.

Ostatnim elementem, jaki wykorzystamy, będzie obrazek, który pojawi się zarówno na pasku narzędziowym, jak i obok danej pozycji menu. Wszystkie obrazki, jakich użyjemy w naszej aplikacji, będą umieszczone w komponencie `ImageList` (zakładka *Win32*). Dodanie obrazka rozpoczniemy od dwukrotnego kliknięcia w komponent `ImageList` i wybrania za pomocą przycisku *Add* odpowiedniego obrazka. Następnie właściwość `Images` komponentu `ActionList` ustawiamy na `ImageList`. Dodatkowo w akcji `AUczen` właściwość `ImageIndex` wypełniamy numerem indeksu obrazka znajdującego się w komponencie `ImageList`. Dla ułatwienia możemy skorzystać z rozwijanej listy właściwości, która zawiera wszystkie obrazki danej listy.

W tym momencie możemy przejść już do tworzenia interfejsów użytkownika. Proces ten rozpoczniemy od menu. Na formatce umieszczamy komponent `MainMenu` (zakładka *Standard*). Pierwszą gałąź menu nazywamy *Obiekty* (właściwość `Caption`), a jej pierwszą pozycję kojarzymy z akcją `AUczen`, wykorzystując właściwość `Action`. Widzimy, że opcje (`Hint`, `ShortCut`, `Caption`) ustawione w akcji `AUczen` zostały zaimportowane do pozycji menu.

Drugi element interfejsu użytkownika, tj. pasek narzędziowy, utworzymy za pomocą komponentu ToolBar (zakładka *Win32*). Po umieszczeniu go na formatce z menu podręcznego możemy dodawać nowe przyciski, wybierając pozycję *New Button*. Tak dodanemu przyciskowi ustawiamy właściwość Action na AUczen. Dodatkowo, aby umożliwić wyświetlanie podpowiedzi, zmieniamy właściwość ShowHint komponentu ToolBar na true.

Widzimy jednak, że żaden z utworzonych elementów nie wyświetla obrazka. Musimy jeszcze skojarzyć oba elementy z komponentem ImageList. W obu tych komponentach odpowiednio ustawiamy właściwość Images (MainMenu i ToolBar).

Utworzone interfejsy użytkownika będą nam umożliwiały (na tym etapie) pokazywanie okna *Uczniowie*. W tym celu przechodzimy do akcji AUczen (klikając dwukrotnie w komponent ActionList) i klikamy w nią dwukrotnie, co przeniesie nas do funkcji obsługującej zdarzenia OnExecute. Zdarzenie to będzie wywoływane w odpowiedzi na kliknięcie w przycisk bądź wybranie odpowiedniej pozycji menu skojarzonych z tą akcją.

W naszym przykładzie dopuszczamy możliwość utworzenia tylko jednego egzemplarza danej klasy. W przypadku istnienia już tego typu okna po prostu je pokazujemy. A zatem zdarzenie to wypełniamy następującym kodem:

```
void __fastcall TFGlowna::AUczenExecute(TObject *Sender)
{
  if (FOknoUczen)
    FOknoUczen->Show();
  else
    FOknoUczen = new TFOknoUczen(this);
}
```

Widzimy, że do poprawnego działania naszego kodu wymagamy, aby po zniszczeniu okna *Uczniowie* wskaźnik FOknoUczen przyjął wartość NULL. Automatyzm tego procesu zapewnimy, umieszczając w zdarzeniu OnDestory kod:

```
void __fastcall TFOknoUczen::FormDestroy(TObject *Sender)
{
  FOknoUczen = NULL;
}
```

Przechodzimy teraz do omówienia drugiego problemu, a mianowicie, aby po naciśnięciu krzyżyka zamykającego okno było ono niszczone, a nie minimalizowane. W tym celu musimy obsłużyć zdarzenie OnClose. Posiada ono parametr Action, przekazywany przez referencję, co sprawi, że wprowadzona modyfikacja będzie widoczna na zewnątrz.

Parametr Action jest typu TCloseAction o następującej definicji:

```
enum TCloseAction {caNone, caHide, caFree, caMinimize};
```

Domyślną opcją w przypadku okien typu MDIChild jest caMinimize, co powoduje minimalizację przy próbie ich zamknięcia. Wystarczy zatem przypisać parametrowi Action wartość caFree, aby okno po zamknięciu było niszczone. Operację te przeprowadzimy dla zdarzenia klasy bazowej TFOkno, dzięki czemu będzie ono wykorzystywane przez wszystkie klasy potomne.

```
void __fastcall TFOkno::FormClose(TObject *Sender, TCloseAction
&Action)
{
```

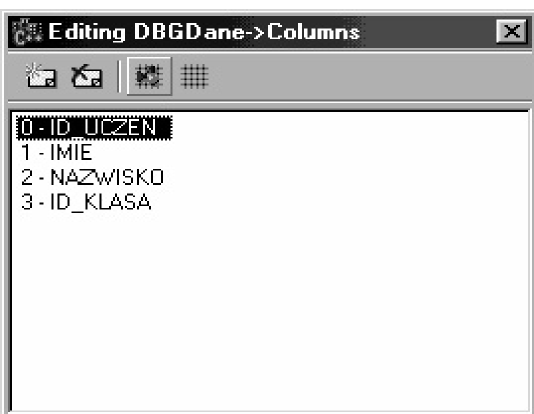
```
    Action = caFree;
  }
```

Możemy się przekonać, że nie jest możliwe ukrycie tego typu okna (co jest naturalną cechą zwykłych okien). W tym celu wystarczy parametrowi `Action` przypisać wartość `caHide`, co przy próbie jego zamknięcia objawi się komunikatem (odpowiednio rys. 8.11).

Rys. 8.11. Okno z komunikatem 'Nie można'

Trzeci problem związany jest z nazwami kolumn prezentowanymi w komponencie `DBGDane`. Ten stan rzeczy możemy zmienić na dwa sposoby. Pierwszy wykorzystuje właściwość `Columns` komponentu `DBGrid`. Drugi natomiast polega na modyfikacji właściwości `DisplayLabel` komponentu `Field`.

Prześledzimy teraz oba sposoby, chociaż w naszej aplikacji zdecydujemy się na drugi sposób. Edytor właściwości `Columns` otwieramy poprzez dwukrotne kliknięcie w komponent `DBGrid` lub wprost z właściwości `Columns` (rys. 8.12).



Rys. 8.12. Okno edytora właściwości `Columns`

Następnie klikamy w przycisk *Add All Fields*, co spowoduje zaczerpnięcie wszystkich pól zbioru danych. Każda z pozycji posiada właściwość `Title`, w której możemy zmodyfikować nie tylko tekst kolumny, lecz także atrybuty jego czcionki. Możemy też ukryć daną kolumnę za pomocą właściwości `Visible` lub całkowicie usuwając daną kolumnę.

Jeżeli zdecydujemy się na drugi sposób (polegający na modyfikacji właściwości `DisplayLabel`), to wprowadzone zmiany będą widoczne we wszystkich komponentach edycyjnych korzystających z danego zbioru danych.

Na wstępie musimy utworzyć statyczne pola zbioru danych. W tym celu klikamy dwukrotnie w komponent `IBDSUczen` znajdujący się w module danych (o nazwie `DM`). Następnie z jego menu podręcznego wybieramy pozycję *Add all fields*, co spowoduje dodanie wszystkich pól zbioru danych. Dalej modyfikując właściwość `DispalyLabel` każdego z pól, wprowadzamy bardziej opisowe nazwy pól. W naszym przypadku będą to kolejno: *Nr*, *Imię*, *Nazwisko* i *Klasa*.

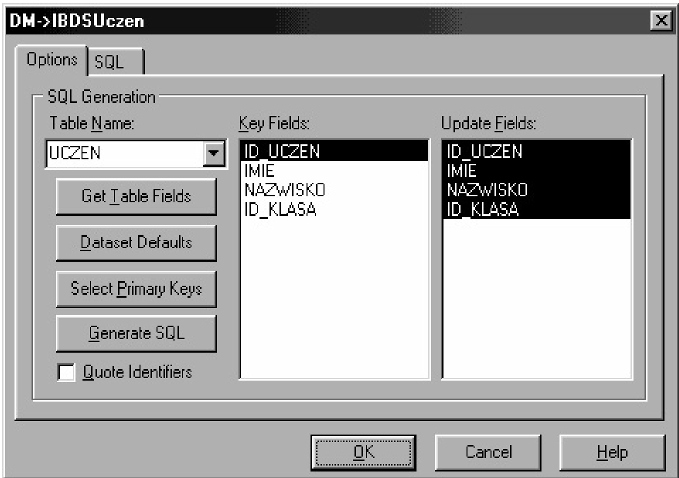
Możemy też zmienić rozmiar kolumn za pomocą właściwości `DisplayWidth`. Czynność tę możemy również przeprowadzić wprost z komponentu `DBGrid`, co znacznie ułatwi dobór odpowiednich wielkości. Umożliwi to częściową realizację punktu czwartego, a dokładniej: po zmniejszeniu kolumny pierwszej przeniesiemy wszystkie kolumny na zakładkę *Dane*. Sprawi to, że w komponencie `DBGrid` będą widoczne trzy kolumny, a do wszystkich pozostałych będziemy mieli dostęp na zakładce *Dane*.

W naszym przypadku będzie to tylko jedna dodatkowa kolumna, ale w przypadku większych tabel rozwiązanie to pozwala na łatwy podgląd całego rekordu. Nasze pola zaprezentujemy z wykorzystaniem komponentów `DBEdit` i opisujących je komponentów `Label`. Możemy to zrobić w sposób standardowy (wybierając z palety komponentów) lub też wykonać tę czynność automatycznie. W tym celu klikamy dwukrotnie w komponent `IBDSUczen`, a następnie po zaznaczeniu wszystkich pól przenosimy je na zakładkę *Dane*. Zaznaczenia tego dokonujemy za pomocą przycisku *Ctrl*, *Shift* lub wybierając z menu podręcznego pozycję *Select all*. Po ich przeniesieniu dostosowujemy odpowiednio rozmiary powstałych komponentów, a także sprawimy, że będą płaskie. Dokonujemy tego za pomocą ustawienia właściwości `Ctl3D` na `false`. Możemy teraz ukryć czwartą kolumnę w komponencie `DBGDane`. Przechodzimy zatem do edytora komponentu `IBDSUczen` i zmieniamy właściwość `Visible` pola *Id_Klasa* na `false`.

Ostatnim i najważniejszym z zakreślonych przez nas problemów jest niemożność edytowania danych tabeli. Wiąże się ona z ogólną cechą wszystkich komponentów bazujących na zapytaniu SQL, a mianowicie, że wszystkie one pozostają w podstawowym kształcie jako zbiory danych tylko do odczytu.

Jednakże komponenty `IBDataSet` posiadają właściwości, które umożliwiają wprowadzenie pełnej edytowalności tychże komponentów. Te właściwości to: `DeleteSQL`, `InsertSQL` i `ModifySQL`. Możemy każdą z nich wypełnić oddzielnie albo możemy też skorzystać z odpowiedniego kreatora. W tym przypadku wybieramy z menu podręcznego komponentu pozycję *Dataset Editor*. W okienku tym widzimy kilka kontrolek.

Rozpoczynamy od `Comboboxa` *Table Name*, wybierając nazwę tabeli, której zawartość będziemy edytować. W naszym przypadku będzie to *Uczeń*. W lewym `ListBoxie` zaznaczamy pole lub pola wchodzące w skład klucza. W prawym natomiast pola, które mają podlegać aktualizacji. W naszym przypadku poprawnie wypełnione okienko powinno wyglądać jak na rys. 8.13.



Rys. 8.13. Okno edytowania tabeli *Uczeń*

Następnie klikamy przycisk *GenerateSQL*, który wygeneruje treści odpowiednich zapytań SQL. Jeżeli przyjrzymy się bliżej tym zapytaniom, to zobaczymy, że są to w gruncie rzeczy zapytania parametryczne. Parametry odpowiadające poszczególnym polom poprzedzone są znakiem dwukropka (`:`).

Druga uwaga dotyczy zapytań `DELETE` i `MODIFY`. Mianowicie, zmiana dokonywana przez te zapytania powinna dotyczyć tylko jednego bieżącego rekordu. W tym celu każde z nich zawiera frazę `WHERE`, w której występuje pole lub pola wchodzące w skład klucza. Parametr jednak różni się od nazwy pola przedrostkiem *OLD_*. Oznacza on, że wartość parametru powinna dotyczyć starej, a nie nowej wartości pola.

Na koniec zwrócimy uwagę na powstałe zapytanie `REFRESH`, którego zawartość przechowywana jest we właściwości `RefreshSQL`. Pozwala ona pobrać zawartość jednego bieżącego rekordu, aby odświeżyć jego zawartość. Działanie takie jest niezbędne, w przypadku gdy rekord zawiera pola, których wartości generowane są przez serwer. Są to m.in. pola z wartościami domyślnymi, z wartościami generowanymi przez triggery oraz pola kalkulowane. Sytuacja ta dotyczy również pól, których zawartość zależy od pól, które uległy modyfikacji. Związane jest to przede wszystkim ze złączeniami tabel, gdyż modyfikacja zazwyczaj dotyczy pól jednej tabeli, podczas gdy pola drugiej udostępniają dane w zależności od pól modyfikowanej tabeli. Dzięki powyższym operacjom rozwiązaliśmy problem opisany w punkcie piątym.

8.1.5. Drugie poprawki aplikacji

Mimo że rozwiązaliśmy podstawowe problemy związane z naszą aplikacją, to nie spełnia one jeszcze wielu podstawowych kryteriów. Tak więc określimy teraz w kilku punktach dodatkowe zmiany, jakim będzie podlegała nasza aplikacja:

1. Możliwość wprowadzania zmian tylko za pośrednictwem formularza.
2. Szybkie wyszukiwanie uczniów po imieniu i nazwisku.
3. Automatyczne wypełnianie wartości *klucza głównego*.
4. Wyświetlanie zamiast identyfikatora klasy jej nazwy.
5. Kojarzenie ucznia z daną klasą za pomocą jej nazwy.

Realizacja punktu pierwszego pozwoli nam uniknąć przypadkowej modyfikacji rekordu, a także oddzieli w sposób logiczny operacje przeglądania danych od ich edycji. Dodawanie, edycję oraz usuwanie bieżącego rekordu będziemy wywoływali za pomocą przycisków `DBNavigatora`. Aktualnie możemy dokonywać zmian wprost w komponencie `DBGDane`, jak również w każdym z komponentów edycyjnych. Za tę sytuację odpowiedzialny jest komponent `DSUczen`, a dokładniej jego właściwość `AutoEdit`. Mianowicie, jeżeli ustawiona jest ona na `true`, to próba modyfikacji danych w komponencie edycyjnym wywołuje metodę `Edit()` zbioru danych, co wprowadza go w stan edycji. Ustawienie właściwości na `false` nie pozwala na automatyczne przechodzenie w tryb edycji.

Dokładniejsza analiza kodu źródłowego ostatecznie wyjaśnia tę kwestię. Przytoczony fragment dotyczy jednego z komponentów edycyjnych. Pozostałe jednak implementują to zachowanie w podobny sposób:

```
procedure TDBEdit.KeyPress(var Key: Char);
begin
  inherited KeyPress(Key);
```

```
    if (Key in [#32..#255]) and (FDataLink.Field <> nil) and
      not FDataLink.Field.IsValidChar(Key) then
    begin
      MessageBeep(0);
      Key := #0;
    end;
    case Key of
      ^H, ^V, ^X, #32..#255:
        FDataLink.Edit;
      #27:
        begin
          FDataLink.Reset;
          SelectAll;
          Key := #0;
        end;
    end;
  end;
```

Widzimy, że metoda `Edit()` obiektu `FDataLink` jest wywoływana przy wciśnięciu dowolnego niebiałego znaku z klawiatury, a także trzech skrótów (*Ctrl* + ...). Obiekt ten jest klasy `TFieldDataLink`, którego metoda `Edit()` przedstawia się następująco:

```
function TFieldDataLink.Edit: Boolean;
begin
  if CanModify then inherited Edit;
  Result := FEditing;
end;
```

Metoda ta korzysta w zasadzie z metody klasy bazowej, którą dla klasy `TFieldDataLink` jest `TDataLink`. Oto kod metody `Edit()` tej klasy:

```
function TDataLink.Edit: Boolean;
begin
  if not FReadOnly and (DataSource <> nil) then DataSource.Edit;
  Result := FEditing;
end;
```

Na zakończenie przechodzimy do właściwej procedury, która to w sposób bezpośredni korzysta z właściwości `AutoEdit`:

```
procedure TDataSource.Edit;
begin
  if AutoEdit and (State = dsBrowse) then DataSet.Edit;
end;
```

Ustawienie właściwości `AutoEdit` na `false` nie blokuje całkowicie możliwości wprowadzania zmian. Mianowicie, możemy tego dokonać za pomocą komponentu `DBGrid`, a dokładniej z pomocą dwóch skrótów. Pierwszy z nich, *Ctrl+Delete*, usuwa bieżący wiersz (wywołując metodę `Delete()`), drugi zaś, *Insert*, dodaje nowy wiersz (wywołując metodę `Insert()`).

Możliwość wprowadzania zmian za pomocą komponentu `DBGrid` blokujemy, ustawiając właściwość `ReadOnly` na `true`. Oczywiście, zmianę tę wprowadzamy w komponencie `DBGDane` z formularza bazowego `FOkno`.

Teraz możemy przejść już do utworzenia formularza edycyjnego. Na wstępie stworzymy formularz bazowy, na którym umieścimy dwa przyciski. Będą to komponenty klasy `TBitBtn`. Odpowiedni wygląd uzyskujemy dzięki właściwości `Kind`. W pierwszym przycisku przyjmie ona wartość `bkOK`, w drugim zaś `bkCancel`.

Automatycznie wraz z tą zmianą odpowiednio ustawiona zostaje właściwość `ModalResult`. Pozwoli ona nam na zamykanie formatki (pokazanej w sposób modalny) za pośrednictwem przycisku, bez konieczności pisania kodu zdarzenia `OnClick`. Ponadto metoda `ShowModal()` formatki zwraca wartość właściwości `ModalResult` przycisku, który doprowadził do zamknięcia formatki, a tym samym pozwala na dokonanie uzależnionych od tego operacji. Dodatkowo modyfikujemy właściwość `Caption` drugiego przycisku, zmieniając wartość *Cancel* na *Anuluj*.

Ostatnim elementem będzie uniemożliwienie zmiany rozmiaru formatki edycyjnej przez użytkownika, co uzyskamy dzięki ustawieniu właściwości `BorderStyle` na wartość `bsDialog`.

W miarę rozwoju aplikacji będziemy dodawali do niego elementy wspólne dla wszystkich formularzy edycyjnych. Formularz taki nazywamy `FEdycja` i zapisujemy w pliku pod nazwą *UEdycja.cpp* oraz przenosimy go na listę formularzy „dynamicznych".

Wykorzystując tak zaprojektowaną formatkę w charakterze klasy bazowej, tworzymy formatkę potomną o nazwie `FEdycjaUczen` i zapisujemy ją w pliku o nazwie *UEdycjaUczen.cpp*. Kontrolki bazodanowe formularza wybieramy z palety komponentów i odpowiednio je kojarzymy ze źródłem danych lub korzystamy z techniki *drag & drop*, która uprości nam ten proces. Mianowicie, klikamy dwa razy w komponent zbioru danych, w naszym przypadku będzie to `IBDSUczen`. W okienku edytora komponentu o nazwie *DM-> IBDSUczen* zaznaczamy wszystkie komponenty pól i przeciągamy je na formatkę `FEdycjaUczen`. Po ich upuszczeniu pojawi się okienko pokazane na rys. 8.14.

Rys. 8.14. Okienko pozwalające na automatyczne dołączenie modułu, w którym zapisany jest kontener `DM`

Nasza formatka jest zatem gotowa do wykorzystania. Wywołanie jej będzie następowało za pomocą odpowiednich przycisków komponentu `DBNavigator`. Oczywiście, wykorzystamy w tym celu `DBNavigator` znajdujący się na formatce bazowej `FOkno`. Aby jednak możliwe było z tego poziomu wywoływanie odpowiednich formatek edycyjnych, klasa `TFOkno` musi posiadać pole wskazujące na odpowiednie okno edycyjne. W naszym przypadku typ tego pola będzie klasą bazową wszystkich okien edycyjnych:

```
protected:
        TFEdycja *FormatkaEdycji;
```

Poprawność tego wpisu musimy uzupełnić poprzez dołączenie do modułu *UOkno* modułu *UEdycja* (oczywiście chodzi o pliki nagłówkowe). Musimy również zadbać o prawi-

dłowe ustawienie tego pola. Najlepiej dokonać tego w ciele konstruktora klas potomnych. W klasie `TFOknoUczen` wpis wyglądał będzie następująco:

```
__fastcall TFOknoUczen::TFOknoUczen(TComponent* Owner)
        : TFOkno(Owner)
{
  FormatkaEdycji = FEdycjaUczen;
}
```

Poprawność tego kodu zapewnimy, dołączając plik nagłówkowy *UEdycjaUczen*. Pokazywanie formatki edycyjnej będzie następowało po naciśnięciu przycisku *Dodaj* lub *Edytuj* komponentu `DBNavigator`. W tym celu obsłużymy zdarzenie `OnClick`. Chodzi tu oczywiście o metodę klasy bazowej `FOkno`, co sprawi, że wszystkie klasy pochodne będą ją dziedziczyły, a tym samym wykorzystywały.

Zdarzenie `OnClick` komponentu `DBNavigator` posiada parametr `Button` typu `TNavigateBtn`, który pozwala zidentyfikować przycisk odpowiedzialny za wywołanie tego zdarzenia. Deklaracja typu `TNavigateBtn` wygląda tak:

```
enum TNavigateBtn {nbFirst, nbPrior, nbNext, nbLast, nbInsert,
nbDelete, nbEdit, nbPost, nbCancel, nbRefresh};
```

W naszym przypadku formatka edycyjna pokazywana będzie po naciśnięciu przycisków: *Dodaj* lub *Edytuj*. Odpowiada to parametrom `nbInsert` i `nbEdit`:

```
void __fastcall TFOkno::DBNavigatorClick(TObject *Sender,
      TNavigateBtn Button)
{
if (Button == nbInsert || Button == nbEdit)
  FormatkaEdycji->ShowModal();
}
```

W powyższym kodzie widzimy przydatność pola `FormatkaEdycji`, które przechowuje wskaźnik formatki edycyjnej. Powyższe rozwiązanie nie jest kompletnym mechanizmem edycyjnym. Mianowicie, po zamknięciu okna zbiór danych pozostaje w trybie edycji. Wyjście z tego trybu może nastąpić za pomocą dwóch metod. Pierwsza, `Post()`, zatwierdza wprowadzone zmiany, druga zaś, `Cancel()`, porzuca je. Naturalnym sposobem byłoby testowanie wartości zwracanej przez funkcje `ShowModal()`.

Tak więc zmodyfikowany kod zdarzenia mógłby wyglądać następująco:

```
void __fastcall TFOkno::DBNavigatorClick(TObject *Sender,
      TNavigateBtn Button)
{
if (Button == nbInsert || Button == nbEdit)
  if (FormatkaEdycji->ShowModal() == mrOk)
    DBNavigator->DataSource->DataSet->Post();
  else
    DBNavigator->DataSource->DataSet->Cancel();
}
```

Takie rozwiązanie sprawi jednak, że w przypadku wystąpienia błędu (np. pozostawienie pustej wartości pola wymaganego) zgłoszony wyjątek wystąpi po zamknięciu okna edycyjnego. W naszym przypadku zmian edycyjnych moglibyśmy dokonać w komponen-

tach umieszczonych na zakładce *Dane*. Mogłoby się jednak zdarzyć, że nie istniałyby inne komponenty edycyjne lub też nasz komponent byłby ostatni (gdyż metoda `DataEvent()` dla poszczególnych obiektów wywoływana jest od końca) na liście `FDataLinks`. Wtedy dostalibyśmy mało zrozumiały komunikat „Cannot focus a disabled or invisible window", który informuje nas o niemożności ustawienia skupienia (*focus*) w okienku edycyjnym, którego wartość została błędnie wypełniona. Co oczywiście jest zrozumiałe, gdyż nasza formatka jest w tym momencie niewidoczna (*invisible*). Dlatego też metody `Post()` i `Cancel()` powinny być wywoływane jeszcze przed zamknięciem formatki. Najlepiej w tym celu wykorzystać zdarzenia poszczególnych przycisków lub też zdarzenie formatki `OnClose`. My wybierzemy sposób drugi.

Będziemy chcieli uniezależnić nasz kod od poszczególnych formatek potomnych i umieścić go w zdarzeniu `OnClose` formularza bazowego. Potrzebujemy zatem pola, które będzie wskaźnikiem na komponent zbioru danych. W sekcji *protected* umieszczamy następującą deklarację:

```
protected:
        TDataSet *DataSet;
```

Ustawienie go na odpowiednią wartość uzyskujemy poprzez uzupełnienie kodu konstruktora:

```
__fastcall TFEdycjaUczen::TFEdycjaUczen(TComponent* Owner)
        : TFEdycja(Owner)
{
  DataSet = DBEdit1->DataSource->DataSet;
}
```

Moglibyśmy wprost przypisać polu `DataSet` wartość `IBDSUczen`, jednakże w taki sposób uniezależniamy się od nazwy konkretnego komponentu. Tak więc w przypadku tego typu zmiany uzyskujemy zawsze poprawne wskazanie. Pozostaje nam jeszcze tylko odpowiednie napisanie kodu zdarzenia `OnClose` formatki `FEdycja`:

```
void __fastcall TFEdycja::FormClose(TObject *Sender, TCloseAction
&Action)
{
  if (ModalResult == mrOk)
    DataSet->Post();
  else
    DataSet->Cancel();
}
```

Na tym zakończyliśmy realizację punktu pierwszego.

Przed oprogramowaniem mechanizmu wyszukiwania, które jest przedmiotem punktu drugiego, wprowadzimy modyfikację kolumn wyświetlanych w komponencie `DBGrid`. Będzie ona dotyczyła w zasadzie komponentów pól.

Zamiast dwóch kolumn (*Imię* i *Nazwisko*) stworzymy jedną wspólną. To nowe pole będzie polem kalkulowanym. Na początku musimy sprawić, aby podstawowe komponenty pól udostępniające zawartość kolumn *Imię i Nazwisko* były niewidoczne. W tym celu ustawiamy w obu komponentach właściwość `Visible` na `false`. Wspólna wartość obu pól będzie wyświetlana w jednym komponencie pola kalkulowanego. Zanim przejdziemy do tworzenia nowego pola, musimy zamknąć zbiór danych, zmieniając wartość właściwości `Active` na `false`.

W tym momencie zajmiemy się bliżej problematyką otwierania i zamykania zbiorów danych. Dotychczas wartość właściwości `Active` równa jest `true`, co oznacza, że zbiór danych otwierany jest w momencie tworzenia modułu danych `DM` i zamykany w trakcie jego niszczenia.

Zbyt duża ilość otwartych zbiorów danych niekorzystnie wpływa na efektywność funkcjonowania aplikacji bazodanowej. Dlatego też otwarciu powinny podlegać tylko te zbiory danych, które są aktualnie wykorzystywane, po czym powinny być natychmiast zamykane. W naszej aplikacji otwieranie i zamykanie zbiorów danych możemy powiązać z tworzeniem i usuwaniem okien potomnych *ChildForm*. Metodę `Open()` umieścimy w kodzie obsługującym zdarzenie `OnCreate`. Natomiast zdarzenie `OnClose` będzie zamykało zbiór danych. Poniższe zdarzenia odnoszą się do klasy bazowej `FOkno`:

```
void __fastcall TFOkno::FormCreate(TObject *Sender)
{
  DBGDane->DataSource->DataSet->Open();
}

void __fastcall TFOkno::FormClose(TObject *Sender, TCloseAction
&Action)
{
  Action = caFree;
  DBGDane->DataSource->DataSet->Close();
}
```

Wracamy już do pola kalkulowanego, które tworzymy, wykorzystując edytor pól *Fields Editor*. W tym celu klikamy dwukrotnie w komponent `IBDSUczen` i z menu podręcznego wybieramy pozycję *New field*. Pole nazywamy IMIĘINAZWISKO, a typ określamy jako `IBString`. Typ pola *Field type* określamy jako *Calculated*. Wypełnione okienko powinno wyglądać w sposób pokazany na rys. 8.15.

Rys. 8.15. Okno definiowania nowego pola (*IMIEINAZWISKO*)

Po kliknięciu przycisku *OK* i utworzeniu pola zmieniamy w *Object Inspectorze* dwie właściwości. Pierwszą, `DisplayLabel`, wypełniamy tekstem *Imię i nazwisko*. Druga właściwość, `Size`, określa rozmiar pola. W naszym przypadku ustawiamy ją na 61

(20 znaków pola IMIĘ + 40 znaków pola NAZWISKO + 1 znak spacji). Wartość pola obliczanego musi być ustawiana w zdarzeniu `OnCalcFields`. Oto jego kod:

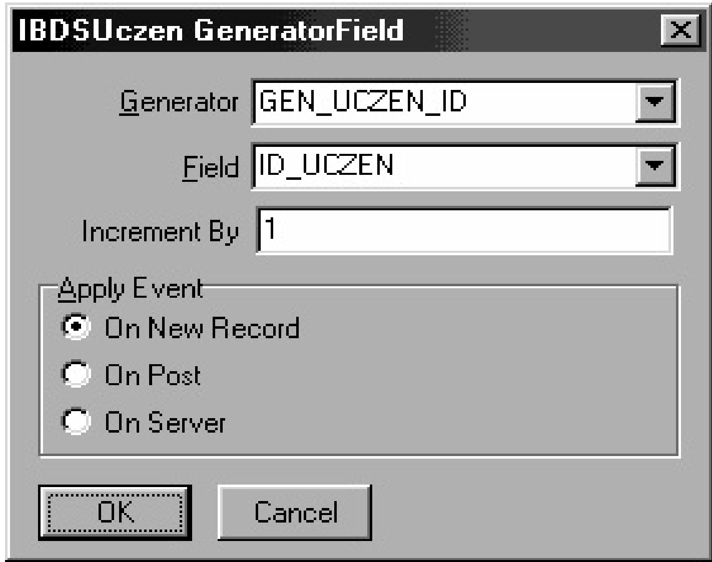
```
void __fastcall TDM::IBDSUczenCalcFields(TDataSet *DataSet)
{
  IBDSUczenIMIĘINAZWISKO->Value = IBDSUczenIMIĘ->Value + " "+
        IBDSUczenNAZWISKO->Value;
}
```

Przejdziemy teraz do mechanizmu wyszukiwania, który zrealizujemy w analogiczny sposób dla wszystkich okien potomnych klasy `TFOkno`. Nakłada to na nas obowiązek ustalenia wybranej kolumny, której wartości będziemy poszukiwali. Wybierzemy pole numer dwa, a zatem to, którego pozycja we właściwości `Column` równa jest 1 (numerowanie kolumn rozpoczyna się od 0). Mechanizm wyszukiwania realizujemy na podstawie zdarzenia `OnChange` komponentu `ESzukaj` (komponent klasy `TEdit`):

```
void __fastcall TFOkno::ESzukajChange(TObject *Sender)
{
  DBGDane->DataSource->DataSet->
Locate(DBGDane->Columns->Items[1]->FieldName, ESzukaj->Text,
TLocateOptions() << loPartialKey << loCaseInsensitive);
}
```

Opcje wyszukiwania `loPartialKey` oraz `loCaseInsensitive` umożliwiają odpowiednio wyszukiwanie częściowe i wyszukiwanie bez względu na wielkość liter.

Problematyka punktu 3 dotyczy zagadnienia unikatowości numeracji używanej dla wartości *klucza głównego*. Mechanizmem wspierającym ze strony serwera będzie generator. Ze strony *Klienta* będzie to właściwość `GeneratorField` komponentu `IBDataSet`. Po wywołaniu jej edytora uzupełniamy dwa pola. Pierwsze `Combobox` o nazwie *Generator* pozwala na wybranie nazwy generatora przeznaczonego dla danego pola. W naszym przypadku będzie to GEN_UCZEN_ID. Drugi, o nazwie *Field*, określa nazwę pola, którego wartość będzie podlegała aktualizacji. Oczywiście, będzie to nazwa pola będącego *kluczem głównym*, czyli ID_UCZEN. Poprawnie wypełnione okienko powinno wyglądać jak na rys. 8.16.



Rys. 8.16. Okno realizacji mechanizmu generacji zawartości pola kluczowego tabeli *Uczeń*

Pozostałe opcje pozostają niezmienione. Pole *Incremented By* określa krok generatora, czyli wartość, o jaką zwiększy się generator po pobraniu nowego numeru. Opcje *Apply Event* określają moment, w którym nastąpi pobranie wartości z generatora. Pierwsza opcja, *On New Record*, pobiera wartość w momencie dodawania nowego wiersza, druga, *On Post*, w trakcie metody `Post()`, trzecia zaś pozostawia tę czynność serwerowi.

Wydawałoby się, że trzecia opcja jest całkowicie zbędna, gdyż bez ustawienia właściwości `GeneratorField` serwer niezależnie wywoła własne mechanizmy (jeżeli zostały zdefiniowane). Jednakże rolą opcji *On Server* jest ustawienie właściwości `Required`, skojarzonego z tym pola, na `false`, co ma zapobiec wyrzuceniu wyjątku (związanego z pozostawieniem pustej wartości pola wymaganego).

Wybór pomiędzy pierwszą a drugą opcją sprowadza się do tego, czy wartość *klucza głównego* potrzebna nam jest w momencie dodawania nowego wiersza, czy dopiero w trakcie jego zatwierdzania. Jeżeli nie jest nam potrzebna znajomość wartości klucza na etapie dodawania nowego wiersza, to lepiej zdecydować się na opcję *On Post*. Zalecenie to wiąże się z faktem, iż pobieranie wartości z generatora skutkuje zwiększeniem jego wewnętrznej wartości. A zatem jeżeli w trakcie dodawania nowego wiersza wyskoczy wyjątek lub użytkownik anuluje wprowadzone zmiany, to pobrana wartość z generatora pozostanie utracona. Dlatego też lepiej przesunąć moment pobrania wartości z generatora na później. Oczywiście, nie ustrzeże to nas przed utraconymi wartościami, lecz tylko zredukuje ich ilość. W naszym przypadku nie możemy jednak skorzystać z tej opcji, gdyż na możliwie wczesnym etapie (dodawanie nowego wiersza) chcemy pokazać użytkownikowi identyfikator *klucza głównego*. Dodatkowo zablokujemy możliwość zmiany tej wartości przez użytkownika. Możemy to uzyskać poprzez ustawienie wartości właściwości `ReadOnly` komponentu pola o nazwie ID_UCZEN na `false`. Drugi sposób polega na zablokowaniu komponentu edycyjnego bądź przez właściwość `ReadOnly`, bądź przez właściwość `Enabled`. W aplikacji wybierzemy pierwszą możliwość.

Realizacja punktu 4 wiązać się będzie ze zmianą treści zapytania SQL, co wpłynie również na właściwość `RefreshSQL`. Oczywiście, rozpoczynamy od modyfikacji głównego zapytania SQL, czyli właściwości `SelectSQL`. Wyświetlenie nazwy klasy wiąże się z pobraniem tej wartości z tabeli *Klasa*, a zatem konieczne jest złączenie obu tabel. Zapytanie SQL realizujące to zadanie może wyglądać następująco:

```
SELECT u.*, k.Nazwa from Uczen u, Klasa k
WHERE u.Id_Klasa = k.Id_Klasa;
```

Pozostaje nam dodanie nowego pola, które stanie się dostępne dzięki temu zapytaniu, a mianowicie pola *Nazwa*. Zmieniamy jeszcze właściwość `DisplayLabel` na *Nazwa*, `Visible` na `false` oraz właściwość `Required` również na `false`.

Aby zobaczyć nazwę klasy, musimy zmienić wartość właściwości `DataField` komponentu `DBEdit` wyświetlającego dotychczas wartość pola *Id_Klasa* na *Nazwa*. Widzimy teraz, że zmiana identyfikatora klasy nie skutkuje zmianą nazwy klasy. Ten stan rzeczy możemy zmienić, wymuszając odświeżanie bieżącego rekordu za pomocą właściwości `ForcedRefresh` ustawionej na `true`. Musimy także zmienić treść zapytania `RefreshSQL` na następującą:

```
SELECT u.*, k.Nazwa from Uczen u, Klasa k
WHERE u.Id_Klasa = k.Id_Klasa
and
  Id_Uczen = :Id_Uczen;
```

Realizacja ostatniego punktu wiąże się z punktem 4. Chcemy mianowicie, aby można było za pomocą nazwy klasy kojarzyć z nią ucznia. Dotychczas musieliśmy znać jej identyfikator, co oczywiście było niezbyt wygodne.

Kojarzenie ucznia z wybraną klasą za pomocą nazwy musi się wiązać ze zmianą odpowiedniej wartości identyfikatora klasy w polu *Id_Klasa*, będącym kluczem obcym w tabeli *Uczeń*. Tego typu powiązanie zapewnia komponent `DBLookupComboBox`. Posiada on tak jak każdy komponent edycyjny właściwości `DataSource` i `DataField`. Pierwszą z nich ustawiamy na `DSUczen`, a drugą na *Id_Klasa*. Widzimy więc, że komponent w rzeczywistości odnosi się do pola *Id_Klasa*, mimo że nie będzie on w sposób bezpośredni prezentował wartości tego pola. Kojarzenie komponentu `DBLookupComboBox` z tabelą *Klasa* (gdyż to właśnie pole *Nazwa* z tej tabeli będzie w nim wyświetlane) rozpoczniemy od komponentu zbioru danych, który będzie zwracał zawartość tej tabeli. W naszym przypadku będzie to komponent `IBQuery` znajdujący się na zakładce *InterBase*. Zwracany przez niego zbiór będzie tylko do odczytu, więc nie musimy stosować komponentu `IBDataSet`. Zapytanie o następującej treści będzie wartością właściwości SQL:

```
SELECT Id_Klasa, Nazwa
FROM Klasa
ORDER BY Nazwa;
```

Komponent `DBLookupComboBox` łączymy z komponentem `IBQuery` za pomocą komponentu źródła danych `DataSource`. W tym celu wykorzystujemy właściwość `ListSource`.

Obok tej zmiany musimy jeszcze ustawić dwie właściwości `KeyField` i `ListField`. Pierwsza z nich służy do wskazania pola będącego kluczem głównym, druga zaś określa pole lub pola, których wartości pojawią się w rozwijanej liście. Jeżeli chcemy użyć kilku pól, to ich nazwy rozdzielamy średnikiem. W naszej aplikacji właściwość `KeyField` przyjmie wartość *Id_Klasa*, natomiast właściwość `ListField` przyjmie wartość *Nazwa*.

Oczywiście, abyśmy mogli korzystać z wartości zwracanych przez zbiór danych, `IBQuery` musi zostać otwarty. Najprostszym rozwiązaniem jest ustawienie w *Object Inspectorze* właściwości `Active` na `true`, co sprawi, że zbiór będzie otwarty przez cały okres pracy naszej aplikacji. Jednakże zmiany dokonane w trakcie pracy naszej aplikacji w tabeli *Klasa* nie będą widoczne w komponencie *IBQuery*, a tym samym w komponencie `DBLookupComboBox`. Dlatego dobrze jest otwierać zbiór w momencie pokazywania formatki i zamykać przy jej chowaniu. Sprawi to, iż po jej otwarciu będziemy widzieli zawsze aktualną zawartość tabeli *Klasa*. Otwarcie zbioru danych oprogramujemy w zdarzeniu `OnShow`, natomiast jego zamknięcie w `OnHide`:

```
void __fastcall TFEdycjaUczen::FormShow(TObject *Sender)
{
  IBQKlasa->Open();
}

void __fastcall TFEdycjaUczen::FormHide(TObject *Sender)
{
  IBQKlasa->Close();
}
```

W działaniu komponentu `DBLookupComboBox` pojawia się mały problem. Mianowicie, po rozwinięciu listy jej długość jest krótsza niż wartość właściwości `DropDownRows`.

Określa ona, ile pozycji listy powinno się pojawić po jej rozwinięciu. Liczba ta jest równa minimum z dwóch wartości DropDownRows i ilości rekordów danego zbioru danych.

Niestety, komponent DBLookupComboBox do określenia ilości wierszy zbioru danych wykorzystuje właściwość RecordCount. Właściwość ta w komponentach pochodnych klasy TIBCustomDataSet (tj. IBQuery, IBDataSet, IBTable) określa tylko ilość wierszy aktualnie pobranych z serwera bazy danych. Dopóki komponent nie zaczerpnie tylu wierszy, ile wynosi właściwość DropDownRows, to liczba pozycji listy komponentu DBLookupComboBox będzie krótsza, niż powinna wynikać z jej wartości.

Poprawną wartość właściwości RecordCount możemy zapewnić jedynie poprzez ściągnięcie wszystkich rekordów z serwera baz danych, co wydaje się strategią dość rozrzutną. Wystarczające jest jednak pobranie tylko tylu wierszy, ile wynosi wartość właściwości DropDownRows. Uzyskamy to, wywołując metodę MoveBy() zbioru danych. Najlepiej wywołać tę metodę bezpośrednio po otwarciu zbioru danych, a zatem zadanie to zrealizujemy w zdarzeniu OnShow:

```
void __fastcall TFEdycjaUczen::FormShow(TObject *Sender)
{
  IBQKlasa->Open();
  IBQKlasa->MoveBy(DBLookupComboBox->DropDownRows - 1);
}
```

W ten sposób zrealizowaliśmy ostatni punkt naszej listy, co oczywiście nie kończy pracy nad naszą aplikacją.

Pozostaje nam zrealizowanie obsługi pozostałych okien. Mianowicie, są to okna reprezentujące nauczycieli, klasy i przedmioty. Ich utworzenie jest jednak analogiczne jak okna obsługującego uczniów. Jedną z różnic jest fakt, iż nie wykorzystujemy wprost generatorów dla wartości *kluczy głównych*. Zamiast tego w bazie danych zostały odpowiednio utworzone *wyzwalacze*, które realizują zadanie uzupełnienia odpowiednimi wartościami *kluczy głównych*. W tym celu pola będące kluczami muszą mieć ustawioną właściwość Required na false, gdyż w przeciwnym wypadku wyskoczy wyjątek zgłoszony przez aplikację w odpowiedzi na brak wartości dla pola wymaganego. Zadanie to możemy zrealizować w sposób bezpośredni poprzez ustawienie właściwości Required na false na etapie projektowania lub też poprzez ustawienie opcji *On Server* we właściwości GeneratorField. Druga różnica związana jest z założeniem, iż pole, względem którego wyszukujemy, zajmuje drugą pozycję w zbiorze danych. Oczywiście, fakt ten nie jest żadnym problemem, gdyż możemy zmienić kolejność pól w zbiorze danych i tych wyświetlanych w komponencie DBGrid.

Dobrze jest jednak dołożyć jako pierwsze pole kolumnę będącą wartością liczbową. Nie chcemy jednak prezentować wartości *klucza głównego*, na które użytkownik nie ma żadnego wpływu. Rozwiązanie nasze polegać będzie na utworzeniu pola obliczanego, które będzie polem porządkowym, co oznacza, że jego wartość równa będzie pozycji danego rekordu w zbiorze danych. Wartość potrzebną do wypełnienia naszego pola uzyskamy dzięki właściwości RecNo zbioru danych. Operację tę zrealizujemy w zdarzeniu przeznaczonym do tego typu operacji, a mianowicie, w zdarzeniu OnCalcFields. Przykładowo dla zbioru danych IBDSNauczyciel będzie ono wyglądać następująco:

```
void __fastcall TDM::IBDSNauczycielCalcFields(TDataSet *DataSet)
{
  IBDSNauczycielLp->Value = IBDSNauczyciel->RecNo;
  IBDSNauczycielIMIĘINAZWISKO->Value = IBDSNauczycielIMIĘ->Value +
" "+ IBDSNauczycielNAZWISKO->Value;
}
```

8.1.6. Oprogramowywanie tabel łączących

Dotychczas utworzone elementy służyły prezentacji i obsłudze tabel podstawowych, tj. *Uczeń*, *Nauczyciel*, *Klasa*, *Przedmiot*. Zajmiemy się teraz wzajemnymi relacjami pomiędzy tymi tabelami, które na poziomie BD przedstawione są za pomocą tabel łączących, a zatem sposób ich obsługi będzie analogiczny jak tabel podstawowych.

Zaczniemy od tabeli *Klasa_Przedmiot*, która reprezentuje związek pomiędzy tabelami *Klasa* i *Przedmiot*. Składa się ona tylko z dwóch pól *Id_Klasa* oraz *Id_Przedmiot*, będących kluczami obcymi. Prezentacja bardziej opisowych pól będzie wiązała się z wykorzystaniem mechanizmu złączeń.

Pierwszym krokiem będzie dodanie nowej zakładki (karty) do komponentu `PageControl` na formatce `FOknoKlasa`. Układamy na niej komponent `DBGrid` oraz `DBNavigator`, a właściwości `Caption` zakładki przypisujemy wartość *Przedmioty*. Na kontenerze `DataModule` umieszczamy komponenty `IBDataSet` oraz `DataSource`, które łączymy ze sobą, a także z nowo położonymi komponentami kontrolnymi na zakładce *Przedmioty*.

Wypełnianie właściwości komponentów rozpoczynamy od nazwania ich (właściwość `Name`), co ułatwi nam projektowanie. Komponent `IBDataSet` przyjmie nazwę `IBDSPrzedmiotKlasy`, a `DataSource` – `DSPrzedmiotKlasy`. Najważniejszym parametrem komponentu `IBDSPrzedmiotKlasy` jest treść zapytania SQL, którą umieszczamy we właściwości `SelectSQL`. Oczywiście, najprostsze zapytanie, takie jak pokazane poniżej, prezentuje tylko wartości *kluczy obcych*, które dla użytkownika są całkowicie nieczytelne:

```
SELECT *
FROM Klasa_Przedmiot;
```

Dlatego też musimy uzupełnić je o nazwę pobraną z tabeli *Przedmiot*. Zadanie to zrealizujemy poprzez mechanizm złączenia przedstawiony poniżej:

```
SELECT kp.*, p.Nazwa
FROM Klasa_Przedmiot kp, Przedmiot p
WHERE kp.Id_Przedmiot = p.Id_Przedmiot;
```

Musimy pamiętać o tym, że lista przedmiotów, jakie pojawią się w komponencie `DBGrid` zakładki *Przedmioty*, musi odpowiadać liście przedmiotów danej klasy, a nie być zestawieniem ich wszystkich. Zadanie to zrealizujemy dzięki sparametryzowaniu zapytania i połączeniu go z komponentem `IBDSKlasa` za pomocą właściwości `DataSource` komponentu `IBDSPrzedmiotKlasy`. Po takim połączeniu każda zmiana pozycji bieżącego rekordu zbioru danych `IBDSKlasa` sprawi zamknięcie i otwarcie zbioru danych `IBDSPrzedmiotKlasy` z nową wartością parametru pobraną z aktualnie bieżącego rekordu. Zapytanie nasze musimy zatem uzupełnić o nazwę parametru tożsamą z nazwą klucza głównego tabeli *Klasa*:

```
SELECT kp.*, p.Nazwa
FROM Klasa_Przedmiot kp, Przedmiot p
WHERE kp.Id_Przedmiot = p.Id_Przedmiot
and Id_Klasa = Id_Klasa;
```

Następnie musimy uczynić nasz zbiór edytowalny, co zrealizujemy dzięki edytorowi *Dataset Editor*. Problemem w takiej sytuacji jest wygenerowana właściwość `RefreshSQL`, która wygląda następująco:

```
SELECT
  Id_Klasa,
  Id_Przedmiot
FROM Klasa_Przedmiot
WHERE
  Id_Klasa = :Id_Klasa and
  Id_Przedmiot = :Id_Przedmiot
```

Z taką treścią zapytania odświeżenie zbioru danych nie będzie uwzględniało pola *Nazwa*, co sprawi, że stanie się ono puste. Dlatego też musimy zmodyfikować je w następujący sposób:

```
SELECT kp.*, p.Nazwa
FROM Klasa_Przedmiot kp, Przedmiot p
WHERE kp.Id_Przedmiot = p.Id_Przedmiot
and
  Id_Klasa = :Id_Klasa and
  Id_Przedmiot = :Id_Przedmiot
```

Jeżeli jednak spróbujemy ustawić właściwość `Active` na `true` zbioru danych `IBDSPrzedmiotKlasy`, to otrzymamy komunikat o zaistniałym błędzie (rys. 8.17).

Rys. 8.17. Komunikat systemu o zaistniałym błędzie w zdaniu SQL

Mówi on mianowicie o niejednoznaczności nazwy pola *Id_Przedmiot*, które występuje równocześnie w tabeli *Klasa_Przedmiot* oraz *Przedmiot*. Rozróżnienia tego dokonamy poprzez dodanie do pola prefiksu *kp.*:

```
SELECT kp.*, p.Nazwa
FROM Klasa_Przedmiot kp, Przedmiot p
WHERE kp.Id_Przedmiot = p.Id_Przedmiot
and
  Id_Klasa = :Id_Klasa and
  kp.Id_Przedmiot = :Id_Przedmiot
```

Odświeżanie bieżącego rekordu – po jego zatwierdzeniu – zagwarantujemy poprzez ustawienie właściwości `ForcedRefresh` na `true`. Do tak przygotowanego komponentu za pomocą edytora pól *Fields Editor* dodajemy wszystkie pola zbioru danych. Dwa pierwsze pola czynimy niewidocznymi dla komponentu `DBGrid`, ustawiając im właściwość `Visible` na `false`. Komponentowi pola *Nazwa* zmieniamy właściwość `Required` na `false`, gdyż pole to nie uczestniczy w procesie edycji.

Edycję zbioru danych zrealizujemy, wykorzystując dodatkową formatkę. Umieścimy na niej komponent `DBLookupComboBox`, który ułatwi nam wybór odpowiedniego

przedmiotu. Jako źródło danych wpisane we właściwości `ListSource` wykorzystamy komponent `DSPrzedmiot` klasy `TDataSource` i powiązany z nim komponent `IBQPrzedmiot` klasy `TIBQuery`. Jego właściwość `SQL` wypełniamy następującym zapytaniem:

```
SELECT Id_Przedmiot, Nazwa
FROM Przedmiot
ORDER BY Nazwa;
```

Fraza `ORDER BY` służy do posortowania danych względem pola nazwisko, co ułatwi użytkownikowi korzystanie z komponentu `DBLookupComboBox`. Otwieranie i zamykanie zbioru danych `IBQPrzedmiot` zrealizujemy analogicznie jak w przypadku formatki edycyjnej `FEdycjaUczen`. Tabela *Klasa_Przedmiot* posiada dwa pola wymagane, które musimy wypełnić. Pierwsze *Id_Przedmiot* wypełnia użytkownik za pomocą komponentu `DBLookupComboBox`, drugie zaś powinno być wypełnione automatycznie, zgodnie z wartością pola *Id_Klasa* bieżącego rekordu zbioru danych `IBDSKlasa`. Zadanie to zrealizujemy w zdarzeniu `OnNewRecord` komponentu `IBDSPrzedmiotKlasy` w następujący sposób:

```
void __fastcall TDM::IBDSPrzedmiotKlasyNewRecord(TDataSet *DataSet)
{
  IBDSPrzedmiotKlasyID_KLASA->Value = IBDSKlasaID_KLASA->Value;
}
```

Problem związany z operacją realizowaną w tym zdarzeniu uaktywni się w przypadku pustego zbioru `IBDSKlasa`, gdyż wtedy pole *Id_Klasa* reprezentować będzie wartość NULL, co zostanie skonwertowane poprzez właściwość `Value` na 0. Przez to operacja `Post()` zatwierdzenia rekordu zbioru danych wyrzuci wyjątek związany z nieistnieniem odpowiadającego tej wartości pola w *tabeli nadrzędnej*. Problem ten rozwiążemy jednak po przedstawieniu zdarzenia `OnClick`.

Wyświetlenie formatki edycji zrealizujemy w zdarzeniu `OnClick` komponentu `DBNavigator` w sposób przedstawiony poniżej:

```
void __fastcall TFOknoKlasa::DBNPrzedmiotClick(TObject *Sender,
      TNavigateBtn Button)
{
if (Button == nbInsert || Button == nbEdit)
  FEdycjaPrzedmiotuKlasy->ShowModal() == mrOk;
}
```

Wydawałoby się, że rozwiązanie przedstawionego powyżej problemu możemy realizować poprzez odpowiednie sformułowanie warunku *if*:

```
void __fastcall TFOknoKlasa::DBNPrzedmiotClick(TObject *Sender,
      TNavigateBtn Button)
{
if ((Button == nbInsert || Button == nbEdit) &&
!DBNPrzedmiot->DataSource->DataSet->DataSource->DataSet->IsEmpty())
  FEdycjaPrzedmiotuKlasy->ShowModal() == mrOk;
}
```

Takie działanie sprawi, że niezależnie od poprawności warunku będziemy w trybie edycji. Dlatego też musimy warunek pustości zbioru danych testować przed wywołaniem

zdarzenia OnClick. Zadanie to pozwala nam zrealizować zdarzenie BeforeAction. Wystarczy wyrzucić w nim wyjątek, co sprawi, iż nie wejdziemy w tryb edycji, jak również do zdarzenia OnClick. W naszym przypadku będzie to cichy wyjątek (EAbort) wyrzucony metodą Abort():

```
void __fastcall TFOknoKlasa::DBNPrzedmiotBeforeAction(TObject *Sender,
      TNavigateBtn Button)
{
  if          (DBNPrzedmiot->DataSource->DataSet->DataSource->DataSet-
>IsEmpty())
    Abort();
}
```

W analogiczny sposób zrealizujemy listę klas uczęszczających na dany przedmiot. Ostatnim ważnym elementem, jaki musimy zrealizować, jest skojarzenie nauczyciela z zajęciami, które prowadzi. W tym celu w module DM umieszczamy komponent IBDataSet oraz DataSource, które nazywamy odpowiednio IBDSNauczycielZajecia i DSNauczycielZajecia. Właściwość Database ustawiamy na IBDatabase. Natomiast wartość właściwości SelectSQL wypełniamy następującą treścią zapytania SQL:

```
SELECT kpn.*, k.Nazwa K_Nazwa, p.Nazwa P_Nazwa
FROM Klasa_Przedmiot_Nauczyciel kpn, Klasa k, Przedmiot p
WHERE kpn.Id_Klasa = k.Id_Klasa and kpn.Id_Przedmiot = p.Id_Przedmiot
and kpn.Id_Nauczyciel = :Id_Nauczyciel
ORDER BY  k.Nazwa, p.Nazwa
```

Właściwości DeleteSQL, InsertSQL oraz ModifySQL wypełniamy automatycznie za pomocą przycisku *Generate SQL* w edytorze *Dataset Editor*. Wypełniona właściwość RefreshSQL musi zostać zmodyfikowana, gdyż nie odpowiada zapytaniu przechowywanemu we właściwości SelectSQL. Modyfikacja zatem wygląda następująco:

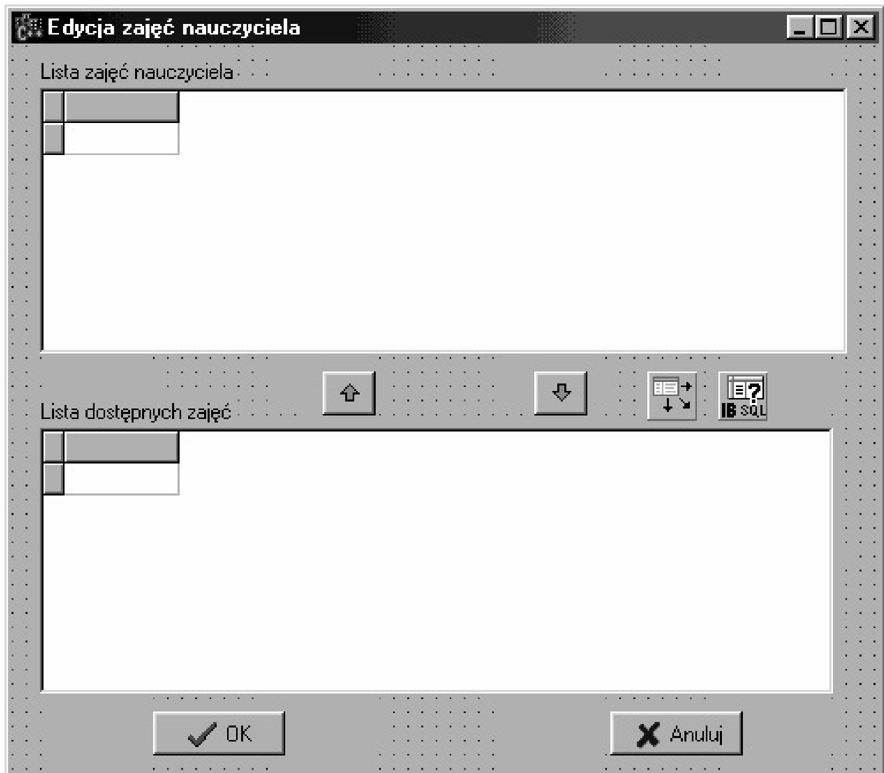
```
SELECT kpn.*, k.Nazwa K_Nazwa, p.Nazwa P_Nazwa
FROM Klasa_Przedmiot_Nauczyciel kpn, Klasa k, Przedmiot p
WHERE kpn.Id_Klasa = k.Id_Klasa and kpn.Id_Przedmiot = p.Id_Przedmiot
and Id_Nauczyciel = :Id_Nauczyciel
and
  kpn.Id_Klasa = :Id_Klasa and
  kpn.Id_Przedmiot = :Id_Przedmiot
```

Poprawność odświeżania zapewnimy, ustawiając właściwość ForcedRefresh na true. Następnie dodajemy w edytorze pól wszystkie pola naszego zapytania. Trzem pierwszym ustawiamy właściwość Visible na false. Dwóm pozostałym natomiast modyfikujemy właściwość Required na false oraz odpowiednio wypełniamy właściwość DisplayLabel.

Wyświetlanie prowadzonych przez danego nauczyciela zajęć zrealizujemy podobnie jak powyższe powiązania (np. *Klasa – Przedmiot*). Dodajemy zatem na formatce FOknoNauczyciel dodatkową zakładkę o nazwie *Zajęcia*. Umieszczamy na niej komponent DBGrid oraz DBNavigator, które łączymy z komponentem IBDSNauczycielZajecia za pośrednictwem komponentu DSNauczycielZajecia.

Przejdziemy teraz do realizacji dodawania i usuwania zajęć w kontekście danego nauczyciela. Zadanie to będzie nieco bardziej skomplikowane niż dodawanie przedmiotu do klasy czy klasy do przedmiotu. Powiązanie nauczyciela z prowadzonymi przez niego

zajęciami wiąże się z odpowiednim ustawieniem dwóch pól w tabeli *Klasa_Przedmiot_Nauczyciel*. Będą to pola: *Id_Klasa* oraz *Id_Przedmiot*. Poza tym pola te nie mogą mieć wartości przypadkowych, lecz muszą odpowiadać wpisom z tabeli *Klasa_Przedmiot*. Dlatego też zrealizowanie tego mechanizmu z wykorzystaniem komponentu `DBLookupComboBox` byłoby mało czytelne dla użytkownika. Oprogramujemy mechanizm dodawania zajęć jako przenoszenie konkretnych wpisów z tabeli *Klasa_Przedmiot* do tabeli *Klasa_Przedmiot_Nauczyciel*, dodając jedynie informację o nauczycielu. Usuwanie będzie polegało wprost na usunięciu rekordu z tabeli *Klasa_Przedmiot_Nauczyciel*. Formatka realizująca to zadanie podana jest na rys. 8.18.



Rys. 8.18. Wygląd formatki do realizacji dodawania i usuwania zajęć w kontekście danego nauczyciela

Dwa najważniejsze elementy naszej formatki do komponenty `DBGrid`. Pierwszy (górny) prezentuje zajęcia danego nauczyciela. Łączymy go zatem z komponentem `DSNauczycielZajecia`. Drugi (dolny) wyświetla wszystkie zajęcia w naszej szkole. Powiązany jest on z komponentami `DSKlasaPrzedmiot` oraz `IBQKlasaPrzedmiot` znajdującymi się na formatce edycyjnej. Najważniejsza właściwość `SQL` komponentu `IBQKlasaPrzedmiot` wypełniona jest następującym zapytaniem:

```
SELECT kp.*, k.Nazwa K_Nazwa, p.Nazwa P_Nazwa
FROM Klasa_Przedmiot kp, Klasa k, Przedmiot p
WHERE kp.Id_Klasa = k.Id_Klasa and kp.Id_Przedmiot = p.Id_Przedmiot
ORDER BY k.Nazwa, p.Nazwa
```

Przypisywanie zajęć nauczycielowi uzyskujemy dzięki przyciskowi „strzałka w górę”. Zdarzenie realizujące to zadanie wygląda następująco:

```
if (!IBQKlasaPrzedmiot->IsEmpty())
{
```

```
  DataSet->Insert();
  DataSet->FieldByName("ID_KLASA")->AsInteger =
IBQKlasaPrzedmiot->FieldByName("ID_KLASA")->AsInteger;
  DataSet->FieldByName("ID_PRZEDMIOT")->AsInteger =
 IBQKlasaPrzedmiot->FieldByName("ID_PRZEDMIOT")->AsInteger;
  DataSet->Post();
  IBQKlasaPrzedmiot->Close();
  IBQKlasaPrzedmiot->Open();
  }
```

Zbiór danych `DataSet` odnosi się do komponentu zbioru danych powiązanego z górnym komponentem `DBGrid`. Przypisanie to dokonywane jest w konstruktorze formatki edycyjnej:

```
__fastcall
TFEdycjaZajecNauczyciela::TFEdycjaZajecNauczyciela(TComponent* Owner)
        : TFEdycja(Owner)
{
  DataSet = DBGrid1->DataSource->DataSet;
}
```

Widzimy, że w powyższym zdarzeniu brakuje przypisania odpowiedniej wartości polu *Id_Nauczyciel*. Czynność ta jest jednak realizowana w zdarzeniu `OnNewRecord` zbioru danych `IBDSNauczycielZajecia`:

```
void __fastcall TDM::IBDSNauczycielZajeciaNewRecord(TDataSet *DataSet)
{
  IBDSNauczycielZajeciaID_NAUCZYCIEL->Value =
IBDSNauczycielID_NAUCZYCIEL->Value;
}
```

Ujmowanie zajęć realizujemy za pomocą przycisku „strzałka w dół”. Zdarzenie realizujące tę czynność wygląda następująco:

```
if (!DataSet->IsEmpty())
  DataSet->Delete();
```

Operacja `Delete()` wykonana na pustym zbiorze danych skutkuje wyrzuceniem wyjątku. Dlatego też wprowadziliśmy zabezpieczenie w postaci testowania pustości zbioru danych. Aktualną zawartość komponentu `IBQKlasaPrzedmiot` uzyskamy dzięki zamykaniu i otwieraniu zbioru w trakcie ukrywania i pokazywania formatki edycyjnej:

```
void __fastcall TFEdycjaZajecNauczyciela::FormHide(TObject *Sender)
{
  IBQKlasaPrzedmiot->Close();
}

void __fastcall TFEdycjaZajecNauczyciela::FormShow(TObject *Sender)
{
  IBQKlasaPrzedmiot->Open();
}
```

Musimy jeszcze napisać zdarzenie `OnClose` formatki edycyjnej, gdyż to odziedziczone z klasy `TFEdycja` działałoby w sposób niepoprawny. Wywołanie metody `Post()` lub `Cancel()` dla zbioru niebędącego w trybie edycyjnym skutkuje wyrzuceniem wyjątku. Po drugie, kliknięcie przycisku *Anuluj* powinno odwołać wszystkie wprowadzone na formatce zmiany. Mechanizmem, który tutaj wykorzystamy, będzie mechanizm transakcji. Mianowicie, jeżeli użytkownik kliknie przycisk *Zatwierdź*, to zatwierdzamy transakcję za pomocą metody `CommitRetaining()`. Natomiast kliknięcie przycisku *Anuluj* powinno skutkować porzuceniem wszystkich wprowadzonych dotychczas (ale jeszcze niezatwierdzonych) zmian, co uzyskamy dzięki wywołaniu metody `RollbackRetaining()`. Zdarzenie `OnClose` wygląda zatem tak:

```
TIBDataSet *IBDataSet = dynamic_cast<TIBDataSet*>(DataSet);
if (ModalResult == mrOk)
      IBDataSet->Transaction->CommitRetaining();
  else
    {
      IBDataSet->Transaction->RollbackRetaining();
      DataSet->Close();
      DataSet->Open();
    };
```

Posiadany przez formatkę wskaźnik `DataSet` klasy `TDataSet` nie zawiera informacji o transakcji, z jaką jest skojarzony nasz komponent. Właściwość `Transaction` pojawia się w komponentach *InterBase* po raz pierwszy w klasie `TIBCustomDataSet`. Jednak bardziej czytelne będzie, jeżeli zrzutujemy ją na klasę pochodną, do której zaliczają się komponenty `IBDataSet`, czyli `TIBDataSet`.

Musimy zdawać sobie sprawę, iż mimo porzucenia zmian poprzednia zawartość zbioru danych nie będzie dostępna, dopóki nie odczytamy ponownie jego zawartości. Czynność tę wykonuje para metod `Close()`/`Open()`. Implementując mechanizm transakcji obsługujący formatkę edycyjną `FEdycjaZajecNauczyciela`, musimy wprowadzić go w sposób jawny także w innych formatkach edycyjnych. Związane jest to z działaniem funkcji `RollbackRetaining()`, która cofa ostatnio wykonane operacje do momentu ich wcześniejszego zatwierdzenia. Spowoduje to więc anulowanie wszystkich operacji wprowadzonych od początku uruchomienia naszej aplikacji. Dlatego też każda operacja edycji powinna po pomyślnym zakończeniu zatwierdzać za pomocą transakcji wprowadzone zmiany (funkcja `CommitRetaining()`).

Ostatnią zmianą, jaką dokonamy, będzie uniemożliwienie przypisania nauczycielowi dwa razy tych samych zajęć, gdyż skutkować to będzie wyrzuceniem wyjątku związanego z naruszeniem unikatowości *klucza głównego* tabeli *Klasa_Przedmiot_Nauczyciel*. Uzyskamy to dzięki odpowiedniej modyfikacji zapytania SQL w zbiorze danych `IBQKlasaPrzedmiot`. Mianowicie, wykluczymy z niego te rekordy (pary *Klasa – Przedmiot*), które zostały już przypisane danemu nauczycielowi, co uniemożliwi ich powtórne dodanie. W tym celu wykorzystamy frazę `EXISTS`, a dokładniej jej zaprzeczenie `NOT EXISTS`:

```
SELECT kp.*, k.Nazwa K_Nazwa, p.Nazwa P_Nazwa
FROM Klasa_Przedmiot kp, Klasa k, Przedmiot p
WHERE kp.Id_Klasa = k.Id_Klasa and kp.Id_Przedmiot = p.Id_Przedmiot
and
NOT EXISTS (SELECT kpn.Id_Nauczyciel
```

```
 FROM Klasa_Przedmiot_Nauczyciel kpn
 WHERE kpn.Id_Klasa = kp.Id_Klasa
 and  kpn.Id_Przedmiot = kp.Id_Przedmiot
 and  kpn.Id_Nauczyciel = :Id_Nauczyciel)
ORDER BY k.Nazwa, p.Nazwa;
```

Widzimy więc, że zawartość naszego zbioru wynikowego zależy od aktualnie przypisanych zajęć, a zatem musi być ona odświeżana po każdorazowej operacji dodania bądź usunięcia wiersza w tabeli *Klasa_Przedmiot_Nauczyciel*. Zdarzenia realizujące te operacje uzupełniamy o następujący kod:

```
IBQKlasaPrzedmiot->Close();
IBQKlasaPrzedmiot->Open();
```

Po drugie parametr *:Id_Nauczyciel* musi zostać przed otwarciem zapytania (jego wykonaniem) zainicjalizowany. Czynność tę zrealizujemy w zdarzeniu `OnShow` następująco:

```
void __fastcall TFEdycjaZajecNauczyciela::FormShow(TObject *Sender)
{
  IBQKlasaPrzedmiot->ParamByName("ID_NAUCZYCIEL")->AsInteger =
    DataSet->DataSource->
DataSet->FieldByName("ID_NAUCZYCIEL")->AsInteger;
  IBQKlasaPrzedmiot->Open();
}
```

Wprowadzone tu modyfikacje rozszerzymy na pozostałe formatki. Mianowicie, operacje edycji powinny kończyć się wywołaniem odpowiednich metod komponentu `IBTransaction`. Druga kwestia dotyczy niedopuszczenia do dodania dwa razy tego samego przedmiotu jednej klasie oraz przypisania dwukrotnie jednej klasy temu samemu przedmiotowi. Pierwsze rozszerzenie, związane z mechanizmem transakcji, dodamy do zdarzenia `OnClose` formatki edycyjnej `FEdycja`, którego kod będzie zatem wyglądał następująco:

```
void __fastcall TFEdycja::FormClose(TObject *Sender, TCloseAction
&Action)
{
TIBDataSet *IBDataSet = dynamic_cast<TIBDataSet*>(DataSet);
  if (ModalResult == mrOk)
  {
    DataSet->Post();
    if (IBDataSet)
      IBDataSet->Transaction->CommitRetaining();
  }
  else
  {
    DataSet->Cancel();
    if (IBDataSet)
      IBDataSet->Transaction->RollbackRetaining();
  }
}
```

Drugie rozszerzenie, związane z niedopuszczeniem dodania dwa razy tego samego przedmiotu jednej klasie, oraz przypisanie dwukrotnie jednej klasy temu samemu przedmiotowi pozwolą nam zrealizować następujące zapytania SQL:

```
Edycja przedmiotu klasy:
SELECT p.Id_Przedmiot, p.Nazwa
FROM Przedmiot p
WHERE NOT EXISTS
(SELECT kp.Id_Przedmiot
FROM Klasa_Przedmiot kp
WHERE kp.Id_Przedmiot = p.Id_Przedmiot
and kp.Id_Klasa = :Id_Klasa)
ORDER BY p.Nazwa;

Edycja klasy przedmiotu:
SELECT k.Id_Klasa, k.Nazwa
FROM Klasa k
WHERE NOT EXISTS
(SELECT kp.Id_Klasa
FROM Klasa_Przedmiot kp
WHERE kp.Id_Klasa = k.Id_Klasa
and kp.Id_Przedmiot = :Id_Przedmiot)
ORDER BY k.Nazwa;
```

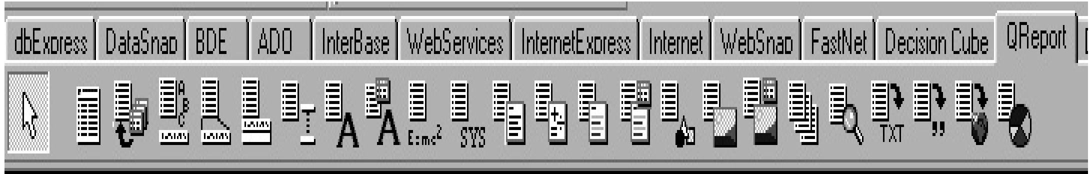
Poza tym w zdarzeniach `OnShow` powinny pojawić się następujące wpisy, realizujące inicjalizację parametrów zapytań SQL:

```
Edycja klasy przedmiotu:
  IBQKlasa->ParamByName("ID_PRZEDMIOT")->AsInteger =
    DataSet->FieldByName("ID_PRZEDMIOT")->AsInteger;

Edycja przedmiotu klasy:
  IBQPrzedmiot->ParamByName("ID_KLASA")->AsInteger =
    DataSet->FieldByName("ID_KLASA")->AsInteger;
```

8.1.7. Tworzenie raportów

Doszliśmy do momentu, w którym nasza aplikacja spełnia wszystkie najważniejsze założenia. Pozwala ona zatem na dodawanie, edycje i usuwanie wierszy wszystkich tabel oraz na ich sprawne przeglądanie. Jednakże każda poważna aplikacja bazodanowa powinna umożliwiać sporządzanie zestawień opracowujących w przejrzystej formie dane zgromadzone w bazie danych. Tego typu opracowania noszą w terminologii bazodanowej nazwę raportów. Narzędziem umożliwiającym tworzenie raportów są komponenty zlokalizowane na zakładce *QReports* (rys. 8.19).



Rys. 8.19. Pasek narzędzi zakładki *QReports*

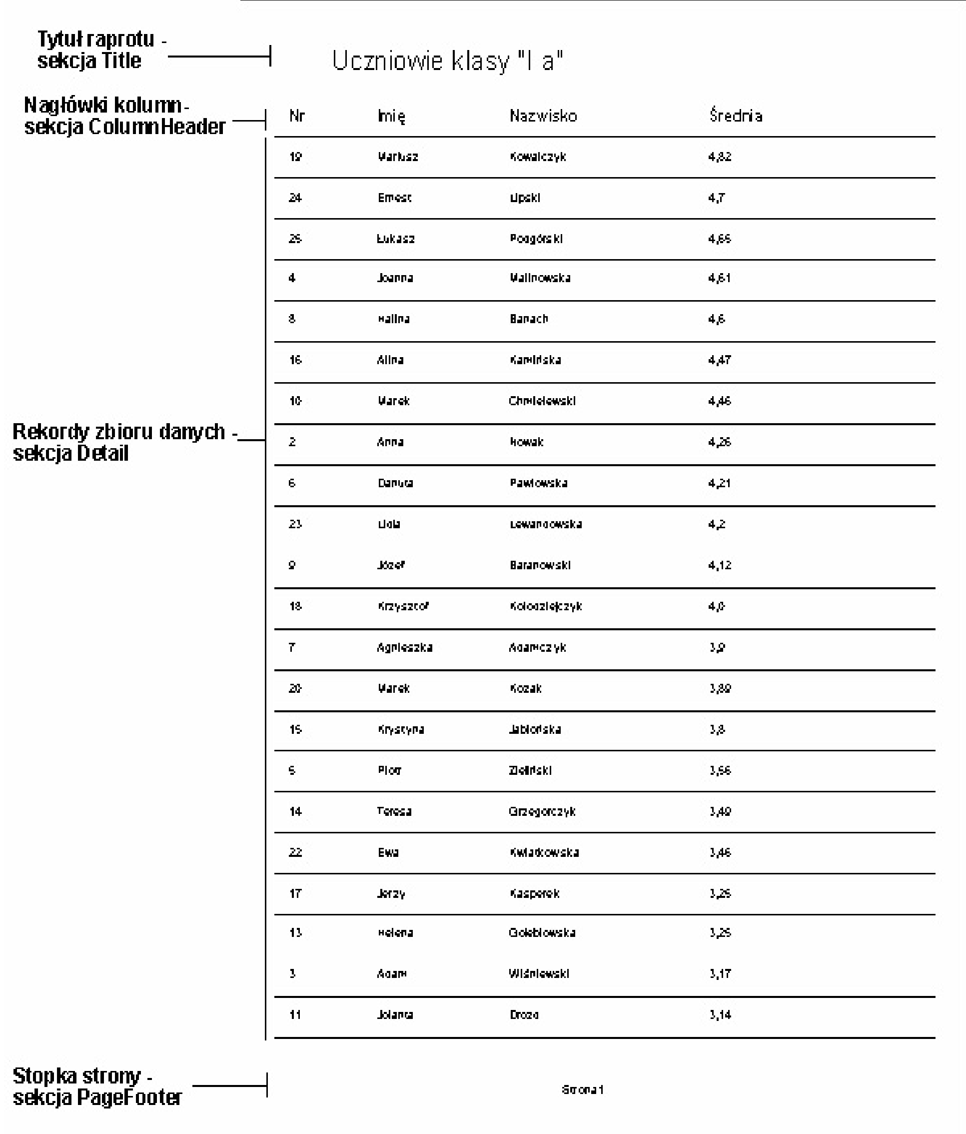
Analizując powyższą zakładkę, widzimy, że znajdują się na niej aż 23 komponenty służące do budowania raportów. Najważniejszy z nich to pierwszy komponent `QuickRep` reprezentujący wydruk. Pozostałe zaś to elementy mogące się na niej znaleźć, spełniając rozmaite funkcje. Pierwszy raport będzie prostym zestawieniem, przedstawiającym wszystkich uczniów danej klasy posortowanych ze względu na średnią ocen.

Komponent `QuickRep` jak każdy inny komponent wizualny musi rezydować na formatce. Jednakże formatka ta nigdy nie będzie pokazywana użytkownikowi, a jedynie będzie pełniła rolę pojemnika na komponent `QuickRep`. Dostęp do wizualnej strony raportu uzyskamy dzięki metodzie `Preview()`. Dodajemy zatem do naszego projektu nową formatkę, którą nazywamy `FRaportUczniowie`, moduł zaś zapisujemy w pliku pod nazwą *URaportUczniowie.cpp*.

Formatka nasza będzie formatką tworzoną dynamicznie, co odpowiednio odnotowujemy w opcjach projektu (menu *Project/Options*). Następnie umieszczamy na formatce komponent `QuickRep` z zakładki *QReport*. Posiada on właściwość `Bands`, która reprezentuje poszczególne sekcje raportu. Sekcje są horyzontalnie ułożonymi paskami, związanymi z ich pozycją na stronie. Mogą być one uaktywniane za pomocą właściwości zbiorowej `Bands`, jak również poprzez dodanie komponentu `QRBand`. Oto opcje właściwości `Bands`:

- `HasColumnHeader` – sekcja *ColumnHeader* drukowana jest w górnej części każdej kolumny raportu. Wykorzystywana jest zatem jako element wyświetlający nagłówki pól zbioru danych;
- `HasDetail` – sekcja *Detail* drukowana jest raz dla każdego rekordu ze zbioru danych skojarzonego za pomocą właściwości `DataSet` komponentu `QuickRep`. Jest to najważniejsza sekcja raportu, na której umieszczamy komponenty drukowalne kontrolki danych, takie jak `QRDBText`;
- `HasPageFooter` – sekcja *PageFooter* drukowana jest na stopce każdej strony. Drukowanie jej na ostatniej stronie uzależnione jest od właściwości `Options->LastPageFooter`. Ustawienie jej wartości na `false` sprawi, że sekcja ta nie będzie drukowana na ostatniej stronie raportu;
- `HasPageHeader` – sekcja *PageHeader* drukowana jest w nagłówku każdej strony. Drukowanie jej na pierwszej stronie uzależnione jest od właściwości `Options->FirstPageHeader`. Ustawienie jej wartości na `false` sprawi, że sekcja ta nie będzie drukowana na pierwszej stronie raportu;
- `HasSummary` – sekcja *Summary* podsumowuje wszystkie rekordy wydrukowane w sekcji *Detail*. Przykładowo może ona zawierać liczbę rekordów zbioru danych czy też zsumowaną wartość pola numerycznego;
- `HasTitle` – sekcja *Title* jest pierwszą sekcją drukowaną w raporcie, określającą jego tytuł.

Obrazowo sekcje raportu przedstawiają się na rys. 8.20.



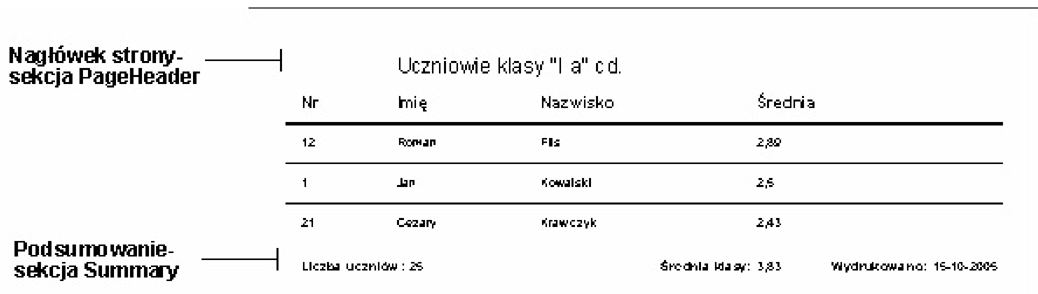
Uczniowie klasy "I a"

Nr	Imię	Nazwisko	Średnia
19	Mariusz	Kowalczyk	4,82
24	Ernest	Lipski	4,7
25	Łukasz	Podgórski	4,66
4	Joanna	Malinowska	4,61
8	Halina	Banach	4,6
16	Alina	Kamińska	4,47
10	Marek	Chmielewski	4,46
2	Anna	Nowak	4,26
6	Danuta	Pawłowska	4,21
23	Lidia	Lewandowska	4,2
9	Józef	Baranowski	4,12
18	Krzysztof	Kołodziejczyk	4,0
7	Agnieszka	Adamczyk	3,9
20	Marek	Kozak	3,89
15	Krystyna	Jabłońska	3,8
5	Piotr	Zieliński	3,56
14	Teresa	Grzegorczyk	3,49
22	Ewa	Kwiatkowska	3,46
17	Jerzy	Kasperek	3,26
13	Helena	Gołębiowska	3,25
3	Adam	Wiśniewski	3,17
11	Jolanta	Drozd	3,14

Strona 1

Rys. 8.20. Raport *Uczniowie klasy 1A*

Druga strona raportu zawiera te sekcje, które nie pojawiły się na poprzedniej stronie. Chodzi mianowicie o sekcję *PageHeader*, która poprzez ustawioną właściwość `FirstPageHeader` na wartość `false` nie pojawiła się pierwszej stronie raportu, oraz o sekcję *Summary*, która pojawia się po wyświetleniu wszystkich rekordów sekcji *Detail* (rys. 8.21).



Uczniowie klasy "I a" cd.

Nr	Imię	Nazwisko	Średnia
12	Roman	Fils	2,89
1	Jan	Kowalski	2,6
21	Cezary	Krawczyk	2,43

Liczba uczniów : 25 Średnia klasy: 3,83 Wydrukowano: 15-10-2005

Rys. 8.21. Raport (sekcja *PageHeader*)

Po położeniu komponentu `QuickRep` na formatce ustawiamy wszystkie opcje właściwości `Bands` na `true`. Następnie układamy na formatce dwa komponenty `IBQuery`. Pierwszy z nich, którego zadaniem będzie zwrócenie listy uczniów danej klasy posortowanych ze względu na średnią ocen, nazywamy `IBQUczniowie`. Treść tego zapytania, a tym samym i właściwości `SQL` jest następująca:

```
SELECT Id_Uczen, Imię, Nazwisko, Srednia
FROM Uczen
WHERE Id_Klasa = :Id_Klasa
ORDER BY Srednia DESC;
```

Jak widzimy, zapytanie to jest zapytaniem parametrycznym, co oznacza, że przed wyświetleniem raportu musimy odpowiednio ustawić wartość parametru *Id_Klasa*. Komponent `QuickRep` łączymy z komponentem `IBQUczniowie` za pomocą właściwości `DataSet`. Rolą drugiego komponentu, `IBQuery`, będzie zwrócenie dla danej wartości parametru nazwy klasy. Komponent ten nazywamy `IBQKlasa`, a wartość jego właściwości `SQL` wypełniamy następującym zapytaniem:

```
SELECT Nazwa
FROM Klasa
WHERE Id_Klasa = :Id_Klasa;
```

W tym momencie przechodzimy do wypełniania poszczególnych sekcji odpowiednimi komponentami drukowalnymi. Pierwsza sekcja, *PageHeader*, zawierać będzie komponent `QRExpr`, którego rolą jest wyświetlanie wartości tekstowej, będącej odpowiednio sformułowanym wyrażeniem. Przechowywane jest ono we właściwości `Expression` i jest kombinacją stałych tekstowych, pól zbioru danych, funkcji, zmiennych oraz operatorów. Wprowadzenie wyrażeń ułatwi nam edytor właściwości `Expression` (patrz rys. 8.22).

Przycisk *Database field* umożliwia wybór pola danego zbioru danych. Wyświetlenie pól jest możliwe tylko wtedy, kiedy zbiór danych posiada komponenty pól. Co oznacza, że w przypadku dynamicznych komponentów pól zbiór danych musi być otwarty. Przycisk *Function* wyświetla listę funkcji, jakie mogą być użyte w wyrażeniu. Kompletne ich zestawienie zawiera tab. 8.1.

Rys. 8.22. Okno edytora właściwości

Tabela 8.1. Opis funkcji do używania w wyrażeniach w oknie na rys. 8.22

Funkcja	*Składnia*	*Opis*
IF	IF(<Exp>,<X>,<Y>)	Zwraca wartość <X> lub <Y> w zależności od prawdziwości wyrażenia <Exp>
STR	STR(<X>)	Konwertuje wartość numeryczną <X> na łańcuch znaków
UPPER	UPPER(<X>)	Konwertuje wszystkie litery łańcucha <X> na duże
LOWER	LOWER(<X>)	Konwertuje wszystkie litery łańcucha <X> na małe
PRETTY	PRETTY(<X>)	Zmienia pierwszą literę łańcucha <X> na dużą, pozostałe zaś na małe
TIME	TIME	Zwraca bieżący czas w postaci łańcucha znaków
DATE	DATE	Zwraca bieżącą datę w postaci łańcucha znaków
COPY	COPY(<X>,<St>,<Len>)	Zwraca podłańcuch o długości <Len> łańcucha przekazanego w parametrze <X>, zaczynając od pozycji <St>
SUM	SUM(<X>)	Zwraca sumę wyrażenia <X>
COUNT	COUNT	Zwraca liczbę iteracji
MAX	MAX(<X>)	Zwraca największą wartość wyrażenia <X>
MIN	MIN(<X>)	Zwraca najmniejszą wartość wyrażenia <X>
AVG	AVG(<X>)	Zwraca średnią wartość wyrażenia <X>
TRUE	TRUE	Wartość logiczna prawda
FALSE	FALSE	Wartość logiczna fałsz
INT	INT(<X>)	Zwraca część całkowitą <X>
FRAC	FRAC(<X>)	Zwraca część ułamkową <X>
SQRT	SQRT(<X>)	Zwraca pierwiastek z <X>
DIV	DIV(<X>,<Y>)	Zwraca wynik dzielenia całkowitego <X> przez <Y>
TYPEOF	TYPEOF(<Exp>)	Zwraca typ wyrażenia
FORMATNUMERIC	FORMATNUMERIC(<F>,<N>)	Formatuje liczbę <N> według formatu <F>, który jest tożsamy z formatowaniem użytym w funkcji FormatFloat()

Poza funkcjami mamy też do dyspozycji kilka zmiennych, które przedstawia tab. 8.2.

Tabela 8.2. Opis zmiennych do używania w wyrażeniach w oknie na rys. 8.22

Zmienna	*Opis*
PAGENUMBER	Przechowuje numer strony
COLUMNNUMBER	Przechowuje numer kolumny
RAPORTTITLE	Przechowuje tytuł raportu
APPSTARTTIME	Przechowuje czas startu aplikacji
APPSTARTDATE	Przechowuje datę startu aplikacji
APPNAME	Przechowuje nazwę aplikacji

Do tego mamy cały szereg operatorów, które możemy podzielić na następujące kategorie:

- arytmetyczne – +, −, *, / ;
- logiczne – NOT, AND, OR ;
- relacyjne – =, <>, <, >, <=, >= ;
- konkatenacji – + .

W naszym przykładzie wykorzystamy stałe dosłowne (które ujmujemy w pojedyncze apostrofy), pole zbioru danych oraz operator konkatenacji (+). Zatem nasze wyrażenie będzie postaci:

```
'Uczniowie klasy '  + '"' + IBQKlasa.NAZWA + '"' + + ' cd.'
```

Za pomocą właściwości Font formatujemy w odpowiedni sposób nasz tekst. Nagłówek strony PageHeader, co oczywiste, nie powinien być wyświetlony na pierwszej stronie raportu. Uzyskamy to, ustawiając właściwość Options->FirstPageHeader komponentu QuickRep na false.

Druga sekcja *Title* zawiera, podobnie jak *PageHeader*, tylko komponent QRExpr, którego właściwość różni się nieznacznie, a mianowicie, nie zawiera stałej tekstowej ' cd.'.

Trzecia sekcja *ColumnHeader* powinna zawierać nagłówki kolumn. W tym celu wykorzystamy komponent QRLabel, który wykorzystywany jest do drukowania tekstów statycznych (tzn. niepochodzących ze zbioru danych). Wyświetlany tekst wpisujemy we właściwości Caption. W naszym przykładzie umieszczamy cztery komponenty QRLabel, którym odpowiednio ustawiamy właściwość Caption, następującymi napisami: *Nr*, *Imię*, *Nazwisko*, *Średnia*.

Kolejna sekcja *Detail* stanowi centralne miejsce naszego raportu i jest drukowana raz dla każdego rekordu zbioru danych. Do wyświetlenia zawartości poszczególnych pól wykorzystamy komponenty QRDBText. Umieszczamy zatem na komponencie QuickRep w jego sekcji *Detail* 4 sztuki. Wszystkim ustawiamy właściwość DataSet na IBQUczniowie. Natomiast właściwość DataField przyjmie odpowiednio wartości: *Id_Uczen*, *Imię*, *Nazwisko*, *Średnia*.

Następna sekcja *Summary* zawierać będzie elementy podsumowujące sekcję *Detail*, jak też cały raport. Będą to zatem informacje o liczbie wierszy, średniej ocen całej klasy oraz dacie sporządzenia raportu. Pierwszą wielkość uzyskamy dzięki komponentowi QRSysData, a dokładniej jego właściwości Data ustawionej na qrsDetailCount. Oprócz tego może ona przyjąć 7 różnych wartości, które opisuje tab. 8.3.

Tabela 8.3. Lista wartości komponentu QRSysData

Wartość	*Opis*
qrsDate	Bieżąca data
qrsDateTime	Bieżąca data i czas
qrsDetailCount	Liczba rekordów sekcji *Detail*
qrsDetailNo	Bieżący numer rekordu sekcji *Detail*
qrsPageNumber	Numer strony raportu
qrsReportTitle	Tytuł raportu
qrsTime	Bieżący czas

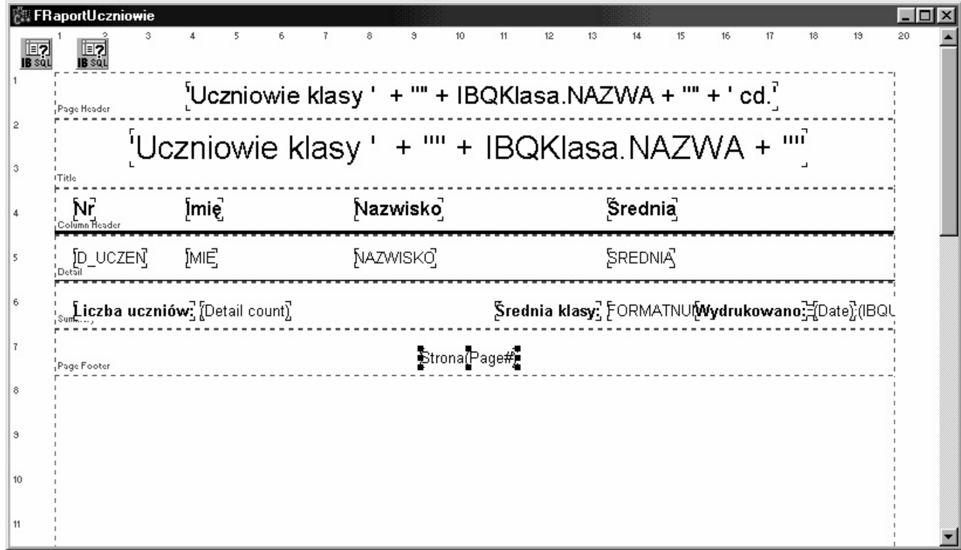
Drugi element podsumowujący sformułujemy na podstawie wyrażenia `Expression` komponentu `QRExpr` w następującej postaci:

```
FORMATNUMERIC('#.0#',AVERAGE(IBQUczniowie.SREDNIA))
```

Trzeci element to komponent `QRSysData` z ustawioną właściwością `Date` na wartość `qrsDate`.

Ostatnia sekcja *PageFooter* zawierać będzie komponent `QRSysData` z ustawioną właściwością `Date` na wartość `qrsPageNumber`. Oznacza to, iż w stopce strony pojawi się informacja o numerze strony raportu.

Gotowy raport będzie wyglądał jak na rys. 8.23.



Rys. 8.23. Formatka z gotowym raportem

Na tym etapie możemy przejść już do implementowania procedury wyświetlania zawartości raportu. W tym celu wykorzystamy metodę `Preview()` komponentu `QuickRep`. Formatka, na której znajduje się nasz raport, będzie tworzona dynamicznie, a zatem prawidłowy kod przedstawia się następująco:

```
FRaportUczniowie = new TFRaportUczniowie(this);
try
{
  FRaportUczniowie->QuickRep->Preview();
}
__finally
{
  delete FRaportUczniowie;
}
```

Powyższy kod umieszczamy w obsłudze zdarzenia `OnExecute` akcji `ARaportUczniowie`, z którą skojarzona jest odpowiednia pozycja w menu *Raporty*.

Zapytanie zwracające zawartość naszego raportu jest zapytaniem parametrycznym, które zwraca listę uczniów danej klasy. Potrzebujemy zatem przed wyświetleniem raportu odpowiednio ustawić wartość parametru *Id_Klasa*. Odpowiedni kod będzie wyglądał tak:

```
void __fastcall TFRaportUczniowie::FormCreate(TObject *Sender)
{
  int ID_KLASA = 0;
  FWyborKlasy = new TFWyborKlasy(this);
  try
  {
    FWyborKlasy->ShowModal();
    ID_KLASA = FWyborKlasy->PobierzId();
  }
  __finally
  {
    delete FWyborKlasy;
  }
  IBQUczniowie->ParamByName("ID_KLASA")->AsInteger = ID_KLASA;
  IBQKlasa->ParamByName("ID_KLASA")->AsInteger = ID_KLASA;
  IBQUczniowie->Open();
  IBQKlasa->Open();
}
```

Wybór odpowiedniej wartości parametru *Id_Klasa* realizujemy na podstawie formatki `FWyborKlasy` zawierającej komponent `ComboBox` z listą wszystkich klas oraz przycisk *OK* (rys. 8.24).

Rys. 8.24. Formatka `FWyborKlasy` pozwalająca na wybór klasy

Zawartość komponentu `ComboBox` stanowić będą nazwy klas pobrane z tabeli *Klasa*. Zadanie to zrealizujemy na podstawie komponentu `IBSQL` klasy `TIBSQL`. Służy on do wykonywania zapytań SQL z minimalnym narzutem. Oznacza to, iż nie posiada standardowego interfejsu dostępu dla kontrolek typu *data-aware* oraz że jest zbiorem jednokierunkowym. W naszym przypadku będzie wykorzystany do zwrócenia listy klas, która zostanie wpisana jako kolejne pozycje komponentu `ComboBox`. Zadanie to zrealizujemy w konstruktorze formatki:

```
__fastcall TFWyborKlasy::TFWyborKlasy(TComponent* Owner)
        : TForm(Owner)
{
  TIBSQL *IBSQL = new TIBSQL(0);
  try
  {
```

```
    IBSQL->Database = DM->IBDatabase;
    IBSQL->SQL->Add("select ID_KLASA, NAZWA from KLASA order by
NAZWA");
    IBSQL->ExecQuery();
    while (!IBSQL->Eof)
    {
      CBKlasy->Items->AddObject(IBSQL->Fields[1]->AsString,
(TObject*)IBSQL->Fields[0]->AsInteger);
      IBSQL->Next();
    }
  }
  __finally
  {
    delete IBSQL;
  }
  CBKlasy->ItemIndex = 0;
}
```

Komponent `IBSQL` posiada właściwość `Database`, którą ustawiamy na wartość `DM->IBDatabase`. Następnie wypełniamy właściwość `SQL` typu `TStrings` odpowiednim zapytaniem. Metoda `ExecQuery()` wykonuje treść zapytania znajdującego się we właściwości `SQL`. Przejrzenie wyników uzyskujemy dzięki metodzie `Next()`. Natomiast osiągniecie końca zbioru testujemy za pomocą właściwości `Eof`. Dostęp do pól bieżącego rekordu możemy uzyskać bądź za pomocą właściwości `Fields`, bądź wykorzystując funkcje `FieldByName()`. W naszym przykładzie wykorzystaliśmy właściwość `Fields`.

Ostatnim ważnym zagadnieniem, na które musimy zwrócić uwagę, jest wypełnienie w odpowiedni sposób komponentu `ComboBox`. Wartości będące odpowiednimi pozycjami na liście Items muszą prezentować użytkownikowi możliwie najlepszą informację o wybieranym obiekcie. Zatem w naszym przypadku będzie to nazwa klasy (pole *Nazwa*). Z drugiej strony wybór ten musi pociągać za sobą odpowiednią korelację z kluczem głównym *Id_Klasa*, który będzie wartością dla parametru *Id_Klasa*. W tym celu wykorzystamy właściwość `Objects`, która pozwala powiązać wyświetlane elementy z obiektami, a dokładniej ze wskaźnikami typu `TObject*`. Oczywiście, moglibyśmy stworzyć obiekt klasy pochodnej po `TObject` i zawierający jedno pole typu `int` reprezentujące wartość *klucza głównego*. Jednakże z uwagi na fakt, iż wskaźniki są 4-bajtowymi liczbami całkowitymi, możliwa jest konwersja na typ `int`. Dlatego też wykorzystamy ten drugi prostszy sposób.

Odwrotne rzutowanie musimy wykonać przy odczytywaniu odpowiedniej wartości z właściwości `Objects`, co widzimy poniżej w metodzie `PobierzId()`:

```
__fastcall TFWyborKlasy::PobierzId()
{
  if (CBKlasy->ItemIndex < 0) return 0;
  return (int)CBKlasy->Items->Objects[CBKlasy->ItemIndex];
}
```

Drugi raport, który zaprezentujemy, będzie odzwierciedlał relację *Master/Detail* (*Nadrzędny/Podrzędny*). W roli *tabeli nadrzędnej* wystąpi tabela *Klasa*, natomiast *tabelę pod-*

rzędną będzie tworzyło złączenie tabel *Przedmiot* i *Klasa_Przedmiot*. W tym celu wykorzystamy dwa komponenty `IBQuery` oraz jeden `DataSource`. Pierwszy komponent `IBQuery` o nazwie `IBQKlasy` będzie pełnił rolę *tabeli nadrzędnej*, drugi zaś o nazwie `IBQPrzedmioty` *tabeli podrzędnej*. Zawartość *tabeli nadrzędnej* będzie pobierana jako wynik następującego zapytania umieszczonego we właściwości `SQL`:

```
SELECT Id_Klasa, Nazwa
FROM Klasa
ORDER BY Nazwa;
```

Drugie zapytanie wykorzystywane w komponencie `IBQPrzedmioty` będzie zapytaniem parametrycznym dokonującym złączenia tabel:

```
SELECT p.Nazwa
FROM Przedmiot p, Klasa_Przedmiot kp
WHERE p.Id_Przedmiot = kp.Id_Przedmiot and
kp.Id_Klasa = :Id_Klasa
ORDER BY Nazwa;
```

Do prawidłowego funkcjonowania naszego modelu należy we właściwości `DataSource` komponentu `IBQPrzedmioty` ustawić komponent `DSKlasy` (powiązany z komponentem `IBQKlasy`).

Tworzony raport będzie zawierał 4 sekcje: *Title*, *Detail*, *Sub Detail*, *Summary*. Sekcja *Sub Detail* służy prezentacji wierszy *tabeli podrzędnej*. W tym celu ustawiamy jej właściwość `DataSet` na `IBQPrzedmioty`. Oprócz tego sekcja ta zawierać będzie komponent `QRDBText` powiązany z polem *Nazwa* tabeli *Przedmiot*. Drugi komponent `IBQKlasy` będzie wartością właściwości `DataSet` komponentu `QucikRep`.

Sekcja *Title* zawierać będzie komponent `QRLabel` z właściwością `Caption` = *Zestawienie przedmiotów wszystkich klas*. W sekcji *Detail* znajdą się dwa komponenty `QRLabel`. Pierwszy opisywał będzie klasę, drugi zaś przedmioty. Oprócz tego pojawi się komponent `QRDBText` połączony z polem *Nazwa* tabeli *Klasa*. Ostatnia sekcja *Summary* będzie prezentowała podsumowanie raportu ze względu na liczbę klas oraz na datę jego sporządzenia.

Oba zadania zrealizujemy, wykorzystując komponent `QRSysData`. W pierwszym właściwość *Data* przyjmie wartość `qrsDetailCount`, w drugim zaś `qrsDate`. Gotowy raport pokazany jest na rys. 8.25.

Title Zestawienie przedmiotów wszystkich klas
Klasa NAZWA
Detail Przedmioty:
Sub Detail NAZWA
Summary Ilość klas: 12 Raport sporządzono: 03-11-2005

Rys. 8.25. Forma raportu *Zestawienie przedmiotów wszystkich klas*

8.2. Aplikacja bazodanowa wypożyczalni filmów 'Matrix'

Baza danych *Wypożyczalni filmów* (niech nosi nazwę *Matrix*) została dość szczegółowo opisana wcześniej: m.in. w podrozdz. 3.1.2 (Prz. 3.2 – 3.4) został opisany projekt BD, w Prz. 3.17, 3.21, 3.23, 3.25 został przeanalizowany proces normalizacji niektórych tabel, na rys. 3.6 został przedstawiony schemat całej bazy.

Aplikacja do obsługi bazy została napisana w języku C++ z wykorzystaniem narzędzia C++ Builder 6 w wersji *Professional*. Wersja ta ma szereg gotowych komponentów, które w znacznym stopniu ułatwiają pracę z bazami *InterBase* i *FireBird*, o czym była mowa w podrozdz. 8.1.

W tej części skupimy się raczej na cechach graficznych aplikacji, jak również na niektórych ważnych aspektach dotyczących zarządzania bazą i jej obsługi.

Na rys. 8.26 przedstawiony jest rzut ekranowy programu Borland C++ Builder 6 *Professional* z elementami analizowanej BD.

Rys. 8.26. Zrzut ekranowy programu Borland C++ Builder 6 *Professional*

Na rys. 8.26 widoczny jest program Borland C++ Builder 6 z odsłoniętą zakładką *InterBase*, na której znajdują się komponenty wspomniane wcześniej. Widoczny jest również otwarty projekt *Wypożyczalnia* i jedna z jego formatek *FPracownicy*, która zostanie omówiona później.

8.2.1. Formatka tytułowa

Formatka główna (tytułowa) aplikacji obsługującej bazę danych wypożyczalni filmów „Matrix” ma wygląd przedstawiony na rys. 8.27.

Na formatce tej mamy następujące przyciski:

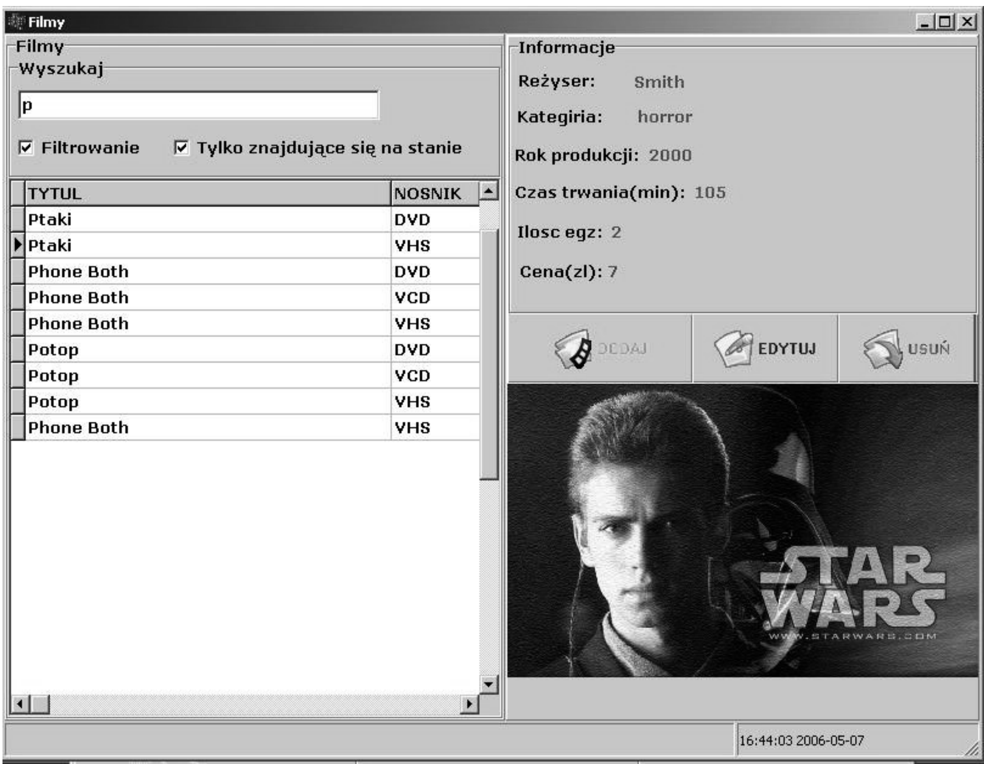
- *FILMY* – dzięki niemu uruchamiamy formatkę pozwalającą na zarządzanie filmami wypożyczalni;
- *KLIENCI* – dzięki niemu uruchamiamy formatkę zarządzającą klientami wypożyczalni;
- *PRACOWNICY* – uruchamia formatkę zarządzającą pracownikami wypożyczalni;
- *WYPOŻYCZENIA* – uruchamia formatkę pozwalającą sprawdzić wypożyczenia poszczególnych klientów, a także dokonać nowych wypożyczeń i zwrotów filmów;
- *RAPORTY* – uruchamia formatkę zawierającą spis dostępnych zestawień;
- *ZAMKNIJ* – kończący pracę z programem.

Rys. 8.27. Zrzut ekranowy formatki tytułowej programu

8.2.2. Formatka *Filmy*

Formatka pozwalająca na zarządzanie filmami oferowanymi przez wypożyczalnię ma wygląd pokazany na rys. 8.28.

Na formatce tej widoczne są tytuły aktualnie oferowanych przez wypożyczalnię filmów wraz z ich nośnikiem. Możliwe jest wyszukanie interesującego nas filmu lub wyfiltrowanie grupy filmów po nazwie, a także wyświetlenie wszystkich filmów lub tylko tych, które znajdują się na stanie (a więc występuje w wypożyczalni przynajmniej jeden egzemplarz filmu). Po prawej stronie dla aktualnie wskazywanego filmu wyświetlane są informacje o reżyserze, kategorii, roku produkcji, czasie trwania, liczbie egzemplarzy i cenie wypożyczenia za dobę. Przycisk *DODAJ* pozwala dodać do oferty nowy film, *EDYTUJ* modyfikować informacje o filmie, a *USUŃ* zdjąć ze stanu dany film (przypisanie zerowej liczby egzemplarzy filmu, dzięki czemu nie będzie on już widoczny na liście, jeżeli będziemy chcieli wyświetlić tylko filmy znajdujące się na stanie).



Rys. 8.28. Zrzut ekranowy formatki *Filmy*

Przykładowo: po wciśnięciu przycisku *USUŃ* wykonuje się następujący kod programu:

```
ModulDanych->IBTFilmy->Edit();

if (MessageBox(NULL, "Czy na pewno chcesz usunąć daną kasetę z
   wypożyczalni?", "Usuwanie", MB_YESNO+ MB_ICONQUESTION +
   MB_DEFBUTTON2 ) == IDYES) {

          ModulDanych->IBTFilmy->FieldByName("ILOSC_SZTUK")->AsInteger =
   0;

          ModulDanych->IBTFilmy->CheckBrowseMode();

        }

        else

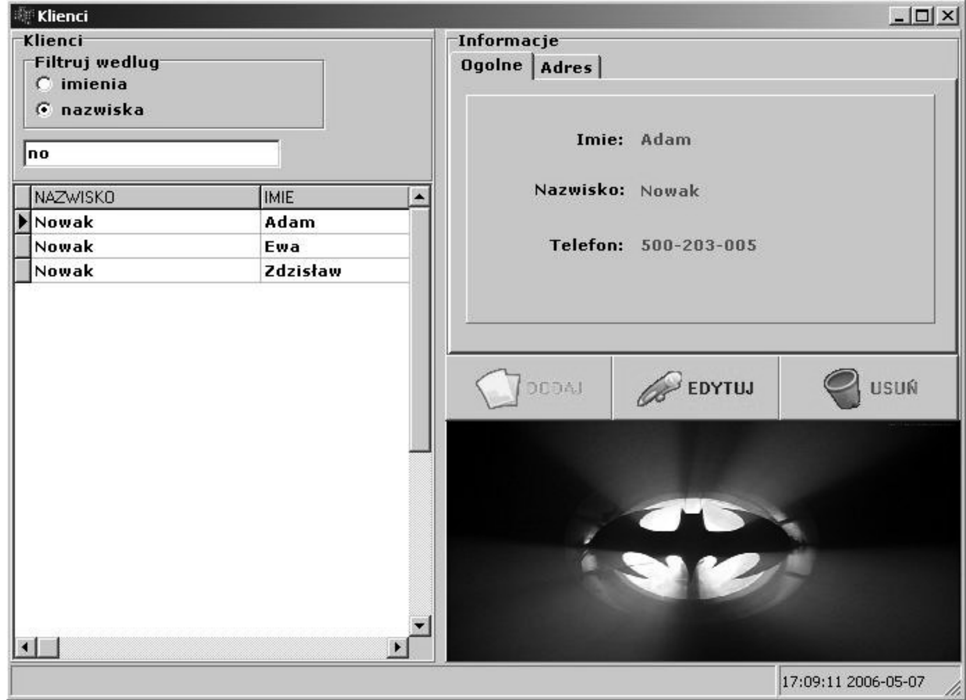
ModulDanych->IBTFilmy->Cancel();
```

Użytkownikowi wyświetlany jest odpowiedni komunikat z potwierdzeniem usunięcia, jeżeli wyrazi on zgodę, polu *ILOSC_SZTUK* przypisywana jest wartość 0, w przeciwnym razie operacja jest anulowana.

8.2.3. Formatka *Klienci*

Formatka służąca do zarządzania klientami wypożyczalni ma następujący wygląd (rys. 8.29).

Na formatce mamy pokazaną listę klientów z możliwością jej filtrowania po imieniu lub nazwisku. Z prawej strony znajdują się dwie zakładki z informacjami ogólnymi i adresowymi o kliencie. Przycisk *DODAJ* służy do zapisania do bazy nowego klienta, przycisk *EDYTUJ* do modyfikowania informacji o kliencie, natomiast *USUŃ* do usunięcia klienta z wypożyczalni (poprzez przypisanie polu *Status* tabeli *Klienci* wartości ‘*N*’ – nieaktywny).



Rys. 8.29. Zrzut ekranowy formatki *Klienci*

Po wciśnięciu przycisku *DODAJ* pojawia się nowa formatka służąca do wprowadzenia informacji o nowym kliencie, a po wciśnięciu przycisku *OK* wykonywany jest następujący kod:

```
ModulDanych->IBDSAdresyKlientow->FieldByName("KOD_POCZTOWY")->AsString
= MEKod>Text;

        if (ModulDanych->IBDSAdresyKlientow->State == dsInsert){

               ModulDanych->IBQTemp->SQL->Text="select
gen_id(GEN_ADRESY_ID,1) ID from RDB$DATABASE";

              ModulDanych->IBQTemp->Active=true;

              ModulDanych->IBDSAdresyKlientow-
>FieldByName("ID_ADRESU")->AsInteger=

ModulDanych>IBQTemp->FieldByName("ID")->AsInteger;

              ModulDanych->IBTKlienci->FieldByName("ADRES")->AsInteger
=

 ModulDanych->IBQTemp->FieldByName("ID")->AsInteger;

              ModulDanych->IBTKlienci->FieldByName("STATUS")->AsString
= "A";

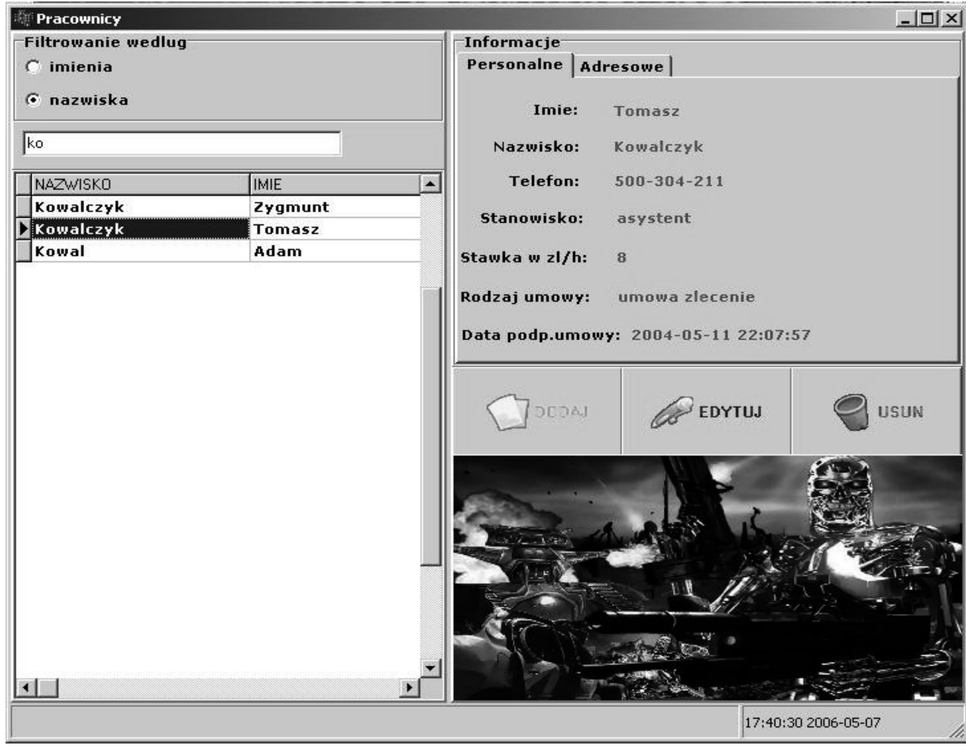
        }
```

```
ModulDanych->IBDSAdresyKlientow->CheckBrowseMode();
ModulDanych->IBTKlienci->CheckBrowseMode();
ModalResult = mrOk;
```

Nowemu klientowi przypisywany jest status '*A*' – aktywny, oprócz tego jest dodawany do bazy nowy adres dla tego klienta, którego identyfikator jest pobierany z odpowiedniego generatora.

8.2.4. Formatka *Pracownicy*

Formatka służąca do zarządzania pracownikami (rys. 8.30) ma strukturę podobną do formatki *Klienci*. Mamy tutaj po lewej stronie listę pracowników z możliwością filtrowania jej po Imieniu i Nazwisku. Z prawej strony podane są na dwóch zakładkach informacje personalne i adresowe pracowników. Przyciski *DODAJ, EDYTUJ* i USUŃ mają podobną funkcjonalność jak w przypadku formatki *Klientów* – pozwalają na dodanie, usunięcie i edycję informacji o wybranym pracowniku. Usunięcie pracownika, tak jak w przypadku klientów, nie polega na całkowitym usunięciu rekordu odpowiadającego pracownikowi z bazy, ale na przypisaniu polu *Status* wartości '*N'* – nieaktywny.



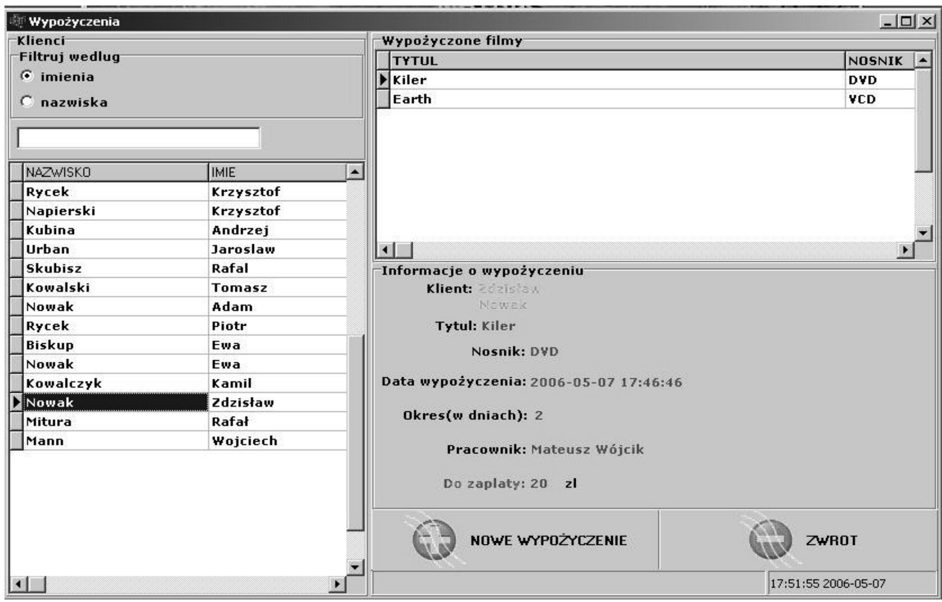
Rys. 8.30. Zrzut ekranowy formatki *Pracownicy*

8.2.5. Formatka *Wypożyczenia*

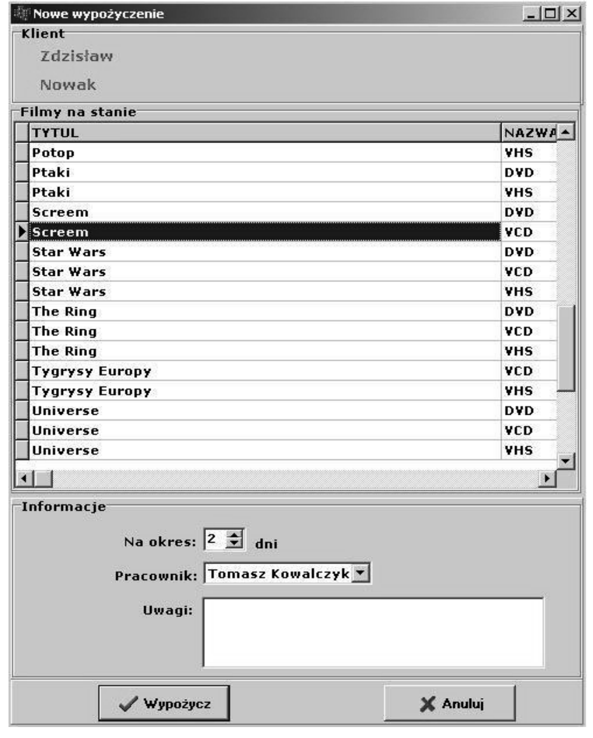
Do obsługi wypożyczeń filmów przez klientów służy formatka *Wypożyczenia* (rys. 8.31).

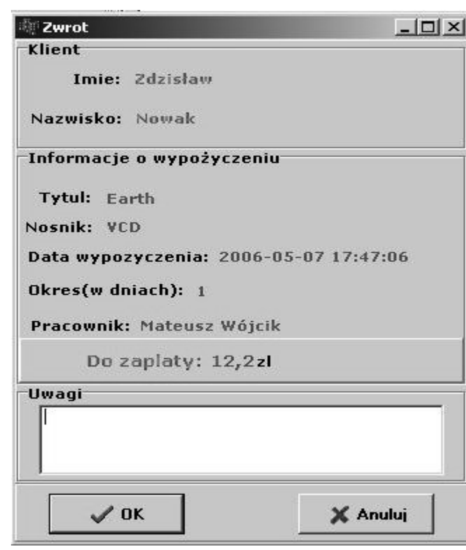
Po lewej stronie mamy listę klientów z możliwością jej filtrowania po Imieniu i Nazwisku, natomiast po prawej jest lista filmów wypożyczonych i jeszcze niezwróconych przez danego klienta. Również po prawej stronie na dole mamy informacje dotyczące

danego wypożyczenia, takie jak: imię i nazwisko klienta, tytuł filmu, nośnik, datę wypożyczenia i okres w dniach, imię i nazwisko pracownika dokonującego wypożyczenia i kwotę należną za wypożyczenie. Przycisk *NOWE WYPOŻYCZENIE* pozwala wpisać do bazy kolejne wypożyczenie filmu, natomiast przycisk *ZWROT* służy do zarejestrowania zwrotu filmu przez klienta. Formatki uruchamiane przez te dwa przyciski pokazane są na rys. 8.32a i 8.32b.



Rys. 8.31. Zrzut ekranowy formatki *Wypożyczenia*





a) Formatka *Nowe wypożyczenia*

b) Formatka *Zwroty*

Rys. 8.32. Zrzuty ekranowe formatek *Nowe Wypożyczenie* (a) i *Zwroty* (b)

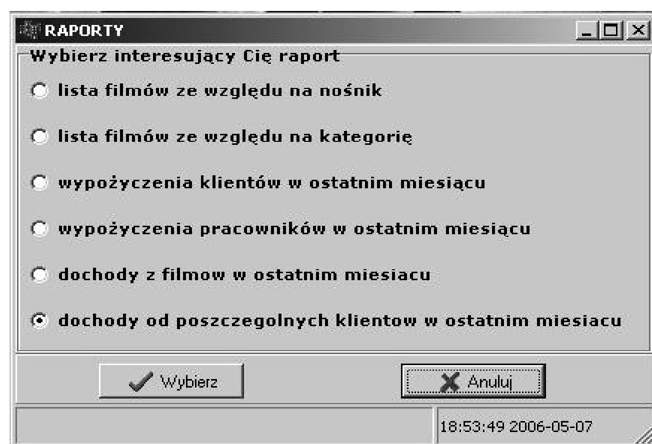
Przykładowo: po naciśnięciu przycisku *Wypożycz* na formatce *Nowe Wypożyczenie* jest wykonywany następujący kod:

```
ModulDanych->IBDSWypozyczoneKasety->FieldByName("ID_KLIENTA")-
>AsInteger =
       ModulDanych->IBTKlienci->FieldByName("ID_KLIENTA")->AsInteger;
        ModulDanych->IBDSWypozyczoneKasety->FieldByName("ID_KASETY")-
>AsInteger =
        ModulDanych->IBQListaFilmow->FieldByName("ID_KASETY")-
>AsInteger;
        ModulDanych->IBDSWypozyczoneKasety->FieldByName("OKRES")-
>AsInteger =
CSDni->Value;
        ModulDanych->IBDSWypozyczoneKasety->CheckBrowseMode();
 ModulDanych->IBQTemp->SQL->Text = "UPDATE KASETY SET ILOSC_SZTUK =
ILOSC_SZTUK - 1 WHERE ID_KASETY = "
+ ModulDanych->IBQListaFilmo->FieldByName("ID_KASETY")->AsString;
        ModulDanych->IBQTemp->ExecSQL();
        ModalResult = mrOk;
```

Uzupełniane są tu odpowiednie pola komponentu `IBDSWypozyczoneKaset`, który odpowiada za obsługę wypożyczeń, ale oprócz tego modyfikowane jest pole *Ilość_Sztuk* tabeli *Kasety* dla wypożyczanego akurat filmu – zmniejszany jest stan kasety o 1. Przy zwrocie filmu wykonywana jest operacja odwrotna – pole *Ilość_Sztuk* zwiększane jest o 1, dzięki czemu ktoś inny może wypożyczyć dany egzemplarz filmu.

8.2.6. Formatka *Raporty*

Formatka *Raporty* dostarcza właścicielowi wypożyczalni „Matrix" wielu informacji, dzięki którym może on podejmować decyzje dotyczące oferowanych filmów w przyszłości, a także oceniać pracę swoich pracowników. Ma ona postać pokazaną na rys. 8.33.



Rys. 8.33. Rzut ekranowy formatki *Raporty*

Do wyboru są następujące raporty:

- lista filmów ze względu na nośnik – pokazuje on filmy dostępne w wypożyczalni w rozbiciu na nośniki. Dzięki temu zestawieniu właściciel może zdecydować, jaki nośnik powinien teraz zamawiać, filmów, na którym nośniku jest niedobór;
- lista filmów ze względu na kategorię – pokazuje filmy dostępne w wypożyczalni w rozbiciu na poszczególne kategorie. Dzięki temu zestawieniu właściciel może zdecydować, jakiej kategorii filmów jest niedobór i wobec tego jakich filmów należy więcej wprowadzić do oferty;
- wypożyczenia klientów w ostatnim miesiącu – pokazuje wszystkich klientów, którzy dokonali wypożyczenia w ostatnim miesiącu wraz z tytułami filmów i datą wypożyczenia. Dzięki temu zestawieniu właściciel wypożyczalni może stwierdzić, które tytuły cieszyły się największym powodzeniem i którzy klienci wykazali się największą aktywnością w ostatnim miesiącu;
- wypożyczenia pracowników w ostatnim miesiącu – pokazuje wszystkich pracowników, którzy obsługiwali wypożyczenia filmów w ostatnim miesiącu wraz z tytułami filmów oraz datą wypożyczenia. Dzięki temu właściciel może stwierdzić, którzy pracownicy są najbardziej aktywni;
- dochody z filmów w ostatnim miesiącu – pokazuje wszystkie filmy, które przyniosły dochód w ostatnim miesiącu, nośniki, a także całkowity dochód. Filmy są posortowane malejąco pod względem całkowitego dochodu. Dzięki temu właściciel wie, wypożyczanie których filmów przynosi największy i najmniejszy dochód i może podjąć decyzję o zamówieniu nowych egzemplarzy lub wycofaniu filmu z oferty;
- dochody od poszczególnych klientów w ostatnim miesiącu – pokazuje wszystkich klientów, na których wypożyczalnia zarobiła w ostatnim miesiącu, poczynając od tych najbardziej dochodowych. Dzięki temu właściciel wie, którzy klienci są najbardziej wartościowi i którym klientom można udzielić ewentualnych rabatów.

Aby wydobyć informacje zawarte w powyższych zestawieniach, niezbędne jest zbudowanie odpowiednich poleceń `SELECT`. Przykładowo: dla ostatniego z wymienionych raportów to polecenie ma następującą postać:

```
SELECT K.Imię||' '||K.Nazwisko Full_Name,
SUM(KA.Cena_Za_Dobe * W.Okres) Dochod
FROM Klienci K, Kasety KA, Wypozyczenia W
WHERE W.Id_Kasety = KA.Id_Kasety
AND    W.Id_Klienta = K.Id_Klienta
AND    W.Data_Wypozyczenia + 30 > Current_Date
GROUP BY K.Imię,K.Nazwisko
ORDER BY 2 DESC;
```

Wybierane jest imię i nazwisko klienta oraz iloczyn ceny za dobę filmu i okresu wypożyczenia dla wypożyczeń z ostatniego miesiąca. Wybrany wynik jest grupowany po imieniu i nazwisku klienta, iloczyny stanowiące dochód są sumowane w ramach każdego z klientów. Ostatecznie wyniki są sortowane po dochodzie malejąco.

Literatura

Abbey M., Corey M., Abramson I., *Oracle 9i. Przewodnik dla początkujących*, Helion 2003.
Allen S., *Modelowanie danych*, Helion 2006.
Banachowski L., *Bazy danych. Tworzenie aplikacji*, Wydawnictwo „Akademicka Oficyna Wydawnicza PLJ", Warszawa 1998.
Barker R., *CASE Method. Modelowanie związków encji*, WNT, Warszawa 1996.
Barker R., Longman C., *CASE Method. Modelowanie funkcji i procesów*, WNT, Warszawa 2001.
Beynon-Davies P., *Systemy baz danych*, WNT, Warszawa 2000.
Codd E., *The Relational Model for Database Management*, Adison-Wesley Pub. Comp. 1990.
Connolly T., Begg C., *Systemy baz danych*, Warszawa 2004.
Date C.J., *Relacyjne bazy danych dla praktyków*, Helion 2005.
Date C.J., *Wprowadzenie do systemów baz danych*, WNT, Warszawa 2000.
Dorobek M., *C++ Builder. Podręcznik*, Mikom, Warszawa 2002.
Dudek W., *Bazy danych SQL. Teoria i praktyka*, Helion 2006.
Elmasri R., Navathe S., *Wprowadzenie do systemów baz danych*, Helion 2005.
Fernandez M.J., *Bazy danych dla zwykłych śmiertelników*, Mikom, Warszawa 2004.
Garcia-Molina H., Ullman J.D., Widom J., *Implementacja systemów baz danych*, WNT, Warszawa 2003.
Gawroński P.N., *InterBase dla „delfinów"*, Helion 2001.
Hall L.C., *Techniczne podstawy systemów klient–serwer*, WNT, Warszawa 1996.
Henderson K., *Bazy danych w architekturze klient/serwer*, Robomatic, Wrocław 2000.
Jakubowski A., *SQL w InterBase dla Windows i Linuksa*, Helion 2001.
Jankowski B., Regmunt A., *Bazy danych: uczymy się na przykładach*, Mikom, Warszawa 2004.
Mendrala D., Szeliga M., *Access 2003 PL. Ćwiczenia praktyczne*, Helion 2006[2].
Muller R.J., *Bazy danych. Język UML w modelowaniu danych*, Mikom, Warszawa 2000.
Otey M., Otey D., *Microsoft SQL Server 2005. Podręcznik programisty*, Helion 2007.
Riordan R.M., *Projektowanie systemów relacyjnych baz danych*, Wydawnictwo RM, Warszawa 2000.
Rudnicki D.D., *Bazy danych*, Toruń 2004.
Stokłosa J, Bilski T., Pankowski T., *Bezpieczeństwo danych w systemach informatycznych*, PWN S.A., Warszawa–Poznań 2001.
Stones R., Neil M., *Bazy danych i PostgreSQL. Od podstaw*, Helion 2003.
Ullman L., *MySQL. Szybki start*, Helion 2007[2].
Ullman S., *Systemy baz danych*, WNT, Warszawa 1988.
Wybrańczyk M., *C++ Builder 6 i bazy danych*, Helion 2005.
Wybrańczyk M., *Delphi 7 i bazy danych*, Helion 2003.

Dokumentacja FireBird 1.5 oraz InterBase 6

Api Guide (InterBase 6).
Data Definition Guide (InterBase 6).
Developer's Guide (InterBase 6).
Embedded SQL Guide (InterBase 6).

FireBird 1.5 Quick Start Guide.
FireBird – Release Notes v.1.5.
Getting Started (Instalation and Migration).
Language Reference (InterBase 6).
Migration from MS-SQL to FireBird.
Operations Guide (InterBase 6).

Strony internetowe

http://pl.wikipedia.org/wiki/SQL.
http://pl.wikipedia.org/wiki/Firebird.
http://www.firebirdsql.org/manual/.
http://www.ibphoenix.com.
http://www.ploug.org.pl/gazetka/20/3.htm.
http://www.dbf.pl/faq/tresc.html?rozdzial=1#o1_5.
http://gskoczylas.rekord.pl/.

Komercjalizacja wiedzy

jako instrument wsparcia rozwoju gospodarczego województwa lubelskiego

Jak sprzedać wiedzę?

Adresaci projektu

- Pracownicy naukowi i naukowo-dydaktyczni uczelni i jednostek naukowych z terenu województwa lubelskiego
- Doktoranci, absolwenci uczelni i studenci zamieszkali, pracujący lub uczący się na terenie województwa lubelskiego

Cel projektu

- Promocja idei przedsiębiorczości akademickiej
- Przygotowanie uczestników do zakładania firm spin-off i spin-out

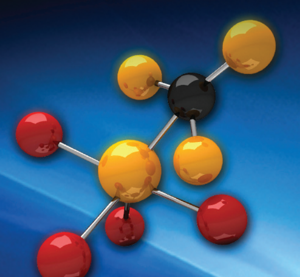
www.lbs.pl

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Komercjalizacja wiedzy

jako instrument wsparcia rozwoju gospodarczego województwa lubelskiego

Formy wsparcia oferowane w ramach projektu



- Szkolenie „Studium przedsiębiorczości akademickiej”
- Usługi doradcze
- Konferencje

Świadczenia przysługujące uczestnikom

- Obszerny zestaw materiałów szkoleniowych
- „Poradnik przedsiębiorczości akademickiej” – wydany na zakończenie projektu
- Wyżywienie na szkoleniach
- Zwrot kosztów dojazdu na szkolenia (osoby spoza Lublina)
- Materiały konferencyjne wydane jako odrębne publikacje

www.lbs.pl

Biuro projektu:

Lubelska Szkoła Biznesu
ul. Konstantynów 1H
20-708 Lublin

Tel. 81 445 46 60
Tel. 81 445 46 68
Email: info@lbs.pl

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego