Relacyjne bazy danych dla praktyków

Autor: C.J. Date
Tłumaczenie: Marek Pętlicki
ISBN: 83-246-0101-5
Tytuł oryginału: Database in Depth
Format: B5, stron: 280

Wszystkie tajniki relacyjnego modelu danych

- Dowiedz się, czym są krotki i relacje
- Poznaj algebrę relacji i zasady normalizacji danych
- Zaprojektuj efektywne schematy relacji i zaimplementuj je w bazie

Relacyjne bazy danych spotykamy niemal w każdej aplikacji komputerowej, często nawet nie zdając sobie z tego sprawy. Wiemy, że w oparciu o nie buduje się aplikacje korporacyjne, witryny internetowe i inne rozbudowane systemy informatyczne. Jednak to nie wszystko – bazy danych są wszędzie – nawet cyfrowy aparat fotograficzny posiada bazę danych, którą wykorzystuje przy doborze parametrów ekspozycji. Wszystkie nowoczesne systemy zarządzania bazami opierają się na modelu relacyjnym, sformułowanym w 1969 roku 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 taką wiedzę, aby w pełni wykorzystać możliwości, jakie oferuje im model relacyjny.

Książka „Relacyjne bazy danych dla praktyków" to szczegółowe omówienie modelu relacyjnego przeznaczone dla użytkowników takich systemów. Nie opisuje konkretnych produktów – przedstawia wszystkie tajniki teorii relacyjnej i wskazuje możliwości wykorzystania tej wiedzy w codziennych zadaniach. Autor książki Chris Date, znany autorytet z dziedziny baz danych, przedstawi Ci koncepcje relacyjne, teorię zbiorów, różnice pomiędzy modelem a implementacją, algebrę relacyjną oraz zagadnienia normalizacji danych.

- Podstawy modelu relacyjnego
- Skalarne i nieskalarne typy danych
- Krotki i relacje pomiędzy nimi
- Zmienne relacyjne i predykaty
- Podstawy algebry relacji
- Ograniczenia w bazach danych
- Postaci normalne
- Teoria projektowania baz danych
- Implementacja modelu relacyjnego

Dzięki wiadomościom zawartym w tej książce sprawniej i efektywniej wykorzystasz możliwości współczesnych systemów zarządzania bazami danych.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 4.

Zmienne relacyjne

W rozdziale 1. poznaliśmy pojęcie zmiennej relacyjnej, której wartości są relacjami. To właśnie zmienne relacyjne powstają w wyniku działania operacji INSERT, DELETE czy UPDATE. Dowiedzieliśmy się też, że INSERT, DELETE oraz UPDATE są odpowiednikami relacyjnych operacji przypisania. Przypominam, że jeśli *R* jest zmienną relacyjną, a *r* jest relacją, która ma być przypisana *R*, wtedy *R* i *r* muszą być zgodnego typu relacyjnego. Druga ważna informacja to taka, że *nagłówki*, *zawartość*, *atrybuty*, *krotki*, *liczność* i *stopień* zdefiniowane formalnie na potrzeby relacji (rozdział 3.) mają zastosowanie również do zmiennych relacyjnych. Nadszedł czas, aby przyjrzeć się bliżej tym zagadnieniom. Jako podstawę do rozważań wykorzystam następujące definicje relacji z bazy dostawców i części zamiennych napisane w języku Tutorial D:

```
VAR S BASE RELATION
  {SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR}
   KEY {SNO};

VAR P BASE RELATION
  {PNO PNO, PNAME NAME, COLOR COLOR, WEIGHT WEIGHT, CITY CHAR }
   KEY {PNO};

VAR SP BASE RELATION
  {SNO SNO, PNO PNO, OTY OTY}
   KEY {SNO, PNO}
   FOREIGN KEY {SNO} REFERENCES S
   FOREIGN KEY {PNO} REFERENCES P;
```

Operacje modyfikujące

Pierwsze ważne spostrzeżenie dotyczy przypisania. Niezależnie od zastosowanej składni przypisanie relacyjne jest operacją *na poziomie* zbiorów. W rzeczywistości wszystkie operacje zdefiniowane w modelu relacyjnym odbywają się na poziomie zbiorów, do czego wrócimy w rozdziale 5. Oznacza to, że INSERT dodaje do zmiennej relacyjnej zbiór krotek, DELETE usuwa ze zmiennej relacyjnej zbiór krotek, natomiast UPDATE modyfikuje zbiór krotek zmiennej relacyjnej. Często co prawda mówi się, że modyfikuje się jakąś konkretną krotkę, należy jednak zawsze pamiętać o tym, że:

- takie stwierdzenie oznacza, że modyfikowany zbiór krotek ma liczność równą jeden;
- aktualizacja zbioru krotek o liczności równej jeden czasem nie jest operacją wykonalną.

Załóżmy na przykład, że w zmiennej relacyjnej S jest zdefiniowane ograniczenie integralnościowe (rozdział 6.) wymuszające, aby S1 i S4 zawsze miały zdefiniowane to samo miasto. W takim przypadku operacja modyfikacji, która ma zadziałać tylko w jednej krotce, po prostu nie zadziała poprawnie. Należy więc zaktualizować obydwie powiązane krotki, na przykład wykorzystując następujące zapytanie (SQL):

```
UPDATE S
SET    CITY = 'Gdańsk'
WHERE  S.SNO = SNO('S1') OR S.SNO = SNO('S4');
```

Powyższe zapytanie oczywiście modyfikuje zbiór złożony z dwóch krotek.

UWAGA Dla zainteresowanych: odpowiednie wyrażenie w języku Tutorial D ma następującą postać (dość podobną):

```
UPDATE S WHERE SNO = SNO('S1') OR SNO = SNO('S4')
   (CITY := 'Gdańsk');
```

Jedną z konsekwencji powyższych własności jest brak w modelu relacyjnym obsługi pozycjonowanych modyfikacji, znanych z języka SQL (zapytania typu `DELETE ... WHERE CURRENT OF cursor`), ponieważ operacje tego typu z definicji odbywają się na poziomie krotki, nie zbioru. Tego typu techniki spotyka się dość często we współczesnych produktach, lecz dzieje się tak głównie dlatego, że produkty te nie są zwykle szczególnie skuteczne w obsłudze ograniczeń integralnościowych. Gdyby wspomniane produkty poprawiły swoją obsługę ograniczeń integralnościowych, mogłoby się okazać, że „modyfikacje pozycyjne" *nie* będą działać. Oznacza to, że aplikacje bardzo popularne dziś jutro straciłyby swoich użytkowników. Jest to moim zdaniem sytuacja, której producenci starają się unikać.

Muszę się do czegoś przyznać. Stosowane przeze mnie określenie „modyfikacji krotki", lub raczej zbioru krotek, jest dość nieprecyzyjne (żeby nie powiedzieć „do niczego"). Jeśli obiekt *V* jest podatny na modyfikację, musi być *zmienną*, nie wartością, nie zbiorem krotek ani relacją, ponieważ jak wiemy z definicji, *wartości* nie mogą być modyfikowane. Mówiąc o modyfikacji krotki *t1* na krotkę *t2* w ramach zmiennej relacyjnej *R*, mamy na myśli *zastąpienie* krotki *t1* w *R* na krotkę *t2*. Ale ten rodzaj rozumowania nadal jest nieprecyzyjny! Tak *naprawdę* to cała oryginalna relacja *r1* ulega wymianie na *r2* w ramach zmiennej relacyjnej *R*. Czym jest zatem *r2*? Załóżmy, że *s1* i *s2* są relacjami zawierającymi odpowiednio krotki *t1* i *t2*. Relacja *r2* powstaje w wyniku wyrażenia `(r1 MINUS s1) UNION s2`. Innymi słowy, „modyfikację krotki *t1* na krotkę *t2* w ramach zmiennej relacyjnej *R*" można postrzegać jako usunięcie *t1* i wstawienie w to miejsce *t2*. To nadal znaczne uproszczenie, ponieważ zakłada możliwość usuwania i podstawiania pojedynczych krotek w zmiennych relacyjnych.

Analogicznie nie należy dosłownie postrzegać operacji typu „modyfikacja atrybutu *A* w krotce *t*" lub w relacji *r*, a nawet w zmiennej relacyjnej *R*. Oczywiście taka operacja (efektywnie) ma miejsce i wygodnie jest określać ją w taki sposób (to znacznie upraszcza analizę), lecz podobnie jak w przypadku „terminologii przyjaznej użytkownikowi" skrytykowanej przeze mnie w rozdziale 1. można przyjąć stosowanie takich uproszczeń jedynie w sytuacjach, gdy nie spowoduje to nieporozumień i nie zagmatwa postrzegania rzeczywistego stanu rzeczy.

Klucze kandydujące

Podstawowe założenia dotyczące kluczy kandydujących opisałem w rozdziale 1., lecz nadszedł czas na doprecyzowanie tego pojęcia. Oto definicja:

DEFINICJA Niech *K* będzie podzbiorem nagłówka zmiennej relacyjnej *R*. *K* jest *kluczem kandydującym* (lub po prostu *kluczem*) *R* wtedy i tylko wtedy, gdy posiada następujące cechy:

DEFINICJA *Unikalność*: Żadna wartość *R* nie może zawierać dwóch różnych krotek o tej samej wartości *K*.

DEFINICJA *Nieredukowalność*: Żaden podzbiór *K* nie ma cechy unikalności.

UWAGA Zgodnie z powszechnie stosowaną praktyką w tej książce stosuję powszechnie określenia typu „*B* jest podzbiorem *A*" oraz „*A* jest nadzbiorem *B*" z założeniem, że *A* i *B* mogą być równe. Gdy pojawi się konieczność wykluczenia tej możliwości, wyraźnie to zaznaczę.

Własność unikalności jest zrozumiała i oczywista, należy jednak powiedzieć kilka słów o własności nieredukowalności. Weźmy pod uwagę zmienną relacyjną *S* i zbiór atrybutów {SNO, CITY} (nazwijmy go *SK*). Ten zbiór atrybutów jest podzbiorem nagłówka zmiennej *S* z pewnością posiadającym cechę unikalności (żadna możliwa wartość *S* nie może mieć dwóch krotek o tej samej wartości *SK*). *SK* jednak nie ma cechy nieredukowalności, ponieważ możemy odrzucić atrybut CITY i to, co zostanie, czyli podzbiór {SNO} również posiada cechę unikalności. Nie można więc uznać *SK* za klucz, ponieważ *jest* „za duży". Natomiast {SNO} ma cechę nieredukowalności i jest poprawnym kluczem.

Dlaczego klucze muszą być nieredukowalne? Jednym z powodów jest poprawna obsługa klauzuli unikalności przez DBMS. Załóżmy, że „okłamaliśmy" DBMS, wymuszając klucz {SNO, CITY}. System DBMS *nie* może w takim przypadku wymusić unikalności identyfikatora dostawcy, może jedynie wymusić unikalność klucza, czyli identyfikatorów dostawców w danym mieście. To jeden z powodów (oczywiście nie jedyny), dlaczego klucze nie mogą zawierać atrybutów niepotrzebnych do unikalnej identyfikacji krotek.

Wszystkie zmienne relacyjne analizowane do tej pory posiadają tylko jeden klucz. Przedstawię dla odmiany kilka relacji zawierających większą liczbę kluczy (te przykłady są z założenia intuicyjne). Zwracam uwagę na częściowe pokrywanie się kluczy w drugim i trzecim przykładzie.

```
VAR TAX_BRACKET BASE RELATION
  {LOW MONEY, HIGH MONEY, PERCENTAGE INTEGER}
   KEY {LOW}
   KEY {HIGH}
   KEY {PERCENTAGE};

VAR ROSTER BASE RELATION
  {DAY DAY_OF_WEEK, TIME TIME_OF_DAY, GATE GATE, PILOT NAME}
   KEY {DAY, TIME, GATE}
   KEY {DAY, TIME, PILOT};
```

```
VAR MARRIAGE BASE RELATION
  {SPOUSE_A NAME, SPOUSE_B NAME, DATE_OF_MARRIAGE DATE}
   /* wykluczamy poligamię */
   /* i możliwość dwukrotnego zawarcia małżeństwa przez tę samą parę */
   KEY {SPOUSE_A, DATE_OF_MARRIAGE}
   KEY {DATE_OF_MARRIAGE, SPOUSE_B}
   KEY {SPOUSE_B, SPOUSE_A};
```

Ten podrozdział zakończę kilkoma uwagami. Po pierwsze, należy pamiętać, że koncepcja kluczy odnosi się do zmiennych relacyjnych, nie do relacji. Dlaczego? Ponieważ stwierdzenie, że podzbiór nagłówka jest kluczem, jest równoznaczne zdefiniowaniu ograniczenia integralnościowego, a dokładniej ograniczenia wymuszającego unikalność. Ograniczenie integralnościowe stosuje się do zmiennych, nie do wartości — ograniczenia stanowią mechanizm kontrolny operacji modyfikujących, a jak wiadomo wartości nie są modyfikowalne. Więcej informacji na ten temat można znaleźć w rozdziale 6.

Po drugie, jeśli *R* jest zmienną relacyjną, musi zawierać przynajmniej jeden klucz. Powodem tego wymogu jest to, że wartościami *R* są relacje, które z definicji nie mogą zawierać duplikatów. W najgorszym przypadku więc wymóg unikalności spełnia kombinacja wszystkich atrybutów *R*[1]. Z tego powodu albo kombinacja wszystkich atrybutów relacji spełnia wymóg nieredukowalności, albo udaje się wyodrębnić podzbiór atrybutów spełniający ten wymóg. W każdym wypadku *zawsze* udaje się znaleźć odpowiedni klucz.

Po trzecie, należy pamiętać, że wartościami klucza są *krotki*. W przypadku zmiennej relacyjnej *S* zawierającej jedyny klucz {SNO} wartość tego klucza dla określonej krotki (np. dla dostawcy S1) wynosi:

```
TUPLE {SNO SNO('S1')}
```

Jak pamiętamy z rozdziału 3., każdy podzbiór krotki jest również krotką. Oczywiście w praktyce można powiedzieć w uproszczeniu, że wartością klucza w tym przykładzie jest S1 lub SNO('S1'), lecz ponownie nie jest to stwierdzenie precyzyjne.

Powinno być już oczywiste, że koncepcja kluczy opiera się (jak większość zagadnień modelu relacyjnego) na koncepcji *równości krotek*. Aby więc wymusić ograniczenie unikalności, musimy być w stanie stwierdzić, kiedy dwie wartości klucza są równe, i to właśnie jest kwestia równości krotek. Nawet w przypadku zmiennej relacyjnej *S*, gdy krotki klucza „wyglądają" jak zwykłe wartości skalarne.

W ramach ostatniej uwagi chciałbym nawiązać do *zależności funkcyjnej*. Nie będę w tym miejscu omawiać szczegółowo tej koncepcji (przyjdzie na to czas w rozdziale 7.), lecz Czytelnik zapewne już się z nią spotkał. Chcę zwrócić uwagę na jeden szczegół. Załóżmy, że *K* jest kluczem zmiennej relacyjnej *R*, a *A* jest atrybutem *R*. Wynika z tego, że *R* z pewnością spełnia następującą zależność funkcyjną:

$$K \rightarrow A$$

W skrócie: zależność funkcyjna $K \rightarrow A$ oznacza, że każde dwie krotki *R* zawierające tę samą wartość *K* będą również zawierały tę samą wartość *A*. Lecz jeśli dwie krotki zawierają tę samą wartość *K*, a *K* jest kluczem, to z definicji jest to ta sama krotka, a w związku z tym *oczywiście mają* tę samą wartość *A*. Innymi słowy: zawsze zachodzi zależność funkcyjna wszyst-

[1] Nie można tego oczywiście powiedzieć o tabelach języka SQL — jak pamiętamy tabele SQL zezwalają na duplikację wierszy i dlatego mogą nie zawierać żadnego klucza.

kich wartości atrybutów od kluczy relacji. Zjawisko to przeanalizuję bardziej szczegółowo w rozdziale 7.

Klucze obce

W rozdziale 1. wyjaśniłem podstawowe założenia kluczy obcych, lecz dokładną definicję podam dopiero teraz (zwracam uwagę na to, że i tutaj znajdziemy związek z koncepcją równości krotek).

DEFINICJA

Załóżmy, że *R1* i *R2* są zmiennymi relacyjnymi (niekoniecznie różnymi), *K* jest kluczem *R1*. Załóżmy też, że *FK* jest podzbiorem nagłówka *R2*, który zawiera dokładnie te same atrybuty co *K* (dopuszczamy możliwość zmiany nazw niektórych atrybutów). *FK* będzie *kluczem obcym* wtedy i tylko wtedy, gdy w danym momencie dla wszystkich krotek w *R2* istnieje wartość *FK* równa (w sposób unikalny) wartości *K* krotki w *R1*.

Jak wiemy, w bazie danych dostawców i części zamiennych w zmiennej relacyjnej SP występują dwa klucze obce {SNO} i {PNO}, które odwołują się do kluczy kandydujących (a w rzeczywistości do kluczy głównych) odpowiednio w relacjach S i P. Oto kolejny przykład:

```
VAR EMP BASE RELATION
  {ENO ENO, ..., MNO ENO, ...}
   KEY {ENO}
   FOREIGN KEY {RENAME (MNO AS ENO)} REFERENCES EMP;
```

Atrybut MNO określa identyfikator menedżera pracownika identyfikowanego przez ENO. Odwołująca się zmienna relacyjna (w definicji *R2*) i zmienna odwoływana (w definicji *R1*) to w tym przypadku ta sama zmienna. Na przykład krotka zmiennej relacyjnej EMP dla pracownika E3 może zawierać w atrybucie MNO wartość E3, co stanowi odwołanie do krotki w tej samej relacji o wartości E2 w atrybucie EMP. Wartości kluczy obcych (podobnie jak wartości kluczy kandydujących) są wartościami krotkowymi, zatem aby zachodziła równość krotek, musimy zmienić nazwę atrybutu MNO w specyfikacji klucza obcego. O jaką równość krotek chodzi? O porównanie wartości klucza obcego z kluczem kandydującym, a jak pamiętamy, aby dwie krotki mogły być równe, muszą być tego samego typu, a „ten sam typ" oznacza, że muszą mieć takie same nazwy i typy atrybutów.

Na marginesie należy wspomnieć, że model relacyjny w swojej oryginalnej postaci wymagał, aby klucze obce były dopasowane nie do dowolnego klucza kandydującego, lecz wyłącznie do klucza *głównego* relacji. Jak jednak stwierdziłem w rozdziale 1., nie uważam, żeby zawsze była konieczność wyboru klucza głównego spośród kluczy kandydujących, a co się z tym wiąże, nie można wymagać, aby klucz obcy był dopasowywany wyłącznie do klucza głównego (akurat pod tym względem zgadzam się z ustaleniami standardu SQL).

SQL obsługuje nie tylko same klucze obce, lecz również pewne *działania związane* z ich obsługą zgodnie z zasadami integralności odwołań. Jednym z takich działań jest dyrektywa CASCADE (która ma zastosowanie przy operacjach usuwania i modyfikacji klucza i definiuje się ją odpowiednio w klauzulach ON DELETE i ON UPDATE). Na przykład instrukcja CREATE TABLE tworząca tabelę dostaw może zawierać następujący fragment tworzący klucz obcy:

```
FOREIGN KEY (SNO) REFERENCES S (SNO) ON DELETE CASCADE
```

Po takim zdefiniowaniu klucza obcego próba usunięcia dostawcy spowoduje usunięcie wszystkich jego dostaw. Wspominam o tym z następujących powodów:

- Tego typu mechanizmy mogą mieć bardzo praktyczne zastosowanie, lecz same w sobie nie wchodzą w skład modelu relacyjnego.
- To jednak nie jest problem, ponieważ model relacyjny jest fundamentem, na którym powinny być budowane bazy danych, lecz nadal to *tylko* fundament. Nie ma powodu, aby w produktach DBMS nie były dodawane funkcje, które w oparciu o fundament pozwolą lepiej obsługiwać model, lecz zarazem nie będą z nim kolidować. Należy przy tym dodać, że takie funkcje uzupełniające powinny wspierać model i być użyteczne. A bardziej konkretnie:
  a) Teoria typów jest najbardziej oczywistym przykładem. W rozdziale 2. Czytelnicy dowiedzieli się, że „typy są niezależne od tabel". Ale także, że prawidłowa obsługa typów jest bardzo pożądana w systemie relacyjnym.
  b) Drugim przykładem mogą być mechanizmy odtwarzania danych i kontroli współużytkowania, które nie są w ogóle omawiane przez model relacyjny, co nie oznacza, że systemy relacyjne nie powinny zawierać takich funkcji. Można co prawda sprzeczać się, czy model relacyjny nie wspomina o takich funkcjach pośrednio, ponieważ wymaga, aby system DBMS poprawnie implementował modyfikacje danych i nie „gubił" niczego „po drodze". Model jednak nie daje żadnych wskazówek co do sposobu, w jaki ma to być zrealizowane.

Uwaga na koniec podrozdziału: klucze obce omówiłem z powodu ich wielkiego znaczenia praktycznego, a także dlatego, że są częścią modelu w jego oryginalnej definicji. Powinienem jednak podkreślić, że nie mają fundamentalnego znaczenia, są po prostu formą abstrakcji, uproszczonej obsługi ograniczeń integralnościowych, które z kolei *mają* znaczenie fundamentalne[2] (o czym przekonamy się w rozdziale 6.). To samo można zresztą powiedzieć i o kluczach kandydujących, lecz w tym przypadku praktyczne zalety opracowania takiej abstrakcji są nieporównywalnie większe.

Perspektywy

Perspektywy (ang. *view*) są czasem określane mianem wirtualnych zmiennych relacyjnych. Jest to taka zmienna relacyjna, która nie istnieje w każdym momencie, lecz sprawia takie wrażenie z punktu widzenia użytkownika. Oto definicja:

DEFINICJA

Perspektywa lub *wirtualna zmienna relacyjna V* jest zmienną relacyjną, której wartość jest generowana w czasie *t* w wyniku określonego wyrażenia relacyjnego. Wyrażenie generujące wartość zmiennej relacyjnej jest określane wraz z definicją *V* i musi wykorzystywać przynajmniej jedną zmienną relacyjną.

[2] Dokładnie z tego powodu Tutorial D nie obsługuje ich w sposób bezpośredni. Jestem jednak pewny, że wersja tego języka przeznaczona na rynek będzie je obsługiwała i pozwolę sobie przyjąć na potrzeby tej książki, że taka obsługa istnieje.

Przeanalizujmy kilka przykładów: „dostawcy z Warszawy" oraz „dostawcy spoza Warszawy (definicja w Tutorial D po lewej, w SQL po prawej):

```
VAR LS VIRTUAL
(S WHERE CITY = 'Warszawa');
```

```
CREATE VIEW LS AS
(SELECT S.*
  FROM S
  WHERE S.CITY = 'Warszawa');
```

```
VAR NLS VIRTUAL
(S WHERE CITY ≠ 'Warszawa');
```

```
CREATE VIEW NLS AS
(SELECT S.*
FROM S
WHERE S.CITY <> 'Warszawa');
```

Nawiasy w powyższych przykładach są zbędne (ale poprawne), dopisałem je w celu zwiększenia czytelności.

Odczyt wartości perspektyw

Powtórzę ważną własność: perspektywy sprawiają wrażenie, jakby istniały w sposób niezależny, innymi słowy, z punktu widzenia użytkownika mają wyglądać i działać tak, jakby były zmiennymi relacyjnymi. W szczególności użytkownik powinien mieć możliwość wykonywania na perspektywach tych samych operacji, co na bazowych zmiennych relacyjnych, a DBMS powinien przenosić te działania na odpowiednie bazowe zmienne relacyjne w sposób zgodny z ostateczną definicją perspektywy. Ostateczną, czyli rozwiniętą z definicji pośrednich, bo perspektywa może być skonstruowana w oparciu o inną perspektywę (skoro perspektywy mają z definicji wyglądać i zachowywać się dokładnie tak, jak bazowe zmienne relacyjne), na przykład:

```
CREATE VIEW LS_STATUS
  AS (SELECT LS.SNO, LS.STATUS
    FROM LS);
```

Odwzorowanie operacji odczytu jest proste. Załóżmy na przykład następujące zapytanie SQL (LS jest perspektywą):

```
SELECT LS.SNO
FROM   LS
WHERE  LS.STATUS > 10
```

Po pierwsze, DBMS zastępuje odwołanie z klauzuli FROM odpowiednim podzapytaniem (z definicji perspektywy):

```
SELECT LS.SNO
FROM  (SELECT S.*
       FROM   S
       WHERE  S.CITY = 'Warszawa') AS LS
WHERE  LS.STATUS > 10
```

To wyrażenie można uprościć w następujący sposób:

```
SELECT S.SNO
FROM   S
WHERE  S.CITY = 'Warszawa'
AND    S.STATUS > 10
```

Opisany proces działa poprawnie dzięki własności *domknięt*ości algebry relacyjnej. Własność domknięt*ości* zakłada między innymi, że wszędzie, gdzie możemy zastosować *nazwę* obiektu

(na przykład w zapytaniu), istnieje bardziej ogólne wyrażenie generujące ten obiekt. W klauzuli FROM można umieścić nazwę tabeli SQL, bardziej ogólne będzie wpisanie tam wyrażenia SQL i dlatego nazwę perspektywy LS możemy zastąpić jej definicją.

Przy okazji warto wspomnieć, że opisany proces nie działał w pierwszych wersjach SQL (a dokładniej pojawił się dopiero w standardzie SQL z roku 1992), ponieważ te wczesne wersje nie obsługiwały domknięcia w sposób poprawny. W wyniku tego niektóre skomplikowane zapytania na niestandardowych tabelach (a dokładniej: na perspektywach) po prostu nie działały, w dodatku często z przyczyn trudnych do wytłumaczenia. Oto prosty przykład:

```
CREATE VIEW V
  AS (SELECT S.CITY, SUM (S.STATUS) AS ST
      FROM S
      CROUP BY S.CITY);

SELECT V.CITY
FROM V
WHERE V.ST > 25
```

To zapytanie w języku SQL zgodnym ze standardem sprzed roku 1992 nie zadziała. I choć standard został poprawiony, nie oznacza to, że zostały poprawione wszystkie produkty! I rzeczywiście, przynajmniej jeden liczący się na rynku produkt nadal (początek roku 2005) nie obsługuje poprawnie tego typu zapytań.

Modyfikacja wartości perspektyw

Przejdźmy do operacji modyfikujących wartości. Zanim zajmę się szczegółami, przyjrzyjmy się perspektywom dostawców z Warszawy i spoza Warszawy (tym razem w Tutorial D):

```
VAR LS VIRTUAL (S WHERE CITY = 'Warszawa');
VAR NLS VIRTUAL (S WHERE CITY ≠ 'Warszawa');
```

Ważne spostrzeżenie: sytuację można odwrócić i utworzyć *relacje bazowe LS i NLS, a relację bazową S z powyższego przykładu skonstruować jako perspektywę wykorzystującą te dwie relacje*:

```
VAR LS BASE RELATION
  {SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR}
   KEY { SNO };

VAR NLS BASE RELATION
  {SNO SNO, SNAME NAME, STATUS INTEGER, CITY CHAR }
  KEY {SNO };

VAR S VIRTUAL (LS UNION NLS);
```

UWAGA

Aby uzyskać pełną równoważność tych dwóch wersji bazy danych, należy zdefiniować pewne ograniczenia. Przede wszystkim każda wartość atrybutu CITY w LS musi być równa `'Warszawa'`, natomiast żadna z wartości CITY w NLS nie może być równa `'Warszawa'`. Więcej tego typu informacji można znaleźć w rozdziale 6.

Przykład ten demonstruje, że *dobór relacji bazowych i pochodnych jest w pełni dowolny*, z czego wynika, że nie ma powodu, aby rozgraniczać relacje bazowe i pochodne. Taką własność nazywa się *regułą wymienności* (bazowych i wirtualnych zmiennych relacyjnych). Oto kilka jej konsekwencji:

- Perspektywy, podobnie jak bazowe zmienne relacyjne, muszą uwzględniać ograniczenia integralnościowe. Ograniczenia integralnościowe z reguły postrzega się jako elementy relacji bazowych, lecz *reguła wymienności* demonstruje, że to nieprawda.
- Perspektywy mają klucze kandydujące, z tego powodu powinienem był umieścić definicje kluczy kandydujących w powyższych definicjach perspektyw. Tutorial D pozwala na to, lecz SQL nie. Perspektywy mogą też mieć zdefiniowane klucze obce, a klucze obce innych relacji mogą wskazywać na perspektywy.
- Nie wspomniałem o tym w rozdziale 1., ale reguła „integralności encji" ma zastosowanie do bazowych zmiennych relacyjnych, nie do perspektyw. Z tego powodu jest sprzeczna z *regułą wymienności*. Oczywiście osobiście odrzucam regułę integralności encji, ponieważ dotyczy NULL-i.
- Wiele produktów SQL i sam standard SQL udostępniają funkcję „identyfikatora wiersza" (ang. *row ID*). Jeśli ta funkcja ma zastosowanie do tabel bazowych i nie ma zastosowania do perspektyw (co jest raczej oczywiste), w prosty sposób stanowi sprzeczność z *regułą wymienności*[3]. Oczywiście identyfikatory wierszy nie występują w modelu relacyjnym, lecz nie oznacza to, że nie mogą być obsługiwane w produktach. Obserwuję jednak, że identyfikatory wierszy służą jako forma identyfikatorów obiektów (w znaczeniu obiektowym, jak również w standardzie SQL i w większości liczących się produktów SQL) i jako takie są zabronione przez model relacyjny! Identyfikatory obiektów są efektywnie *wskaźnikami*, a model relacyjny jawnie zabrania stosowania wskaźników.

Wracając do podstawowego tematu rozważań: perspektywy *muszą* być modyfikowalne, ponieważ w przeciwnym wypadku byłaby to najbardziej bezpośrednia sprzeczność z *regułą wymienności*.

Jak wiadomo, obsługa tego wymogu w języku SQL jest dość słaba, zarówno w standardzie, jak i w produktach. Najczęściej istnieje jedynie możliwość modyfikacji perspektyw zdefiniowanych jako proste selekcje lub projekcje pojedynczych zmiennych relacyjnych (choć nawet tu należy oczekiwać problemów). Załóżmy na przykład następującą perspektywę (identyczną z zaprezentowaną w rozdziale 1.):

```
CREATE VIEW SST_WROCLAW
  AS (SELECT S.SNO, S.STATUS
      FROM   S
      WHERE  S.CITY = 'Wrocław');
```

Ta perspektywa jest projekcją selekcji tabeli bazowej S i dlatego można na przykład wykonać następującą operację:

```
DELETE
FROM    SST_WROCLAW
WHERE   SST_WROCLAW.STATUS > 15;
```

Ta operacja jest równoważna następującej:

```
DELETE
FROM    S
WHERE   S.CITY = 'Wrocław'
AND     S.STATUS > 15;
```

[3] Być może jest również sprzeczna z *zasadą informacji* (rozdział 8.).

Niewiele produktów obsługuje jednak możliwość modyfikacji wartości perspektyw w przypadkach o większym niż przykładowy poziomie komplikacji.

Niestety błądzę teraz w obszarze, który jest obiektem znacznych kontrowersji. Moja osobista opinia jest taka, że problem modyfikacji perspektyw został w znacznym stopniu rozwiązany (w teorii), lecz nie wszyscy zgodzą się ze mną, a głębsza dyskusja na ten temat wymaga analizy, która wybiega poza możliwości tej książki. Z tego powodu mogę jedynie polecić inną książkę: *Databases, Types, and the Relational Model: The Third Manifesto* (Third Edition, Addison-Wesley, 2006) autorstwa C.J. Date i Hugh Darwena. W tej książce problematyka perspektyw jest omówiona bardziej szczegółowo.

Kilka uwag

Istnieje kilka zagadnień, które należy omówić przy okazji perspektyw. Po pierwsze, jak powszechnie wiadomo, ale i tak warto o tym wspomnieć, perspektywy służą dwóm podstawowym celom:

- Z perspektywy *V* korzysta użytkownik, który ją zdefiniował i jest oczywiście świadomy wyrażenia *X* będącego jej definicją. Użytkownik ten może zastosować *V* w każdym miejscu, gdzie pasuje *X*. W takim zastosowaniu *V* jest po prostu skrótem.
- Z perspektywy *V* korzysta użytkownik, który jest po prostu poinformowany o tym, że *V* istnieje i można z niej korzystać, lecz z reguły *nie* zna wyrażenia *X* (w każdym razie nie powinien). Dla niego *V* jest w gruncie rzeczy zwykłą zmienną relacyjną (bazową). I to właśnie tego typu zastosowania perspektyw są szczególnie ważne i analizę perspektyw przeprowadzałem właśnie z myślą o nich.

Gdy objaśniałem podstawy perspektyw na początku tego podrozdziału, napisałem, że wyrażenie relacyjne służące do definicji perspektywy powinno wykorzystywać przynajmniej jedną zmienną relacyjną. Dlaczego? Ponieważ w przeciwnym razie wynikowa „wirtualna zmienna relacyjna" nie będzie zmienną relacyjną! A dokładniej: nie będzie *zmienną* i z pewnością nie będzie można modyfikować jej zawartości. Będzie natomiast czymś, co można nazwać *stałą relacyjną*. Na przykład (składnia jest oczywiście fikcyjna):

```
CONST PERIODIC_TABLE (RELATION {
      TUPLE {ELEMENT 'Wodór', SYMBOL 'H', ATOMICNO 1},
      TUPLE {ELEMENT 'Hel', SYMBOL 'He', ATOMICNO 2},
      ...
      TUPLE {ELEMENT 'Uran', SYMBOL 'U', ATOMICNO 92}
});
```

Dostępność stałych relacyjnych może wydać się ciekawym pomysłem, lecz z pewnością nie należy postrzegać takich obiektów jako zmiennych relacyjnych. Nie sądzę bowiem, że objaśniając zasady programowania, należy przyjąć, że stałe są zmiennymi.

Przy okazji naszych analiz wystąpiło bardzo niefortunne zjawisko kolizji terminologicznej, szczególnie wśród Czytelników o podejściu akademickim, lecz zapewne również wśród Czytelników o podejściu biznesowym. Jak pamiętamy z rozdziału 1., perspektywę można postrzegać jako pochodną zmiennej relacyjnej. Istnieje jeszcze inna forma pochodnej zmiennej relacyjnej, zwana *migawką* (ang. *snapshot*). Jak sugeruje nazwa, migawka jest pochodną, jest jednak też obiektem rzeczywistym, nie wirtualnym. Oznacza to, że jest reprezentowana nie tylko za pomocą definicji pobierającej wartości z innych zmiennych relacyjnych, lecz także

(przynajmniej koncepcyjnie) przez własną kopię danych. Na przykład (ponownie wymyślam fikcyjną składnię):

```
VAR LSS SNAPSHOT (S WHERE CITY = 'Warszawa')
    REFRESH EVERY DAY;
```

Definicja migawki jest podobna do definicji zapytania, z następującymi różnicami:

- Wynik zapytania jest zapisywany w bazie danych pod określoną nazwą (w przykładzie jest to LSS) jako zmienna relacyjna tylko do odczytu (dla zwykłych operacji, ponieważ może zmieniać swoją zawartość w wyniku odświeżania, co omawia kolejny punkt).
- Migawka jest odświeżana okresowo (w przykładzie klauzula EVERY DAY powoduje odświeżanie zrzutu raz dziennie) — aktualna wartość migawki jest porzucana, zapytanie jest wywoływane ponownie i wynik tego wywołania jest ponownie zapisywany jako nowa wartość migawki.

Przykładowa migawka reprezentuje zatem dane, które były aktualne co najwyżej 24 godziny temu.

Migawki są ważne w hurtowniach danych, systemach rozproszonych i wielu innych kontekstach. We wszystkich przypadkach ich stosowanie stanowi efekt kompromisu, który jest akceptowalny (lub nawet wymagany) i efektywnie oznacza „obraz" danych w określonym punkcie czasu. Przykładem takiego przypadku są aplikacje raportujące i księgowe: wymagają z reguły, aby dane były „zamrożone" w danym punkcie czasu (na przykład na koniec okresu rozliczeniowego). Migawki pozwalają na takie zamrożenie bez konieczności blokowania innych aplikacji korzystających z bazy danych.

Problem polega jednak na tym, że migawki w niektórych kręgach są znane nie pod własną nazwą, lecz jako *zmaterializowane perspektywy* (ang. *materialized views*). Migawki *nie są* jednak perspektywami! Podstawowa własność perspektyw z punktu widzenia modelu relacyjnego dotyczy właśnie tego, że *nie są* zmaterializowane! Jak zaobserwowaliśmy, operacje dokonywane na perspektywach są tłumaczone na bazowe zmienne relacyjne. Z tego powodu określenie „zmaterializowana perspektywa" jest po prostu wewnętrzną sprzecznością. Co gorsza, pojęcie *perspektywa* bywa stosowane jako skrót pojęcia „zmaterializowana perspektywa" (w niektórych kręgach), więc problem się pogłębia, co grozi w konsekwencji dewaluacją prawidłowego znaczenia terminu. W tej książce termin *perspektywa* stosuję oczywiście w jego oryginalnym znaczeniu, lecz należy pamiętać, że nie w każdym kontekście oznacza dokładnie to samo.

Zmienne relacyjne i predykaty

Dotarliśmy do najważniejszego (pod wieloma względami) punktu rozdziału. Można go podsumować następująco: zmienne relacyjne można postrzegać na wiele sposobów. Wiele osób postrzega je jako pliki w tradycyjnym znaczeniu informatycznym. Być może jako dość abstrakcyjne pliki (*ustrukturalizowane*, to chyba jeszcze lepsze słowo), lecz koniec końców: pliki. Istnieje jednak inny sposób ich postrzegania, który jak mam nadzieję, pomoże lepiej zrozumieć, o co naprawdę chodzi w modelu relacyjnym.

Weźmy pod uwagę zmienną relacyjną dostawców S. Jak wszystkie zmienne relacyjne powinna ona reprezentować określony fragment rzeczywistości. A dokładniej: nagłówek tej zmiennej relacyjnej reprezentuje pewien *predykat*, czyli ogólną definicję reprezentacji określonego

fragmentu rzeczywistości (ogólną, bo *sparametryzowaną*, o czym za chwilę). Predykat ten ma następującą postać:

> Dostawca SNO ma podpisany kontrakt z naszą firmą, ma nazwisko SNAME, status STATUS, a jego siedziba znajduje się w mieście CITY.

Ten predykat to *założona interpretacja*, czyli inaczej *znaczenie* zmiennej relacyjnej S.

Predykaty można postrzegać jako *funkcje o wartości logicznej*. Predykat jak wszystkie funkcje posiada parametry, zwraca wynik, którym może być wartość logiczna TRUE lub FALSE. W przypadku prezentowanego wyżej predykatu parametrami są SNO, SNAME, STATUS i CITY (odpowiadające atrybutom zmiennej relacyjnej) i przyjmują wartości określonych typów (odpowiednio SNO, NAME, INTEGER i CHAR). Gdy ta funkcja jest wywołana (lub jak mówią logicy: *gdy tworzony jest egzemplarz predykatu*), za parametry podstawiane są odpowiednie argumenty. Załóżmy, że podstawiamy odpowiednio argumenty S1, Kowalski, 20 i Warszawa. Otrzymamy następującą *propozycję*:

> Dostawca S1 ma podpisany kontrakt z naszą firmą, ma nazwisko Kowalski, status 20, a jego siedziba znajduje się w mieście Warszawa.

Propozycja to instrukcja logiczna zwracająca bezwarunkowo wartość TRUE lub FALSE. Oto kilka przykładów:

> Edward Abbey napisał *The Monkey Wrench Gang*.
>
> William Szekspir napisał *The Monkey Wrench Gang*.

Pierwsze wyrażenie ma wartość TRUE, drugie ma wartość FALSE. Nie wolno zakładać, że propozycje zawsze przyjmują wartość TRUE. Jednakże propozycje rozpatrywane w kontekście zmiennych relacyjnych powinny mieć wartość TRUE z następujących powodów:

- *Każda* zmienna relacyjna ma skojarzony predykat nazywany *predykatem zmiennej relacyjnej*.
- Załóżmy, że zmienna relacyjna *R* ma skojarzony predykat *P*. Każda krotka *t* występująca w *R* może być uznana za reprezentację propozycji *p* utworzonej przez wywołanie (utworzenie egzemplarza) predykatu *P* przez podstawienie wartości atrybutów *t* jako argumentów *P*.
- Zakładamy ponadto (ważne!), że każda propozycja *p* uzyskana w ten sposób ma wartość TRUE.

Z tego powodu, biorąc przykładową wartość zmiennej relacyjnej *S*, zakładamy, że następujące propozycje mają wartości TRUE:

> Dostawca S1 ma podpisany kontrakt z naszą firmą, ma nazwisko Kowalski, status 20, a jego siedziba znajduje się w mieście Warszawa.
>
> Dostawca S2 ma podpisany kontrakt z naszą firmą, ma nazwisko Nowak, status 10, a jego siedziba znajduje się w mieście Wrocław.
>
> Dostawca S3 ma podpisany kontrakt z naszą firmą, ma nazwisko Kwiatkowski, status 30, a jego siedziba znajduje się w mieście Wrocław.

I tak dalej. Co więcej, jeśli jakaś krotka może teoretycznie występować w zmiennej relacyjnej, lecz w danym punkcie czasu się w niej nie znajduje, zakładamy, że wynik propozycji o wartościach z jej atrybutów jest równy FALSE (czyli przyjmujemy założenie, że relacja jest skończonym światem w danym punkcie czasu). Weźmy na przykład następującą krotkę:

```
TUPLE {SNO SNO('S6'), SNAME NAME('Śmigły'), STATUS 30, CITY 'Szczecin'}
```

Taka krotka jest całkowicie poprawna, lecz w danym punkcie czasu nie występuje w zmiennej relacyjnej S, więc zakładamy, że w tym punkcie czasu następująca propozycja ma wartość FALSE:

> Dostawca S6 ma podpisany kontrakt z naszą firmą, ma nazwisko Śmigły, status 30, a jego siedziba znajduje się w mieście Szczecin.

Innymi słowy, zmienna relacyjna zawiera w danym punkcie czasu *wszystkie* i *tylko* te krotki, dla których propozycje (egzemplarze predykatów) w tym punkcie czasu mają wartość TRUE.

DEFINICJA

Więcej terminologii: Niech *P* będzie predykatem zmiennej relacyjnej *R*, niech wartość *R* w danym punkcie czasu będzie relacją *r*. Relacja *r* (jej zawartość) stanowi *rozszerzenie* (ang. *extension*) *P* w danym punkcie czasu. Rozszerzenie jest zmienne w czasie, lecz predykat (ang. *predicate* lub *intension*) jest stały.

Wyrażenia relacyjne

Omówione koncepcje mają bezpośredni związek z wyrażeniami relacyjnymi. Na przykład weźmy pod uwagę następujące wyrażenie, które reprezentuje projekcję relacji dostawców zawierającą wszystkie atrybuty, oprócz CITY:

```
S {SNO, SNAME, STATUS}
```

Wynik zawiera wszystkie krotki relacji S postaci:

```
TUPLE {SNO s, SNAME n, STATUS t}
```

które występują w relacji S i mają postać (dla dowolnej wartości c atrybutu CITY):

```
TUPLE {SNO s, SNAME n, STATUS t, CITY c}
```

Innymi słowy, wynik reprezentuje bieżące rozszerzenie następującego predykatu:

> Istnieje takie miasto CITY, że dostawca SNO ma kontrakt z naszą firmą, ma nazwisko SNAME, status STATUS, a jego siedziba znajduje się w mieście CITY.

Należy zauważyć, że ten predykat posiada trzy parametry, a odpowiadająca mu relacja (projekcja relacji dostawców z pominięciem atrybutu CITY) ma trzy atrybuty — CITY nie jest parametrem, w logice tego typu element nazywa się *zmienną związaną* (ang. *bound variable*). Ten predykat nie wykorzystuje bowiem wartości atrybutu CITY, zakłada jedynie, *że takowy istnieje* (dodatkowe informacje na temat zmiennych związanych oraz kwalifikatorów można znaleźć w dodatku A). Innym sposobem wykazania, że ten predykat ma trzy parametry, jest analiza następującego predykatu, który jest logicznym odpowiednikiem powyższego:

> Dostawca SNO ma kontrakt z naszą firmą, ma nazwisko SNAME, status STATUS, a jego siedziba znajduje się w jakimś mieście (*innymi słowy: w dowolnym mieście, nie wiemy gdzie i nie jest to ważne*).

Z powyższych rozważań wynika, że perspektywy są reprezentacją określonych predykatów. Weźmy za przykład perspektywę SST zdefiniowaną następująco:

```
VAR SST VIRTUAL (S {SNO, SNAME, STATUS});
```

Predykat zmiennej relacyjnej dla tej perspektywy będzie miał następujące brzmienie:

> Dostawca SNO ma kontrakt z naszą firmą, ma nazwisko SNAME, status STATUS, a jego siedziba znajduje się w jakimś mieście.

W tym miejscu warto przedstawić jeszcze jedną uwagę dotyczącą predykatów i propozycji. Jak wspominałem, predykat jest zbiorem parametrów. Jak zwykle ten zbiór może być pusty. Jeśli jest, predykat staje się propozycją! Oczywiście przede wszystkim staje się stwierdzeniem, które bezwarunkowo przyjmuje wartość TRUE lub FALSE. Innymi słowy, propozycja jest *zdegenerowanym* predykatem, wszystkie propozycje są zatem predykatami, lecz nie wszystkie predykaty są propozycjami.

Dalsze uwagi na temat relacji i typów

Rozdział 2. ma tytuł „Relacje i typy". Niestety w tamtym rozdziale nie miałem możliwości wyjaśnienia najważniejszej różnicy pomiędzy tymi dwoma koncepcjami. W tym miejscu już mam tę możliwość.

Jak stwierdziłem wyżej, baza danych w dowolnym punkcie czasu może być uznana za kolekcję propozycji o wartości TRUE. Za przykład może posłużyć propozycja *Dostawca S1 ma podpisany kontrakt z naszą firmą, ma nazwisko Kowalski, status 20, a jego siedziba znajduje się w mieście Warszawa*. Co więcej, pokazałem, że wartości atrybutów pojawiające się w takich propozycjach (w przykładzie: S1, Kowalski, 20 i Warszawa) są wartościami atrybutów krotki odpowiadającej danej propozycji i oczywiście każda wartość atrybutu jest odpowiedniego typu. Wynika z tego następująca własność:

Typy są zbiorami elementów, o których można mówić,
relacje są prawdziwymi stwierdzeniami na temat tych elementów.

Innymi słowy, typy dają nam słowniki — czyli elementy, o których możemy mówić. Relacje dają nam możliwość mówienia o tych elementach (ciekawa analogia pomagająca zrozumieć tę własność: *typy są dla relacji tym, czym rzeczowniki są dla zdań*). Na przykład jeśli ograniczymy nasze dociekania tylko do relacji dostawców, zobaczymy, że:

- Możemy mówić o identyfikatorach dostawców, nazwiskach, liczbach i ciągach znaków — i o niczym więcej.
- Możemy stwierdzić fakty o następującej treści: „Dostawca o zadanym identyfikatorze ma kontrakt z naszą firmą, posiada nazwisko, status określony liczbą całkowitą, a jego siedziba mieści się w mieście określonym ciągiem znaków" — i nic więcej. Nic więcej, oprócz *wniosków wynikających* z tych stwierdzeń. Możemy na przykład stwierdzić dość sporo na temat dostawcy S1, na przykład, że *dostawca S1 ma kontrakt z naszą firmą, ma nazwisko Kowalski, status 20, a jego siedziba mieści się w jakimś mieście*, którego nazwa nie jest istotna. To ostatnie stwierdzenie może bardzo przypominać projekcje relacji, i właśnie o to chodziło.

Ten stan rzeczy ma co najmniej trzy istotne konsekwencje. Baza danych ma za zadanie „reprezentować fragment rzeczywistości". W związku z tym zachodzą następujące własności:

1. Typy i relacje są *niezbędne* — bez typów nie mielibyśmy o czym mówić, bez relacji nie mielibyśmy nic do powiedzenia.
2. Typy i relacje są *wystarczające* — z logicznego punktu widzenia nie potrzebujemy nic innego. W szczególności w celu zaprezentowania stanu świata w danym punkcie czasu nie potrzebujemy zmiennych relacyjnych, są one potrzebne jedynie do tego, aby rejestrować zmiany zachodzące w obserwowanym świecie.
3. Typy i relacje *to nie to samo*. Należy unikać tego typu skojarzeń! W rzeczywistości niektóre produkty komercyjne usiłują wmówić właśnie coś takiego (choć niekoniecznie w sposób dosłowny). Mam nadzieję, że produkt opierający się na tak poważnym błędzie logicznym jest skazany na niepowodzenie. Produkty, które mam na myśli, nie są bowiem produktami relacyjnymi, z reguły funkcjonują w oparciu o ujęcie zorientowane obiektowo lub usiłują połączyć obiekty z tabelami SQL (jeden z produktów, które szczególnie mam na myśli, w rzeczywistości praktycznie zniknął już z rynku). Dalsze szczegóły tej analizy znacznie wykraczają poza tematykę książki.

Podrozdział chciałbym zakończyć nieco bardziej formalnym spojrzeniem na tematy w nim omówione. Jak stwierdziłem, bazę danych można postrzegać jako kolekcję prawdziwych propozycji. W rzeczywistości baza danych wraz z operatorami mającymi zastosowanie do propozycji z tej bazy danych (a ściślej: do zbiorów propozycji) jest *systemem logicznym*. Pisząc tu „system logiczny", mam na myśli system formalny — jak na przykład geometria Euklidesa — posiadający *aksjomaty* (prawdy założone) i *reguły związków*, na podstawie których można za pomocą tych aksjomatów dowodzić *twierdzenia* (prawdy wynikające). To było niesamowicie przewidujące ze strony Codda, że gdy w 1969 roku wynalazł model relacyjny, stwierdził, iż baza danych (pomimo nazwy) nie jest po prostu kolekcją *danych*, lecz zbiorem prawdziwych *stwierdzeń* (propozycji). Te propozycje (podstawowe, to znaczy reprezentowane przez bazowe zmienne relacyjne) są aksjomatami w zadanym systemie logicznym. Reguły związków to te reguły, na podstawie których jedne propozycje można przekształcić w inne, innymi słowy, są to reguły zastosowania operatorów algebry relacyjnej. Gdy system generuje wynik wyrażenia relacyjnego (na przykład wynik zapytania), tak naprawdę tworzy nową prawdę w oparciu o daną, a w rezultacie: udowadnia twierdzenie!

Gdy ktoś zrozumie i zaakceptuje powyższe spostrzeżenie, zauważy, że dzięki temu na potrzeby „problemu bazy danych" dostępny staje się cały mechanizm formalnej logiki. Innymi słowy, następujące pytania staną się zagadnieniami logicznymi (i jako takie mogą być poddane analizie logicznej i można na nie uzyskać logiczne odpowiedzi):

- W jaki sposób użytkownik powinien postrzegać bazę danych?
- Jak powinny wyglądać ograniczenia integralnościowe?
- Jak powinien wyglądać język zapytań?
- Jaki jest najlepszy sposób implementacji zapytań?
- Bardziej ogólnie: w jaki sposób wyliczać wyrażenia relacyjne?
- W jaki sposób wyniki powinny być prezentowane użytkownikowi?
- Przede wszystkim: w jaki sposób projektować bazy danych?

Oczywiście rozumie się samo przez się, że model relacyjny zawiera bardzo bezpośrednie odpowiedzi na te pytania. Jest to przy okazji powód, dla którego uważam, że model ten jest skończony, pewny i poprawny i że z pewnością przetrwa. Z tego powodu też uważam, że inne modele danych po prostu „nie należą do tej samej ligi". Tak naprawdę mam wątpliwości, czy te inne „modele" zasługują w ogóle na miano modeli w takim samym sensie, w jakim rozumiany jest model relacyjny. Z pewnością większość tych modeli jest zdefiniowana ad hoc, zamiast w oparciu o solidne fundamenty, jak ma to miejsce w przypadku modelu relacyjnego, który jest oparty na teorii zbiorów i logice predykatów. Szersze omówienie tego tematu zawiera rozdział 8.

Podsumowanie

Najważniejsza część tego rozdziału to podrozdział omawiający predykaty (wraz z poprzedzającym go podrozdziałem na temat relacji i typów). Każda zmienna relacyjna *R* ma przypisany predykat *P* (*predykat zmiennej relacyjnej* dla *R*). Predykat *P* jest założoną interpretacją dla *R* i jest niezmienny w czasie. Jeśli bieżącą wartością *R* jest relacja *r*, wtedy *r* jest *bieżącym rozszerzeniem P*. Rozszerzenie predykatu jest zmienne w czasie. Baza danych wraz z jej operatorami może być zatem postrzegana jako *system logiczny*.

Z tego rozdziału wynikają następujące ważne wnioski:

- Tylko wartości relacyjne mogą być modyfikowane, określenia „modyfikacja krotki" czy „modyfikacja atrybutu" są wygodne, lecz nieprecyzyjne. Modyfikacja odbywa się zawsze na poziomie zbioru krotek (nie jednej krotki).
- Każda zmienna relacyjna posiada przynajmniej jeden klucz (kandydujący). Klucze mają własność unikalności i nieredukowalności. Wartościami kluczy są krotki.
- Niektóre zmienne relacyjne posiadają klucze obce. SQL obsługuje określone „działania związane z zachowaniem integralności", jak CASCADE, które choć bardzo użyteczne, nie są elementem modelu relacyjnego. Co więcej, klucze obce też nie mają znaczenia fundamentalnego.
- Operacje na perspektywach są zaimplementowane za pomocą odwzorowania na odpowiednie bazowe zmienne relacyjne. Proces odwzorowania działa przede wszystkim dzięki własności domknięcia. Jest prosty dla operacji odczytu, lecz w przypadku modyfikacji jest znacznie bardziej skomplikowany. *Reguła wymienności* przyjmuje, że nie ma potrzeby rozgraniczania pomiędzy bazowymi a wirtualnymi zmiennymi relacyjnymi.
- „Typy są dla relacji tym, czym rzeczowniki są dla zdań".
- Typy i relacje są konieczne i wystarczające, aby zaprezentować dane. Oczywiście mówimy tu o poziomie logicznym. Jak doskonale wiemy, na poziomie fizycznym użyteczne są inne konstrukcje. Ta różnica wynika z tego, że na obydwu poziomach abstrakcji najważniejsze zadania nieco się różnią. Między innymi dlatego poziom fizyczny jest celowo pozostawiony poza zakresem zainteresowania modelu relacyjnego.

Ćwiczenia

Ćwiczenie 4.1. Wyjaśnić własnymi słowami, dlaczego określenie typu „ta operacja UPDATE modyfikuje status dostawców z Warszawy" jest nieprecyzyjne. Zaproponować poprawną formę tego określenia.

Ćwiczenie 4.2. Dlaczego „pozycjonowane modyfikacje" znane z języka SQL są przykładem błędnej koncepcji?

Ćwiczenie 4.3. Podać definicje w języku SQL zmiennych relacyjnych TAX_BRACKET, ROSTER oraz MARRIAGE z podrozdziału „Klucze kandydujące".

Ćwiczenie 4.4. Dlaczego określenie „relacja ma klucz" nie ma sensu?

Ćwiczenie 4.5. W treści rozdziału pojawił się argument na to, że własność nieredukowalności kluczy ma znaczenie. Podać inne argumenty.

Ćwiczenie 4.6. „Wartości kluczy nie są skalarami, lecz krotkami". Wyjaśnić dlaczego.

Ćwiczenie 4.7. Załóżmy, że zmienna relacyjna *R* jest stopnia *n*. Jaką maksymalną liczbę kluczy może mieć *R*?

Ćwiczenie 4.8. Zmienna relacyjna EMP z podrozdziału „Klucze obce" jest przykładem *samoodwołującej się* zmiennej relacyjnej. Wymyślić przykładowe dane dla tej zmiennej relacyjnej. Czy ten przykład stanowi uzasadnienie dla stosowania NULL-i? (*Poprawna odpowiedź*: nie, ale jest doskonałą demonstracją tego, jak kusząca może być idea stosowania NULL-i). Co można zrobić w tym przykładzie, jeśli NULL-e są zabronione?

Ćwiczenie 4.9. Język SQL nie obsługuje dostępnej w Tutorial D funkcji modyfikacji nazwy klucza obcego. Dlaczego?

Ćwiczenie 4.10. Czy można wymyślić dwie zmienne relacyjne *R1* i *R2*, które zawierają klucze obce odwołujące się wzajemnie do siebie?

Ćwiczenie 4.11. Przeanalizować wykorzystywany przez siebie produkt SQL. Jakie działania związane z integralnością odwołań obsługuje? Które z nich są rzeczywiście użyteczne? Czy można wymyślić inne, nieobsługiwane przez produkt, lecz potencjalnie użyteczne?

Ćwiczenie 4.12. Model relacyjny nie wspomina o *procedurach wyzwalanych* (ang. *triggered procedures*), często nazywanych *wyzwalaczami* (ang. *triggers*). Czy to pominięcie stanowi problem? Jeśli tak, dlaczego? Czy procedury wyzwalane są konieczne? Czy są w ogóle potrzebne?

Ćwiczenie 4.13. Niech perspektywa LSSP będzie zdefiniowana następująco (SQL):

```
CREATE VIEW LSSP
  AS (SELECT S.SNO, S.SNAME, S.STATUS, SP.PNO, SP.OTY
      FROM   S, SP
      WHERE  S.SNO = SP.SNO
      AND    S.CITY = 'Warszawa');
```

Na tak zdefiniowanej perspektywie wykonamy następujące zapytanie:

```
SELECT DISTINCT LSSP.STATUS, LSSP.OTY
FROM   LSSP
WHERE  LSSP.PNO IN
      (SELECT P.PNO
       FROM   P
       WHERE  P.CITY <> 'Warszawa')
```

W jaki sposób mogłoby wyglądać zapytanie wynikowe (rozwinięcie zapytania z zastosowaniem definicji perspektywy) na odpowiedniej bazowej zmiennej relacyjnej?

Ćwiczenie 4.14. Jakie klucze posiada perspektywa LSSP z powyższego ćwiczenia?

Ćwiczenie 4.15. Przeanalizować wykorzystywany przez siebie produkt SQL. Czy istnieją teoretycznie poprawne zapytania, które nie działają w tym produkcie? Jeśli tak, określić dokładnie, jakie warunki muszą być spełnione, aby zapytanie nie zadziałało. Jakie jest wytłumaczenie producenta dotyczące braku pełnej obsługi tego typu zapytań? Uwaga: ćwiczenie dotyczy tylko zapytań odczytujących, nie modyfikujących.

Ćwiczenie 4.16. Przeanalizować wykorzystywany przez siebie produkt SQL. Jakie obsługuje operacje modyfikujące na perspektywach? Odpowiedzi udzielić w sposób jak najbardziej precyzyjny.

Ćwiczenie 4.17. Wykorzystując bazę danych dostawców i części zamiennych lub dowolną inną bazę danych, podać kilka przykładów ilustrujących stwierdzenie, że podział zmiennych relacyjnych na bazowe i wirtualne jest czysto umowny.

Ćwiczenie 4.18. Przeanalizować wykorzystywany przez siebie produkt SQL. W jaki sposób (na pewno znajdzie się kilka!) ten produkt narusza *regułę wymienności*?

Ćwiczenie 4.19. Przedstawić różnice pomiędzy perspektywami a migawkami. Czy SQL obsługuje migawki? Czy którykolwiek z produktów znanych Czytelnikowi je obsługuje?

Ćwiczenie 4.20. Co to jest „zmaterializowana perspektywa"? Dlaczego stosowanie tego terminu nie jest zalecane?

Ćwiczenie 4.21. Zdefiniować terminy *propozycja* i *predykat*. Przedstawić przykłady.

Ćwiczenie 4.22. Określić predykaty dla zmiennych relacyjnych P i SP z bazy danych dostawców i części zamiennych.

Ćwiczenie 4.23. Co należy rozumieć przez pojęcia *predykat* i *rozszerzenie*?

Ćwiczenie 4.24. Niech *DB* będzie dowolną bazą danych znaną Czytelnikowi, a *R* dowolną zmienną relacyjną w *DB*. Jaki jest predykat dla *R*? Uwaga: celem tego ćwiczenia jest zastosowanie fundamentalnych koncepcji omówionych w treści rozdziału do własnych danych i nauczenie Czytelnika myślenia o danych w takim ogólnym zakresie. Oczywiście to ćwiczenie nie ma uniwersalnego rozwiązania.

Ćwiczenie 4.25. Weźmy pod uwagę perspektywy LS i NLS z podrozdziału „perspektywy". Jakie są predykaty dla tych zmiennych relacyjnych? Czy gdyby te perspektywy były bazowymi zmiennymi relacyjnymi, zmieniłoby się brzmienie predykatów?

Ćwiczenie 4.26. Jaki jest predykat dla perspektywy LSSP z ćwiczenia 4.13?

Ćwiczenie 4.27. Wytłumaczyć *założenie zamkniętego świata*.

Ćwiczenie 4.28. Klucz jest zbiorem atrybutów, pusty zbiór jest również prawidłowym zbiorem, z tego można wywnioskować, że *pusty* klucz jest kluczem o pustym zbiorze atrybutów. Czy taki klucz ma jakiekolwiek praktyczne zastosowanie?

Ćwiczenie 4.29. Jaki jest predykat dla zmiennej relacyjnej stopnia zerowego? Czy to pytanie ma w ogóle sens? Uzasadnić odpowiedź.

Ćwiczenie 4.30. Każda zmienna relacyjna posiada wartość w postaci relacji. Czy twierdzenie odwrotne jest również prawdziwe? To znaczy, czy każda relacja jest wartością zmiennej relacyjnej?