Matematyka stosowana

Bazy danych

Zbigniew Jurkiewicz
http://www.mimuw.edu.pl/~zbyszek

Uniwersytet Warszawski, 2013

Streszczenie. Wprowadzenie do baz danych

Wersja internetowa wykładu:
http://mst.mimuw.edu.pl/lecture.php?lecture=bad

(może zawierać dodatkowe materiały)

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego.

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Skład w systemie LaTeX, z wykorzystaniem m.in. pakietów beamer oraz listings. Szablony podręcznika i prezentacji: Piotr Krzyżanowski; koncept: Robert Dąbrowski.

Spis treści

Celem wykładu jest zaznajomienie studentów z podstawowymi pojęciami i koncepcjami systemów baz danych. Omówimy podstawowe zasady modelowania danych i projektowania baz danych, relacyjny model danych, standardowy język baz danych SQL, teorię projektowania relacyjnych baz danych (normalizacja, rozkłady).

W drugiej części wykładu zajmiemy się innymi modelami baz danych: logicznym, obiektowym i semistrukturalnym. Na koniec podamy kilka uwag o fizycznych aspektach implementacji baz danych.

1. Wprowadzenie

Bazą danych nazywa się utrwalony zbiór danych, opisujących pewien fragment rzeczywistości i przeznaczonych do wspólnego wykorzystania przez różne programy. Jest to chyba najpopularniejsze wykorzystanie komputerów w życiu codziennym.

Przykładowo, w bazie danych banku mogą być gromadzone informacje na temat klientów, ich rachunków i operacji na tych rachunkach. Będą tam także informacje o udzielanych kredytach i ich spłatach.

Dane w bazie danych są zorganizowane w specyficzny sposób, zgodnie z modelem danych. Powinny odzwierciedlać rzeczywistość w sposób z nią zgodny.

1.1. Systemy informacyjne

Komputery obecnie służą przede wszystkim do przechowywania informacji i zarządzania nią, a nie do szybkich obliczeń numerycznych. Duże projekty informatyczne używające baz danych to często najbardziej złożone przedsięwzięcia podejmowane przez firmy. Na przykład systemy informowania kierownictwa (MIS – *Management Information Systems*) mogą obejmować:
— dane finansowe firmy;
— wyniki badań rynku, materiały marketingowe;
— informacje o aktualnych i potencjalnych klientach.

1.2. Zastosowania baz danych

Tradycyjne zastosowania baz danych obejmują tekie dziedziny jak:
— Obsługa wpłat i wypłat w banku.
— Rezerwacja biletów lotniczych.
— Przeglądanie katalogu bibliotek UW w internecie.
— Zakupy w supermarkecie.
— Kartoteki pracowników (kadry i płace).
— Księgowość

W miarę upowszechniania informatyki pojawiło się coraz więcej innych zastosowań, z zupełnie nowymi problemami:
— Multimedialne bazy danych: obrazy, muzyka, filmy.
— Geograficzne systemy informacyjne (*Geographic Information Systems* — GIS).
— Systemy analizy danych (*Online Analytical Processing* — OLAP) wykorzystujące *hurtownie danych*.
— Poszukiwanie prawidłowości w danych (*Data mining*).
— Naukowe i medyczne bazy danych
— Wyszukiwarki informacji: Google, Amazon itp.

Bazy danych charakteryzują się przy tym pewną specyfiką, wyróżniającą je wśród innych działów informatyki.

Do opisywania operacji na bazie danych używa się ograniczonych języków programowania — jest to chyba jedyna znacząca dziedzina, gdzie używa się w praktyce języków programowania o mocy słabszej niż maszyna Turinga.

Ale za to występują specyficzne problemy związane z
— optymalizacją wykonania zapytań dla wielkich zbiorów danych;
— synchronizacją wielu transakcji wykonywanych równocześnie

1.3. Model danych

Przed skonstruowaniem bazy danych powinniśmy zapoznać się ze strukturą informacji, które zamierzamy w niej przechowywać. Służą do tego *modele danych*, pozwalające opisać własności danych w sposób ścisły (często sformalizowany) z wykorzystaniem języka matematyki. Przykłady takich modeli to
— relacje (odpowiadające tabelom) w modelu relacyjnym
— drzewa lub grafy w modelach semistrukturalnych

Oprócz opisywania struktury danych model powinien także określać dozwolone operacje na danych oraz sposób nakładania ograniczeń (więzów) zapewniających poprawność bazy danych.

1.4. Przykład relacji

Przykładowa relacja może mieć postać następującej tabeli

Narty

model	producent
Cool Minx	Atomic
Jewel Cristal	Salomon

1.5. Schemat bazy danych

Strukturę danych i powiązania między nimi opisuje się w schemacie bazy danych. *Schemat relacji* podaje nazwę relacji i listę jej atrybutów. Dodatkowo można podać typy atrybutów

Przykłady:
— Narty(model,producent)
— Narty(model:string,producent:string)

Schemat bazy danych obejmuje schematy wszystkich relacji zawartych w bazie danych. W ten sposób *baza danych* jest po prostu kolekcją relacji.

Popatrzmy na zalety relacji:
— Bardzo prosty model
— Często intuicyjnie (zwłaszcza wśród biurokratów ;-) pasuje do danych
— Znane własności matematyczne
— Stanowi podstawę języka SQL

Pora na przykład nieco większego schematu:

Narty(model, producent)
Wypożyczalnie(nazwa, adres, telefon)
Narciarze(nazwisko, adres, telefon)
Lubi(narciarz, narty)
Wypożycza(wypożyczalnia, narty, cena)
Korzysta(narciarz, wypożyczalnia)

Podkreśleniem zaznaczamy *klucz*: atrybut (czasem atrybuty) taki, że w żadnych dwóch wierszach tabeli nie może mieć takiej samej wartości

Klucz to szczególny przypadek *ograniczenia* (więzów), nakładanego na bazę danych aby zagwarantować pewne warunki poprawności.

1.6. Realizacja baz danych

Najprościej byłoby wczytać na początku całą bazę danych do pamięci operacyjnej. Niestety współczesne bazy danych często mają rozmiary przekraczające 100 GB, co raczej wyklucza taie podejście. Poza tym na systemach z 32-bitową przestrzenią adresową można byłoby bezpośrednio odwoływać się do 4 GB danych.

Ponadto wykluczyłoby to (lub co najmniej bardzo utrudniło) współbieżny dostęp do bazy danych. Niemniej w pewnych sytuacjach, dla małych specjalizowanych baz danych może to być czasem sensowna realizacja.

1.6.1. Realizacja na plikach

Kolejna możliwość to skorzystać z plików dyskowych. System plików Linuxa czy Macintosha to bardzo prosty rodzaj bazy danych. Dane przechowuje się na dysku w nazwanych kawałkach (*plikach*), nie posiadających żadnej wewnętrznej struktury.

Zalety to pozorna prostota zarządzania (cały mechanizm jest po prostu fragmentem systemu operacyjnego), wystarczy od czasu do czasu zachować kopię archiwalną dysku na CD.

Niestety są również wady. Co stanie się, gdy w trakcje wykonywania operacji na bazie danych nastąpi jakaś awaria (błąd dysku, zanik zasilania, niedozwolona operacja)?

Po naprawieniu zawartość plików zapewne będzie niespójna, zwłaszcza biorąc pod uwagę, że system nie zawsze zapisuje zmiany w plikach na dysku w takiej kolejności, w jakiej je wprowadzaliśmy. Często stosuje się tzw. opóźniony zapis, ponieważ pozwala to zwiększyć wydajność systemu.

1.6.2. Wady plików

To jednak dopiero początek kłopotów. Przyjrzyjmy się bliżej poniższemu scenariuszowi.

Załóżmy, że tabelę Narciarze zapisaliśmy w osobnym pliku, każdy wiersz w osobnej linii, rozdzielając pola na przykład znakiem pionowej kreski.

Pierwszy kłopot to modyfikacje

— Jeśli próbujemy zmienić adres jednego z narciarzy, a nowy adres jest dłuższy niż stary, to zamażemy (być może częściowo) jego telefon.
— A jeżeli jest dużo dłuższy, to może i następny wiersz.
— Wniosek: trzeba zapamiętać całą resztę pliku i zapisać na nowo.

Jeśli natomiast nowy adres będzie krótszy niż stary, to z kolei powstanie „dziura” — fragment pliku nie zawierający żadnej użytecznej informacji.

Popatrzmy teraz na następującą sytuację. Co się stanie, gdy *dwie* osoby *równocześnie* spróbują wykonać taką operację modyfikacji?

— Każda z nich zapamięta dotychczasową „resztę” pliku, a następnie obie zaczną wpisywać nową wartość.
— Być może obie zmiany wykonają się poprawnie.
— A być może jedna z nich zostanie utracona.
— Najgorzej będzie, gdy wpisywane zmiany zostana przemieszane tak, że żadna z nich nie będzie poprawna.

1.6.3. Synchronizacja dostępu

Potrzebujemy więc jakiegoś mechanizmu synchronizacji. Najprościej byłoby użyć *blokad*:
— Ten kto pierwszy zaczął modyfikować rezerwuje cały plik tylko dla siebie.
— Pozostali muszą czekać aż skończy.
Jest to świetne rozwiązanie, gdy współbieżność jest niewielka (a najlepiej, gdy wcale jej nie ma ;-).

Słabo sprawdza się jednak w przypadku systemu rezerwacji biletów lotniczych, gdy kilkadziesiąt żądań napływa równocześnie.

1.6.4. Zalety realizacji w bazie danych

Skorzystanie ze specjalizowanego *systemu zarządzania bazami danych* (ang. DBMS od *Data Base Management System*) ma w tej sytuacji szereg zalet.

— Abstrakcja od szczegółów fizycznej implementacji, niezależność programów od danych, zwłaszcza od sposobu ich przechowywania.
— Komunikacja na poziomie modelu danych — relacji, z użyciem specjalizowanego języka dostępu do danych: najczęściej SQL.
— Opis struktury bazy danych (schemat) także przechowywany w bazie jako tzw. *relacje katalogowe* stanowiące słownik bazy danych.
— Zawarte w *katalogu* informacje opisujące strukturę bazy danych nazywa się *metadanymi*

1.7. SQL

Podstawowym językiem komunikacji z relacyjnymi bazami danych jest SQL. Zawiera on zarówno konstrukcje do definiowania schematu danych jak i do operowania na zawartości bazy. Popatrzmy na dwa przykłady:
— Definiowanie tabeli

```
CREATE TABLE Narty (
  model VARCHAR(20) PRIMARY KEY,
  producent VARCHAR(20)
);
```

— Usuwanie definicji tabeli

```
DROP TABLE Narty;
```

1.8. System zarządzania bazami danych

Realizacja wydajnych baz danych nie jest prosta. Dlatego potrzebujemy wyspecjalizowanego narzędzia: *systemu zarządzania bazami danych* (DBMS, od *Data Base Management System*).

W takim systemie baza danych zorganizowana jest tak, aby ułatwić modyfikacje i współbieżną pracę.
— Do przyśpieszenia wyszukiwania można tworzyć indeksy.
— Do synchronizacji zadań wielu użytkowników używa się mechanizmu *transakcji*.

Dodatkowe korzyści z używania DBMS to:
— Kontrola nadmiarowości („redundancji”)
— Poufność, ograniczanie niepowołanego dostępu
— Trwała pamięć dla obiektów i struktur danych programów
— Wdrażanie więzów integralności

— Archiwizacja i odtwarzanie
— Automatyczne uruchamianie akcji wywoływane zmianą zawartości bazy

1.9. Transakcje

Transakcje to proste lub złożone operacje na bazie danych, charakteryzowane często skrót ACID od angielskich nazw wymaganych cech:
— A*tomicity* czyli niepodzielność: transakcja ma być wykonana w całości albo wcale.
— C*onsistency* czyli spójność: transakcja przeprowadza bazę danych z legalnego stanu w (być może inny) legalny stan.
— I*solation* czyli izolacja: gdy dwie transakcje są przeprowadzane jednocześnie, ich działania nie mogą na siebie wpływać.
— D*urability* czyli trwałość: jeśli transakcja zakończy się pomyślnie, to jej wyniki nie mogą zostać utracone z powodu awarii systemu.

1.9.1. Role

Typowe role związane z bazami danych to:
— Administratorzy baz danych (*DataBase Administrators* — DBA)
— Projektanci baz danych
— Użytkownicy
  — doraźni
  — zaawansowani
— Programiści
— Analitycy systemów (często nazywani administratorami danych).

1.10. Laboratorium

W czasie zajęć w laboratorium będziemy używać publicznie dostępnego DBMS Postgres.

1.10.1. Logowanie do Postgresa

Należy zalogować się do Linuxa na jakąś maszynę w laboratorium.

Chcąc pracować interakcyjnie logujemy się do Postgresa pisząc:

```
psql -h labdb bd nasz-login
```

`psql` to program do doraźnego dostępu i administrowania bazami danych w Postgresie używający SQL. *nasz-login* oznacza login w systemie Linux.

Zostaniemy poproszeni o hasło. Początkowe hasło jest proste i **musi** być zmienione jak najszybciej. Proszę *nie* używać swoich haseł z Linuxa, ponieważ w pewnych okolicznościach mogą być one widoczne (wprawdzie tylko dla administratorów, ale zawsze to już coś). Po wprowadzeniu poprawnego hasła powinniśmy zobaczyć prompt

```
bd=>
```

1.10.2. Zmienianie hasła

W odpowiedzi na prompt wpisujemy

```
ALTER USER nasz-login WITH PASSWORD 'nowe-hasło';
```

Każde polecenie SQL należy kończyć średnikiem.

Przypominam, że SQL nie rozróżnia (poza napisami) dużych i małych liter. Konwencjonalnie będę używał dużych liter do wyróżniania słów kluczowych SQL.

1.10.3. Polecenia `psql`

Oprócz instrukcji SQL możemy wpisywać polecenia `psql`. Polecenia `psql` rozpoczynają się lewym ukośnikiem (*backslash*em). Polecenie `\q` służy do kończenia pracy.

Inne przydatne polecenia dają dostęp do interakcyjnej pomocy. `\h` z nazwą instrukcji SQL (np. `\h select` lub `\h create table`) podaje krótki opis składni. Aby dostać listę poleceń `psql`, należy napisać `\?`.

Podczas pracy działa „historia" — poprzednie polecenia można przywoływać strzałkami pionowymi i edytować.

1.10.4. Tworzenie tabeli

W `psql` można wykonywać dowolne instrukcje SQL. Można założyć tabelę używając instrukcji

```
CREATE TABLE nazwa-tabeli (
  lista-atrybutów-i-ich-typów
);
```

Tekst instrukcji można wprowadzać w jednej lub kilku liniach (prompt w kolejnych zmieni się wtedy na `bd`), dopóki nie wpiszemy średnika kończącego instrukcję.

Przykład tworzenia tabeli:

```
CREATE TABLE Zwierz (
  waga int,
  imie char(10)
);
```

1.10.5. Klucz główny

Każda tabela powinna mieć zadeklarowany klucz główny:

```
CREATE TABLE nazwa-tabeli (
  ...,
  id typ PRIMARY KEY,
  ...
);
```

Klucz główny może składać się z kilku kolumn:

```
CREATE TABLE nazwa-tabeli (
  atrybuty i ich typy,
  PRIMARY KEY (a,b,c)
);
```

1.10.6. Wstawianie wierszy

Po utworzeniu tabeli można wstawiać do niej wiersze. Najprościej robi się to instrukcją `INSERT`:

```
INSERT INTO nazwa-tabeli
  VALUES(wartość, ...);
```

Kolejność wartości powinna odpowiadać kolejności kolumn w deklaracji tabeli, np. aby wstawić wiersz `(10, 'Kropka')` do tabeli `Zwierz` piszemy

```
INSERT INTO Zwierz VALUES(10, 'Kropka');
```

1.10.7. Przeszukiwanie tabeli

Wiersze tabeli można obejrzeć instrukcją:

```
SELECT *
FROM nazwa-tabeli;
```

Program `psql` wyświetla wyszukane wiersze w postaci tabelki:

```
SELECT * FROM test;

 waga  | imie
-------+------------
 10    | Kropka
(1 row)
```

1.10.8. Usuwanie tabeli

Tabelę usuwamy instrukcją

```
DROP TABLE nazwa-tabeli;
```

Czasami trzeba użyć modyfikatora `CASCADE`

```
DROP TABLE test CASCADE;
```

Jeżeli chcemy zmienić definicję tabeli, przed wczytaniem zmienionej definicji należy usunąć tabelę.

1.10.9. Informacje o tabelach w bazie danych

Informacje o utworzonych tabelach są trzymane w słownikowych tabelach systemowych, przede wszystkim w tabeli `pg_tables`. Można wyszukać nazwy wszystkich swoich tabel podając zapytanie:

```
SELECT tablename
FROM pg_tables
WHERE tableowner = nasz-login;
```

Listę kolumn tabeli można też obejrzeć używając polecenia `psql`:

```
\d nazwa-tabeli
```

1.10.10. Wczytywanie instrukcji SQL z pliku

Zamiast wpisywać instrukcje SQL interakcyjnie można umieścić je w pliku i następnie załadować ten plik poleceniem

```
\i nazwa-pliku
```

na przykład

```
bd=> \i foo.sql
```

1.10.11. Wczytywanie obcych danych

PostgreSQL pozwala wczytywać i wypisywać dane z tabeli używając *plików separowanych*. Służy do tego instrukcja COPY:

```
COPY tabela TO 'plik';
```

Domyślnym separatorem jest znak tabulacji, ale można to zmienić:

```
COPY tabela FROM 'plik' USING DELIMITERS '|';
```

Ponieważ jednak w instrukcji `COPY` plik musi znajdować się w katalogu dostępnym dla procesów serwera bazy danych, zwykły użytkownik na ogół nie będzie mógł z niego skorzystać. Zamiast nazwy pliku można jednak podać `stdin` lub `stdout`, co spowoduje czytanie ze standardowego wejścia i pisanie na standardowe wyjście (w środowisku użytkownika)

```
COPY tabela TO stdout;

COPY tabela FROM stdin USING DELIMITERS '|';
```

Można wtedy przekierować wejście lub wyjście programu takiego jak `psql`, np.

```
psql -h labdb bd >gatunki.txt
bd=> COPY Gatunki TO stdout USING DELIMITERS '|';
bd=> \q
```

2. Algebra relacji

Algebra relacji jest modelem teoretycznym do opisywania semantyki relacyjnych baz danych, zaproponowanym przez T. Codda, twórcę koncepcji relacyjnych baz danych. Jest to algebra, w której dziedzinę stanowią relacje. Zmienne występujące w wyrażeniach tej algebry odpowiadają pojedynczym relacjom.

Operatory algebry relacji zostały dobrane tak, aby odpowiadały typowym operacjom występującym w zapytaniach podczas wyszukiwania informacji z tabel w bazie danych.

Tak określona algebra miała być *językiem zapytań* (*query language*) dla relacyjnych baz danych.

2.1. Relacje

Relacje w algebrze relacji reprezentujemy ich nazwami. Z nazwą każdej relacji jest związany jej schemat — ciąg nazw atrybutów (odpowiadających kolumnom modelowanej tabeli), np.
— $R(A, B, C)$
— `Student(indeks,imię,nazwisko)`
Nazwy atrybutów w schemacie relacji muszą być różne, dlatego w literaturze schemat jest czasem definiowany jako *zbiór* nazw atrybutów, a nie ciąg.

Ponieważ relacje miały być abstrakcyjnym modelem tabel, ich elementy nazywa się często *krotkami* — odpowiadają one wierszom tabel z bazy danych.

2.2. Operacje

Zestaw operacji jest spory. Obejmuje po pierwsze typowe operacje teoriomnogościowe: sumę zbiorów ($\cup$), iloczyn zbiorów ($\cap$) i różnicę zbiorów ($-$). Wymaga się, aby oba argumenty miały ten sam schemat atrybutów. Jeśli nazwy atrybutów różnią się, to należy użyć operacji przemianowania (o czym za chwilę).

Iloczyn kartezjański ($R \times S$) także jest zdefiniowany klasycznie. Ponieważ jednak argumenty mogą mieć atrybuty o tych samych nazwach, nazwy kolumn w schemacie wynikowym trzeba czasem poprzedzać nazwami relacji, z których pochodzą, np. dla relacji $R(A, B)$ i $S(B, C)$ schematem ich iloczynu kartezjańskiego będzie $R \times S(A, R.B, S.B, C)$, tak jak w poniższym przykładzie

R1 =

A	B
1	2
3	4

R2 =

B	C
5	6
7	8
9	10

R1 × R2 =

A	R1.B	R2.B	C
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

Lepiej jednak w takiej sytuacji użyć przemianowania.

2.2.1. Selekcja

Oprócz operacji teoriomnogościowych w algebrze relacji określono kilka operacji (a właściwie rodzin operacji) specyficznych dla niej. Pierwsza z nich to *selekcja* (wybór) $\sigma_{warunek}(R)$. Zgodnie z nazwą wybiera ona z relacji tylko te krotki, dla których jest spełniony podany warunek.

Przypuśćmy, że w mamy relację `Zwierzaki` o następującej postaci:

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50
Lis	Fufu	6,35
Krokodyl	Czako	75,00

Wartością wyrażenia $\sigma_{gatunek='Papuga'}$`Zwierzaki` jest:

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50

2.2.2. Rzutowanie

Podobna do selekcji jest operacja *rzutowania* (projekcji) $\pi_{kolumna_1,\ldots,kolumna_n}(R)$: z relacji wybieramy tylko podane kolumny. Zwróćmy uwagę, że mogłoby to prowadzić do utworzenia relacji, w której niektóre wiersze byłyby takie same. Ponieważ jednak relacje są zbiorami, więc takie duplikaty są automatycznie eliminowane, tak jak w poniższym przykładzie dla wyrażenia $\pi_{gatunek,waga}$`Zwierzaki`:

gatunek	waga
Papuga	3,50
Papuga	5,35
Lis	6,35
Krokodyl	75,00

Ze względów praktycznych warto zdefiniować uogólnione rzutowanie, w którym oprócz nazw kolumn dozwolone są dowolne wyrażenia oparte na kolumnach, np. arytmetyczne. Trzeba je tylko wtedy nazwać:

$$A + B \to C$$

Ponadto dopuszcza się, aby pewne kolumny wystąpiły wielokrotnie, wymagane jest jednak przemianowanie jak powyżej.

A oto przykład: dane jest relacja R

R =

A	B
1	2
3	4

Wartością wyrażenia $\pi_{A+B\to C,A,A\to A1}$R będzie

C	A	A1
3	1	1
7	3	3

Wspominaliśmy wcześniej o operacji *przemianowania* $\rho_S(R)$: nie powoduje ona żadnych zmian w zawartości relacji, lecz służy jedynie
— do zmiany nazwy relacji: $\rho_S(R)$
— lub zmiany nazw jej atrybutów $\rho_{R(X,Y,Z)}R$,
— a czasem jednego i drugiego: $\rho_{S(X,Y,ZX)}R$.

Złączenie $R \underset{\theta}{\bowtie} S$: podobne do iloczynu kartezjańskiego, ale łączy się ze sobą tylko pary wierszy spełniające podany warunek

— $R \underset{\theta}{\bowtie} S = \sigma_\theta(R \times S)$
— θ oznacza dowolny warunek na kolumny łączonych relacji, np. $A < C$.
— Złączenie theta, w którym warunek jest prostą równością pary atrybutów, nazywa się *złączeniem równościowym*.
— Pojęcie porzuconej krotki (*dangling tuple*): wiersza z jednej z relacji, do którego nie pasuje żaden wiersz z drugiej relacji.

— Zwierzaki

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50
Lis	Fufu	6,35
Krokodyl	Czako	75,00

Gatunki

nazwa	kontynent
Papuga	Ameryka
Lis	Europa
Krokodyl	Afryka

— Zwierzaki $\underset{gatunek=nazwa}{\bowtie}$ Gatunki

gatunek	imię	waga	nazwa	kontynent
Papuga	Kropka	3,50	Papuga	Ameryka
Papuga	Lulu	5,35	Papuga	Ameryka
Papuga	Hipek	3,50	Papuga	Ameryka
Lis	Fufu	6,35	Lis	Europa
Krokodyl	Czako	75,00	Krokodyl	Afryka

— Notacja: $R \bowtie S$.
— Łączone relacje muszą mieć co najmniej jedną wspólną kolumnę o tej samej nazwie.
— Warunkiem złączenia jest równość dla wszystkich par atrybutów o tych samych nazwach.
— W wyniku zostaje tylko jedna kolumna z pary kolumn o tych samych nazwach.

— Zwierzaki

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50
Lis	Fufu	6,35
Krokodyl	Czako	75,00

Gatunki

gatunek	kontynent
Papuga	Ameryka
Lis	Europa
Krokodyl	Afryka

— Zwierzaki ⋈ Gatunki

gatunek	imię	waga	kontynent
Papuga	Kropka	3,50	Ameryka
Papuga	Lulu	5,35	Ameryka
Papuga	Hipek	3,50	Ameryka
Lis	Fufu	6,35	Europa
Krokodyl	Czako	75,00	Afryka

— Pozwala na nazywanie relacji wynikowych: $\rho_{RS(A,B,X,C,D,E)}(R \times S)$.
— Uproszczone notacja: $R1(A1, B, X, C, D, E) := (R \times S)$.

2.3. Wyrażenia

— Ponieważ jest to algebra, więc operacje można składać otrzymując wyrażenia złożone.
— Równoważność wyrażeń można wykorzystać przy optymalizacji, zastępując dane wyrażenie równoważnym mu, lecz bardziej efektywnym.

— Zwierzaki

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50
Lis	Fufu	6,35
Krokodyl	Czako	75,00

— Znajdź pary zwierzaków (imiona) tego samego gatunku

$$\pi_{Z1.imie,Z2.imie}(\rho_{Z1}Zwierzaki \underset{\substack{Z1.gatunek=Z2.gatunek \wedge \\ Z1.imie<Z2.imie}}{\bowtie} \rho_{Z2}Zwierzaki)$$

— Zgodnie z matematyczną definicją relacji jako zbioru utożsamia się jednakowe krotki (powstające np. podczas rzutowania).
— Można rozszerzyć tę algebrę na wielozbiory, dopuszczając powtórzenia.
— Powstaje jednak problem odpowiedniej semantyki dla operacji iloczynu i różnicy teoriomnogościowej.
— Intuicyjnie zdefiniowane rozszerzenia operacji na wielozbiory zastosowane do relacji dają relacje z wyjątkiem sumy, która dla dwóch relacji może dać wielozbiór.
— Przestają zachodzić niektóre prawa algebry relacji, np.

$$(R \cup S) - T = (R - T) \cup (S - T)$$

— Operator eliminacji powtórzeń $\delta(R)$.

— Operator grupowania z ewentualną agregacją

$$\gamma_{A,MIN(B)\rightarrow MinB}(R)$$

— Zauważmy, że

$$\gamma_{A_1,\ldots,A_n}(R) = \sigma(R)$$

jeśli A_i to wszystkie atrybuty R.

— Operator sortowania $\tau_{C,B}(R)$.
— Nie jest to operator algebry relacji ani wielozbiorów, lecz ewentualnej algebry list, dlatego powinien być zewnętrznym operatorem wyrażenia!

— naturalne: $R \overset{\circ}{\bowtie} S$
  — jako wypełniaczy brakujących wartości w dołączonych kolumnach używa się $\perp$;
— lewostronne: $R \overset{\circ}{\bowtie}_L S$
  — brane są tylko porzucone krotki z pierwszego argumentu;
— prawostronne: $R \overset{\circ}{\bowtie}_R S$;
— wersje theta powyższych (z warunkiem u dołu).

— Zwierzaki

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50
Lis	Fufu	6,35
Krokodyl	Czako	75,00

Gatunki

gatunek	kontynent
Papuga	Ameryka
Lis	Europa
Krokodyl	Afryka
Krowa	Europa

— Zwierzaki $\bowtie$ Gatunki

gatunek	imię	waga	kontynent
Papuga	Kropka	3,50	Ameryka
Papuga	Lulu	5,35	Ameryka
Papuga	Hipek	3,50	Ameryka
Lis	Fufu	6,35	Europa
Krokodyl	Czako	75,00	Afryka
Krowa	$\perp$	$\perp$	Europa

2.4. Zastosowania algebry relacji

— Zapisywanie zapytań (np. modelowanie semantyki)
— Nakładanie ograniczeń na poprawność bazy danych (*więzy*). Przykłady:

$$R \cap S = \emptyset \quad \text{(styl równościowy)}$$
$$R \cap S \subseteq \emptyset \quad \text{(styl teoriomnogościowy)}$$

— Integralność referencyjna

$$\pi_{klucz\text{-}zewnętrzny}(R) \subseteq \pi_{klucz}(S)$$
$$\pi_{klucz\text{-}zewnętrzny}(R) - \pi_{klucz}(S) = \emptyset$$

— Zależności funkcyjne

$$A \to B: \qquad \sigma_{R.A=R1.A \wedge R.b \neq R1.B}(R \times \rho_{R1}(R)) = \emptyset$$

2.5. Zadania

Ćwiczenie 2.1. Dane są dowolne relacje $R(x)$, $S(x)$, $T(x)$ algebry relacji oraz wyrażenia (zapytania)

$$Q_1: \quad (R \cup S) - T$$
$$Q_2: \quad (R - T) \cup (S - T)$$

Które z poniższych stwierdzeń są prawdziwe
1. Q_1 i Q_2 dają ten sam wynik
2. Odpowiedź na Q_1 może mieć mniej elementów niż odpowiedź na Q_2
3. Q_1 i Q_2 mogą dać inne wyniki.

Rozwiązanie. Pierwsze.

Ćwiczenie 2.2. Dane są relacje R i Q, każda zawierająca n krotek. Relacja $R \bowtie Q$ ma
1. co najmniej n krotek
2. co najwyżej n^2 krotek
3. zawsze $2n$ krotek

Rozwiązanie.
1. nie
2. tak
3. nie

3. Język SQL — część 1

SQL jest to język wysokiego poziomu do komunikacji z bazami danych (ściślej: z systemami zarządzania bazami danych). Język ten zorientowany jest na operowanie zbiorami i jak gdyby opisywanie wyniku. Podajemy „co ma być zrobione", a nie „jak to zrobić". To DBMS sam określa „najlepszy" sposób wykonania polecenia. Dzięki temu możliwa staje się dość wyrafinowana *optymalizacja wykonywania zapytań*.

3.1. Przykładowa baza danych

W naszych przykładach używać będziemy „zoologicznej" bazy danych o następującym schemacie

Gatunki(nazwa,kontynent,groźny,chroniony)
Zwierz(imię,gatunek,wiek,waga)
Potrawa(nazwa,koszt)
Jada(gatunek,potrawa,ile)

Podkreślone atrybuty oznaczają klucze główne.

3.2. Zapytania

Do zadawania zapytań służy tylko jedno polecenie: `SELECT`. Pełna jego składnia jest złożona, dlatego obejrzymy na razie uproszczoną postać

```
SELECT jakie atrybuty
FROM z jakich tabel
WHERE jakie warunki muszą spełniać wybrane
  wiersze
```

Zaczniemy od prostego zapytania: „Jak nazywają się lwy?" (czyli jakie są ich imiona).

```
SELECT imie
FROM Zwierz
WHERE gatunek = 'lew';
```

Wynikiem tego zapytania będzie

imie
Kocio
Puszek
...

Najprostszą realizację tego zapytania można opisać następująco:

1. Weź tabelę podaną we frazie FROM.
2. Wybierz wiersze używając warunku z frazy WHERE (selekcja).
3. Wybierz tylko kolumny wskazane frazą SELECT (rzutowanie).

Pierwsze bardziej formalne podejście do semantyki operacyjnej mogłoby wyglądać tak

1. Wprowadzamy *zmienną krotkową* (np. nazywającą się tak, jak tabela), przebiegającą po kolejnych wierszach (krotkach) tabeli.
2. Sprawdzamy czy „bieżący" wiersz spełnia warunek z frazy WHERE.
3. Jeśli tak, obliczamy wyrażenia we frazie SELECT używając tego wiersza i dopisujemy nowy wiersz do wyniku.

3.2.1. Fraza SELECT

Fraza SELECT musi być pierwszą frazą zapytania. Oprócz nazw kolumn i wyrażeń nad nimi można w niej używać dodatkowo specjalnego symbolu `*` oznaczającego „wszystkie atrybuty relacji". Na przykład aby wynik zawierał wszystkie kolumny tabeli piszemy:

```
SELECT *
FROM Zwierz
WHERE gatunek = 'lew';
```

i otrzymujemy

imie	gatunek	wiek	waga
Kocio	lew	4	120
Puszek	lew	7	87
...	...	...	...

3.2.2. Fraza WHERE

Fraza WHERE jest opcjonalna, jej pominięcie oznacza, że wynik będzie zawierał odpowiedniki wszystkich wierszy źródłowej tabeli.

W warunkach umieszczanych we frazie WHERE można umieszczać typowe wyrażenia złożone zawierające

— Operatory arytmetyczne: +, -, *, /.
— Operatory porównywania: =, <>, <, >, <=, >=.
— Spójniki logiczne: AND, OR, NOT.

na przykład:

```
SELECT imie
FROM Zwierz
WHERE gatunek = 'lew' AND wiek > 4;
```

3.2.3. Wartości puste: NULL

W bazie danych często przechowujemy niekompletną informację. Czasem jest to zamierzone, na przykład w tabeli z ocenami studentów z różnych przedmiotów pewne kolumny mogą nie zawierać ocen, ponieważ nie odbyły się jeszcze egzaminy.

W innych przypadkach informacja nie jest znana, na przykład może nie być wiadomo, kto jest producentem komputera otrzymanego w ramach darowizny, choć niewątpliwie taki producent istnieje.

We wszystkich takich sytuacjach używa się w SQL wyróżnionej wartości `NULL`, umieszczając ją w odpowiednich kolumnach w danym wierszu.

Należy bardzo uważać na wartości NULL w warunkach. Logika dla warunków w SQL jest *trójwartościowa*: **true**, **false**, **unknown**. Jakiekolwiek normalne porównanie z wartością NULL daje wynik **unknown**, podobnie jest dla operacji arytmetycznych (inaczej mówiąc NULL jest „zaraźliwe").

Dlatego do sprawdzania wartości pustych należy używać specjalnych operatorów porównania IS NULL i IS NOT NULL.

3.2.4. Inne warunki

W podanych dalej warunkach elementarnych SQL operator można zwykle poprzedzać dodatkowo symbolem `NOT` (oczywiście z odwróceniem znaczenia).

Wyrażenie

wartość `IN` *zbiór*

bada przynależność *wartości* do *zbioru*. Zbiór może byc podany jawnie przez wyliczenie elementów

$(elem_1, ..., elemx_n)$

lub jako zapytanie wewnętrzne.

Wyrażenie

wartość `BETWEEN` *a* `AND` *b*

sprawdza, czy *wartość* należy do podanego przedziału domkniętego [a,b].

Wyrażenie

napis `LIKE` *wzorzec*

oznacza dopasowanie napisu do wzorca. We wzorcu (który także powinien być napisem) `%` oznacza dowolny ciąg znaków, zaś _ dowolny pojedynczy znak.

3.2.5. Inne wyrażenia

W klauzuli `SELECT` można używać wyrażenia

```
CASE WHEN warunek THEN wartość
     ...
     ELSE wartość
END
```

na przykład do kategoryzacji wartości

```
SELECT imie, nazwisko,
       CASE WHEN wiek IS NULL THEN 'nie wiadomo'
            WHEN wiek >= 18 THEN 'dorosły'
            ELSE 'nieletni'
       END
FROM ...
```

Do wartości NULL przyda się wyrażenie

`COALESCE(`*v1*`,` *v2*`)`.

Jego wartością jest *v1*, o ile nie jest `NULL`, w przeciwnym razie *v2*.

3.2.6. Eliminacja powtórzeń

SQL nie jest algebrą relacji, dlatego powtórzenia nie są automatycznie eliminowane z tabel. Do usuwania potwórzeń z wyników zapytań służy modyfikator `DISTINCT` we frazie SELECT

```
SELECT DISTINCT kontynent
FROM Gatunki;
```

Przy braku DISTINCT każdy kontynent zostałby wypisany wielokrotnie.

Operacje teoriomnogościowe `UNION`, `INTERSECT` i `EXCEPT` automatycznie eliminują powtórzenia, o ile nie zastosowano modyfikatora `ALL`

```
(SELECT gatunek
 FROM Zwierz
 WHERE waga > 100)
UNION ALL
(SELECT gatunek
 FROM Zwierz
 WHERE wiek > 10);
```

3.3. Tworzenie tabel

Do tworzenia tabel służy konstrukcja CREATE TABLE

```
CREATE TABLE nazwa(
  kolumna typ więzy-spójności,
  ...
);
```

Deklarujemy w niej kolumny tabeli, ich typy oraz dodatkowe ograniczenia poprawności.
Tabele usuwamy konstrukcją DROP TABLE `DROP TABLE nazwa;`
Generalnie w SQL wszelkie polecenia tqorzenia obiektów w bazie danych mają postać

CREATE *typ-obiektu nazwa* ...;

zaś polecenia usuwania

DROP *typ-obiektu nazwa*;

Utworzymy teraz niektóre tabele, których używaliśmy w przykładach zapytań. Zaczniemy od tabeli Gatunki

```
CREATE TABLE Gatunki (
  nazwa VARCHAR(30) PRIMARY KEY,
  kontynent VARCHAR(25),
  grozny BOOLEAN,
  chroniony BOOLEAN
);
```

Tworzenie tabeli Zwierz wygląda podobnie

```
CREATE TABLE Zwierz (
  imie VARCHAR(20) PRIMARY KEY,
  gatunek VARCHAR(30) REFERENCES Gatunki,
  wiek INTEGER,
  waga NUMERIC
);
```

3.3.1. Typy danych

Większośc typów w SQL to typy znane z innych języków programowania. Najbardziej przydatne to

— `CHAR(n)`: napis o długości n znaków,
— `VARCHAR(n)`, `VARCHAR2(n)`: napis o zmiennej długości nie przekraczającej n znaków,
— `NUMERIC(n)`, `NUMERIC(n, m)`: liczba o zadanej precyzji
— `INTEGER`, `INT`: liczba całkowita,
— `DATE`: data,
— `BOOLEAN`: wartość logiczna (prawda lub fałsz).

SQL zawiera wiele funkcji do konwersji między typami, ale oprócz nich można używać uniwersalnej konstrukcji `CAST`, np.

```
CAST(wczoraj AS TEXT)
```

zamieni wartość typu `DATE` z kolumny `wczoraj` na tekst.

3.3.2. Tabele robocze

Tabele zdefiniowane przez CREATE TABLE są trwale przechowywane w bazie danych i aby się ich pozbyć należy użyć DROP TABLE. Czasem jednak potrzebujemy tabeli tylko na czas obliczeń do przechowywania wyników częściowych. Służa do tego tabele robocze.

Są one widoczne *tylko* w sesji, w której zostały utworzone i znikają automatycznie po jej zakończeniu. Tworzymy je używając rozszerzonej postaci CREATE TABLE

```
CREATE TEMPORARY TABLE nazwa(
  ...
);
```

albo po prostu zapisując wynik zapytania

```
SELECT ... INTO TEMPORARY TABLE nazwa
FROM ...
...;
```

3.3.3. Więzy spójności

Terminem *więzy spójności* określa się elementarne warunki na poprawność bazy danych, zapisane składniowo w definicji tabeli lub innego obiektu. Podaje się je po nazwie i typie kolumny której dotyczą. Jeśli natomiast ograniczenie dotyczy kilku kolumn (np. klucz główny składający się z dwóch kolumn), zapisujemy je osobną deklaracją.

Deklaracja `NOT NULL` zakazuje umieszczania wartości pustych w kolumnie, której dotyczy. Przy wstawianiu wierszy zawsze należy podać jakąś wartość dla tej kolumny.

Deklaracja `UNIQUE` mówi, że wartości w tej kolumnie (lub kolumnach) są unikalne — żadne dwa wiersze nie mogą zawierać tej samej wartości.

Deklaracja `PRIMARY KEY` określa klucz główny. Równoważna jest deklaracji `UNIQUE NOT NULL`, no i może wystąpić tylko raz.

Deklaracja `CHECK` *warunek* umożliwia podanie wyrażenia SQL, określającego dodatkowe ograniczenia na poprawność danych.

I wreszcie deklaracja `REFERENCES` *nazwa-tabeli*. Określa ona kolumnę jako *klucz obcy* (ang. *foreign key)*. Klucz obcy zawiera odwołanie do innej tabeli przez umieszczenie w deklarowanej kolumnie wartości z klucza głównego do tamtej tabeli. Oznacza to, że dozwolone są tylko takie wartości, które występują jako klucze w tabeli, do której się odwołujemy. Takie ograniczenie jest czasem nazywane „więzami integralności referencyjnej".

Oczywiście dla klucz obcego trzeba zadbać o to, żeby typy kolumn, po których łączymy, były zgodne.

W deklaracji tabeli Zwierz wystąpiła deklaracja kolumny gatunek

```
...
  gatunek VARCHAR(30) REFERENCES Gatunki,
...
```

Oznacza to, że kolumna `gatunek` może zawierać tylko wartości z klucza głównego tabeli `Gatunki`, czyli z kolumny `nazwa` w tej tabeli.

W deklaracjach odwołań do innych tabel można dodatkowo określić wymagane zachowanie w przypadku usuwania lub modyfikacji wartości klucza obcego w jego macierzystej tabeli, np.:

```
... ON DELETE SET NULL,
... ON UPDATE CASCADE
```

Pierwsza z tych deklaracji mówi, że w przypadku usunięcia danej wartości klucza obcego z macierzystej tabeli należy jego wartość zastąpic przez NULL. Przy braku takiej deklaracji usunięcie tej wartości spowodowałoby błąd.

Druga deklaracja mówi, że w przypadku zmiany wartości klucza na inną należy zmienioną wartość umieścić również w miejscach odwołań (tzw. „kaskada modyfikacji". Wartość klucza jest zresztą zmieniana niezwykle rzadko, na ogół jako wynik błędu przy wpisywaniu.

3.3.4. Cykliczne zależności referencyjne

W przypadku kluczy obcych należy zwracać uwagę na ewentualne cykle. Powstający problem jest odmianą powszechnie znanego problemu „kury i jajka" (co było pierwsze?). Poniższe dwa polecenia zawsze powodują błąd

```
CREATE TABLE Kura (
  imie CHAR(8) PRIMARY KEY,
  jajko INTEGER REFERENCES Jajko
);
CREATE TABLE Jajko (
  numer INTEGER PRIMARY KEY,
  kura CHAR(8) REFERENCES Kura
);
```

W przypadku deklaracji klucza obcego wymaga się, aby istniała tabela, do której się odwołujemy. Tutaj podczas tworzenia tabeli `Kura` system napotyka frazę odnoszącą się do tabeli `Jajko`, która jeszcze nie istnieje!

Zmiana kolejności poleceń też nic nie pomoże, ponieważ analogiczne odwołanie występuje w tabeli Jajko.

3.4. Polecenia modyfikacji schematu

Aby poradzić sobie z tym problemem musimy sięgnąć do poleceń modyfikacji schematu bazy danych. Utworzymy najpierw tabele bz określania więzów kluczy obcych:

```
CREATE TABLE Kura (
  imie CHAR(8) PRIMARY KEY,
  jajko INTEGER
);
CREATE TABLE Jajko (
  numer INTEGER PRIMARY KEY,
  kura CHAR(8)
);
```

Nowe więzy do istniejącej tabeli można dodać poleceniem:

```
ALTER TABLE tabela
  ADD CONSTRAINT nazwa ograniczenie;
```

W naszym przypadku potrzebne będą dwa polecenia:

```
ALTER TABLE Kura ADD CONSTRAINT Kura_Jajko
  FOREIGN KEY (jajko) REFERENCES Jajko(numer)
  INITIALLY DEFERRED DEFERRABLE;
ALTER TABLE Jajko ADD CONSTRAINT Jajko_Kura
  FOREIGN KEY (kura) REFERENCES Kura(imie)
  INITIALLY DEFERRED DEFERRABLE;
```

Fraza `INITIALLY DEFERRED DEFERRABLE` żąda od SQL *odroczenia sprawdzania więzów* do chwili zatwierdzenia transakcji, np. aby wstawić ('Czubatka', 1) do tabeli `Kura` i (1, 'Czubatka') do tabeli `Jajko` użyjemy:

```
INSERT INTO Kura VALUES ('Czubatka', 1);
INSERT INTO Jajko VALUES (1, 'Czubatka');
COMMIT;
```

Bez opóźnionego sprawdzania więzów nie można byłoby wstawić żadnego wiersza do tabel `Kura` ani `Jajko`, ponieważ już pierwszy `INSERT` naruszałby więzy, chyba że dopuścimy wartości puste (`NULL`) w kolumnie klucza obcego.

3.4.1. Usuwanie więzów

Nazwane więzy można usuwać poleceniem:

```
ALTER TABLE tabela DROP CONSTRAINT nazwa;
```

Należy zawsze pamiętać, aby przed usunięciem tabel zawsze przedtem usunąć ręcznie więzy cykliczne, w przeciwnym razie SQL nie pozwoli na usunięcie tabel.

```
ALTER TABLE Jajko DROP CONSTRAINT Jajko_Kura;
ALTER TABLE Kura DROP CONSTRAINT Kura_Jajko;
DROP TABLE Jajko;
DROP TABLE Kura;
```

W DROP TABLE można użyć modyfikatora CASCADE, ale nie zawsze radzi on sobie z takimi sytuacjami.

3.5. Funkcje agregujące

Funkcje agregujące są przeznaczone do obliczania wartości parametrów „statystycznych", takich jak średnia czy suma, dotyczących całej tabeli (lub wybranych grup wierszy), a nie pojedynczych wierszy.

```
SELECT AVG(waga)
FROM Zwierz
WHERE gatunek = 'Niedźwiedź';
```

oblicza średnią wagę niedźwiedzi, czyli średnią z wartości w kolumnie `waga` dla wierszy zawierających `'Niedźwiedź'` w kolumnie `gatunek`.

Standardowe funkcje agregujące to `AVG`, `COUNT`, `MAX`, `MIN` i `SUM`. Z wyjątkiem wyrażenia `COUNT(*)` wartości puste są pomijane.

Funkcji `COUNT` warto przyjrzeć się dokładniej. Zlicza ona wiersze i często ma argument zastępczy `*`:

```
SELECT COUNT(*)
FROM Zwierz
WHERE gatunek = 'Niedźwiedź';
```

Jeśli zamiast * jej argumentem jest nazwa kolumny, to nie są liczone wiersze, zawierające tej kolumnie wartości puste.

Natomiast poprzedzenie takiego argumentu dodatkowo modyfikatorem DISTINCT spowoduje obliczenie, ile *różnych* wartości występuje w tej kolumnie, na przykład

```
SELECT COUNT(DISTINCT gatunek)
FROM Zwierz;
```

policzy, ile mamy różnych gatunków w tabeli Zwierz.

3.6. Grupowanie

Dzielenie wierszy na grupy frazą GROUP BY ułatwia równoczesne obliczanie parametrów statystycznych dla wybranych podzbiorów wierszy. Zapytanie

```
SELECT gatunek, AVG(waga)
FROM Zwierz
GROUP BY gatunek;
```

poda średnią wagę dla każdego gatunku w tabeli Zwierz.

Zauważmy, że eliminację powtórzeń można rzeprowadzić grupowaniem zamiast używać DISTINCT:

```
SELECT kontynent
FROM Gatunki
GROUP BY kontynent;
```

Warunkiem frazy WHERE można ograniczyć grupowanie tylko do wybranych wierszy

```
SELECT gatunek, AVG(waga)
FROM Zwierz, Gatunki
WHERE Zwierz.gatunek = Gatunki.gatunek
  AND kontynent = 'Afryka'
GROUP BY gatunek;
```

Czasem jednak chcemy dla całych grup, a nie pojedynczych wierszy. Służy do tego fraza HAVING. Polecenie

```
SELECT gatunek, AVG(waga)
FROM Zwierzaki, Gatunki
WHERE Zwierzaki.gatunek = Gatunki.gatunek
GROUP BY gatunek
HAVING COUNT(*) > 2;
```

odrzuci wszystkie grupy zawierające mniej niż 3 elementy.

Uwaga: Chcąc znaleźć najwyższą średnią po grupach, nie możemy po prostu napisać MAX(AVG(*wyrażenie*)) [Oracle akceptuje taką konstrukcję, ale nie jest to zgodne ze standardem SQL]. Można jednak napisać proste zapytanie zagnieżdżone:

```
SELECT MAX(średnia_z_ocen)
FROM (SELECT AVG(ocena) AS średnia_z_ocen
      FROM Oceny
      GROUP BY indeks) Średnie;
```

3.7. Zadania

Baza danych biblioteki jest oparta na następującym schemacie:

```
CREATE TABLE Ksiazki (
  nrk NUMERIC(5) PRIMARY KEY,
  tytul VARCHAR(20) NOT NULL,
  autor VARCHAR(25),
  wydawca VARCHAR(20),
  rok_wyd NUMERIC(4),
  data_zakupu DATE,
  cena NUMERIC(6,2));

CREATE TABLE Czytelnicy (
  nrcz NUMERIC(4) PRIMARY KEY,
  nazwisko VARCHAR(20) NOT NULL,
  imie VARCHAR(15) NOT NULL,
  zawod VARCHAR(15));

CREATE TABLE Wypozyczenia (
  nrk NUMERIC(5) NOT NULL REFERENCES Ksiazki,
  nrcz NUMERIC(4) NOT NULL REFERENCES Czytelnicy,
  data_wyp DATE NOT NULL,
  data_zwr DATE,
  PRIMARY KEY(nrk, nrcz, data_wyp));
```

Zapisz w SQL następujące zapytania:

Ćwiczenie 3.1. Która obecnie wypożyczona książka jest najdłużej trzymana i przez kogo (może być kilka takich książek — należy podać wszystkie)? Podaj autora, tytuł oraz imię i nazwisko czytelnika.

Rozwiązanie.

Ćwiczenie 3.2. Kto czytał najdroższą książkę wydaną przed 1989 rokiem (może być kilka takich książek — podaj dla wszystkich imię i nazwisko czytelnika)?

Rozwiązanie.

Ćwiczenie 3.3. Podaj numery katalogowe i tytuły pięciu (lub więcej, jeśli jest „remis") książek o największej liczbie wypożyczeń.

Rozwiązanie.

W bazie danych znajdują się tabele:

```
CREATE TABLE Osoby (
  id NUMERIC(5) PRIMARY KEY,
  nazwisko VARCHAR(20) NOT NULL,
  imie VARCHAR(15) NOT NULL,
  miasto VARCHAR(20));

CREATE TABLE Agenci (
  id NUMERIC(4) PRIMARY KEY,
  imie VARCHAR(15) NOT NULL,
  nazwisko VARCHAR(20) NOT NULL);

CREATE TABLE Ubezpieczenia (
  polisa NUMERIC(5) PRIMARY KEY,
  data_od DATE NOT NULL,
  data_do DATE NOT NULL CHECK (data_do > data_od),
  wariant CHAR(1),
  ag_id NUMERIC(4) NOT NULL REFERENCES(Agenci),
```

```
    os_id NUMERIC(5) NOT NULL REFERENCES(Osoby));
```

Zapisz w SQL następujące polecenia:

Ćwiczenie 3.4. Jaka jest maksymalna liczba ubezpieczeń jednej osoby?

Rozwiązanie.

```
SELECT MAX(ile)
FROM (SELECT COUNT(*) AS ile
      FROM Ubezpieczenia
      GROUP BY os_id) maksy;
```

Ćwiczenie 3.5. Który agent nie zawarł żadnego ubezpieczenia? Podaj jego imię i nazwisko (może być kilku takich agentów).

Rozwiązanie.

```
SELECT imie,nazwisko
FROM Agenci LEFT JOIN Ubezpieczenia ON id=ag_id
GROUP BY id,imie,nazwisko
HAVING COUNT(polisa)=0;
```

Ćwiczenie 3.6. Który klient ma najwięcej ubezpieczeń? Podaj imię, nazwisko i liczbę ubezpieczeń. Uwaga: może być kilku o tej samej liczbie — wtedy należy podać wszystkich!

Rozwiązanie.

4. Język SQL — część 2

4.1. Wstawianie wierszy

Nowe wiersze do tabeli wstawiamy poleceniem INSERT

```
INSERT INTO tabela
VALUES (wartość,...);
```

na przykład

```
INSERT INTO Gatunki
VALUES ('krowa','Europa',FALSE,FALSE);
```

Wartości można podawać w innej kolejności niż w definicji tabeli, wtedy jednak trzeba po nazwie tabeli podać w nawiasach listę nazw kolumn w odpowiedniej kolejności. Dzięki temu można nie podawać wartości dla kolumn dopuszczających NULL.

```
INSERT INTO Gatunki(nazwa,chroniony,kontynent)
VALUES ('krowa',FALSE,'Europa');
```

Daty zapisujemy wyrażeniem

```
DATE '2008-03-11'
```

chociaż dla kolumny typu DATE podanie napisu jest też akceptiowane i powoduje wykonanie automatycznej konwersji.

Czas podajemy wyrażeniem

```
TIME '15:00:07'
```

zaś łączny zapis daty i czasu ma postać

```
TIMESTAMP '2008-03-11 15:00:10'
```

4.2. Modyfikacja wierszy

Zawartość niektórych kolumn może ulegać zmianom. Takich zmian dokonuje się poleceniem UPDATE

```
UPDATE tabela
SET kolumna = wartość, ...
WHERE warunek;
```

Zmiana dotyczy wszystkich wierszy, dla których jest spełniony warunek. Jeśli warunek został pominięty, zmiana dotyczy *wszystkich wierszy* w tabeli!

Czasem zmiana służy do uzupełnienia lub korekty informacji

```
UPDATE Gatunki
SET grozny = FALSE
WHERE nazwa = 'krowa';
```

Często dokonanie zmian jest wymuszone przez zmiany w otaczającym świecie

```
UPDATE Zwierz
SET wiek = wiek + 1;
```

4.3. Usuwanie wierszy

Usuwanie wierszy to najprostsza operacja modyfikacji

```
DELETE FROM tabela
WHERE warunek;
```

Usuwane są wszystkie wiersze, dla których jest spełniony warunek. Jeśli warunek został pominięty, usuwane są *wszystkie wiersze w tabeli*!

```
DELETE FROM Gatunki
WHERE nazwa = 'krowa';
```

4.4. Zapytania na kilku tabelach

Czasem poszukiwana informacja znajduje się w kilku tabelach. Aby zapytanie dotyczyło kilku tabel, należy je podać we frazie FROM

Jeśli w używanych tabelach powtarzają się nazwy atrybutów, należy użyć notacji

tabela.atrybut

Zaczniemy od przykładu zwykłego złączenie dwóch relacji. Zapytajmy o imiona wszystkich zwierzaków pochodzących z Afryki.

```
SELECT imie
FROM Zwierz, Gatunki
WHERE Gatunki.nazwa = Zwierz.gatunek
  AND kontynent = 'Afryka';
```

Bardzo niepraktyczna semantyka formalna mogłaby wyglądać tak

1. Utwórz iloczyn kartezjański wszystkich tabel z frazy FROM.
2. Używając otrzymanego wyniku, postępuj jak dla zapytania opartego na pojedynczej tabeli.

Warto też obejrzeć uproszczoną semantykę operacyjną

— Z każdą tabelą z frazy FROM związujemy ukrytą zmienną krotkową (zwykle nazywającą się tak jak tabela i wskazującą na „bieżący" wiersz).
— Przy dwóch tabelach:
  — Dla każdej wartości zmiennej krotkowej z pierwszej tabeli znajdujemy wszystkie „pasujące" wiersze z drugiej tabeli, przechodząc po niej jej zmienną krotkową.
  — Dla każdego znalezionego wiersza łączymy go z wierszem z pierwszej tabeli i przetwarzamy jak dla pojedynczej tabeli (tzn. sprawdzamy warunek itd.).
— Analogicznie postępujemy przy większej liczbie tabel wprowadzając kolejne zagnieżdżone pętle iteracyjne.

4.4.1. Jawne zmienne krotkowe

Czasami w zapytaniu chcemy dwukrotnie użyć tej samej tabeli. Aby móc odróżniać te wystąpienia tabeli, we frazie FROM po nazwie tabeli można umieścić zmienną krotkową.

Oczywiście można tak zrobić dla dowolnej tabeli, np. aby w innych frazach używać krótszej nazwy.

Pora na przykład samozłączenia — złączenia tabeli z nią samą. Mamy napisać zapytanie spełniające następujące warunki:

— Podaj wszystkie pary zwierzaków (ich imiona) tego samego gatunku.
— Unikaj par identycznych typu (Kropka,Kropka).
— W ramach pary zachowaj porządek alfabetyczny, tzn. (Kropka,Puszek) a nie (Puszek,Kropka).

```
SELECT z1.imie,z2.imie
FROM Zwierz z1, Zwierz z2
WHERE z1.gatunek = z2.gatunek
  AND z1.name < z2.name;
```

4.5. Podzapytania

Nawiasowane wyrażenie SELECT (*podzapytanie*) można umieszczać we frazach WHERE i FROM (a w pewnych sytuacjach także we frazie SELECT, ale nie próbujcie robić tego w domu).

We frazie FROM po podzapytaniu *musi* wystąpić zmienna krotkowa, a wynik podzapytania traktowany jest w zewnętrznym zapytaniu tak, jak dodatkowa tabela (a ściślej perspektywa). Przykład: podaj imiona najcięższych zwierzaków z każdego gatunku

```
SELECT imie
FROM Zwierz, (SELECT gatunek,MAX(waga) AS maks
              FROM Zwierz
              GROUP BY gatunek) mwg
WHERE Zwierz.gatunek = mwg.gatunek
  AND waga = maks;
```

Jeśli używamy podzapytania we frazie WHERE w zwykłym porównaniu, to powinno ono zwracać pojedynczą wartość. Przykład: podaj imiona zwierzaków, które ważą najwięcej

```
SELECT imie
FROM Zwierz
WHERE waga = (SELECT MAX(waga)
              FROM Zwierz);
```

Podzapytania bardzo często używa się w połączeniu z operatorem IN. Może ono wtedy zwracać wiele wartości, ale może mieć tylko jedną kolumnę w wyniku.

Podaj imiona wszystkich zwierzaków pochodzących z Afryki.

```
SELECT imie
FROM Zwierz
WHERE gatunek IN (SELECT nazwa
                  FROM Gatunki
                  WHERE kontynent = 'Afryka');
```

4.6. Złączenia

— Warunki łączące dla złączeń można zapisywać we frazie FROM
— Jeszcze raz imiona wszystkich zwierzaków pochodzących z Afryki.

```
SELECT imie
FROM Zwierz JOIN Gatunki
  ON (Gatunki.nazwa = Zwierz.gatunek)
WHERE kontynent = 'Afryka';
```

Standard SQL-92 podaje operatory złączeń do używania we frazie FROM:

`T1 CROSS JOIN T2`	Iloczyn kartezjański
`T1 NATURAL JOIN T2`	Złączenie naturalne (równościowe po kolumnach o tych samych nazwach)
`T1 INNER JOIN T2`	Zwykłe złączenie
`T1 LEFT OUTER JOIN T2`	Złączenia zewnętrzne
`T1 RIGHT OUTER JOIN T2`	
`T1 FULL OUTER JOIN T2`	

Po takim wyrażeniu dodatkowo podajemy

`USING (`*kolumna*`, ...)`	nazwy kolumn po których łączymy
`ON` *warunek*	warunek ograniczający na złączenie

4.7. Perspektywy

W SQL relacja to tabela lub *perspektywa*.

Tworzenie perspektywy

```
CREATE VIEW nazwa [(atrybut ...)] AS zapytanie;
```

na przykład

```
CREATE VIEW GatunkiAfryki AS
  SELECT *
  FROM Gatunki
  WHERE kontynent = 'Afryka';
```

Usuwanie perspektywy

```
DROP VIEW nazwa;
```

Uproszczona semantyka operacyjna dla zapytań z perspektywami:

— Nazwę perspektywy we frazie `FROM` w zapytaniu zastępuje się relacjami, na podstawie których ją utworzono.

— Warunki z definicji perspektywy dołącza się do warunków zapytania.

Modyfikacje są dozwolone tylko dla *aktualizowalnych* perspektyw:

— zbudowanych na podstawie pojedynczej tabeli, oraz

— obejmujących wszystkie atrybuty nie posiadające wartości domyślnych.

Warto zabronić operacji wstawiania i modyfikacji perspektywy dających wiersze, które nie bedą należeć do perspektywy, używając podczas jej tworzenia frazy `WITH CHECK OPTION`:

```
CREATE VIEW GatunkiAfryki AS
  SELECT *
  FROM Gatunki
  WHERE kontynent = 'Afryka'
  WITH CHECK OPTION;
```

4.8. Kursory

— Kursory służą do krokowego przeglądania wyniku zapytania. Używa się ich przede wszystkim w procedurach składowanych.

— Kursor deklarujemy poleceniem `DECLARE`

```
DECLARE kursor_gatkon CURSOR FOR
SELECT gatunek, kontynent FROM Gatunki;
```

— Z kursora można pobierać kolejne wiersze używając polecenia

`FETCH [` *kierunek* `] [` *ile* `] IN | FROM` *cursor*

— Parametr *kierunek* definiuje kierunek pobierania wierszy i może być równy
  `FORWARD` pobiera następne wiersze (zachowanie domyślne)
  `BACKWARD` pobiera poprzednie wiersze.
— Parametr *ile* określa, ile wierszy należy pobrać i może być równy
  n Liczba ze znakiem podająca liczbę wierszy do pobrania. Podanie liczby ujemnej zamienia znaczenie `FORWARD` i `BACKWARD`.
  `ALL` Wszystkie pozostałe wiersze.
  `NEXT` Równoważny podaniu `1`.
  `PRIOR` Równoważny podaniu `-1`.

Aby pobrać dwa kolejne wiersze z kursora

```
=> FETCH 2 FROM kursor_gatkon;

gatunek      kontynent
----------------------
lew          Afryka
bóbr         Europa
```

Po kursorze można się cofać

```
FETCH -1 FROM kursor_gatkon;
```

lub

```
-- Pobierz poprzedni wiersz
FETCH BACKWARD 1 FROM kursor_gatkon;
```

— Kursor można pozycjonować bez pobierania wierszy poleceniem
  `MOVE [` *kierunek* `] [` *ile* `] IN | FROM` *kursor*
  Znaczenie parametrów jest takie, jak dla `FETCH`.
— Aby przestawić kursor o 5 wierszy do przodu
  `MOVE 5 FROM kursor_gatkon;`
— Na zakończenie kursor należy zamknąć
  `CLOSE kursor_gatkon;`

— Uwaga: kursory działają tylko wewnątrz transakcji, czyli przed ich użyciem należy wykonać
  `BEGIN WORK;`
  (o ile nie jesteśmy wewnątrz otwartej transakcji), a potem (niekoniecznie natychmiast) zamknąć transakcję
  `COMMIT WORK;`
— Nie jest możliwa aktualizacja bieżącego wiersza kursora, trzeba używać niezależnego polecenia `UPDATE`.
— SQL92 nie zawiera polecenia `MOVE`, ale za to pozwala na absolutne pozycjonowanie kursora, co w PostgresSQL nie jest zrealizowane.

4.9. Asercje i dziedziny

— Nie występują nigdzie poza standardem. Składnia:
  `CREATE ASSERTION` *nazwa* `CHECK (`*warunek*`);`
— Przykład użycia:

```
CREATE ASSERTION DodatniaWaga
  CHECK (NOT EXISTS (SELECT * FROM Zwierz
                     WHERE waga < 0));
```

Służą do określania typów danych. Polecenie

```
CREATE DOMAIN AdresTyp AS
  VARCHAR(40) DEFAULT 'Nieznany';
```

tworzy nowy typ, którego można użyć wewnątrz `CREATE TABLE`

```
CREATE TABLE Studenci (
  indeks CHAR(6) PRIMARY KEY,
  imie VARCHAR(15) NOT NULL,
  nazwisko VARCHAR(15) NOT NULL,
  adres AdresTyp
);
```

Dziedzinę usuwamy poleceniem

```
DROP DOMAIN Adres;
```

4.10. Indeksy

— Polecenie `CREATE INDEX` definiuje nowy indeks dla podanej tabeli. Pojawiło się dopiero w SQL-99.

```
CREATE [ UNIQUE ] INDEX nazwa-indeksu ON tabela
  (kolumna [, ...])
  [ WHERE warunek ]
```

— Parametr `UNIQUE` powoduje sprawdzanie duplikatów w tabeli podczas tworzenia indeksu i przy każdej modyfikacji.
— Utworzony indeks będzie oparty na kluczu powstałym przez konkatenację podanych kolumn.
— Do usuwania indeksu służy polecenie `DROP INDEX`.

— Utworzymy indeks na kolumnie `kontynent` tabeli `Gatunki`

```
CREATE INDEX IndGat ON Gatunki(kontynent);
```

— Indeks może obejmować kilka kolumn.

```
CREATE INDEX IndKontChron
  ON Gatunki(kontynent,chroniony);
```

— Usuwanie indeksu:

```
DROP INDEX IndGat;
```

— Istnieje też inna postać definicji indeksu:

```
CREATE [ UNIQUE ] INDEX nazwa-indeksu ON tabela
  ( funkcja( kolumna [, ... ]) )
  [ WHERE warunek ]
```

— Służy ona do definiowania *indeksów funkcyjnych*, gdzie wartością klucza indeksowego jest wynik wywołania określonej przez użytkownika *funkcji*, której parametrami są podane kolumny indeksowanej tabeli.
— Przykładowo, użycie funkcji `upper(`*kolumna*`)` pozwoli podczas indeksowania ignorować rozróżnianie dużych i małych liter.

```
CREATE INDEX test1_idx ON test1 (upper(kol1));
```

— Wartość funkcji używanej w indeksie musi zależeć jedynie od jej argumentów. Podczas jej tworzenia należy ją oznaczyć jako ustaloną (*immutable*).

— Jeśli w definicji indeksu występuje klauzula `WHERE`, to powstanie *indeks częściowy*, zawierający pozycje tylko dla wierszy tabeli spełniających podany warunek.
— Na przykład w tabeli zamówień można by zdefiniować indeks tylko dla wierszy zawierających `'tak'` w kolumnie `zapłacono`.
— Wyrażenie w klauzuli `WHERE` może odwoływać się tylko do kolumn indeksowanej tabeli, nie wolno też używać podzapytań ani funkcji agregujących.

4.11. Sekwencje

— Służą do otrzymania kolejnych wartości dla kolumn typu całkowitego.
— Tworzenie

```
CREATE SEQUENCE sekw_kat
  INCREMENT BY 1 START WITH 1;
```

— Generowanie kolejnej wartości funkcją `nextval` (jej argumentem jest *nazwa* generatora):

```
SELECT nextval('sekw_kat');
```

— Wywołanie `nextval` można też umieścić we frazie `DEFAULT` definicji kolumny w poleceniu `CREATE TABLE`.

— Do otrzymania bieżącej wartości generatora sekwencji służy funkcja `curval`.
`SELECT curval('sekw_kat');`
zaś do ustawienia na konkretną wartość funkcja `setval`
`SELECT setval('sekw_kat', 12);`
— Zamiast umieszczać wywołanie `nextval` we frazie `DEFAULT`, można jako typ takiej kolumny podać `SERIAL`. Zostanie wtedy automatycznie utworzona sekwencja, a kolumna będzie w rzeczywistości typu INT4.

4.12. Varia

Zmiana hasła użytkownika

`ALTER USER` *nazwa* `PASSWORD` *'nowe-hasło'*`;`

4.13. Laboratorium: typy danych

4.13.1. Napisy

Wbudowane operacje na napisach

Najczęściej używane operacje to:
— `length(nap)` podaje długość napisu `nap` w znakach;
— `trim(nap)` zwraca napis `nap` z usuniętymi początkowymi i końcowymi spacjami. Warianty: `trim(BOTH, nap)`, `trim(LEADING, nap)`, `trim(TRAILING, nap)`.
— `substr(str,m,n)` zwraca fragment napisu `str`, rozpoczynający się od znaku o numerze `m` o długości `n` znaków. Parametr `n` można pomijać, otrzymamy wtedy całą resztę napisu. Oczywiście wynki jest napisem.
— `substring(kol FROM m FOR n)` — to samo.
— `rpad(kol,n[,znak])` zwraca napis w kolumnie `kol` uzupełniony na końcu spacjami do szerokości `n`. Opcjonalny trzeci argument podaje inny znak do wypełniania.

— `lpad(kol,n[,znak])` jak poprzednia, ale uzupełnia na początku.
— `lower(kol)` zamienia dużę litery w napisie na małe.
— `upper(kol)` zamienia małe litery w napisie na dużę.
— `initcap(kol)` ustawia pierwszą literę na dużą.
— `position(str1 IN kol)` szuka napisu `str1` w napisie `str2` i zwraca numer pierwszego znaku znalezionego wystąpienia napisu `str1`.
— `strpos(kol,pos)` — to samo.
— `str1 || str2` zwraca konkatenację napisów `str1` i `str2`.

Przypuśćmy, że w kolumnie student mamy zapisaną informację w postaci `'nazwisko imię'`, a chcemy odwrócić tę kolejność na postać `'imię nazwisko'`:

```
substr(student, position(' ' IN student) + 1)
|| ' '
|| substr(student, 1, position(' ' IN student))
```

4.13.2. Daty i czas

W Postgresie daty i czas są obsługiwane zgodnie ze standardem SQL2. Cztery podstawowe wbudowane typy to `DATE`, `TIME`, `TIMESTAMP` i `INTERVAL`. Typ `TIMESTAMP` obejmuje zarówno datę jak i czas.

Typów takich jak `DATE` używa się tak samo jak innych, na przykład do tworzenia kolumn tabeli

```
CREATE TABLE x(a int, b date);
```

Zewnętrzna reprezentacja dat

Przed wyświetleniem daty zamieniane są automatycznie na napis. Do konwersji używa się funkcji `to_char`, według ustalonego *formatu* domyślnego. U nas domyślnym formatem jest ISO, tzn. `'YYYY-MM-DD'`, na przykład

```
SELECT b FROM x;

B
----------
2004-04-01
```

Sposób wyświetlania daty można zmienić wywołując samemu `to_char` z własnym formatem

```
SELECT to_char(b, 'YYYY/MM/DD') AS b
FROM x;

B
----------
2004/04/01
```

Funkcja `to_char` ma składnię:

`to_char(`*data*`, '`*format*`')`

W *formacie* można używać rozmaitych specyfikacji, najpopularniejsze to:

MM	Miesiąc cyframi (np. 07)
MON	Skrócona nazwa miesiąca (np. JUL)
MONTH	Pełna nazwa miesiąca (np. JULY)
DD	Dzień cyframi (np. 24)
DY	Skrócona nazwa dnia tygodnia (np. FRI)
YYYY	Rok czterema cyframi (np. 2004)
YY	Dwie ostatnie cyfry roku (np. 04)

Przy wczytywaniu dat używa się funkcji `date`, zamieniającej napis na datę zgodnie z domyślnym formatem. Zwykle nie wywołuje się jej jawnie, ponieważ Postgres tam, gdzie jest wymagany argument typu data, automatycznie zamienia napis na datę wołając `date`, na przykład

```
insert into x values(99, '2004-05-31');
```

W innych sytuacjach trzeba jawnie wywoływać funkcję `date`.

Chcąc użyć innego formatu należy wywołać funkcje `to_date`:

```
INSERT INTO x
VALUES(99, to_date('2004/05/31', 'yyyy/mm/dd'));
```

Funkcja `to_date` ma składnię:

`to_date(`*napis*`, '`*format*`')`

gdzie dla formatu obowiązują te same opcje co dla `to_char`.

Domyślny format daty zmienia się instrukcją `SET DATESTYLE`, na przykład:

```
SET DATESTYLE TO SQL, DMY;
```

Dozwolone wartości pierwszego parametru to: `ISO`, `SQL`, `POSTGRES`, a drugiego: `EUROPEAN`, `US`, `NONEUROPEAN`.

Bieżąca data i czas

Standardowe zmienne `current_date` i `current_time` zwracają bieżącą datę i czas.

```
SELECT current_date AS "Bieżąca data", current_time AS "Teraz";

Bieżąca data   Teraz
----------------------------
2004-01-01     21:18:27
```

Zmienna `current_timestamp` podaje zarówno datę jak i czas.

Operacje na datach

Daty można porównywać standardowymi operatorami porównania `=`, `!=`, `>`, itp.

Daty można odejmować od siebie, Otrzymując wynik typu `TIMESPAN`.

Do dat można dodawać liczby lub odejmować je od nich, na przykład `current_date + 1` to dzień jutrzejszy.

Po przekształceniu na napis funkcją `to_char`, można na datach wykonywać wszelkie operacje dla napisów, na przykład

`to_char(`*date*`, 'DD-MON-YY') LIKE '%JUN%'`

zwraca prawdę jeśli mamy do czynienia z datą czerwcową.

4.13.3. Liczby

Dla liczbe określone są funkcje `abs`, `round` i `trunc`.

4.14. Zadania

Ćwiczenie 4.1. Dane jest tabela $R(a, b, c)$ oraz zapytania w SQL

Q_1: `SELECT DISTINCT a,b FROM R;`
Q_2: `SELECT a,b FROM R GROUP BY a,b;`

Które z poniższych stwierdzeń są prawdziwe
1. Q_1 i Q_2 dają ten sam wynik.
2. Odpowiedź na Q_1 może zawierać mniej wierszy niż odpowiedź na Q_2.
3. Q_1 i Q_2 mogą dać inne wyniki.

Rozwiązanie. Pierwsze.

Ćwiczenie 4.2. Dana jest tabela `Sprawdzian`

student	**kolokwium**	**egzamin**
A	45	NULL
B	NULL	90
C	100	80

Czy następujące zapytanie w SQL

```
SELECT student
FROM Sprawdzian
WHERE (kolokwium > egzamin AND egzamin > 75) OR kolokwium < 50
```

zwróci wiersz dla studenta
1. A
2. B
3. C

Rozwiązanie.
1. Tak
2. Nie
3. Tak

Ćwiczenie 4.3. Dana jest tabela R(A,B,C,D) oraz szkielet zapytania:

```
SELECT [...]
FROM R
GROUP BY a,b;
```

Które wyrażenie wolno wstawić w miejsce [...]?
1. MIN(c+d)
2. a,b
3. b,c

Rozwiązanie. Pierwsze i drugie.

5. Modelowanie danych

„All models are wrong, but some are useful."

George Box

Jak najlepiej budować *aplikacje*?

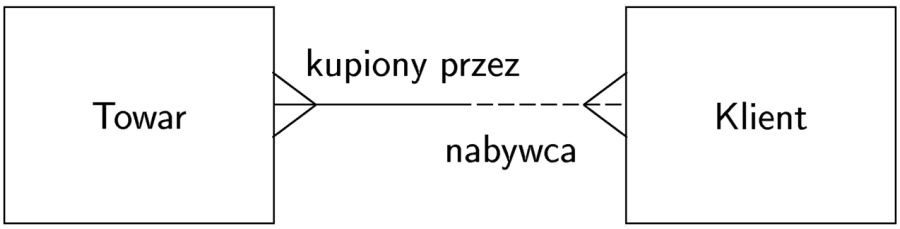
— Dobrze działający system to taki system, którego istnienie jest niezauważalne dla posługujacych się nim (przynajmniej dopóki się nie popsuje).
— Przykłady łatwo wskazać:
  — umywalka
  — winda

Przebieg projektu

1. Uzgodnić z użytkownikami kształt przyszłego systemu.
2. Opisać w postaci specyfikacji uzgodnione wymagania.
3. Zaprojektować sposób realizacji systemu.
4. Zrealizować system i wdrożyć go.

5.1. Diagramy związków/encji (ERD)

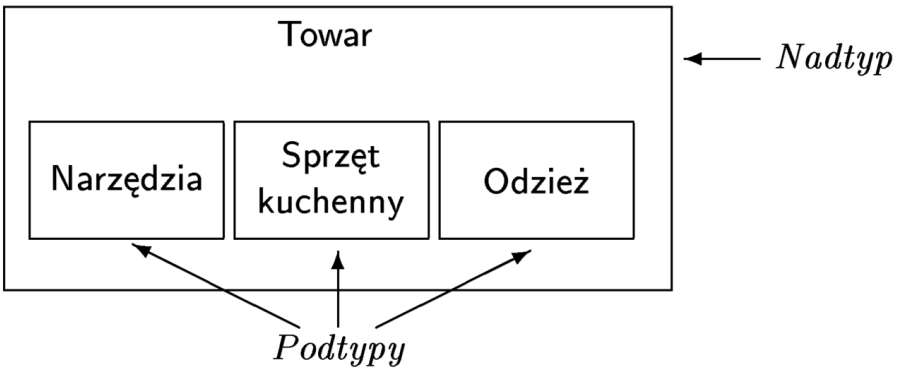
— Diagramy związków-encji, spopularyzowane przez Bachmana i Chena.
— Powinny zawierać związki pomiędzy danymi pamiętanymi, tzn. takimi, które nie mogą być wyprowadzone z innych danych.
— Obecnie coraz częściej używane tylko do modelowania baz danych podczas projektowania fizycznego.
— Dwa podstawowe składniki:
  — encje
  — związki
— Encje opisują (w uproszczony sposób) obiekty, wchodzące w zakres naszego zainteresowania.



Rysunek 5.1. Przykładowy diagram związków encji.

Konstrukcje obiektowe na ERD

— Niektóre encje mogą odpowiadać podklasom, np. towary sprzedawane przez pewną firmę (modelowane encją `Towar`) mogą dzielić się na sprzęt, oprogramowanie i materiały pomocnicze.
— Niektóre relacyjne bazy danych (np. PostgreSQL) pozwalają reprezentować takie hierarchie tabel, jednak podklasy powinny być rozłączne.



Rysunek 5.2. .

5.2. Projektowanie bazy danych

Kolejne kroki
1. Atrybuty i zależności, grupowanie
2. Odwzorowanie encji/obiektów i związków w relacje/tabele
3. Normalizacja i denormalizacja.
4. Strojenie bazy, definiowanie ścieżek dostępu.

Elementy projektowania
— Projektowanie obejmuje projekt logiczny bazy danych i projekt implementacji fizycznej.
— Projektowanie logiczne bazy danych polega na zdefiniowaniu tabel, określeniu ścieżek dostępu (*access paths*) dla tabel (np. indeksów) oraz dostosowaniu indeksów do potrzeb aplikacji.
  — Warto korzystać z perspektyw — upraszcza to zwłaszcza projektowanie formularzy i raportów w wielu narzędziach.
— Projektowanie implementacji fizycznej dotyczy rozmieszczenia plików bazy danych na dyskach, planów archiwowania i odtwarzania oraz integracji z narzędziami systemu operacyjnego.

5.3. Obiektowe podejście do modelowania

UML: obiektowy język modelowania
— UML (*Unified Modeling Language*) został zaprojektowany przez Boocha, Jacobsena i Rumbaugh.
— W 1997 roku Object Management Group przyjęła UML 1.1 jako swój standard przemysłowy.

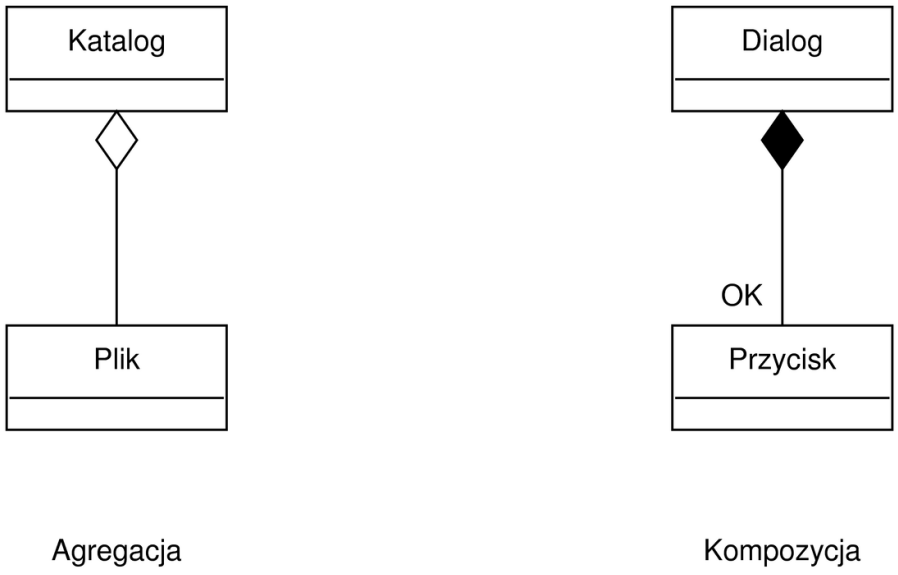
Diagramy klas

— Służą do modelowania struktury systemu.
— Używane w
  — modelowaniu dziedziny systemu (modelowanie biznesowe)
  — projektowaniu (w tym bazy danych), pojawiają się wtedy klasy „techniczne”
  — inżynierii odwrotnej (*reverse engineering*)

Klasy
— Podstawowym składnikiem diagramu klas są nazwane klasy.
— Najprostsza reprezentacja klasy nie zawiera nic więcej.
— Oprócz nazwy klasa może zawierać
  — atrybuty
  — metody.

Powiązania Rodzaje powiązań:
— Asocjacja
— Agregacja i kompozycja
— Dziedziczenie
— Zależność
— *refinement* — realizacja lub uszczegółowienie

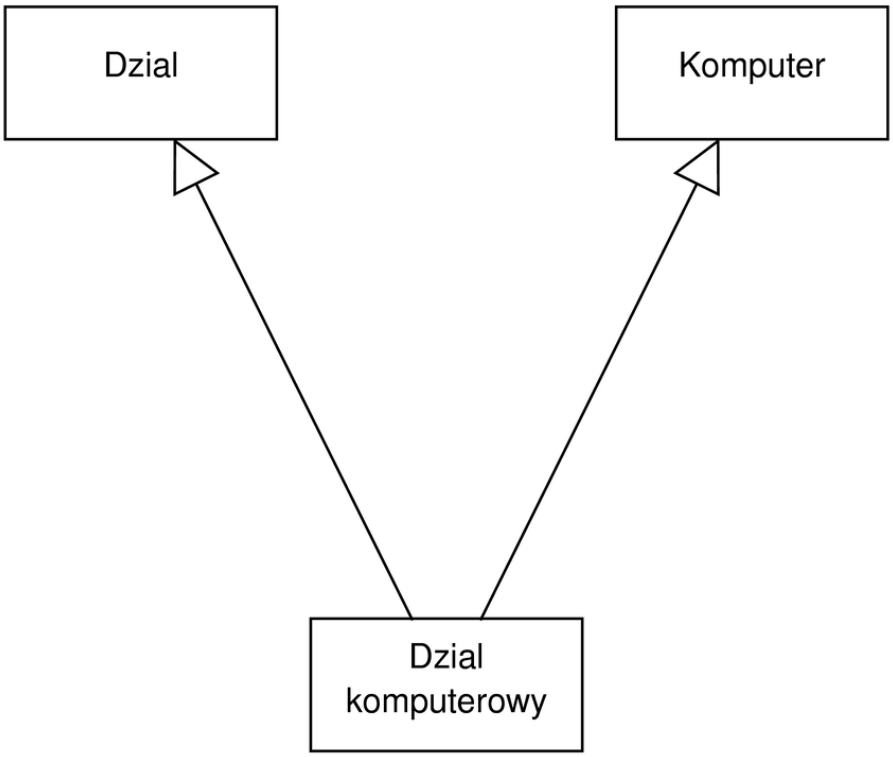
Klasa może być w relacji agregacji z wieloma klasami nadrzędnymi, natomiast w relacji kompozycji tylko z jedną nadrzędną.



Rysunek 5.3. .

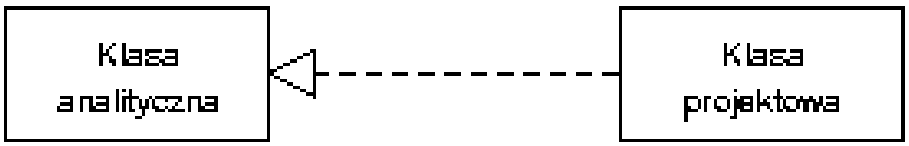
Zależności Zależność między klasami oznacza, że klasy nie mają powiązania, natomiast klasa zależna otrzymuje obiekt tej drugiej klasy np. jako parametr jednej ze swoich metod.

Dziedziczenie Przykład błędnego dziedziczenia:



Rysunek 5.4. .

Refinement
— Wiąże ze sobą dwa opisy tej samej rzeczy na różnych poziomach abstrakcji.
— W przypadku gdy wiąże typ z klasą realizującą go zwane jest *realizacją*.
— Może być użyte do modelowania różnych implementacji tej samej rzeczy.



Rysunek 5.5. .

Kiedy używać agregacji?
— Czy są jakieś operacje na całości, które są automatycznie stosowane do składowych?

Kiedy wybierać agregację, a kiedy dziedziczenie?
— Jeśli rodzaj obiektu (np. Student) może ulec zmianie (np. z dziennego na zaocznego) bez zmiany reszty atrybutów, to przy dziedziczeniu będzie to wymagało zmiany klasy obiektu!

Atrybuty i asocjacje Czasem mamy wątpliwości, czy w danej sytuacji użyć atrybutu czy asocjacji. Ogólna zasada jest następująca:

— Atrybuty służą do łączenia obiektów z *wartościami*. Wartość to element nie posiadający *identity*, np. liczba 1.
— Asocjacje łączą obiekty z obiektami.
Wartości zawsze mogą być zapisane bezpośrednio i odtworzone. Z obiektami jest trudniej, bo należy uwzględnić wszystkie związki.

5.4. Przypadki użycia

Model przypadków użycia
— Modelu tego używa się przede wszystkim do opracowania specyfikacji wymagań.
— Dwa podstawowe składniki: aktorzy i przypadki użycia.
— Model przypadków użycia wyznacza granice systemu.
  — Najczęściej są to granice budowanego systemu (aplikacji), jednak przypadki użycia służą także do modelowania przedsiębiorstwa (tzw. *przypadki biznesowe*).
  — Nie należy jednak mieszać tych dwóch podejść w jednym modelu.

Przypadki użycia
— Każdy przypadek to realizacja jakiejś rzeczywistej *potrzeby* użytkownika (aktora), nazwa przypadku powinna to uwzględniać, a nie opisywać czynności systemu. `Wyświetlanie wyników klasówki` to zła nazwa, lepszą będzie `Przeglądanie wyników klasówki`.
— Przypadki użycia powinny odpowiadać kolejnym fragmentom podręcznika użytkownika. Jeśli używamy pakietów do grupowania dużej liczby przypadków użycia, to każdy pakiet odpowiada rozdziałowi podręcznika.
— Scenariusze (opisy) przypadków użycia należy pisać z perspektywy użytkownika, a nie systemu.
— Nie należy tworzyć przypadków użycia zgrupowanych wokół „obiektów”, np. `Przetwarzanie informacji o studentach`. Mają one zwykle za dużo aktorów i zbyt długie specyfikacje.

5.5. Diagram stanów

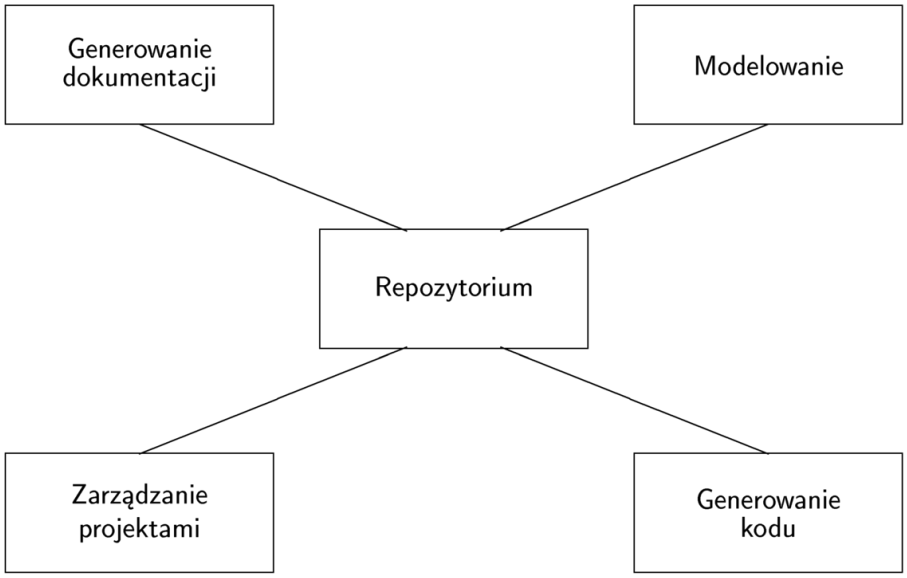
— Do opisywania historii klas korzysta się z sieci przejść, zwanych też diagramami stanów. Są one uogólnionymi automatami skończonymi.
— Sieć taka składa się z wierzchołków, reprezentujących stany klasy, oraz powiązań między nimi, odpowiadających przejściu z jednego stanu do drugiego.
— Diagram stanów opisuje historię życia obiektu danej klasy.
  — Jeśli operacje na obiekcie danej klasy mogą być wykonywane w dowolnej kolejności, to diagram ten jest najprawdopodobniej zbędny.
  — Jeśli jednak klasa przedstawia (*reifikuje*) przypadek użycia, proces, zarządcę interfejsu itp., wtedy kolejnośc kroków staje się istotna. Trzeba więc określić mechanizm dostarczania i obsługi zdarzeń.

Stany i przejścia
— *Stan* stanowi abstrakcję wartości zmiennych (parametrów itp.) obiektu taką, że ich wszystkie te kombinacje dają jednakową *jakościowo* odpowiedź na zdarzenia.
— *Przejścia* etykietuje się warunkami, których spełnienie powoduje wskazaną zmianę stanu. Zamiast warunków można też stosować zdarzenia, powodujące wskazaną zmianę stanu.
— Oprócz tego z poszczególnymi przejściami mogą być związane akcje sterujące, uaktywniające odpowiednie procesy.

Uwagi o diagramach stanów
— Przy bardziej złożonym zachowaniu stosuje się sieci zagnieżdżone, zwane też diagramami Harela. W sieciach takich każde przejście dotyczące stanu złożonego dotyczy wszystkich jego podstanów wewnętrznych.
— Modelując zachowanie obiektów staramy się pokazać dwie rzeczy: zmiany stanu oraz interakcje z innymi obiektami.
— Robiąc diagramy stanów dla klas trzeba pamiętać, że klasa musi mieć atrybuty lub związki, przez zmianę których realizuje zmianę stanu. Inaczej mówiąc, stany są reprezentowane pośrednio ich wartościami.

5.6. Narzędzia CASE

— Pakiet CASE to środowisko do modelowania i tworzenia aplikacji.
— Centralną częścią takiego pakietu jest wspólne repozytorium.
— Dzięki temu możliwa jest równoczesna wspólna praca nad wieloma projektami, z dzieleniem wspólnych fragmentów między nimi.
— Dotyczy to zarówno pojedynczych obiektów, jak i całych modeli.



Rysunek 5.6. .

Funkcje typowego narzędzia CASE
— rysowanie diagramów (obejmuje znajmość semantyki symboli i reguł poprawności)
— repozytorium modeli i ich elementów (np. gdy zmienimy nazwę klasy, powinno to być widoczne na wszystkich diagramach)
— wspieranie nawigacji po modelach
— możliwość pracy kilku użytkowników
— generowanie (szkieletu) kodu
— inżynieria odwrotna (*reverse engineering*)
— integracja z innymi narzędziami
— obsługa poziomów abstrakcji modelu

— wymiana modeli (eksport i import)

Realizacja narzędzi CASE
— Do zarządzania repozytorium używane są zwykle systemy relacyjnych baz danych, np. SQL Anywhere (Sybase).
— Możliwa jest często inżynieria odwrotna, tzn. wciąganie do modeli analitycznych i projektowych obiektów z już istniejących programów i baz danych.

5.7. Zadania

Ćwiczenie 5.1. Dane są dwie encje E_1 i E_2 oraz związek R między nimi. Podczas realizacji w relacyjnej bazie danych używającej SQL wprowadzenie encji pośredniczącej jest konieczne jeśli związek R jest
— 1–1
— 1–n
— m–n.

Rozwiązanie. m–n, czyli wiele-do-wielu

6. Teoria projektowania relacyjnych baz danych

6.1. Zależności funkcyjne

Zależność funkcyjna występuje wtedy, gdy po ustaleniu wartości pewnych atrybutów relacji wartości jakichś innych atrybutów tej relacji są jednoznacznie wyznaczone (unikalne).

Używa się notacji: $X \to Y$, gdzie X i Y są zbiorami atrybutów z relacji R.

Mówimy, że *zależność funkcyjną* $X \to Y$ *zachodzi* w R, jeśli każde dwa wiersze mające te same wartości atrybutów z X *muszą* mieć te same wartości atrybutów z Y.

Formalnie mozna to wyrazić następująco

$$(\exists f : X \to Y)(\forall (x, \ldots, y) \in R(X, \ldots, Y))\, y = f(x)$$

choć funkcja f nie jest oczywiście znana (albo czasem nie jest łatwa do obliczenia).

6.1.1. Rozbicie zależności

Zależność funkcyjną, której prawa strona zawiera kilka atrybutów

$$X \to A_1 A_2 \ldots A_n$$

można zastąpić zbiorem zależności o pojedynczych prawych stronach

$$X \to A_1, X \to A_2, \ldots, X \to A_n$$

Operacja ta jest oczywiście odwracalna.

Przykład: $A \to BC$ można zastąpić przez $A \to B$ i $A \to C$.

Uwaga: *Nie wolno* rozbijać *lewych* stron zależności!

6.1.2. Przykłady zależności

Jeśli w przykładowym ZOO każdy zwierzak nazywa się inaczej, to dla relacji Zwierz(imie,gatunek,waga,wiek) zachodzi zależność:

$$\text{imie} \to \text{gatunek}$$

Jeśli przyjęto (raczej naturalną) zasadę *„Żadne dwa wykłady nie odbywają się o tej samej godzinie w tej samej sali”*, to mamy zależność funkcyjną

$$\text{godzina sala} \to \text{wykład}$$

Jak określa się zależności dla danej relacji? Jedynie przez odpytanie użytkowników.

Nie należy natomiast liczyć na to, że wykryjemy je na podstawie zawartości tabeli w bazie danych. W takiej tabeli może być pozornie spełniona pewna zależność, która przestanie zachodzić za pięć minut (po dodaniu do tabeli nowych wierszy)! W ten sposób można jedynie upewnić się, że podana zależność *nie* zachodzi.

6.1.3. Zależności trywialne

Definicja 6.1. Zależność $X \to Y$ nazywamy *trywialną*, jeśli spełnione jest $Y \subseteq X$.

Uwaga 6.1. Zależności trywialne zachodzą *zawsze* i nie dostarczają żadnej informacji.

6.2. Klucze

Jednym z ważniejszych pojęć w projektowaniu relacyjnych baz danych jest pojęcie *klucza* relacji. Zaczniemy od pomocniczej definicji.

Definicja 6.2. Podzbiór N zbioru atrybutów relacji $R(A_1, A_2, \ldots, A_n)$ nazywamy jej *nadkluczem*, jeśli zachodzi zależność funkcyjna

$$N \to A_1 A_2 \ldots A_n$$

Czyli zbiór wszystkich atrybutów relacji na pewno jest jej nadkluczem.

Definicja 6.3. Podzbiór K zbioru atrybutów relacji $R(A_1, A_2, \ldots, A_n)$ nazywamy jej *kluczem*, jeśli jest on nadkluczem i żaden jego podzbiór nie jest nadkluczem.

Inaczej mówiąc, klucze relacji to jej minimalne nadklucze. Obejrzymuy teraz przykłady kluczy.

Niech w relacji Zwierz zachodzi zależność

$$\text{imie} \to \text{gatunek wiek waga}$$

— {imie,gatunek} jest nadkluczem w relacji Zwierz
— Nie jest jednak kluczem, bo {imie} też jest nadkluczem
— {imie} jest kluczem i nie ma innych kluczy

6.2.1. Skąd się biorą klucze?

Skąd się biorą klucze? Na ogół są wyznaczane z zależności funkcyjnych.

Czasem jednak otrzymane w ten sposób klucze obejmują zbyt wiele kolumn, co jest w praktyce niewygodne. Wprowadza się wtedy arbitralnie tzw. *klucze sztuczne*), dodając do relacji R nowy atrybut KS jako klucz. Oznacza to nałożenie dodatkowej zależności funkcyjnej

$$KS \to A_1 A_2 \ldots A_n$$

gdzie $A_1, A_2, \ldots, A_n$ są atrybutami relacji R

6.2.2. Wyprowadzanie zależności

Pewne zależności można wyprowadzić z innych. Przykład: jeśli zachodzi

$$A \to B$$

i zachodzi

$$B \to C$$

to musi zachodzić

$$A \to C$$

Można to sprawdzić z definicji. Inny sposób to skorzystać z reguł wnioskowania Armstronga.

6.2.3. Domknięcie zbioru atrybutów

Definicja 6.4. *Domknięciem zbioru atrybutów* Y (względem danego zbioru zależności) nazywamy taki zbiór atrybutów Y^+, że

1. $Y \subseteq Y^+$
2. Dla dowolnej zależności $U \to W$, jeśli $U \subseteq Y^+$, to $W \subseteq Y^+$

Jeśli $U \subseteq V^+$, to zachodzi zależność funkcyjna $V \to U$. *Klucz* jest to więc taki minimalny zbiór atrybutów, że jego domknięcie jest zbiorem wszystkich atrybutów relacji.

Dla przykładu policzymy *domknięcie* atrybutu A z naszego przykładu

1. Krok bazowy:
$$A^+ := A$$
2. Indukcja: ponieważ $A \to B$ i $A \subseteq A^+$, to
$$A^+ := A^+ \cup B = AB$$
3. Indukcja: ponieważ $B \to C$ i $B \subseteq A^+$, to
$$A^+ := A^+ \cup C = ABC$$

Wniosek: ponieważ $C \subseteq A^+$, więc zachodzi $A \to C$.

Spróbujmy innego przykładu. Dla zbioru zależności

$$AB \to C, BC \to AD, D \to E, CF \to B$$

domknięciem $\{A, B\}^+$ jest $\{A, B, C, D, E\}$, ponieważ

$$\begin{array}{ll} AB \to C & C \in \{A, B\}^+ \\ BC \to AD & D \in \{A, B\}^+ \\ D \to E & E \in \{A, B\}^+ \end{array}$$

6.2.4. Domknięcie zbioru zależności

Definicja 6.5. *Domknięcie zbioru zależności* Z (oznaczane Z^+) jest to zbiór wszystkich zależności, które można wyprowadzić z Z.

Definicja 6.6 (Równoważność zbiorów zależności)**.** Dwa zbiory zależności Z_1 i Z_2 są równoważne, jeśli ich domkniecia są równe: $Z_1 \equiv Z_2$ wtw $Z_1^+ = Z_2^+$.

Definicja 6.7 (Minimalny zbiór zależności)**.** Minimalny zbiór zależności jest to taki zbiór zależności, który nie jest równoważny żadnemu swojemu podzbiorowi.

— **Uwaga:** to jest uproszczona definicja, pełna jest bardziej skomplikowana.
— Minimalny zbiór zależności dla danego wyjściowego zbioru zależności nie jest wyznaczony jednoznacznie.

6.2.5. Reguły wnioskowania Armstronga

Reguły wnioskowania Armstronga służą do wyprowadzania zależności z innych zależności.

$$\begin{array}{lll} \text{Zwrotność} & & Y \subseteq X \ \vdash X \to Y \\ \text{Rozszerzanie} & & X \to Y \ \vdash XZ \to YZ \\ \text{Przechodniość} & X \to Y, Y \to Z & \vdash X \to Z \end{array}$$

6.3. Projektowanie schematu

6.3.1. Redundancja

Jeden z celów, który staramy się osiągnąć podczas projektowania schematu bazy danych to unikanie *redundancji* (nadmiarowości). Redundancja oznacza, że niektóre informacje są zapisane w bazie danych wielokrotnie. Prowadzi to do anomalii podczas modyfikacji i usuwania

— *Anomalia modyfikacji*: informacja zostaje uaktualniona tylko w niektórych miejscach
  — Przykład: zmiana adresu studenta na nowy, jeśli informacja o studencie i jego adresie trzymana jest w kilku miejscach.
— *Anomalia usuwania*: wraz z usunięciem ostatniego wiersza szczegółowego znika informacja ogólna
  — Gdyby informacje o adresie studenta trzymać przy przedmiotach, na które jest zarejestrowany, to po zakończeniu sesji (a przed nową rejestracją) adres ten by zniknął.

6.3.2. Przykład złego projektu

Obejrzyjmy przykład złego projektu i jego konsekwencje. Przypuśćmy, że mamy relację `Zwierzaki(gatunek,imie,waga,kontynent)` z następującymi zależnościami

imie → gatunek waga kontynent
gatunek → kontynent

Przykładowa zawartość takiej relacji

gatunek	imie	waga	kontynent
Papuga	Kropka	3,50	???
Papuga	Lulu	5,35	Ameryka
Papuga	Hipek	3,50	???
Lis	Fufu	6,35	Europa
Krokodyl	Czako	75,00	Afryka

Występuje redundancja, ponieważ wartości dla ??? można łatwo odgadnąć (czy jak kto woli wyznaczyć) na podstawie zależności.

6.4. Normalizacja

Normalizacja to proces, służący do przekształcenia schematu zawierającego redundancję w schemat, który jej nie zawiera. Zależnie od potrzeb określa się różne poziomy normalizacji — zakresy usuwania redundancji, nazywane *postaciami normalnymi*.

6.4.1. Postacie normalne

Dotychczas formalnie zdefiniowano pięć (poziomów) postaci normalnych, choć tylko trzy pierwsze są powszechnie używane podczas projektowania baz danych.

Postacie normalne są ponumerowane kolejno, ale istnieje postać pośrednia BCNF między 3 i 4 poziomem. Postacie o wyższych numerach automatycznie (z definicji) spełniają warunki dla niższych postaci, dlatego relacja w drugiej postaci normalnej jest automatycznie w pierwszej postaci normalnej.

Odwrotne wynikanie oczywiście nie zachodzi; drugą postać normalną otrzymujemy z pierwszej po nałożeniu dodatkowego warunku.

Proces normalizacji polega na dekompozycji tabel aż do otrzymania pożądanej postaci.

6.4.2. Pierwsza postać normalna (1NF)

Warunkiem pierwszej postaci normalnej jest to, by każdy atrybut w relacji przyjmował tylko *wartości niepodzielne*. Przez wartości niepodzielne rozumiemy takie pojedyncze wartości, jak używane w atrybutach „numer klienta" czy „nazwisko klienta".

Relacja w pierwszej postaci normalnej nie może zawierać atrybutu, w którym można upakować kilka wartości, np. odddzielając je przecinkami.

Daty można traktować jako wartości niepodzielne lub tworzyć oddzielne atrybuty dla dnia, miesiąca i roku.

6.4.3. Druga postać normalna (2NF)

Aby stwierdzić, czy relacja będąca w pierwszej postaci normalnej jest także w drugiej, należy określić klucze relacji.

Każdej wartości klucza powinien jednoznacznie odpowiadać pojedynczy wiersz w tabeli. Na przykład dla relacji `Zamówienie_klienta` kluczem może być `numer_zamówienia`.

Warunkiem na drugą postać normalną jest to, aby każdy niekluczowy atrybut zależał funkcyjnie od *całego* klucza. Niedozwolone są więc tzw. *zależności częściowe*.

Uwaga: przypominamy, że kluczy może być kilka!

6.4.4. Trzecia postać normalna (3NF)

Dla sprawdzenia, czy relacja będąca w drugiej postaci normalnej jest także w trzeciej, bada się zależności między atrybutami niekluczowymi.

Atrybuty niekluczowe powinny zależeć funkcyjnie wyłącznie od klucza i niczego więcej. Wykluczamy w ten sposób *zależności przechodnie*.

6.4.5. Postać normalna Boyce-Codda (BCNF)

Bardziej restrykcyjna niż trzecia postać normalna jest postać normalna Boyce-Codda, lecz jest ona rzadziej wykorzystywana w aplikacjach komercyjnych.

Relacja R jest w tej postaci, jeśli jest w 1NF oraz dla każdej nietrywialnej zależności $X \to Y$ zachodzącej w R, lewa strona zależności X jest nadkluczem.

W odróżnieniu od 3NF sprowadzenie do BCNF nie gwarantuje zachowania zależności funkcyjnych przy rozkładzie.

Popatrzmy na inną definicję 3NF, lepiej pokazującą różnicę między nią a BCNF.

Definicja 6.8 (3NF)**.** Relacja jest w 3NF jeśli jest w 1NF oraz dla każdej nietrywialnej zależności $X \to Y$:

— lewa strona zależności X jest nadkluczem
— *lub*
— prawa strona zależności Y zawiera tylko atrybuty z kluczy (bo kluczy może być kilka)

6.4.6. Dekompozycja do BCNF

Dekomponując relację R do BCNF pozbywamy się kolejno zależności naruszających ją. Przedtem należy zminimalizować zbiór zależności, ale pominiemy ten etap.

1. Dla relacji R wybierz ze zbioru zależności F zależność $X \to Y$ naruszającą BCNF (jeśli nie ma takiej, to relacja jest już w BCNF).
2. Oblicz X^+
   — Będzie zawierać tylko część atrybutów R, bo inaczej X byłoby nadkluczem.
3. Zastąp R dwoma relacjami o schematach

— $R_1 = X^+$
— $R_2 = R - (X^+ - X)$
4. Zrzutuj zbiór zależności F na nowe relacje.
5. Powtarzaj dopóki relacje nie będą w BCNF.

6.4.7. Zachowanie zależności

Czasem zależności powodują kłopoty przy przejściu do BCNF. Weźmy tabelę Gdzie(adres,miasto,kod-poczto
i dwie zależności
adres miasto → kod-pocztowy
kod-pocztowy → miasto
Mamy tu dwa klucze: {adres,miasto} oraz {adres,kod-pocztowy}. Ale zależność kod-pocztowy → miasto narusza postać BCNF, czyli musimy dokonać dekompozycji na dwie tabele
Gdzie1(adres,kod-pocztowy)
Gdzie2(miasto,kod-pocztowy)
Popatrzmy jednak na poniższy (ciut złośliwy) przykład

Gdzie2

adres	kod
Banacha 2	01-234
Banacha 2	01-235

Gdzie1

miasto	kod
Warszawa	01-234
Warszawa	01-235

Pozornie wszystko jest w porządku (żadna zależność nie jest naruszona), ale po złączeniu będzie już inaczej

Gdzie

adres	miasto	kod
Banacha 2	Warszawa	01-234
Banacha 2	Warszawa	01-235

Naruszona jest zależność adres miasto → kod-pocztowy.
Takie kłopotliwe układy zależności nie są rzadkie, popatrzmy na inny przykład zależności dla fikcyjnej sieci kin
kino → miasto
film miasto → kino
Pierwsza z nich jest oczywista: każde kino znajduje się tylko w jednym mieście. Druga może wynikać z prowadzonej „polityki" wyświetlania, aby kina w tym samym mieście nie konkurowały ze sobą filmami. Skutek jest taki sam jak poprzednio.

6.5. Zależności wielowartościowe

6.5.1. Czwarta postać normalna (4NF)

W większości baz danych wystarcza dekompozycja do trzeciej postaci normalnej. Mogą jednak czasem występować anomalie wstawiania, powodowane zależnością wielowartościową.
Będzie tak między innymi wtedy, gdy pewne dwa niekluczowe atrybuty przyjmują dla każdej wartości innego atrybutu tylko po kilka wybranych wartości, niezależnie od innych atrybutów.
Na przykład na seminariach korzysta się z kilku książek. Każde seminarium ma tylko jedną nazwę, ale może mieć kilku prowadzących. Należy wówczas dokonać dekompozycji na dwie osobne relacje: prowadzący seminaria oraz teksty do seminariów, w przeciwnym razie wypadałoby umieścić w relacji dla danego seminarium wszystkie kombinacje prowadzących i książek.

Przykład zależności wielowartościowej

— Tabela

Aktorzy(nazwisko,ulica,miasto,film,rok)

podaje adresy aktorów (mogą mieć kilka) i filmy, w których grali.
— Każdy aktor mógł grać w wielu filmach i może mieć kilka adresów.
— Ale w pojedynczym wierszu możemy zapisać tylko jeden film i jeden adres.
— Trudno będzie wtedy znajdować wszystkie filmy aktorów mieszkających w Warszawie.
— W zasadzie powinniśmy zapisać wszystkie kombinacje adresów z filmami.

Definicja zależności wielowartościowej

Definicja 6.9 (Zależność wielowartościowa)**.** *Zależność wielowartościowa*

$$A_1 \dots A_k \twoheadrightarrow B_1 \dots B_l$$

dla relacji $R(A_1, \dots, A_k, B_1, \dots, B_l, C_1, \dots, C_m)$ oznacza, że jeśli dwie krotki są zgodne na składowych A_i, to można w nich zamienić składowe B_i i otrzymane krotki będą także w relacji R.

— Inaczej mówiąc, lewa strona każdej takiej zależności nie wyznacza pojedynczej wartości, lecz *zbiór* wartości, np.

nazwisko $\twoheadrightarrow$ ulica,miasto
nazwisko $\twoheadrightarrow$ film,rok

Reguły dla zależności wielowartościowych

Stwierdzenie 6.1 (Promocja)**.** *Każda zależność funkcyjna jest zależnością wielowartościową, czyli jeśli zachodzi*

$$X \to Y$$

to zachodzi także

$$X \twoheadrightarrow Y$$

— Niestety odwrotna implikacja nie jest prawdziwa!

Stwierdzenie 6.2 (Uzupełnianie)**.** *Jeśli dla relacji $R(X, Y, Z)$ zachodzi*

$$X \twoheadrightarrow Y$$

to zachodzi także

$$X \twoheadrightarrow Z$$

Stwierdzenie 6.3 (Przechodniość)**.** *Jeśli dla relacji $R(X, Y, Z, V)$ zachodzą*

$$X \twoheadrightarrow Y \quad Y \twoheadrightarrow Z$$

to zachodzi także

$$X \twoheadrightarrow Z$$

Dodatkowe uwagi

— Podobnie jak dla zależności funkcyjnych, nie wolno rozbijać lewej strony zależności.
— Ale dla zależności wielowartościowych nie wolno również rozbijać prawej strony!

Czwarta postać normalna — definicja

Definicja 6.10 (Nietrywialna zależność wielowartościowa)**.** Zależność wielowartościowa $X \twoheadrightarrow Y$ *dla relacji* R jest *nietrywialna*, jeśli
— $Y \not\subseteq X$ oraz
— $X \cup Y$ nie są wszystkimi atrybutami R

Definicja 6.11 (4NF)**.** Relacja R jest w czwartej postaci normalnej (4NF), jeśli dla każdej nietrywialnej zależności $X \twoheadrightarrow Y$ w R, X jest nadkluczem R.

Wnioski
— Jeśli relacja jest w 4NF, to jest też w BCNF.
— Odwrotne zawieranie *nie* zachodzi.

Rozkład do 4NF
— Podobnie jak dla BCNF, szukamy zależności $X \twoheadrightarrow Y$ naruszającej 4NF.
— Rozbijamy relację $R(X, Y, Z)$ na dwie relacje:
$R_2(X, Y)$
$R_1(X, Z)$
— Kończymy, gdy już nie ma takich zależności.

Przykład rozkładu do 4NF
— Zaczynamy od relacji

Aktorzy(nazwisko,ulica,miasto,film,rok)

i zależności

nazwisko $\twoheadrightarrow$ ulica,miasto
nazwisko $\twoheadrightarrow$ film,rok

— W skład klucza wchodzą wszystkie kolumny tabeli.
— Obie zależności naruszają więc 4NF (bo nie są trywialne).
— Wybieramy do rozkładu pierwszą z nich.
— Otrzymujemy dwie tabele

Aktorzy1(nazwisko,ulica,miasto)
Aktorzy2(nazwisko,film,rok)

z tymi samymi zależnościami.
— Ponieważ obie zależności stały się trywialne, kończymy.

6.6. Zadania

Ćwiczenie 6.1. Czy któraś z podanych zależności funkcyjnych jest zależnością trywialną?
1. $A \to A$
2. $AB \to BC$
3. $BC \to C$

Rozwiązanie. Pierwsza i trzecia.

Ćwiczenie 6.2. Dla relacji $R(A, B, C, D)$ nie określono żadnych zależności funkcyjnych. Relacja R:
— nie ma żadnego klucza
— ma jeden klucz
— ma 4 klucze.

Rozwiązanie. Ma jeden klucz, składający się z *wszystkich* atrybutów relacji.

Ćwiczenie 6.3.

Rozwiązanie.

Ćwiczenie 6.4. Które z poniższych stwierdzeń są prawdziwe:
1. Każda relacja w 3NF jest także w BCNF.
2. Każda relacja w BCNF jest także w 3NF.
3. Każda relacja dwuargumentowa jest w BCNF.

Rozwiązanie. Drugie i trzecie.

Ćwiczenie 6.5. Dana jest tabela $R(A, B, C, D, E)$ z jedynymi zależnościami funkcyjnymi

$$A \to B, AB \to CD, D \to ABCE$$

Który z poniższych zbiorów atrybutów jest kluczem R?
1. A
2. AB
3. CD

Rozwiązanie. Tylko A.

Ćwiczenie 6.6. Dana jest tabela $R(A, B, C, D)$ z jedynymi zależnościami funkcyjnymi

$$A \to B, B \to C, C \to A$$

będąca w pierwszej postaci normalnej (1NF).

W jakich jeszcze postaciach jest ta tabela?
1. 2NF
2. 3NF
3. BCNF

Rozwiązanie. 2NF i 3NF.

7. Transakcje i współbieżność

7.1. Współbieżność

Współbieżność działania to cecha wszystkich współczesnych systemów komputerowych (pierwsze komputery pracowały sekwencyjnie, co oszczędzało wielu kłopotów, lecz niestety było dość wolne).

Przykład wsoółbieżnych działań: dwie osoby *równocześnie* pobierają 500 złotych z tego samego konta używając (różnych) bankomatów. System bazy danych powinien zadbać, żeby *obie* operacje zostały odnotowane, tzn. stan konta należy zmniejszyć dwukrotnie.

Podobna sytuacja występuje w innych programach. W wielu prostych edytorach tekstu jeśli dwie osoby równocześnie modyfikują ten sam dokument, utrwalone zostają tylko zmiany jednej z nich. Obecnie jest to jednak rzadko obserwowane, bo pracuje się głównie na komputerach osobistych — użytkownicy jawnie przesyłają sobie dokument.

Ale pamiętajmy, że istnieją rozproszone systemy plików (choćby NFS) i np. w laboratorium studenckim z serwerami plików może się zdarzyć, że dwie osoby będą modyfikować ten sam plik.

7.2. Transakcje

Transakcje to jedno z podstawowych pojęć współczesnych systemów baz danych. Umożliwiają one współbieżny dostęp do zawartości bazy danych, dostarczając niezbędnych mechanizmów synchronizacji.

Istotą transakcji jest integrowanie kilku operacji w jedną niepodzielną całość.

Popatrzmy na przykład wymagającu użycia transakcji. W systemie bankowym jest wykonywany przelew 100 złp z konta bankowego jednego klienta (np. Kangurzycy) na konto innego klienta (np. Tygrysa). W SQL wygląda to zapewne następująco

```
UPDATE Konta SET saldo = saldo - 100.00
    WHERE klient = 'Kangurzyca';
UPDATE Konta SET saldo = saldo + 100.00
    WHERE klient = 'Tygrys';
```

Co stanie się, jeśli po wykonaniu pierwszego polecenia nastąpi awaria dysku?

Podobny problem występuje nawet przy pojedynczym poleceniu SQL, jeśli modyfikuje ono wiele wierszy.

Rozwiązanie takich problemów to transakcyjny system baz danych. Gwarantuje on zapisanie modyfikacji w sposób trwały *przed* zakończeniem transakcji, a jeśli się nie uda to transakcja jest wycofywana.

7.2.1. Semantyka bazy danych

Semantykę bazy danych określa się opisując zbiór *legalnych stanów*, czyli ograniczając dozwoloną zawartość tabel.

Operacje modeluje się jako funkcje:

$$operacja\text{: Stan} \rightarrow \text{Stan}$$

Operacje powinny przeprowadzać legalne stany w legalne stany. Jednak w czasie wykonywania powiązanego ciągu operacji *przejściowo* baza danych może przyjmować nielegalne stany. Takie ciągi obudowujemy *transakcjami* i traktujemy transakcje jako niepodzielne operacje.

7.2.2. Transakcja

Transakcja to ciąg operacji do wspólnego niepodzielnego wykonania.

Współbieżne wykonywanie transakcji wymaga zachowania własności ACID (*Atomicity, Consistency, Isolation, Durability*):

— *niepodzielności*: „wszystko-lub-nic", transakcja nie może być wykonana częściowo;
— *integralności*: po zatwierdzeniu transakcji muszą być spełnione wszystkie warunki poprawności nałożone na bazę danych;
— *izolacji*: efekt równoległego wykonania dwu lub więcej transakcji musi być szeregowalny;
— *trwałości*: po udanym zakończeniu transakcji jej efekty na stałe pozostają w bazie danych.

7.2.3. Wycofanie i zatwierdzenie transakcji

W trakcie wykonywania transakcja może być *wycofana* w dowolnym momencie. Wszelkie wprowadzone przez nią zmiany danych zostaną wtedy zignorowane.

Realizacja tego wymaga „tymczasowego" wykonywania transakcji. Zmiany danych są tylko obliczane i zapisywane w specjalnym *dzienniku transakcji*.

Po zakończeniu wykonywania transakcji następuje jej *zatwierdzenie*, w wyniku czego zmiany są utrwalane w bazie danych.

Konflikty współbieżności

— Odwiedźmy bazę danych piwiarni i zajmijmy się tabelą Sprzedaje(bar,piwo,cena).
— Przypuśćmy, że w barze „U Szwejka" sprzedaje się tylko dwa gatunki piwa: Okocim po 2,50 zł i Żywiec po 3,50 zł.
— Dzielny redaktor gazety postanowił zbadać (używając naszej bazy danych), jaka jest najwyższa i najniższa cena piwa „U Szwejka".
— W tym samym czasie szef piwiarni zdecydował, że przestaje sprzedawać dotychczasowe piwa i przerzuci się na Heineken po 4,50 zł.

— Pan redaktor wykonuje dwa następujące zapytania (po lewej stronie ich umowne nazwy)

```
(max)  SELECT MAX(cena) FROM Sprzedaje
       WHERE bar = 'U Szwejka';

(min)  SELECT MIN(cena) FROM Sprzedaje
       WHERE bar = 'U Szwejka';
```

— A „równocześnie" szef piwiarni wykonał dwa inne polecenia SQL

```
(del)  DELETE FROM Sprzedaje
       WHERE bar = 'U Szwejka';

(ins)  INSERT INTO Sprzedaje
       VALUES('U Szwejka','Heineken',4.50);
```

Przeplecione polecenia

— Przypuśćmy, że powyższe polecenia zostały wykonane w następującej kolejności: max, del, ins, min.
— Popatrzmy na efekty:

Ceny	Operacja	Wynik
{2.50,3.50}	max	3,50
{2.50,3.50}	del	
{}	ins	
{4.50}	min	4,50

— A więc ostatecznie MAX(...) ¡ MIN(...)!

— Aby tego uniknąć, powinniśmy operacje poszczególnych osób pogrupować w transakcje.
— Wtedy obie operacje pana redaktora wykonają się bezpośrednio po sobie, nie wiadomo tylko, czy przed, czy po zmianie „repertuaru".

Problem wycofywania
— Szef piwiarni po wykonaniu (bez użycia transakcji) ciągu operacji (del)(ins) postanowił wycofać drugą z nich (`ROLLBACK`)
— Jeśli redaktorowi udało się „wstrzelić" zapytanie między (ins) i `ROLLBACK`, zobaczy wartość (4,50), której nigdy nie było w bazie danych.
— Rozwiązaniem jest znowu użycie transakcji:
  — Efekty transakcji nie są widziane przez innych, dopóki transakcja nie zostanie zatwierdzona (`COMMIT`).

Blokady
— Dla zapobiegania konfliktom używa się wewnętrznie *blokowania* dostępu do elementów danych używanych przez transakcję.
— Poziomy *ziarnistości* blokad:
  — cała baza danych,
  — pojedyncza relacja,
  — blok wierszy,
  — pojedynczy wiersz.

Rodzaje transakcji
— *Bezpośrednie* (*direct*);
— *Konwersacyjne* — kilkakrotna wymiana informacji klient/serwer;
— *Wiązane* (*chained*) – wymagają przechowywania kontekstu;
— *Zagnieżdżone*
— *Długotrwałe.*
— *Kolejkowane* – wykonywane z opóźnieniem, np. w celu grupowania;

7.3. Transakcje w SQL

Początek transakcji jest zwykle domyślny — pierwsza operacja na bazie danych wykonana przez aplikację.

W Postgresie przez można podać jawnie

```
BEGIN [WORK]
```

co jest najczęściej używane do wyjścia z trybu *autocommit* podczas pracy interakcyjnej.

Zakończenie transakcji następuje przez zatwierdzenie

```
COMMIT;
```

lub anulowanie (*wycofanie*)

```
ROLLBACK;
```

Uwaga: przy wystąpieniu błędu (np. naruszenie ograniczeń) ma miejsce niejawne wycofanie transakcji.

Domyślnie transakcje zezwalają na zapis. Rezygnuje się jednak z tego np. wtedy, gdy chcemy dokonać dłuższego skomplikowanego przeszukania spójnego stanu bazy danych.

Transakcję należy wtedy poprzedzić deklaracją

```
SET TRANSACTION LEVEL READ ONLY;
```

W transakcji takiej nie mogą wystąpić operacje modyfikacji, ale za to nie są widoczne zmiany dokonywane przez inne współbieżne transakcje.

Domyślnie przyjmowany jest poziom READ WRITE, tak jak gdyby podano

```
SET TRANSACTION LEVEL READ WRITE;
```

7.4. Poziomy izolacji transakcji

Poziom izolacji dla transakcji ustalamy korzystając z polecenia

```
SET TRANSACTION ISOLATION LEVEL
    [READ COMMITTED | SERIALIZABLE];
```

Poziom izolacji opisuje tylko, jak dana transakcja chce widzieć bazę danych (nie dotyczy więc bezpośrednio innych transakcji).

Poziom izolacji `SERIALIZABLE` gwarantuje semantykę sekwencyjną dla transakcji (ACID) przez wycofywanie transakcji naruszajacych ją.

Poziom `READ COMMITTED` powoduje przy modyfikacjach czekanie na zwolnienie (jawnej lub ukrytej) blokady wierszy. Odczyt nie jest jednak powtarzalny: kilka kolejnych wywołań tego samego zapytania w ramach tej samej transakcji może dać różne wyniki, ponieważ transakcja ma dostęp do wszystkich zatwierdzonych już modyfikacji z innych transakcji.

7.4.1. Transakcje w SQL

Standard dopuszcza również poziomy izolacji:

— `REPEATABLE READ`, gdy odczyty w ramach transakcji dają zawsze te same wiersze co poprzednio, ale mogą się pojawić dodatkowe wiersze: „fantomy". Żadne wiersze nie mogą jednak zniknąć.

— `READ UNCOMMITED`, zezwalający na tzw. *brudne odczyty* (*dirty reads*): odczytanie danych zmodyfikowanych przez inną transakcję, która potem zostaje wycofana.
W tym przypadku domyślnym poziomem transakcji jest `READ ONLY`, ponieważ `READ WRITE` jest na ogół zbyt ryzykowny.

7.5. Blokady w Oracle

— Blokady można zakładać na całą tabelę

```
LOCK TABLE tabela
IN [SHARE | EXCLUSIVE] MODE
[NOWAIT];
```

— `SHARE` oznacza blokadę dzieloną (tylko przeciw zmianom).
— `EXCLUSIVE` to wyłączna blokada dostępu (w celu dokonania zmian).
— `NOWAIT` chroni przed czekaniem, gdy nie można natychmiast założyć blokady.
— Zdjęcie blokad następuje przez wykonanie `COMMIT` lub `ROLLBACK`.

— Lepiej jednak zakładać blokady na wybrane wiersze, np. gdy transakcja odczytuje pewne wiersze, a następnie dokonuje (zwykle w nich) zmian, można użyć

```
SELECT ... FOR UPDATE [NOWAIT];
```

— Taka blokada też jest ważna do końca transakcji.

Realizacja transakcji
— Dziennik transakcji do zapisywania wszystkich operacji.
— Rejestrowanie wycofań w dzienniku.
— Podczas odtwarzania powtarzamy tylko operacje z zatwierdzonych transakcji.

Znaczniki czasowe
— Inne podejście, dobre gdy głównie odczyty.
— Optymistyczne, wycofanie transakcji gdy konflikt = fizycznie niemożliwy ciąg (lock wycofuje tylko gdy blokada).
— Transakcja utożsamiana z momentem startu.

8. Programowanie

Zajmiemy się teraz programami używającymi baz danych. Mogą one znajdować się w dwóch miejscach:
— na serwerze bazy danych jako *procedury składowane* (składniowo mogą to być funkcje);
— na maszynie wykonującej programy aplikacyjne, zwanej potocznie *klientem* bazy danych, choć w rzeczywistości może to być serwer aplikacji (np. serwer WWW), obsługujący wielu klientów równocześnie.

Zaczniemy od procedur składowanych. Są one przechowywane w bazie danych na serwerze i tam też są wykonywane. Aby użyć takiej funkcji, należy ją wywołać bezpośrednio z polecenia SQL lub pośrednio przez umieszczenie jej wywołania w wyzwalaczu.

8.1. Funkcje

Funkcje definiuje się w Postgresie używając konstrukcji

```
CREATE FUNCTION nazwa(parametr typ, ...)
RETURNS typ-wyniku AS $$
  treść-funkcji
$$ LANGUAGE nazwa-języka;
```

W starszych wersjach Postgresa treść funkcji otaczało się apostrofami, było to jednak niewygodne, bo wymagało dublowania wszystkich apostrofów wewnątrz treści.

Najłatwiej zdefiniować funkcje w języku SQL. Treść takiej funkcji to ciąg poleceń SQL lub pojedyncze polecenie, np.

```
CREATE FUNCTION dodaj7(i int4) RETURNS int4 AS $$
  SELECT i + 7;
$$ LANGUAGE sql;
```

Ponieważ w Postgresie polecenia `SELECT` można używać do obliczania wyrażeń nie związanych z bazą danych

```
bd> SELECT 2 + 5 AS siedem;
siedem
------
7
```

więc najprostsze funkcje można łatwo testować, na przykład pisząc:

```
bd> SELECT dodaj7(4) AS wynik;
wynik
-----
11
```

Funkcje napisane w SQL zwracają jako wartość wynik ostatniego polecenia w treści, musi to być `SELECT`. Jeśli wynik nie ma być zbiorem (co wynika z typu funkcji), będzie to pierwszy wiersz z tego zapytania (warto więc zadbać o ewentualne `ORDER BY` ;-).

Jeśli wynik zapytania był pusty, to jest zwracane NULL.

Dla funkcji SQL, która z założenia nie zwraca nic rozsądnego typem wyniku powinno być `void`, nie może ona wtedy kończyć się zapytaniem `SELECT`

```
CREATE FUNCTION oczyść () RETURNS void AS $$
  DELETE FROM Zwierz
  WHERE waga <= 0;
$$ LANGUAGE SQL;

SELECT oczyść();
```

8.2. PL/pgSQL

Normalnie jednak procedury bazy danych definiuje się nie w SQL, lecz w językach proceduralnych. Takim językiem dla serwera bazy danych w Postgresie jest najczęściej PL/pgSQL. Nazwa jest dość dziwna, ale pochodzi od języka PL/SQL, używanego przez DBMS Oracle.

Aby móc pisać procedury składowane w jakimś języku programowania, musi być on zainstalowany w bazie danych na serwerze. Nie dotyczy to języków SQL ani C, bo ich obsługa jest wbudowana w serwer.

Dla innych standardowych języków zawartych w dystrybucji Postgresa istnieje skrypt powłoki `createlang`, wykonujący instalację wybranego języka. Tak więc aby zainstalować PL/pgSQL w bazie danych `template1` wystarczy napisać (w powłoce na serwerze)

```
createlang plpgsql template1
```

Można zamiast tego użyć polecenia `CREATE LANGUAGE`, ale jest to bardziej kłopotliwe (choć konieczne, jeśli nie mamy bezpośredniego dostępu do komputera z serwerem).

Do usuwania języka proceduralnego służy polecenia `DROP LANGUAGE` albo skrypt `droplang`.

Język programowania PL/pgSQL (Procedural Language/postgreSQL) rozszerza SQL o typowe konstrukcje spotykane w proceduralnych jezykach imperatywnych. Podstawową jednostką w PL/SQL jest *blok*, programy buduje się z zagnieżdżonych bloków.

W blokach wolno używać instukcji SQL służących do manipulowania danymi (np. `SELECT`, `INSERT`, `UPDATE`, `DELETE`) oraz instrukcji sterowania transakcjami. Poza tym w blokach można używać typowych instrukcji takich jak przypisanie, instrukcja warunkowa, pętle, wywołania procedur.

Dla instrukcji `SELECT` jest używana postać rozszerzona, pozwalająca umieszczać wyszukane wartości w zmiennych PL/pgSQL. Instrukcje definiowania danych, takie jak `CREATE`, `DROP` czy `ALTER`, nie są dozwolone.

Bloki PL/pgSQL umieszcza się w treści funkcji, które można uruchamiać zwykłą instrukcją `SELECT` z poziomu programu `psql`.

8.2.1. Bloki

Blok jest podstawową konstrukcją języka PL/pgSQL. Składnia bloku wygląda następująco:

```
[DECLARE
  deklaracje zmiennych, stałych i procedur lokalnych¿]
BEGIN
  instrukcje
END;
```

(nawiasy kwadratowe oznaczają część opcjonalną, nie są elementem składni).

Przykład bloku poniżej

```
DECLARE
  a NUMERIC(5);
BEGIN
  SELECT COUNT(*) INTO a
  FROM EMP
  WHERE ENAME LIKE 'A%';
  IF a > 0 THEN
    INSERT INTO TEMP VALUES(a);
  END IF;
  RAISE NOTICE 'OK';
END;
```

8.3. Wyzwalacze

Wyzwalacze są to procedury wykonywane automatycznie przy zajściu pewnego zdarzenia, np. wstawieniu nowego wiersza do określonej tabeli. Pierwotnie wyzwalacze nie miały służyć do zapewnienia legalności stanów bazy (od tego są warunki integralności i asercje), lecz do zapewnienia legalności *przejść* między stanami.

Ponieważ jednak większość DBMS nie implementuje asercji (nie jest to zresztą łatwe), wyzwalaczy najczęściej używa się do realizacji złożonych ograniczeń, których nie mozna wyrazić w poleceniach takich jak `CREATE TABLE`.

Opis wyzwalacza obejmuje trzy składowe

— *Zdarzenie*: modyfikacja pewnej tabeli, np. „wstawienie to tabeli Gatunki".
— *Warunek*: wyrażenie Booleowskie w SQL.
— *Akcje*: polecenia do wykonania, najczęsciej zapisywane w SQL lub PL/SQL.

Składnia wyzwalacza napisanego w SQL (mniej więcej zgodna ze standardem) jest następująca:

```
CREATE [OR REPLACE] TRIGGER nazwa
{BEFORE | AFTER} INSERT OR DELETE OR UPDATE
ON tabela
FOR EACH {ROW | STATEMENT}
EXECUTE
polecenie;
```

Dla wyzwalaczy określa się, czy ich akcje mają być wykonane przed czy po właściwej operacji (`BEFORE` lub `AFTER`). Ponadto dla wyzwalacza określony jest jeden z dwóch poziomów: wiersza lub zdania.

Wyzwalacz poziomu zdania jest odpalany tylko raz dla całego polecenia SQL, natomiast wyzwalacz poziomu wiersza jest odpalany niezależnie dla każdego modyfikowanego wiersza.

Spróbujmy napisać wyzwalacz, który przy usunięciu wiersza z tabeli Gatunki w odpowiadających mu wierszach w tabeli Zwierz ustawi NULL w kolumnie gatunek.

Oczywiście to samo można osiągnąc łatwiej używając więzów klucza zewnętrznego, ale zobaczymy jak można to zrobić wyzwalaczem.

```
CREATE TRIGGER DelGat
AFTER DELETE ON Gatunki
FOR EACH ROW EXECUTE
UPDATE Zwierz
SET gatunek = NULL
WHERE gatunek = OLD.gatunek;
```

A teraz inny przykład. Załóżmy, że dla każdego gatunku w tabeli Gatunki chcemy w dodatkowej osobnej kolumnie przechowywać liczbę zwierzaków tego gatunku. Zaczniemy od dodania odpowiedniej kolumny

```
ALTER Gatunki
ADD ile INTEGER DEFAULT 0 CHECK (ile > 0);
```

Teraz pora na wyzwalacze

```
CREATE TRIGGER InsZwierz
AFTER INSERT ON Zwierz
FOR EACH ROW EXECUTE
UPDATE Gatunki
SET ile = ile + 1
WHERE gatunek = NEW.gatunek;

CREATE TRIGGER DelZwierz
AFTER DELETE ON Zwierz
FOR EACH ROW EXECUTE
UPDATE Gatunki
SET ile = ile - 1
WHERE gatunek = OLD.gatunek;
```

8.3.1. Wyzwalacze w Postgresie

Wyzwalacze w Postgresie definiuje się w PL/pgSQL, używając uprzednio zdefiniowanych bezargumentowych funkcji zwracających specjalny typ `TRIGGER`.

Funkcja połączona z wyzwalaczem otrzymuje dane za pośrednictwem struktury `TriggerData`, a nie przez zwykłe parametry funkcyjne, dlatego procedur tych nie należy wywoływać bezpośrednio.

Sa one wywoływane niejawnie przez wyzwalacz, ilekroć wystąpi zdarzenie z nim związane. Zdarzeniem może być próba wykonania jednej z operacji SQL `INSERT`, `DELETE` lub `UPDATE`.

Deklaracja wyzwalacza w PL/pgSQL ma następującą składnię:

```
CREATE TRIGGER nazwa
  BEFORE | AFTER INSERT | DELETE | UPDATE [OR ...]
  ON tabela
  [FOR EACH ROW | STATEMENT]
  EXECUTE PROCEDURE nazwa-funkcji( [argumenty] );
```

Funkcja wyzwalacza musi zostać zdefiniowana *przed* utworzeniem wyzwalacza. Nazwa wyzwalacza musi być unikalna dla danej tabeli i przydaje się przy usuwaniu wyzwalacza. Poza tym wyzwalacze tego samego typu uruchamia się w kolejności alfabetycznej według ich nazw.

Modyfikatory BEFORE i AFTER określają, czy funkcja wyzwalacza ma być wywoływana przed czy po właściwej akcji.

Można określić do trzech rodzajów zdarzeń (INSERT, DELETE lub UPDATE) uruchamiających wyzwalacz używając spójnika `OR`. Przykłady:

```
... INSERT ON R ...
... INSERT OR DELETE OR UPDATE ON R ...
```

Opcja `FOR EACH ROW` określa, że wyzwalacz jest poziomu wiersza, tzn. będzie odpalany osobno dla każdego zmienianego wiersza tabeli. Domyślnie wyzwalacz jest poziomu całej instrukcji.

Argumenty wywołania w definicji wyzwalacza są rzadko używane i powinny być literałami. Dzięki nim mozna tej samej funkcji używać w różnych wyzwalaczach. Najprostszy przykład

to funkcja wpisująca informacje do dodatkowej tabeli zawierającej dziennik modyfikacji, gdzie parametrem mogłaby być nazwa modyfikowanej tabeli.

Usuwając wyzwalacz trzeba oprócz jego nazwy podać nazwę tabeli

```
DROP TRIGGER nazwa ON tabela;
```

W treści funkcji związanej z wyzwalaczem są dostępne dwie zmienne rekordowe `NEW` i `OLD`, odnoszące się do nowej i starej zawartości wiersza. Procedura wyzwalacza dla wstawiania i modyfikacji powinna zwracać odpowiedni rekord, zwykle jest to po prostu `NEW`.

Można jednak zwracać NULL, co w wyzwalaczach typu `BEFORE` oznacza, że wykonanie wyzwalacza nie powiodło się i właściwa operacja powinna zostać zignorowana.

Jeśli w funkcji wyzwalacza używa się poleceń SQL, mogą one spowodować uruchomienie innych wyzwalaczy, nazywa się to kaskadowaniem wyzwalaczy. Nie istnieje żadne ograniczenie na liczbę poziomów wywołań kaskadowych, w szczególności możliwe są wywołania rekurencyjne.

Za unikanie nieskończonej rekursji jest odpowiedzialny programista!

Czas na jakiś przykład. Załóżmy, że mamy tabelę

```
CREATE TABLE Gatunki (
  nazwa VARCHAR(30) PRIMARY KEY,
  kontynent VARCHAR(11),
  chroniony BOOLEAN,
  przysmak VARCHAR(15)
);
```

Utworzymy w Postgresie wyzwalacz dbający o to, aby nazwa kontynentu rozpoczynała się dużą literą:

```
CREATE FUNCTION normkont () RETURNS TRIGGER AS $$
BEGIN
  IF NEW.kontynent IS NOT NULL THEN
    NEW.kontynent := lower(NEW.kontynent);
    NEW.kontynent := initcap(NEW.kontynent);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER gatwyzw1
  BEFORE INSERT OR UPDATE ON Gatunki
  FOR EACH ROW
  EXECUTE PROCEDURE normkont();
```

Wykonanie polecenia `CREATE TRIGGER` jedynie utworzyło wyzwalacz nie wykonując go. Aby uruchomić wyzwalacz (np. w celu przetestowania go) należy spowodować zajście odpowiedniego zdarzenia, w tym przypadku powinno to być wstawienie lub modyfikacja wiersza dla tabeli `Gatunki`.

8.4. Programowanie aplikacji

Dostęp użytkownika do baz danych odbywa się zwykle przez programy, uruchamiane na stacji roboczej użytkownika (nazywanej często *klientem*). Programy takie można pisać w dowolnym języku programowania, musi on jednak być wyposażony w interfejs programisty dla SQL (tzw. API — *Application Programmer Interface*). Interfejs taki może być specyficzny dla danego systemu DBMS lub uniwersalny (np. ODBC, JDBC).

8.4.1. Interfejs dla języka C

Autorzy systemów DBMS prawie zawsze dostarczają interfejs dla języka C. Obejrzymy zatem najpierw podstawowy interfejs systemu PostgreSQL dla języka C, zawarty w bibliotece `libpq`.

Po pierwsze w programie należy dołączyć odpowiedni plik nagłówkowy:

```
#include "libpq-fe.h"
```

Korzystanie w programie z bazy danych wymaga nawiązania połączenia:

```
PGConn *polaczenie;

polaczenie =
  PQconnectdb("dbname=bd,host=rainbow,user=ja");
if (PQstatus(polaczenie) == CONNECTION_BAD)
  fprintf(stderr, "Brak polaczenia\n");
  PQfinish(polaczenie);
  exit(1);
```

Załóżmy, że połączenie powiodło się i w bazie danych znajduje się tabela `Gatunki`, zawierająca m.in. kolumny `gatunek` i `kontynent`. Chcemy wyszukać z tej tabeli, z jakiego kontynentu pochodzi gatunek o podanej nazwie (np. wczytanej z klawiatury).

```
PGresult *wynik;

wynik = PQexec(polaczenie,
               "SELECT kontynent FROM Gatunki "
               "WHERE gatunek = 'szop pracz'");
if (PQresultStatus(wynik) == PGRES_TUPLES_OK &&
    PQntuples(wynik) == 1)
  printf("Szop pracz pochodzi z %s\n",
         PQgetvalue(wynik, 0, 0));
else
  fprintf(stderr, "Brak odpowiedzi\n");
PQfinish(polaczenie);
```

8.4.2. Zanurzony (embedded) SQL

Embedded SQL jest to technika bezpośredniego wpisywania poleceń SQL wśród instrukcji języka zanurzającego, w tym przypadku języka C. W PostgreSQL służy do tego narzędzie o nazwie ECPG.

Program w C z wbudowanymi poleceniami SQL jest najpierw przetwarzany preprocesorem `ecpg` na program w języku C. Preprocesor rozpoznaje polecenia SQL zawarte w programie i zastępuje je wywołaniami funkcji z odpowiednie biblioteki CLI dla SQL.

Otrzymany program przetwarza się normalnym kompilatorem C na program wykonywalny.

```
ecpg -I/usr/include/ecpg test1.pgc
cc -I/usr/include/ecpg -o test1 test1.c -L/usr/local/lib -lecpg
```

Wszystkie polecenia SQL w programach muszą być poprzedzane frazą `EXEC SQL` i kończyć się średnikiem („;”). Można je umieszczać gdziekolwiek w programie w C pilnując jedynie, aby deklaracje poprzedzały polecenia wykonywalne.

Przykład:

```
int main ()

  EXEC SQL BEGIN DECLARE SECTION;
```

```
int w;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT wiek INTO :w
         FROM Zwierz
         WHERE imie='Kropka';
...
printf("Kropka waży %d kilo\n", w);
...
```

8.5. Laboratorium: poprawność bazy danych

Baza danych zawiera tabele:
— `Komputer(producent, model, typ)`,
— `PC(model, szybkość, ram, dysk, cdrom, cena)`,
— `Laptop(model, szybkość, ram, dysk, ekran, cena)`.
Zdefiniuj w SQL następujące ograniczenia:

Ćwiczenie 8.1. Komputery PC o szybkości mniejszej niż 150 MHz nie mogą być sprzedawane drożej niż za 2500 złotych lub muszą mieć co najmniej 64 MB pamięci RAM.

Rozwiązanie.

```
ALTER TABLE PC ADD CONSTRAINT polcen
CHECK (szybkość >= 150 OR cena <= 2500 OR ram >= 64);
```

Uwaga: jeśli jakieś już istniejące wiersze nie będą spełniały tego warunku, to polecenie zakończy się błędem. Należy je najpierw skorygować, np. poleceniem UPDATE.

Ćwiczenie 8.2. Laptopy o ekranie mniejszym niż 11 cali, które nie mają dysku co najmniej 1GB, są sprzedawane poniżej 3600 złotych.

Rozwiązanie.

```
ALTER TABLE Laptop ADD CONSTRAINT polcen1
CHECK (ekran >= 11 OR dysk >= 1 OR cena < 3600);
```

Zdefiniuj w PL/SQL następujące ograniczenie:

Ćwiczenie 8.3. Żaden laptop nie może być sprzedawany taniej niż PC o tej samej lub mniejszej szybkości i pamięci RAM.

Rozwiązanie.

```
CREATE FUNCTION pclaptopfun () RETURNS TRIGGER AS $$
DECLARE
  gorna_cena INTEGER;
BEGIN
  IF NEW.ram IS NULL OR NEW.szybkosc IS NULL OR NEW.cena IS NULL
    RETURN NEW;
  ELSE
    SELECT MAX(cena) INTO gorna_cena
    FROM PC
    WHERE szybkosc <= NEW.szybkosc AND ram <= NEW.ram;
    IF gorna_cena > NEW.cena
      RAISE EXCEPTION 'Za mała cena';
      RETURN NULL;
```

```
    ELSE
      RETURN NEW;
    END IF;
  END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER pclaptop
```

9. Programowanie w logice

9.1. Wprowadzenie

Programowanie w logice to jeden z paradygmatów współczesnego programowania. Programu przestaje być przepisem wykonania krok po kroku pewnego algorytmu, lecz jest traktowana jak teoria w logice.

W tym podejściu koncentrujemy się na *deklaratywnym* opisaniu, *co* ma być rozwiązane (odpowiednie postawienie problemu), a nie na szczegółowej *proceduralnej* specyfikacji, *jak* należy to rozwiązać.

Języki do programowania w logice są oparte na rachunku predykatów pierwszego rzędu, mają więc zdecydowany posmak relacyjny. W tym podejściu wywołanie procedury odpowiada podaniu twierdzenia do udowodnienia i jest często podawane interpreterowi jako *zapytanie*. Wykonanie programu oznacza poszukiwanie dowodu.

Jedną z konsekwencji tego podejścia jest specyficzne, typowe dla matematyki traktowanie zmiennych — mogą mieć one nadaną wartość tylko raz.

Najbardziej znanym językiem programowania w logice jest Prolog. Nie jest oczywiście możliwe nauczenie (ani nawet opisanie) Prologu w ramach jednego wykładu.

Pobieżne zaznajomienie się z koncepcjami programowania w logice pozwoli jednak zrozumieć, skąd wywodzi się koncepcja dedukcyjnych baz danych, o której opowiemy w następnym wykładzie. Operować będziemy głównie przykładami programów.

9.2. Warszawskie metro

Warszawskie metro ma (na razie) tylko jedną linię, ale w zupełności wystarczy to na pierwszy przykład użycia Prologu. Rozpoczniemy od zapisania faktów: informacji o kolejności stacji na linii.

```
connected(bielany,słodowiec).
connected(słodowiec,marymont).
connected(marymont,placWilsona).
connected(placWilsona,dworzecGdański).
  ...
```

Składnia nie jest skomplikowana. Prolog używa dla literałów składni pochodzącej z rachunku predykatów

$$predykat(term,...term)$$

Nazwy zmiennych (identyfikatory) muszą rozpoczynać się wielką literą lub znakiem podkreślenia ('_')'. Identyfikator rozpoczynający się małą literą jest stałą — symbolem.

9.2.1. Zapytania

Prologu można używać jak prostej bazy danych. Możemy na przykład zapytać interpreter Prologu, czy Marymont i Plac Wilsona są ze sobą połączone.

```
| ?- connected(marymont,placWilsona).
yes
```

Następnie zapytamy o stacje połączone z Marymontem

```
| ?- connected(marymont,X).
X = placWilsona

yes
```

W zapytaniu użyliśmy zmiennej `X`. Interpreter zwrócił wartość związaną ze zmienną `X` jako jedyną odpowiedź.

Używając zmiennych możemy spytać o wszystkie połączenia

```
| ?- connected(Start,End).
Start = bielany
End = słodowiec ? ;

Start = słodowiec
End = marymont ? a

Start = marymont
End = placWilsona

Start = placWilsona
End = dworzecGdański

no
```

Interpreter, z którego pochodzą przykłady, po podaniu pierwszej odpowiedzi czeka na polecenie użytkownika. Naciśnięcie średnika to prośba o kolejną odpowiedź, zaś litera a (*all*) oznacza żądanie podania wszystkich pozostałych odpowiedzi.

9.3. Reguły

Zdefiniujmy dwie stacje jako położone *blisko* jeśli są one połączone bezpośrednio lub pomiędzy nimi jest tylko jedna stacja. W rachunku predykatów wygląda to tak

$$\forall x \forall y. connected(x, y) \implies blisko(x, y)$$
$$\forall x \forall y. \forall z. connected(x, z) \wedge connected(z, y)) \implies blisko(x, y)$$

Tę samą informację można zapisać regułami Prologu

```
blisko(X,Y) :- connected(X,Y).
blisko(X,Y) :- connected(X,Y),connected(Y,Z).
```

— Najpierw łatwe zapytanie
```
| ?- blisko(bielany,marymont).
yes
```
— Teraz oczekujemy kilku odpowiedzi
```
| ?- blisko(bielany,What).
What = słodowiec ? ;

What = marymont

yes
```

9.4. Definiowanie silni

Wprowadzenie do języka programowania nie liczy się, jeśli nie obejmuje silni (albo największego wspólnego dzielnika). Pora więc na nią

Najpierw definicja funkcyjna

$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{dla} \quad n > 0.$$

którą przekształcimy na relację $factorial$

$$factorial(0, 1)$$
$$(\forall n)(\forall f) n > 0 \wedge factorial(n-1, f) \implies factorial(n, n \cdot f)$$

```
factorial(0,1).

factorial(N,F) :-
   N > 0,
   N1 is N - 1,
   factorial(N1,F1),
   F is N * F1.
```

9.5. Klauzule

— Program składa się z dwóch *klauzul*.
— Pierwsza z nich to klauzula unarna: tylko *nagłówek*, bez treści. Jest to skrót następującego zapisu:
`factorial(0,1) :- true.`
— Klauzule unarne bez zmiennych służą do zapisywania pojedynczych *faktów*.
— Druga klauzula to reguła, ponieważ posiada niepustą *treść*.
— Treść to wszystkie literały następujące po ':-' (możemy go czytać „jeśli").
— Literały w treści oddzielamy przecinkami, przecinek pełni rolę operatora koniunkcji.
— Standardowo nazwy zmiennych rozpoczynają się dużymi literami.
— Klauzule mają intepretację deklaratywną
  — Klauzula pierwsza mówi: „silnia o wynosi 1".
  — Klauzula druga mówi: „silnia N wynosi F jeśli $N > 0$ oraz jeśli $N1$ oznacza $N - 1$, to silnia $N1$ wynosi $F1$ i $F = N * F1$".

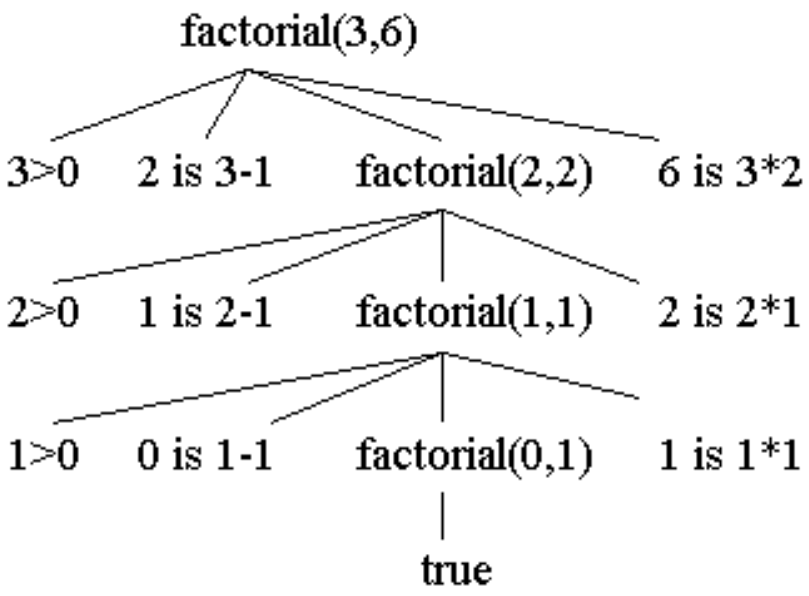
9.6. Zapytania

Chcąc obliczyć silnię od 3 musimy podać w *zapytaniu* zmienną, na której ma znaleźć się wynik — niech to będzie W:

```
?-  factorial(3,W).
W=6
```

Graf obliczeń W zapytaniu nie muszą wystąpić zmienne:

```
?- factorial(3,6).
yes
?- factorial(5,2).
no
```



Rysunek 9.1. .

Inna definicja silni

```
silnia(N,F) :- factorial(N,1,F).

factorial(0,F,F).

factorial(N,A,F) :-
    N > 0,
    A1 is N * A,
    N1 is N - 1,
    factorial(N1,A1,F).
```

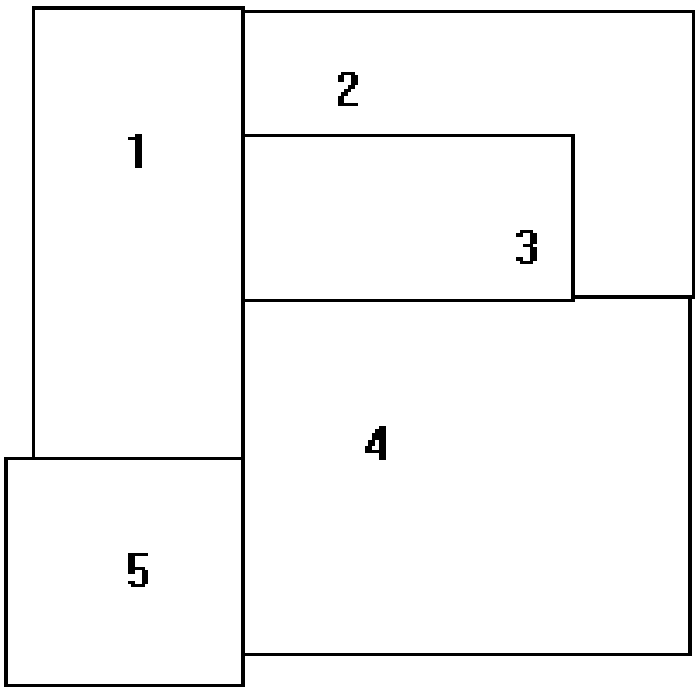
— Wprowadziliśmy drugi parametr pełniący rolę *akumulatora*.
— Dzięki temu nasza definicja ma postać iteracyjną, ponieważ wywołanie rekurencyjne jest *redukcyjne*.
— Płacimy za to zmniejszoną czytelnością.

9.7. Kolorowanie map

Zasada:
— Żadne dwa sąsiadujące regiony nie mogą mieć tego samego koloru.

Sąsiedztwo Informacje o sąsiedztwie można w Prologu zapisać następująco:

```
adjacent(1,2).        adjacent(2,1).
adjacent(1,3).        adjacent(3,1).
adjacent(1,4).        adjacent(4,1).
adjacent(1,5).        adjacent(5,1).
adjacent(2,3).        adjacent(3,2).
adjacent(2,4).        adjacent(4,2).
adjacent(3,4).        adjacent(4,3).
```



Rysunek 9.2. Przykładowa mapa.

```
adjacent(4,5).          adjacent(5,4).
```

Po załadowaniu tych faktów do interpretera można badać sąsiedztwo regionów:

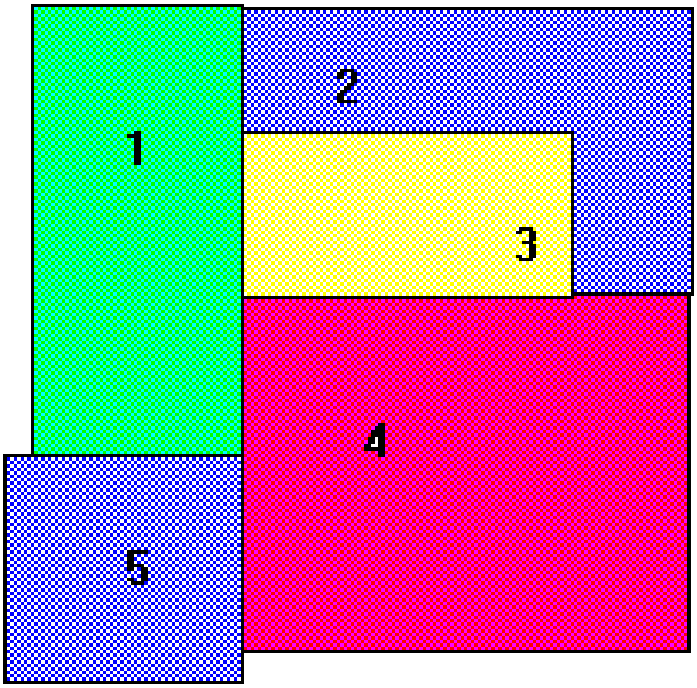
```
?- adjacent(2,3).
yes
?- adjacent(5,3).
no
?- adjacent(3,R).
R = 1 ;
R = 2 ;
R = 4 ;
no
```

Kolorowania Kolorowania także możemy opisać klauzulami unarnymi.

```
color(1,red,a).      color(1,red,b).
color(2,blue,a).     color(2,blue,b).
color(3,green,a).    color(3,green,b).
color(4,yellow,a).   color(4,blue,b).
color(5,blue,a).     color(5,green,b).
```

Kolorowania oznaczyliśmy symbolami `a` i `b`.

Konflikty Błędne kolorowanie musi zawierać konflikt — dwa sąsiednie regiony pokolorowane tym samym kolorem:



Rysunek 9.3. Kolorowania.

```
conflict(Coloring) :-
   adjacent(X,Y),
   color(X,Color,Coloring),
   color(Y,Color,Coloring).
```

Teraz możemy zbadać nasze kolorowania:

```
?- conflict(a).
no
?- conflict(b).
yes
?- conflict(Which).
Which = b
```

Konfliktowość mozna też zdefiniować inaczej:

```
conflict(R1,R2,Coloring) :-
   adjacent(R1,R2),
   color(R1,Color,Coloring),
   color(R2,Color,Coloring).
```

Polimorfizm Prolog dopuszcza równoczesne definiowanie predykatów o tej samej nazwie i różnej liczbie parametrów, wewnętrznie nazywając je 'conflict/1' i 'conflict/3'.

Więcej o konfliktach Możemy poznać konfliktowe regiony, a nawet ich (wspólny) kolor

```
?- conflict(R1,R2,b).
R1 = 2   R2 = 4
```

```
?- conflict(R1,R2,b),color(R1,C,b).
R1 = 2   R2 = 4   C = blue
```

9.8. Rozpoznawanie zwierząt

Na koniec obejrzymy przykład programu do identyfikacji zwierząt na podstawie zaobserwowanych cech. Program korzysta z własnej „bazy danych", zawierającej informacje o cechach poszczególnych gatunków. Przypominamy, że nazwy zmiennych w klauzulach rozpoczynają się dużymi literami (lub podkreśleniem).

```
zoolog :- hypothesize(Zwierzak),
      write('Przypuszczam, że ten zwierzak to: '),
      write(Zwierzak),
      nl,
      undo.
```

`write` i `nl` to przykłady pozalogicznych *predykatów obliczalnych*, wywołujących rozmaite akcje — w tym przypadku służących do wypisywania.

9.8.1. Hipotezy

Predykat `hypothesisze` określa możliwe hipotezy — gatunki, które jesteśmy w stanie rozpoznawać.

```
hypothesize(gepard)   :- gepard, !.
hypothesize(tygrys)   :- tygrys, !.
hypothesize(żyrafa)   :- żyrafa, !.
hypothesize(zebra)    :- zebra, !.
hypothesize(struś)    :- struś, !.
hypothesize(pingwin)  :- pingwin, !.
hypothesize(albatros) :- albatros, !.
hypothesize(nie_wiem).    /* nie udało się */
```

Specjalny predykat wbudowany `!` to tzw. *odcięcie*. Powoduje on, po udanym dojściu do niego, pominięcie pozostałych gałęzi (czyli klauzul) predykatu `hypothesisze`. W tym przypadku po rozpoznaniu jednego z gatunków nie próbujemy już rozpoznawać innych.

9.8.2. Reguły identyfikacji

Reguły identyfikacji zrealizujemy bezargumentowymi predykatami nazwanyni nazwami gatunków.

```
gepard :- ssak,
          drapieżnik,
          verify(ma_płowy_kolor),
          verify(ma_ciemne_plamy).
tygrys :- ssak,
          drapieżnik,
          verify(ma_płowy_kolor),
          verify(ma_czarne_paski).
żyrafa :- kopytny,
          verify(ma_długą_szyję),
          verify(ma_długie_nogi).
zebra :- kopytny,
         verify(ma_czarne_paski).
```

```
struś :- ptak,
         verify(nie_lata),
         verify(ma_długą_szyję).
pingwin :- ptak,
           verify(nie_lata),
           verify(pływa),
           verify(jest_czarnobiały).
albatros :- ptak,
            verify(występuje_w_opowieściach_morskich),
            verify(dobrze_lata).
```

9.8.3. Reguły klasyfikacji

Reguły klasyfikacji pozwalają nam zapisać wspólną informację tylko w jednym miejscu, unikając dublowania informacji.

```
ssak       :- verify(ma_sierść), !.
ssak       :- verify(daje_mleko).
ptak       :- verify(ma_pióra), !.
ptak       :- verify(lata),
              verify(znosi_jajka).
drapieżnik :- verify(je_mięso), !.
drapieżnik :- verify(ma_ostre_zęby),
              verify(ma_pazury),
              verify(ma_oczy_z_przodu).
kopytny :- ssak,
           verify(ma_kopyta), !.
kopytny :- ssak,
           verify(przeżuwacz).
```

9.8.4. Zadawanie pytań

Prostą obsługę dialogu z użytkownikiem nie jest trudno zapisać bezpośrednio w Prologu.

```
ask(Pytanie) :-
    write('Czy zwierzak ma następującą cechę: '),
    write(Pytanie),
    write('? '),
    read(Odpowiedz),
    nl,
    ( (Odpowiedz == tak ; Odpowiedz == t)
      ->
       assert(tak(Pytanie)) ;
       assert(nie(Pytanie)), fail).
```

Aby uniknąć wielokrotonego zadawania tych samych pytań odpowiedzi będziemy zapamiętywać. Po otrzymaniu odpowiedzi `tak` na pytanie P zapisujemy tę informację jako fakt

`tak(`S`).`

Podobnie po otrzymaniu odpowiedzi `nie` na pytanie P zapisujemy tę informację jako fakt

`nie(`S`).`

Wbudowany predykat `fail` powoduje niepowodzenie (nigdy nie może być „udowodniony”) i konieczność wycofania się z danej gałęzi.

Predykat `verify` zagląda najpierw do bazy faktów, a dopiero potem ewentualnie zadaje pytanie użytkownikowi.

```
verify(S) :- tak(S), true, !.
verify(S) :- nie(S), fail, !.
verify(S) :- ask(S).
```

Po rozpoznaniu :-(lub nie) kolejnego zwierzaka bazę faktów tak/nie należy oczyścić

```
undo :- retract(tak(_)), fail.
undo :- retract(nie(_)), fail.
undo.
```

10. Dedukcyjne bazy danych

10.1. Bazy danych z perspektywy logiki

Spojrzenie na bazy danych oczami logika pozwala jednolicie opisać szereg używanych pojęć. Do opisu relacyjnych baz danych używa się zwykle rachunku predykatów bez symboli funkcyjnych.

Fakty zapisane w wierszach tabel reprezentuje się literałami, np.

Ojciec(Jan,Piotr)

gdzie pierwszy symbol (Ojciec) odpowiada nazwie tabeli, a pozostałe zawartości pewnego wiersza w tej tabeli.

Taki sposób opisu obejmuje również najprostsze realizacje baz danych, np. gdy fakty przechowuje się w postaci pojedynczej listy, wczytywanej do pamięci z pliku podczas ładowania bazy danych i zapisywanej na pliku przy zamykaniu bazy.

Tak więc reprezentacją relacji `Zwierzaki`:

gatunek	imię	waga
Papuga	Kropka	3,50
Papuga	Lulu	5,35
Papuga	Hipek	3,50
Lis	Fufu	6,35
Krokodyl	Czako	75,00

będą następujące literały

Zwierzak(Papuga,Kropka,3.50)
Zwierzak(Papuga,Lulu,5.35)
Zwierzak(Papuga,Hipek,3.50)
Zwierzak(Lis,Fufu,6.35)
Zwierzak(Krokodyl,Czako,75.00)

(dla nazwy relacji w rachunku predykatów użyjemy liczby pojedynczej, bo jest naturalniejsza).

10.1.1. (CWA:Closed World Assumption)

W relacyjnych bazach danych obowiązuje założenie o zamkniętości świata.

Definicja 10.1 (Założenie o zamkniętości świata)**.** Przyjmuje się, że jeśli fakt nie znajduje się w odpowiedniej tabeli w bazie danych, to nie zachodzi (jest fałszywy). Inaczej mówiąc, jeśli wiersz

$$\langle w_1, w_2, \ldots, w_n \rangle$$

nie znajduje się w relacji R, to zachodzi

$$\neg R(w_1, w_2, \ldots, w_n)$$

Mówiąc potocznie, jeśli czegoś nie wiem, to powinienem przyjąć, że nie jest to prawdziwe. W przeciwnym razie odpowiedzi na zapytanie SELECT w SQL musiałyby się kończyć frazą „... i być może jeszcze coś, o czym nie wiem".

Dla bazy danych z podaną przed chwilą relacją Zwierzaki będzie więc spełnione (na przykład):

¬Zwierzak(Papuga,Czako,2.00)

i wiele innych zanegowanych literałów.

10.1.2. Domain closure assumption

Założenie o zamkniętości dziedziny (a właściwie dziedzin) mówi, że nie istnieją inne „obiekty" niż występujące w tabelach bazy danych.

Definicja 10.2 (Założenie o zamkniętości dziedziny)**.** Nie ma innych indywiduów niż te znajdujące się w bazie danych. Inaczej mówiąc, nie ma innych stałych niż te, które występują w wierszach bazy danych.

Założenie to spełniają systemy ograniczające niejawnie swoją dziedzinę „referencji". Przykład: algebra relacji (bo wszelkie wyrażenia powstają przez składanie relacji początkowych).

W relacyjnych bazach danych używających SQL założenia tego nie spełniają oczywiście liczby całkowite i rzeczywiste. Zwykle natomiast spełniają je napisy, np. nie ima innych nazw zwierzaków niż te podane w tabeli Zwierz.

Unique name assumption

Definicja 10.3 (Założenie o unikalności nazw)**.** Indywidua (stałe) o różnych nazwach są różne. Inaczej mówiąc, nie ma aksjomatów równościowych dla stałych.

Unikalność nazw: przykład

— W naszej przykładowej bazie

Zwierzak(Papuga,Kropka,3.50)
Zwierzak(Papuga,Lulu,5.35)
Zwierzak(Papuga,Hipek,3.50)
Zwierzak(Lis,Fufu,6.35)
Zwierzak(Krokodyl,Czako,75.00)

— zachodzi

Zwierzak(Krokodyl,Fufu,6.35)

— ponieważ zakładamy, że zachodzi

Krokodyl $\neq$ Lis

Zapytania

— Do zapisywania zapytań będziemy używać formuł zawierających zmienne.
— Ograniczymy się do formuł prostych.
— Zapytania mają wtedy postać wzorców — struktur podobnych do faktów, mogących jednak zawierać zmienne.
— Zmiennymi będą symbole poprzedzone znakami zapytania, np.

```
Ojciec(Jan,?x)
```

jest zapytaniem o wszystkie dzieci Jana.

— Odpowiedź na zapytanie składa się z bezpośrednio wyszukanych faktów pasujących do zapytania oraz z faktów wyprowadzonych przy użyciu reguł.

Klauzule

Definicja 10.4 (Klauzula)**.** *Klauzulą* nazywać będziemy formułę postaci

$$P_1 \wedge P_2 \wedge \cdots \wedge P_k \rightarrow N_1 \vee N_2 \vee \cdots \vee N_l$$

lub też równoważnie

$$\neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_k \vee N_1 \vee N_2 \vee \cdots \vee N_l$$

Wyrażenia $P_1, \ldots, P_k$ nazywać będziemy *poprzednikami* (lub *przesłankami*, zaś wyrażenia $N_1, \ldots, N_l$ *następnikami* lub *wnioskami* klauzuli.

Rodzaje klauzul

— Klauzule bez poprzedników, zawierające tylko jeden następnik bez zmiennych

$$\rightarrow N(a_1, \ldots, a_m).$$

służą do reprezentowania faktów.

— Klauzule bez następników

$$P_1 \wedge P_2 \wedge \cdots \wedge P_k \rightarrow .$$

reprezentują negację poprzedników i mogą służyć do zapisywania ograniczeń (więzów poprawności). Stan bazy danych jest legalny, jeśli nie jest spełniona koniunkcja poprzedników, np.

$$\text{Ojciec(?x,?y)} \wedge \text{Matka(?x,?y)} \rightarrow .$$

— Taka klauzula mająca tylko jeden poprzednik bez zmiennych zapisuje jawnie negację faktu, np.

$$\text{Ojciec(Jan,Piotr)} \rightarrow .$$

oznacza, że Jan nie jest ojcem Piotra.

— Klauzula o pojedynczym następniku

$$P_1 \wedge P_2 \wedge \cdots \wedge P_k \rightarrow R$$

służy w dedukcyjnych bazach danych do definiowania relacji R.

— Jak zobaczymy dalej, pełna definicja relacji może wymagać kilku klauzul.

— W relacyjnych bazach danych klauzule takie (tak jak i następne) służą do zapisywania ograniczeń.

— Klauzule postaci

$$\rightarrow N_1 \vee N_2 \vee \cdots \vee N_k$$

wyrażają wiedzę niekompletną: wiemy, że spełniony jest jeden lub więcej następników, ale nie wiemy który.

— Przykład:

$$\rightarrow \text{Student(Jan,Matematyka)} \vee \text{Student(Jan,Biologia)}$$

— Klauzulą o pełnej postaci

$$P_1 \wedge P_2 \wedge \cdots \wedge P_k \rightarrow N_1 \vee N_2 \vee \cdots \vee N_l$$

można interpretować jako warunkowe informacje o niekompletnej wiedzy.

— Można też ich używać jako ograniczeń, np. aby zapisać, że każdy ma co najwyżej dwoje rodziców użyjemy

```
Rodzic(?p,?a) ∧ Rodzic(?q,?a) ∧ Rodzic(?r,?a)
  → (?p = ?q) ∨ (?p = ?r) ∨ (?q = ?r)
```

— Klauzula pusta, czyli klauzula bez poprzedników i następników, oznacza fałsz.
— Klauzula taka nie powinna nigdy wystąpić w niesprzecznej bazie danych.

10.2. Dedukcyjne bazy danych

W dedukcyjnych bazach danych dopuszcza się tylko niektóre rodzaje klauzul z programowania w logice.
— Nakłada sie dodatkowe ograniczenia.
  — Brak symboli funkcyjnych.
  — Tylko jeden następnik.
    — Takie dedukcyjne bazy danych określa się jako *definite*.
  — Zmienna z konkluzji reguły musi wystąpić w jej treści.
    — Zapewnia to spełnienie założenia o zamkniętości dziedziny.
  — Jako predykaty obliczalne dozwolona tylko arytmetyka.
    — Zauważmy, że formalnie narusza to założenie o zamkniętości dziedziny.
— Jednym z takich rozszerzeń relacyjnych baz danych jest Datalog.

Datalog

— W Datalogu klauzule zapisujemy jako reguły.
— Każdy predykat jest
  — albo ekstensjonalny: zdefiniowany wyłącznie przez fakty zapisane w bazie danych,
  — albo intensjonalny: zdefiniowany wyłącznie przez reguły.
— Reguły mają nieco zmienioną postać zapisu

następnik `<-` *poprzednik* `AND ...`

przy czym zarówno następnik jak i poprzedniki mają postać wzorców, na przykład

```
P(?x,?y,?z) <- Q(?x,?y,?z) AND x > 10
```

— Ponieważ takie reguły, podobnie jak klauzule, odpowiadają implikacjom z logiki, posiadają one zarówno interpretację logiczną jak i proceduralną.
— Na przykład regule

```
Dziadek(?x,?z) <- Ojciec(?x,?y) AND Ojciec(?y,?z)
```

odpowiada implikacja

$$(\forall x, y, z)\ \text{Ojciec}(x, y) \wedge \text{Ojciec}(y, z) \rightarrow \text{Dziadek}(x, z)$$

— Jeśli reguła nie zawiera poprzedników, to jej następnik jest zawsze spełniony.
— Podobnie jak fakty, reguły przechowuje się także w bazie danych.

Reguły a zapytania

— Reguła pasuje do zapytania, jeśli jej konkluzja pasuje do zapytania.
— Znajdowanie odpowiedzi na zapytanie przy użyciu reguł polega na znalezieniu reguł pasujących do zapytania i dla każdej z nich wyszukanie w sposób spójny (tzn. tak, żeby wartości zmiennych były zgodne) odpowiedzi na podzapytania powstałe z przesłanek (czyli rekursja :-) i złożenie otrzymanych wyników, podstawiając otrzymane wartości zmiennych na zmienne w konkluzji.

— Ostateczną odpowiedzią na zapytanie jest lista wyszukanych faktów (dla reguł będą to konkretyzacje ich konkluzji).

Zalety

— Pozwalają otrzymywać odpowiedzi wymagające domknięcia przechodniego.
— Przykład:
  — Mamy tabelę Zawiera(obiekt,składnik,ile) dotyczącą pojazdów (np. rowerów, sanek itp.) podającą obiekty i ich bezpośrednie składowe.
  — Chcemy otrzymać informację o wszystkich częściach wchodzących w skład obiektu 'rower'.

Przykładowa tabela

Zawiera

obiekt	składnik	ile
rower	koło	2
koło	piasta	1
koło	obręcz	1
obręcz	szprycha	20
koło	opona	1
rower	rama	1
rama	siodełko	1
rama	kierownica	1

Definicja predykatu Części

— Definicja w Datalogu

```
Czesci(?calosc,?czesc)
  <- Zawiera(?calosc,?czesc,?_).

Czesci(?calosc,?czesc)
  <- Zawiera(?calosc,?skladnik,?_)
     AND Czesci(?skladnik,?czesc).
```

— i samo zapytanie

```
Czesci('rower',?co) ?
```

Realizacja zapytań

— W odróżnieniu od programowania w logice w Datalogu nie używa się nawracania.
— Budowa odpowiedzi odbywa się *bottom-up*: rozpoczynamy od pustej relacji `Czesci` i iteracyjnie dodajemy do niej wszystkie wiersze, które w tym momencie można otrzymać z reguł.
— Optymalizacja: w kążdym kroku używam reguł tylko dla wierszy dodanych w poprzednim kroku.
— Zauważmy, że jest to typowy algorytm wyznaczania minimalnego punktu stałego (tzw. *minimalnego modelu*).
— Z otrzymanej *ekstensji* relacji wybieramy tylko wiersze pasujące do zapytania.

Przykład obliczania odpowiedzi

— Rozpoczynam od pustej tabeli `Czesci`.
— Krok 1: z pierwszej reguły dodaję do niej wiersze `<'rower','koło'>`, `<'koło','piasta'>`, `<'koło','obręcz'>`, `<'obręcz','szprycha'>`, `<'koło','opona'>`, `<'rower','rama'>`, `<'rama','siodeł` `<'rama','kierownica'>`.

— Krok 2: używając drugiej reguły dodaję wiersze ¡'rower','piasta'¿, ¡'rower','obręcz'¿, ¡'rower','opona'¿, ¡'rower','siodełko'¿, ¡'rower','kierownica'¿, ¡'koło','szprycha'¿.
— Krok 3: używając drugiej reguły dodaję wiersz ¡'rower','szprycha'¿.
— Krok 4: ponieważ żadna reguła nie produkuje już nowych wierszy, kończę iterację i wybieram wiersze ¡'rower','koło'¿, ¡'rower','rama'¿, ¡'rower','piasta'¿, ¡'rower','obręcz'¿, ¡'rower','opona'¿, ¡'rower','siodełko'¿, ¡'rower','kierownica'¿, ¡'rower','szprycha'¿.

Rekursja w SQL

— Zapytania rekurencyjne w SQL-3 definiuje się konstrukcją

```
WITH [RECURSIVE] R(...) AS
    zapytanie używające R
  SELECT ...;
```

— Można tak zdefiniować kilka pomocniczych relacji wzajemnie rekurancyjnych.

Przykład rekursji w SQL

— Zapiszemy nasze zapytanie z Datalogu w SQL

```
WITH RECURSIVE Czesci(calosc,czesc) AS
    (SELECT obiekt,skladnik FROM Zawiera)
    UNION
    (SELECT obiekt,czesc
     FROM Zawiera, Czesci
     WHERE skladnik = calosc)
  SELECT czesc FROM Czesci
  WHERE calosc = 'rower';
```

Negacja

— W poprzednikach reguł Datalogu można poprzedzać predykaty negacją `NOT`.
— Taki literał jest spełniony, jeśli ta negacja jest prawdziwa dla tego predykatu w minimalnym modelu.
— Aby można to było sprawdzać, program z negacjami musi być *stratyfikowalny*: reguły predykatów intensjonalnych dzielimy na warstwy tak, by
  — Definicja predykatu nie była w wyższej warstwie, niż jego wykorzystanie w definicjach innych predykatów.
  — Negacja predykatu występowała w wyższej warstwie niż jego definicja.
— Minimalne punkty stałe wyznaczamy *kolejno* poczynając od najniższej warstwy.
— Podobne ograniczenie na poprawność rekursji z negacją obowiązuje w SQL-3.

11. Hurtownie danych

Tradycyjne bazy danych są skonfigurowane na wiele małych, prostych zapytań. Rozpowszechnienie baz danych spowodowało jednak zainteresowanie wykorzystaniem zgromadzonej informacji nie tylko do obsługi codziennej działalności, lecz także do analiz statystycznych i wspierania podejmowania decyzji o znaczeniu strategicznym dla firmy.

Takie nowe aplikacje używają mniej, ale bardziej czasochłonnych i złożonych zapytań. Ale za to operują na większych bazach danych, często obejmujących materiał wieloletni.

Powstały więc nowe architektury przeznaczone do efektywnej obsługi złożonych zapytań analitycznych.

11.1. Hurtownia danych

Zanim jednak zaczniemy analizować dane należy je ze sobą połączyć. Informacja bowiem może pochodzić z rozmaitych operacyjnych baz danych. Proces łączenia takiej często niejedorodnej informacji nazywamy *integracją danych*.

Najpopularniejszy sposób integracji danych to użyciw *hurtowni danych*. Kopiujemy dane źródłowe z operacyjnych baz danych do pojedynczej zbiorczej bazy danych (*hurtowni*) i dbamy o ich aktualność.

Aktualność osiąga się przez *doładowanie*: okresową rekonstrukcję hurtowni (np. w nocy lub na koniec tygodnia), polegającą na jej uzupełnieniu o nowe dane.

Hurtownie są często używane bezpośrednio do obsługi zapytań analitycznych, lepiej jednak tworzyć z nich specjalizowane *tematyczne* bazy danych.

11.2. OLTP

Większość codziennych operacji na operacyjnej bazie danych to *On-Line Transaction Processing* (OLTP): krótkie, proste i częste zapytania lub modyfikacje, każde dotyczące niewielkiej liczby krotek.

Przykłady: odpowiedzi na zapytania z interfejsu WWW (np. o rozkład jazdy pociągów, bankomaty, sprzedaż biletów lotniczych.

11.3. OLAP

Obecnie rośnie znaczenie programów typu *On-Line Analytical Processing* (OLAP). Zawierają one niewiele zapytań, ale dużo bardziej złożonych — ich przetwarzanie może trwać godzinami.

Zapytania takie przy tym nie wymagają bezwzględnie aktualnej bazy danych, bowiem zwykle służą poszukiwaniu ogólnych prawidłowości.

Integralność danych zapewniana podczas konstrukcji (lub doładowania)
— Później sprawdzanie integralności zbędne, bo tylko zapytania (bez modyfikacji).
— Brak transakcji.

Przykłady aplikacji OLAP

— Amazon analizuje zamówienia swoich klientów, aby wyświetlić im ekran powitalny z produktami z zakresu ich zainteresowań.
— Analitycy w Wal-Mart szukają towarów z rosnącą sprzedażą w jakimś regionie. Pojęcie o skali aplikacji daje informacja o używanym sprzęcie: system Teradata z 900 CPU, 2700 dysków, 23 TB pamięci dyskowej.

Typowa architektura
— Lokalne bazy danych w oddziałach obsługują OLTP.
— Nocą kopiuje się lokalne bazy danych do centralnej hurtowni danych.
— Analitycy używają hurtowni do zapytań OLAP.

Co to jest hurtownia?
— Kolekcja rozmaitych danych
  — zorientowana tematycznie
  — przeznaczona dla menadżerów, do podejmowania decyzji
  — często kopia danych operacyjnych
  — z dodatkowymi informacjami (np. podsumowania, historia)
  — zintegrowana
  — zmienna w czasie
  — nieulotna
— Kolekcja narzędzi do:
  — zbierania danych
  — czyszczenia i integracji danych
  — analizy i raportowania
  — monitorowania i zarządzania hurtownią

Konieczność integracji danych
— Obecnie duże firmy (np. banki, towarzystwa ubezpieczeniowe) posiadają co najmniej kilka operacyjnych baz danych.
— Obok własnych danych operacyjnych często korzysta się również ze źródeł zewnętrznych: wyników badań rynkowych, publicznych administracyjnych baz danych itp.
— Zanim więc przystąpi się do analizy zawartej w nich informacji, należy dokonać *integracji danych* z różnych źródeł.

Integracja danych
— Informacje ze źródłowych (najczęściej operacyjnych) baz danych są łączone w zbiorczą bazę danych, nazywaną *hurtownią danych* (*data warehouse*).
— Oddzielenie od systemów operacyjnych umożliwia operowanie również danymi historycznymi — potrzebne np. do analizy trendów.

Realizacja hurtowni danych
— Dwa sposoby realizacji:
  — Realna zbiorcza baza danych wraz procedurami okresowego doładowywania nowymi informacjami źródłowymi.
  — Hurtownia wirtualna, oparta na koncepcji *mediatora* i *wrapper*ów.

Wybór źródeł danych
— Idealnym rozwiązaniem przy wyborze danych do załadowania do hurtowni danych byłoby wstępne określenie wszystkich zapytań, które będą generowane przez aplikacje odpalone na hurtowni danych i ustalenie wszystkich tabel i pól zawartych w tych zapytaniach.

— Zaletą hurtowni wirtualnej jest możliwość zbadania, jakie dane źródłowe są potrzebne w używanych zapytaniach.
— Po przeanalizowaniu logów wygenerowanych przez zapytania z wszystkich aplikacji łatwiej określić schemat hurtowni.

Łączenie danych
— Łączenie to okazja do usunięcia błędów.
— Przede wszystkim jednak dane są uspójniane.
— Przykład: pole `klient` może
  — w jednej z baz oprócz imienia i nazwiska obejmować tytuł (np. „prof.”),
  — w drugiej na tytuł jest przeznaczone osobne pole,
  — a w trzeciej imię i nazwisko trzymane są w osobnych polach, a tytułu brak.
— Czyszczenie danych obejmuje też przeformatowanie danych i eliminację duplikatów.

Kroki budowy hurtowni
1. Analiza źródłowych danych pod względem rozmieszczenia, spisanie rodzaju i zawartości danych.
2. Projekt hurtowni danych zawierający spis dostępnych źródłowych baz danych wraz z narzędziami scalającymi.
3. Wydobywanie odpowiednich danych z baz źródłowych, przekształcenie wydobytych danych tak aby pasowały do zaprojektowanej hurtowni danych. Załadowanie przekształconych danych do hurtowni danych.

Słownik (repozytorium) metadanych Zawiera informacje:
— skąd pochodzą dane (z jakiej bazy danych i jakiej tabeli);
— co reprezentują, np. kolumna `salesyr` to „roczna sprzedaż brutto towaru wyrażona w złotych”;
— jak dane są przechowywane (format zapisu), np. `salesyr` jako `fixed-decimal`;
— kto odpowiada za aktualizację danych (dział/rola), np. „księgowość”;
— kiedy dana jest zwykle zmieniana, np. „na koniec każdego roku”.

11.3.1. Problem jakości danych

Większość przedsiębiorstw nie ma najmniejszego pojęcia o jakości swoich danych i jest zawsze naiwna w tym względzie. Dopiero przy próbie integracji kilku źródeł danych wychodzą na jaw rozmaite błędy.

Analizując jakość danych (lub jej brak) dobrze jest wprowadzić kategorie usterek w danych. Dane mogą być:
— Brakujące (lub częściowo brakujące)
— Błędne
— Nieaktualne
— Niespójne
— Wprowadzające w błąd

Brakujące dane są stosunkowo łatwe do wychwycenia, trudniej jest natomiast je poprawiać. Dane błędne mogą być trudniejsze do wykrycia, choć ich poprawianie może czasem być łatwiejsze.

Dane wprowadzające w błąd mogą wystąpić, gdy robione są zmiany retrospektywne, a reprezentacja czasu nie jest poprawna. Dzieje się tak na przykład, gdy nastąpi zmiana adresu i (z uwagi na niepoprawne zastosowanie retrospekcji) nowy ades jest mylnie stosowany do poprzednich danych.

Błędne dane

— Wartości spoza zakresu poprawności. Są one zwykle łatwe do rozpoznania.
  — Ten typ błędu zdarza się, kiedy, powiedzmy, płeć klienta, która powinna mieć wartość "F" lub "M", zawiera inną wartość, taką jak "X".
  — Odnosi się to także do wartości numerycznych, takich jak wiek. Poprawne wartości mogłyby byćzdefiniowane między 18 a 100 lat. Jeśli wiek znajduje się poza tym zakresem, to uznaje się, że wystąpił błąd.
— Błędy referencyjne. Jest to każdy błąd naruszający więzy spójności referencyjnej.
— Błedy poprawności. Są to błędy w danych niemal niemożliwe do wykrycia: wartości, które po prostu zostały wstawione niepoprawnie, ale są z zakresu poprawności. Na przykład wstawiono wiek klienta jako 26, podczas gdy powinno być 62. Błędy takie pochodzą z operacyjnych baz danych i w zasadzie tam właśnie powinny być korygowane.

Ujednolicenie danych Jednolitość jest problemem dla wielu przedsiębiorstw, zwłaszcza tych, które używają różnych typów narzędzi.

— Niektóre różnice dotyczą kwestii podstawowych, jak choćby kodowanie znaków.
  — W większości systemów stosuje się kod ASCII (American Standard Code for Information Interchange).
  — Są jednak wyjątki. IBM oparł wszystkie systemy „mainframe" na całkowicie odmiennym kodowaniu znaków, zwanym EBCDIC (Extended Binary Coded Decimal Interchange Code).
  — Litera „P" w kodzie ASCII ma wartość dziesiętną 80, a w EBCDIC wartość 215 (znakiem, który ma wartość 80 w EBCDIC jest „&").
— Inne różnice dotyczą dat. Większość systemów zarządzania bazami danych obsługują typ "Date", ale format przechowywania jest zależny od DBMS.

— Jeszcze subtelniejsze są różnice wewnątrz różnych aplikacji tworzonych przy użyciu tych samych narzędzi.
  — Może się to na przykład zdarzyć, że jeden z projektantów postanowi zapisywać adresy klientów w pięciu polach po 25 znaków każde, inny zaś zdecyduje się gromadzić je w jednym polu typu Varchar(100).
  — Kolor czarny w jednej aplikacji reprezentuje się napisem "BLACK", w drugiej napisem "BL", a w trzeciej "BL" oznacza kolor niebieski.
— Różnice semantyczne: w jednej bazie danych odróżnia się półciężarówki od mikrobusów, a w drugiej nie.
— Katalogi nazw i adresów pozwalają naprawić błędy przy wprowadzaniu danych i pozwolą uzupełnić brakujące informacje takie jak kod adresowy, powiat itp.
— Takie naprawianie typowych błędów wymaga jednak interwencji człowieka, który zatwierdzi poprawki programu.

Problem wydajności i skalowalności

— Relacyjne bazy danych używają metod przyśpieszania dostępu takich jak haszowanie lub indeksy do wybrania małej liczby pożądanych rekordów bez potrzeby skanowania całej tabeli lub bazy danych.
— Takie metody przyśpieszania dostępu są wysoce efektywne w odpowiedziach na zapytania o pojedyncze pole (lub małą liczbę pól), kiedy wynikiem jest niewielka część całej tabeli.
— Przykładem takich zapytań jest: „znajdź wszystkich 25 letnich programistów".
— Odpowiedzi na takie zapytania są szybkie, gdyż może być stworzony indeks dla kolumny „wiek" lub kolumny „zawód".

— Klasyczne metody dostępu sa mało pomocne w odpowiedziach na zapytania, których rezultatem jest znaczna część tabeli.
— Przykładem jest „znajdź wszystkich młodych pracowników”.

12. Analityczne bazy danych („kostki danych”)

— Inaczej *On-Line Analytical Processing* : OLAP
— Rosnące znaczenie: 8 mld. dol. w 1998 roku.
— Od komputerów biurkowych po olbrzymie konfiguracje:
— Wiele modnych haseł, zaklęć
  — zwijanie i rozwijanie, drążenie, MOLAP, obracanie.

Podstawowe zagadnienia
— Co to jest analityczna baza danych?
— Modele i operacje
— Implementacja analitycznej bazy danych
— Kierunki rozwojowe

12.1. Raporty macierzowe

Pierwszym wsparciem dla analizy danych były *raporty macierzowe*. Zwykle były używane przez specjalistów od finansów lub zarządzania.

W systemie sprzedaży będzie na przykład potrzebny raport o klientach i ich wzorcach kupowania z podziałem na powiaty. Zamiast analizować wzorce dla każdego towaru, dzielimy towary na kategorie.

Raporty macierzowe przypominają arkusz kalkulacyjny. W kolumnach raportu będą wypisane u góry poszczególne kategorie towarów, w wierszach powiaty, a każda z komórek będzie pokazywać liczbę pozycji sprzedanych w tej kategorii.

Data Mart
— Mała hurtownia
— Obejmuje tylko część tematyki organizacji, np.
  — marketing: klienci, towary, sprzedaż
— Model dostosowany do potrzeb działu.
— Najczęściej informacja wstępnie zagregowana
  — Eliminacja zbędnych detali
  — Wybieramy pewien krytyczny poziom szczegółowości.

Narzędzia do zapytań i analiz
— Budowanie zapytań
— Generatory raportów
  — porównania: wzrost, spadek
  — trendy,
  — grafy
— Arkusze kalkulacyjne
— Interfejs WWW
— Data Mining

Inne operacje
- Funkcje po czasie
  - np. średnie po różnych okresach
- Atrybuty obliczane
  - np. marża = sprzedaż * stopa
- Zapytania tekstowe, np.
  - znajdź dokumenty zawierające słowa A i B
  - uporządkuj dokumenty według częstości występowania słów X, Y i Z

Modele danych i operatory
- Modele danych
  - relacja
  - gwiazda i płatek śniegu
  - kostka: rozwinięcie idei arkusza kalkulacyjnego (tablice wielowymiarowe)
- Operatory
  - slice & dice
  - roll-up, drill down
  - pivoting
  - inne

Wielowymiarowy model danych
- Najczęściej używa się wielowymiarowych bazy danych z uwagi na analityczny model danych o postaci kostki wielowymiarowej obejmujący:
  - *fakty* zwane też *miarami* (*measures*), np. liczba sprzedanych samochodów;
  - *wymiary* (*dimensions*), np. miesiące, regiony sprzedaży.

Wymiary
- Wymiary tworzą zazwyczaj hierarchie, np. dla czasu będzie to rok-kwartał-miesiąc-dzień.
- Dzięki temu możliwa jest interakcyjna zmiana poziomu szczegółowości (ziarnistości) oglądanej informacji.
- W bardziej złożonych przypadkach hierarchie mogą rozgałęziać się, np. podział na tygodnie jest niezgodny z podziałem na miesiące.

Baza danych
- Źródłem danych jest najczęściej hurtownia danych (rzeczywista lub wirtualna).
- Bezpośrednie trzymanie w bazie danych informacji o faktach dla wszystkich poziomów szczegółowości może być kosztowne
  - Tylko dla najczęściej używanych poziomów hierarchii.
  - Pozostałe są wyliczane na bieżąco w razie potrzeby.
- Przy agregowaniu miar warto pamiętać o różnych regułach liczenia, np.
  - Wielkość sprzedaży jest na ogół sumowana
  - Temperatura lub cena będą raczej uśredniane.
- W analitycznej bazie danych trzymane są w zasadzie dane zagregowane.
- Aby obejrzeć dane szczegółowe (*drill-through*) konieczne jest sięgnięcie do hurtowni danych lub bazy operacyjnej.
- Ponieważ jest to kosztowne czasowo, taka potrzeba nie powinna występować zbyt często.

Operacje na danych
- Przecinanie i rzutowanie (*slice and dice*)
- Zmiana poziomu szczegółowości: drążenie lub rozwijanie (*drill-down*) i zwijanie (*roll-up*),

— obracanie (*pivot*): zmienia położenie wymiaru na „wykresie”.

Podejścia do budowy bazy OLAP

1. ROLAP = „relacyjny OLAP”: dopasowujemy relacyjny DBMS do schematu gwiazdy.
2. MOLAP = „wielowymiarowy OLAP”: używamy specjalizowanego DBMS z modelem w rodzaju „kostka danych”.

Schemat gwiazdy

— *Schemat gwiazdy* to typowy sposób organizacji danych dla relacyjnej bazy danych dla OLAP.
— Obejmuje:
  — *Tabelę faktów*: olbrzymi zbiór faktów takich jak informacje o sprzedaży.
  — *Tabele wymiarów*: mniejsze, statyczne informacje o obiektach, których dotyczą fakty.
— Uogólnienie: model płatka śniegu.
  — Hierarchie tabel dla poszczególnych wymiarów: normalizacja wymiarów.

Przykład schematu gwiazdy

— Chcemy gromadzić w hurtowni danych informacje o sprzedaży piwa: bar, marka piwa, piwosz, który je zakupił, dzień, godzina oraz cena.
— Tabelą faktów będzie relacja:

```
Sprzedaż(bar,piwo,piwosz,dzień,godzina,cena)
```

— Tabele wymiarów zawierają informacje o barach, piwach i piwoszach:

```
Bary(bar,adres,licencja)
Piwa(piwo,prod)
Piwosze(piwosz,adres,tel)
```

Atrybuty wymiarów i atrybuty zależne

— Dwa rodzaje atrybutów w tabeli faktów:
  — *Atrybuty wymiarów*: klucze tabel wymiarów.
  — *Atrybuty zależne*: wartości wyznaczone przez atrybuty wymiarów krotki.

Przykład: atrybut zależny

— cena jest atrybutem zależnym w przykładowej relacji Sprzedaż.
— Jest ona wyznaczona przez kombinację atrybutów wymiarów: bar, piwo, piwosz i czas (kombinacja atrybutów daty i godziny).

Techniki ROLAP

— *Indeksy bitmapowe*: dla każdej wartości klucza indeksowego w tabeli wymiaru (np. dla każdego piwa w tabeli Piwa) tworzymy wektor bitowy podający, które krotki w tabeli faktów zawierają tę wartość.
— *Perspektywy zmaterializowane*: w hurtowni przechowujemy gotowe odpowiedzi na kilka użytecznych zapytań (perspektywy).

Typowe zapytanie OLAP

— Zapytanie OLAP często zaczyna się od „star join”: złączenia naturalnego tabeli faktów z wszystkimi lub większością tabel wymiarów.
— Przykład:

```
SELECT *
FROM Sprzedaż,Bary,Piwa,Piwosze
WHERE Sprzedaż.bar = Piwa.bar
  AND Sprzedaż.piwo = Piwa.piwo
  AND Sprzedaż.piwosz = Piwosze.piwosz;
```

— Rozpoczyna się złączeniem gwiaździstym.
— Wybiera interesujące krotki używając danych z tabel wymiarów.
— Grupuje po jednym lub więcej wymiarach.
— Agreguje niektóre atrybuty wyniku.

Przykład zapytania OLAP

— Dla każdego baru w Poznaniu podaj całkowitą sprzedaż każdego piwa wytwarzanego przez browar Anheuser-Busch.
— Filtr: adres = “Poznań” i prod = “Anheuser-Busch”.
— Grupowanie: po bar i piwo.
— Agregacja: Suma po cena.

Przykład: SQL

```
SELECT bar, piwo, SUM(cena)
FROM Sprzedaż NATURAL JOIN Bary
     NATURAL JOIN Piwa
WHERE addr = 'Poznań'
  AND prod = 'Anheuser-Busch'
GROUP BY bar, piwo;
```

Perspektywy zmaterializowane

— Bezpośrednie wykonanie naszego zapytania dla tabeli Sprzedaż i tabel wymiarów może trwać za długo.
— Jeśli utworzymy perspektywę zmaterializowaną zawierającą odpowiednie informacje, będziemy mogli znacznie szybciej podać odpowiedź.

Przykład: Perspektywa zmaterializowana

— Jaka perspektywa mogłaby nam pomóc?
— Podstawowe wymagania:
1. Musi łączyć co najmniej Sprzedaż, Bary i Piwa.
2. Musi grupować co najmniej po bar i piwo.
3. Nie musi wybierać barów w Poznaniu ani piw Anheuser-Busch.
4. Nie musi wycinać kolumn adres ani prod.

— A oto przydatna perspektywa:

```
CREATE VIEW BaPiS(bar, adres, piwo,
                  prod, sprzedaż) AS
  SELECT bar, adres, piwo, prod,
         SUM(cena) sprzedaż
  FROM Sprzedaż NATURAL JOIN Bary
       NATURAL JOIN Piwa
  GROUP BY bar, adres, piwo, prod;
```

— Ponieważ bar → adres oraz piwo → prod, jest to pozorne grupowanie, konieczne ponieważ adres i prod występują we frazie SELECT.

— Przeformułowane zapytanie z użyciem zmaterializowanej perspektywy BaPiS:

```
SELECT bar, piwo, sprzedaż
FROM BaPiS
WHERE adres = 'Poznań'
  AND prod = 'Anheuser-Busch';
```

Aspekty materializacji
— Typ i częstość zapytań
— Czas odpowiedzi na zapytania
— Koszt pamięci
— Koszt aktualizacji

MOLAP i kostki danych
— Klucze tabel wymiarów stają się wymiarami hiperkostki.
  — Przykład: dla danych z tabeli Sprzedaż mamy cztery wymiary: bar, piwo, piwosz i czas.
— Atrybuty zależne (np. cena) występują w punktach (kratkach) kostki.

Marginesy
— Kostka często zawiera również agregacje (zwykle SUM) wzdłuż hiper-krawędzi kostki.
— *Marginesy* obejmują agregacje jednowymiarowe, dwuwymiarowe, ...

Przykład: marginesy
— Nasza 4-wymiarowa kostka Sprzedaż obejmuje sumy cena dla każdego baru, każdego piwa, każdego piwosza i każdej jednostki czasu (zapewne dni).
— Zawiera też sumy cena dla wszystkich par bar-piwo, trójek bar-piwosz-dzień, ...

Struktura kostki
— Każdy wymiar należy traktować jako mający dodatkową wartość *.
— Punkt wewnętrzny z jedną lub więcej współrzędną * zawiera agregaty po wymiarach z *.
— Przykład: Sprzedaż('Pod Żaglem', 'Bud', *, *) zawiera sumę piwa Bud wypitego w „Pod Żaglem" przez wszystkich piwoszy w dowolnym czasie.

Rozwijanie (drill-down)
— *Drill-down* = „deagregacja" = rozbija agregację na jej składniki.
— Przykład: po stwierdzeniu, że „Pod Żaglem" sprzedaje się bardzo mało piw Okocim, rozbić tę sprzedaż na poszczególne gatunki piw Okocim.

Zwijanie (roll-up)
— *Roll-up* = agregacja po jednym lub więcej wymiarach.
— Przykład: mając tabelę podającą jak wiele piwa Okocim wypija każdy piwosz w każdym barze, zwinąć ją do tabeli podającej ogólną ilość piwa Okocim wypijanego przez każdego z piwoszy.

Roll-Up i Drill-Down: przykłady
— Anheuser-Busch dla piwosz/bar

	Jim	Bob	Mary
Joe's Bar	45	33	30
Nut-House	50	36	42
Blue Chalk	38	31	40

— Zwinięcie po Bary
— A-B / piwosz

Jim	Mary	Bob
133	100	112

— Rozwinięcie po Piwa
— Piwa A-B / piwosz

	Jim	Bob	Mary
Bud	40	29	40
M'lob	45	31	37
Bud Light	48	40	35

Zmaterializowane perspektywy kostek danych
— Dla kostek danych warto robić perspektywy zmaterializowane agregujące po jednym lub więcej wymiarach.
— Wymiary nie powinny być całkowicie agregowane — można ewentualnie grupować po atrybucie z tabeli wymiaru.

Przykład
— Zmaterializowana perspektywa dla naszej kostki Sprzedaż mogłaby:
1. Agregować całkowicie po piwosz.
2. Nie agregować wcale po piwo.
3. Agregować po czasie według tydzień.
4. Agregować po miasto dla barów.

Indeksy
— Tradycyjne techniki
  — B-drzewa, tablice haszujące, R-drzewa, gridy, ...
— Specyficzne
  — listy odwrócone
  — indeksy bitmapowe
  — indeksy złączeniowe
  — indeksy tekstowe

Użycie list odwróconych
— Zapytanie:
  — Podaj osoby dla których wiek = 20 i imię = „Fred”
— Lista dla wiek = 20: r4, r18, r34, r35
— Lista dla imię = „Fred”: r18, r52
— Odpowiedzią jest przecięcie: r18

Użycie bitmap
— Zapytanie:
  — Podaj osoby dla których wiek = 20 i imię = „Fred”
— Mapa dla wiek = 20: 1101100000
— Mapa dla imię = „Fred”: 0100000001
— Odpowiedzią jest przecięcie: 010000000000
— Dobre jeśli mało wartości danych.
— Wektory bitowe można kompresować.

13. Obiektowe bazy danych – wprowadzenie

13.1. Modele danych

We współczesnych bazach danych są rozpowszechnione dwa podstawowe modele danych:
— Dominuje *relacyjny model danych*, w którym organizacja danych opiera się na pojęciu zbioru.
— Ostatnio dużą popularność zdobywa *obiektowy model danych*, będący w pewnym sensie mocno rozszerzoną wersją dawnego sieciowego modelu danych. Swoją popularność zawdzięcza dobremu dopasowaniu do obiektowych języków programowania.

13.2. Programowanie obiektowe

Na czym polega w uproszczeniu programowanie obiektowe? Podstawowym pojęciem jest *obiekt*, traktowany jako kontener zawierający pewien zbiór wartości. Z obiektem związuje się zbiór specyficznych operacji do obserwacji i zmiany stanu obiektu (czyli wartości w nim zawartych).

Musi istnieć możliwość odwołania się do obiektu, aby wykonać którąś jego operację, dlatego obiekty są nazwane (niekoniecznie bezpośrednio).

Obiekty mogą się do siebie odwoływać.
— obiekty *elementarne*: odwołują się tylko do swoich klas
— obiekty *złożone*: odwołują się także do innych obiektów (*bezpośrednio zależą* od nich)
Obiekt może jednak przestać istnieć, może się więc zdarzyć niepowodzenie podczas odwoływania się do niego.

13.2.1. Obiekty trwałe w programowaniu

— Często zachodzi potrzeba dłuższego przechowania niektórych obiektów (np. między sesjami)
— Takie obiekty nazywamy *trwałymi* (ang. *persistent*)
— Obiekty trwałe można zachować na dysku w repozytorium obiektów
— Program korzysta z repozytorium za pośrednictwem pamięci buforowej (*cache*, obiekty ładuje się na żądanie.
— Jeśli dany obiekt ma być trwały, to wszystkie obiekty do których się odwołuje też muszą być trwałe.
— Jako nazwy takiego obiektu (niekoniecznie jedynej) używa się PID (*persistent object identifier*)
— Repozytoria obiektów trwałych nie są dzielone, jeśli dodamy współbieżny dostęp otrzymamy prawdziwą obiektową bazę danych.

13.3. Relacyjny model danych

Podstawowe składowe:
— *relacje* odpowiadające zarówno typom encji (*entity*), jak i typom związków (*relationship*);
— *wiersze* (krotki) reprezentujące egzemplarze (wystąpienia) encji i związków;

— *kolumny* reprezentujące atrybuty;
— zbiór wartości, które można umieszczać w danej kolumnie, nazywa się jej *dziedziną*. Relacje są związane tylko wtedy, gdy mają kolumny o wspólnej dziedzinie.

Relacyjny model danych — operacje
— Operacje na danych opisuje się:
  — algebrą relacji;
  — rachunkiem relacji (specjalizacja rachunku predykatów pierwszego rzędu).
— W praktyce używa się języka SQL łączącego obie te metody.

Wady modelu relacyjnego
— Zorientowany na rekordy (dziedzictwo implementacyjne).
— Konieczność wcześniejszego określenia „schematu" bazy danych, dopiero potem można coś do bazy wstawiać.
— Nie można wstawiać danych nie pasujących do schematu, oficjalnie nie wolno dodawać nowych atrybutów.
— Za słaby do reprezentacji wiedzy: tylko jedna konstrukcja (tabela), atomowe atrybuty.

Zagnieżdżony relacyjny model danych
— Dopuszcza atrybuty o wartościach zbiorowych.
— Wartości zbiorowe reprezentuje się osobnymi, wewnętrznymi tabelami.

13.4. Obiektowy model danych

— Jedno podstawowe pojęcie do modelowania świata — *obiekt*.
— Z obiektem związany jest jego stan i zachowanie.
— Stan definiuje się wartościami własności (atrybutów) obiektu, mogą one być
  — wartościami elementarnymi (liczby, napisy) lub
  — obiektami, zawierającymi z kolei inne własności.
— Tak więc obiekty można definiować rekurencyjnie.
— Na rynku ok. 25 produktów, np. GemStone, ONTOS, ObjectStore, ENCORE.
  — Nie są to jednak jeszcze pełne i uniwersalne bazy danych.
— Dlatego wiele relacyjnych DBMS uzupełniono o możliwości obiektowe, np. Oracle.

Własności obiektowego modelu danych
— Zachowanie obiektu opisuje się zbiorem metod — procedur operujących na stanie obiektu.
— Każdy obiekt jest jednoznacznie identyfikowany systemowym identyfikatorem (OID).
— Proste definiowanie obiektów złożonych.
— Obiekty o takich samych własnościach i zachowaniu grupuje się w klasy. Obiekt może być egzemplarzem tylko jednej klasy lub wielu.
— Klasy łączy się w hierarchie dziedziczenia, w których podklasa *dziedziczy* własności i metody nadklasy, dokładając swoje specyficzne.
— Niektóre modele pozwalają na przesłanianie (*overriding*) — zmianę odziedziczonej własności lub metody.

Różnice w stosunku do modelu relacyjnego
— Atrybuty mogą być wielowartościowe.
  — Eliminuje to potrzebę używania większości złączeń równościowych.

— Możliwość definiowania związków odwrotnych.
— Nie trzeba definiować kluczy, ich rolę pełnią OIDy.
  — Eliminuje to modyfikowanie wartości klucza.
— Dwa rodzaje równości: identyczność i równość wartości.

— Złączenia używane gdy warunki w zapytaniu dotyczą porównywania wartości atrybutów. W przypadku „kluczy zewnętrznych" bezpośrednia nawigacja z użyciem OID.
— Przy ładowaniu powiązanych obiektów z bazy danych do pamięci (buforowanie) OIDy zastępuje się wskaźnikami (*pointer swizzling*), co wielokrotnie przyśpiesza nawigację.
— Wersjonowanie (długie transakcje) — tylko w niektórych OBD.

13.4.1. Zapytania

Tryby dostępu Dwa tryby dostępu:
— *Nawigacyjny*: mamy OID obiektu i zaczynając od niego przechodzimy po kolejnych referencjach. Zwykle używany w programach.
— *Język zapytań* (często podobny do SQL): zapytanie opisuje zbiór obiektów. Deklaratywność.

Przykład Przykład: *Znaleźć wszystkie projekty z budżetem ponad 100000 złp i mające sponsora z Warszawy*
— Relacyjnie
```
{p | (∃f)(∃a) Projekt(p) AND Firma(f) AND Adres(a)
 AND p.budżet > 100000 AND p.sponsor = f
 AND f.adres = a AND a.miasto = 'Warszawa'}
```
— Z użyciem wyrażeń ścieżkowych (co pozwala unikać zmiennych „roboczych")
```
{p | Projekt(p) AND p.budżet > 100000
  AND p.sponsor.adres.miasto = 'Warszawa'}
```

Wersje obiektów
— Potrzebne np. w systemach CAD do eksploracji wariantów.
— Każda wersja ma własny OID, ale zachowujemy *związki wyprowadzenia* między wersjami, najczęściej tworzące hierarchię.
— Hierarchia wersji zwykle trzymana w specjalnym *obiekcie generycznym*.
— Dwa rodzaje odwołań do wersji
  — *specyficzna* (albo *statyczna*): konkretna wersja
  — *generyczna* (albo *dynamiczna*): do *domyślnej* wersji (zwykle ostatniej).

13.4.2. Problemy

— Brak optymalizacji zapytań. Główny problem to wyrażenia ścieżkowe, trudno dla nich budować indeksy.
— Brak dobrej formalizacji, ale są już algebry podobne do algebry relacji.
— Pięć typowych operacji: suma (*union*, różnica (*difference*), selekcja (*select*), generowanie (*generate*), odwzorowanie (*map*).
— Jednak brak powiązania tych operacji z niskopoziomowymi operacjami fizycznymi (takiego jak w algebrze relacji RBD), stąd ich arbitralność.

— Brak uniwersalnych języków zapytań. Zwykle brak zagnieżdżonych podzapytań, funkcji agregujących, grupowania. Brak automatycznego wsparcia dla obsługi ekstensji klasy — zbioru jej egzemplarzy; użytkownik musi sam zdefiniować *kolekcję* (*collection*) i pilnować jej aktualizacji przy dodawaniu i usuwaniu obiektów.

— Kompozycyjność
  — W modelu relacyjnym wynik zapytania jest (anonimową) relacją.
  — Można go więc obrobić kolejnym zapytaniem, co umożliwia składanie zapytań.
  — W modelu obiektowym jest gorzej, obiekty ze zbioru wynikowego mogą nie należeć do żadnej klasy.
— Brak wsparcia dla redefiniowania klas (odpowiednik `ALTER` z SQL), np. dodawania lub usuwania atrybutów.
— Brak perspektyw (ale może są zbędne).
— Brak sensownego mechanizmu definiowania uprawnień, nie wiadomo jaka ziarnistość: obiekt, zbiór, klasa, fragment hierarchii?
— Bardzo ograniczone więzy spójności, konieczność realizacji metodami.
— Kłopoty z współbieżnością, ręczne operowanie blokadami.
  — Problem długich transakcji (unika się ich w systemach OLTP dla relacyjnych baz danych).
  — Propozycja: wspólna publiczna baza danych + prywatne bazy danych użytkowników.

13.5. Przykłady obiektowych baz danych

13.5.1. O_2

Definiowanie klasy wygląda w O_2 następująco:

```
class Instytut
  type tuple : (obszar-badań : string,
                nazwa-instytutu : string,
                adres : Adres,
                grupa-badawcza : set(Zespół))
  public read nazwa-instytutu, write obszar-badań
  method init(string, string, Adres, set(Zespół)) :
    Instytut is public;
```

Zamiast **tuple** można użyć **list** lub **set**, a nawet zagnieżdżać je w sobie.

Deklaracja metody występująca powyżej podaje jedynie jej *sygnaturę*.

Kolekcje muszą być tworzone ręcznie, mamy dwie możliwości. Możemy utworzyć kolekcję jako klasę

```
class Instytuty type set(Instytut);
```

albo też jako nazwaną wartość złożoną

```
name instytuty: set(Instytut);
```

Programy Standardowym językiem do pisania programów jest CO_2

```
execute co2 {
  o2 Instytut tmp;

  tmp = new(Instytut);
  instytuty = set(tmp);
}
```

Zapytania Zapytania można zadawać używając języka zapytań OQL, stanowiącego rozszerzenie SQL:

```
select x from x in instytuty
where x.obszar-badań = 'Bazy danych';
```

Można w nim używać wyrażeń ścieżkowych:

```
select x from x in instytuty
where x.obszar-badań = 'Bazy danych'
  and x.adres.kraj = 'Włochy';
```

13.5.2. Orion

```
make-class 'Instytut'
  superclasses: nil
  attributes: '((obszar-badań: domain string)
                (nazwa-instytutu: domain string)
                (adres: domain Adres)
                (grupa-badawcza:
                  domain (set-of Zespół)))
```

Uwagi:
— Można określać wartości domyślne dla atrybutów
— Wielodziedziczenie

Zapytania w Orionie
— Brak złączeń, rzutowanie tylko na jeden atrybut lub wszystkie.
— Powód: dzięki temu wynik zawsze należy do jakiejś klasy.

```
(Instytut select :I (:I obszar-badań = 'Bazy danych'))

(Instytut select (:I ((:I obszar-badań = 'Bazy danych')
                      and (:I adres kraj = 'Włochy'))))
```

13.5.3. ODMG

Propozycja ODMG

```
interface Instytut
  (extent Instytuty
   key nazwa)
  {attribute string nazwa;
   attribute string obszar-badań;
   attribute Adres adres;
   relationship Zespół grupa-badawcza
   inverse Zespół::afiliacja};
```

Zapytania w ODMG Dozwolone wyrażenia ścieżkowe

```
select distinct x from Instytuty x
where x.obszar-badań = 'Bazy danych'
  and x.adres.kraj = 'Włochy';
```

13.5.4. Implementacja

Rodzaje OID:
— *logiczne*: bez związku z adresem w pamięci zewnętrznej
— *fizyczne*: na podstawie położenia

Logiczne OID

— Orion: ¡id-klasy,id-egzemplarza¿. Definicje atrybutów i metod trzymane w obiekcie reprezentującym klasę, więc potrzebny szybki dostęp do niego. Migracja obiektu z klasy do klasy trudnsa, bo zmienia OID.
— GemStone: informacja o klasie w samym obiekcie, a nie w OID, wymaga wczesnego pobrania obiektu.

Fizyczne OID
— O_2 używa Wiss (Wisconsin Storage Subsystem), obiekt przechowywany jako rekord Wiss, OID jest identyfikatorem rekordem (RID).
  — Przy zmianie strony *forward reference*.
  — Wymagane tymczasowe OID dla obiektów utworzonych w pamięci, otrzymują trwały OID dopiero przy zatwierdzaniu (*commit*).
— Orion: w wersji rozproszonej OID dodatkowo zawierał identyfikator położenia, nie zmieniający się nawet przy migracji.

Zagadnienia otwarte
— Czy klasy są obiektami? Inaczej mówiąc, czy schemat trzymany jest osobno?
— Jakie są dozwolone związki między klasami?
— Czy i jak jest wspierana nawigacja?
— Jakie są operacje na obiektach?
— Jak identyfikuje się obiekty?
— Jak wybiera się obiekty?
— Jaką rolę pełni klasyfikacja?
— Czy klasy obiektów mogą ewoluować?
— Czy w rozproszonym systemie sieć powinna być widoczna?
— Czy jest specjalna obsługa obiektów aktywnych?
— Czy „świat" obiektów jest zamknięty (*closed*) czy otwarty?

— Sikha Bagui *Achievements and Weaknesses of Object-Oriented Databases*, Journal of Object Technology, vol. 2, no. 4, July-August 2003, str. 29–41, http://www.jot.fm/issues/issue_2003_07/column2.

13.6. Laboratorium: Obiektowe własności PostgreSQL

13.6.1. Dziedziczenie

Dziedziczenie to jedno z podstawowych pojęć obiektowych baz danych. PostgreSQL umożliwia nakładanie dziedziczenia na tabele.

Zobaczmy to na przykładzie. Utworzymy dwie tabele: tabelę miast i tabelę stolic. Ponieważ stolice także są miastami, chcemy uniknąc dublowania informacji. Jeden ze sposobów to

```
CREATE TABLE Stolice (
  nazwa       text,
  populacja   real,
  temperatura real
  kraj        char(3)
);

CREATE TABLE Inne_miasta (
  nazwa       text,
  populacja   real,
```

```
  temperatura real
);
```

gdzie listę wszystkich miast otrzymamy z:

```
CREATE VIEW miasta AS
  SELECT nazwa, populacja, temperatura FROM Stolice
    UNION
  SELECT nazwa, populacja, temperatura FROM Inne_miasta;
```

W PostgreSQL lepszym rozwiązaniem jest:

```
CREATE TABLE Miasta (
  nazwa       text,
  populacja   real,
  temperatura real
);

CREATE TABLE Stolice (
  kraj char(3)
) INHERITS (Miasta);
```

Tabela `Stolice` dziedziczy wszystkie kolumny (`nazwa`, `populacja`, `temperatura`) z macierzystej tabeli `Miasta`. W tabeli tej występuje jednak dodatkowa kolumna `kraj`.

Poniższe zapytanie podaje nazwy wszystkich miast (także stolic), w których średnia temperatura roczna jest wyższa niż 10degC:

```
SELECT nazwa, temperatura
  FROM Miasta
  WHERE temperatura > 500;
```

natomiast innym zapytaniem możemy otrzymać listę obejmującą jedynie te z nich, które nie są stolicami:

```
SELECT nazwa, temperatura
  FROM ONLY Miasta
  WHERE temperatura > 500;
```

Fraza `ONLY` powoduje, że zapytanie obejmujące jedynie wiersze znajdujące się bezpośrednio w tabeli `Miasta`, nie zaś w tabelach dziedziczących z niej. Frazy `ONLY` można używać w poleceniach `SELECT`, `UPDATE` i `DELETE`.

W PostgreSQL tabela może dziedziczyć również z kilku tabel. Struktura dziedziczenia nie może jednak zawierać cykli.

13.6.2. Definiowanie typów

W PostgreSQL można definiować typy w sposób zbliżony do SQL3. Definicja ma postać:

```
  CREATE TYPE typ AS OBJECT (
  lista atrybutów i metod
  );
```

Używając jej możemy zdefiniować typ dla punktów:

```
CREATE TYPE punkt_t AS OBJECT (
  x NUMERIC,
  y NUMERIC
);
```

Po zdefiniowaniu typu obiektowego używa się podobnie jak typów standardowych, np. możemy teraz zdefiniować typ dla odcinków:

```
CREATE TYPE odcinek_t AS OBJECT (
  początek punkt_t,
  koniec   punkt_t
);
```

Możemy teraz utworzyć tabelę zawierającą odcinki wraz z ich identyfikatorami (ID):

```
CREATE TABLE Odcinki (
  ID      INT,
  odcinek odcinek_t
);
```

Typy usuwamy poleceniem `DROP`:

```
DROP TYPE odcinek_t;
```

Przed usunięciem typu trzeba jednak oczywiście usunąć wszystkie tabele oraz inne typy, które używają danego typu a type, więc powyższe polecenie zakończy się niepowodzeniem. Trzeba najpierw usunąć tabelę `Odcinki`.

Typów zdefiniowanych można również używać bezpośrednio de definiowania tabel (odpowiednik „rowtype" z SQL3). W poleceniu `CREATE TABLE` można zastąpić listę kolumn słowem kluczowym `OF` i nazwą typu, np.

```
CREATE TABLE Odcinki OF odcinek_t;
```

Tworzenie obiektów zdefiniowanego typu

PostgreSQL dla każdego definiowanego typu tworzy automatycznie funkcję konstruktora o tej samej nazwie. Obiekt typu `punkt_t` tworzy się wywołując funkcję `odcinek_t` od (podanej w nawiasach) listy wartości atrybutów, np.

```
INSERT INTO Odcinki
VALUES(27, odcinek_t(punkt_t(0.0, 0.0),
                     punkt_t(3.0, 4.0)));
```

13.6.3. Zapytania ze zdefiniowanymi typami

Dostęp do składowych obiektu uzyskuje się używając notacji kropkowej. Jeśli O jest pewnym obiektem typu T, którego jedną ze składowych (atrybutów lub metod) jest A, to `O.A` odnosi się do tej składowej w obiekcie O.

Możemy wyszukać współrzędne początków owszystkich odcinków

```
SELECT o.odcinek.początek.x, o.odcinek.początek.y
FROM Odcinki o;
```

Użycie aliasu jest w takich sytuacjach obowiązkowe. Zapytanie

```
SELECT o.odcinek.koniec
FROM Odcinki o;
```

wyszuka końce wszystkich odcinków (wypisując je ewentualnie jako wywołania konstruktora.

13.6.4. Odwołania do obiektów (referencje)

Dla typu t wyrażenie `REF` t oznacza jego typ referencyjny („ID obiektu"). Można go używać w definicjach kolumn:

```
CREATE TABLE Odcinki (
  początek REF punkt_t,
  koniec   REF punkt_t
);
```

Używając takiego typu trzeba pamiętać o pewnych ograniczeniach. Konkretne referencje muszą odnosić się do wierszy tabeli podanego typu. Na przykład dla tabeli

```
CREATE TABLE Punkty OF punkt_t;
```

możemy umieścić w tabeli `Odcinki` referencje do wierszy z tej tabeli:

```
INSERT INTO Odcinki
SELECT REF(pp), REF(qq)
FROM Points pp, Points qq
WHERE pp.x < qq.x;
```

Referencje nie mogą natomiast odnosić się do obiektów występujących w kolumnach innych tabel, ani nie mogą być tworzone ad hoc (np. umieszczając w poleceniu `INSERT` klauzulę `VALUES(REF(punkt_t(1,2)), REF(punkt_t(3,4)))`).

Przechodzenie po referencji zapisuje się używając notacji kropkowej, np. poniższe zapytanie podaje współrzędne x początków i końców wszystkich odcinków w tabeli `Odcinki`.

```
SELECT ll.początek.x, ll.koniec.x
FROM Lines2 ll;
```

13.6.5. Tablice

W Postgresie jako typów kolumn można używać typów tablicowych. Wartością atrybutu może być wtedy cała tablica, tak jak dla kolumny `b` poniżej.

a	b
-	- - - - - -
-	- - - - - -
-	- - - - - -

Aby było to możliwe, należy najpierw zdefiniować relację jako typ, używając frazy `AS TABLE OF`, np.

```
CREATE TYPE wielobok_t AS TABLE OF punkt_t;
```

definiuje typ `wielobok_t` jako relację, której krotki są typu `punkt_t`, tzn. mają dwa atrybuty `x` i `y`.

Teraz można zdefiniować relację, w której jedna z kolumn będzie reprezentowała wieloboki, tzn. zbiory punktów, np.

```
CREATE TABLE Wieloboki (
  nazwa VARCHAR2(20),
  punkty wielobok_t)
NESTED TABLE punkty STORE AS TabPunkty;
```

Relacje odpowiadające poszczególnym wielobokom nie są zapisywane bezpośrednio jako wartości atrybutu `punkty`, lecz trzyma się je w pojedynczej tabeli, której nazwę należy zadeklarować. Do tabeli tej nie można się odwoływać bezpośrednio.

Przy wstawianiu wierszy do relacji `Wieloboki` używa się konstruktora typu dla zagnieżdżonej relacji (`wielobok_t`). Wartość zagnieżdżonej relacji podaje się w nim jako rozdzielaną przecinkami listę wartości odpowiedniego typu (`punkt_t`), najczęściej także używając konstruktora.

```
INSERT INTO Wieloboki
VALUES('kwadrat',
       wielobok_t(punkt_t(0.0, 0.0), punkt_t(0.0, 1.0),
                  punkt_t(1.0, 0.0), punkt_t(1.0, 1.0)));
```

Wierzchołki tego kwadratu podaje poniższe zapytanie:

```
SELECT punkty
FROM Wieloboki
WHERE nazwa = 'kwadrat';
```

14. Semistrukturalne bazy danych – wprowadzenie

Rozpowszechnienie Internetu i jego zasobów informacyjnych, dostępnych w postaci stron WWW, pociągnęło naturalnie za sobą myśl o wykorzystaniu ich jako baz danych. Stwarza to jednak szereg problemów.

Dane przechowywane w postaci stron HTML czy plików XML nie posiadają tak regularnej struktury, jak relacyjne bazy danych. Do określenia takich źródeł danych zaczęto używać terminu *dane semistrukturalne*.

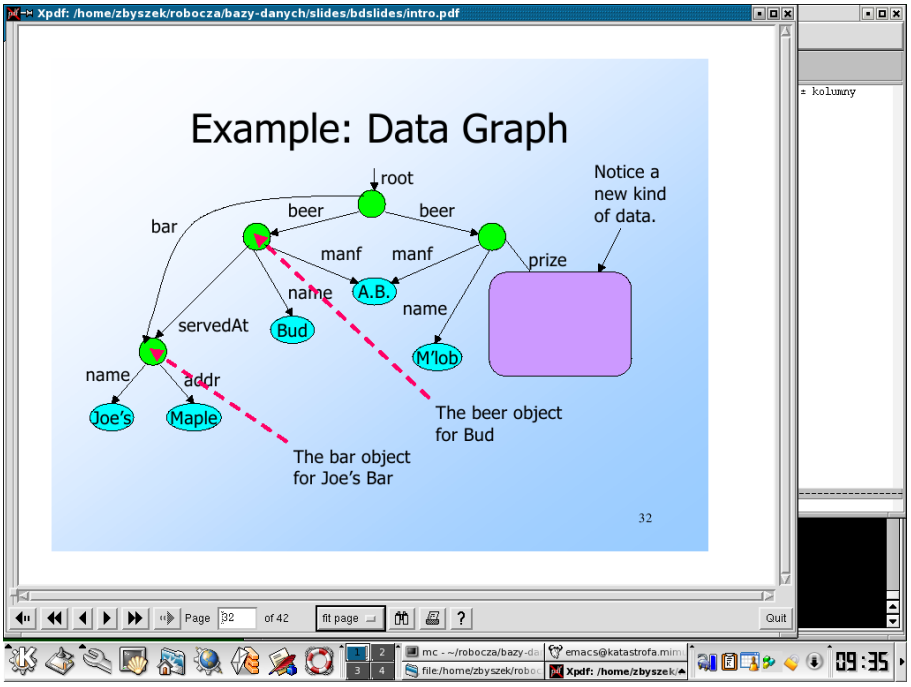
14.1. Dane semistrukturalne

Dane semistrukturalne to nowy model danych oparty na drzewach. Reprezentacja danych jest bardziej elestyczna niż w relacyjnych bazach danych — jako podstawę matematyczną modelu wybrano graf skierowany.

Schemat bazy jest często wpisany bezpośrednio w dane, można to określić jako dane „samo-opisujące się". Taka struktura bardzo dobrze pasuje do plików w XML, popularnej notacji do przechowywania informacji opartej na SGML.

Model demistrukturalny przydaje się też przy integracji informacji ze źródeł o nieco różnej strukturze, zwłaszcza przy budowie hurtowni wirtualnych.

Przykład takiej (niewielkiej) bazy danych możemy obejrzeć poniżej



Rysunek 14.1. Przykład.

14.2. Graf semistrukturalny

Grafem semistrukturalnym nazywamy graf skierowany o następujących własnościach:
— Jego wierchołkami są obiekty.
— Krawędzie są etykietowane atrybutami obiektu, z którego wychodzą.
— Wartości (atomowe) znajdują się wyłącznie w liściach drzewa.

Jak widać z powyższej definicji, model semistrukturalny daje dużą elastyczność: brak jest ograniczeń na etykiety wychodzących krawędzi ani na liczbę następników wierzchołka.

14.3. Zapytania

Języki zapytań (np. Lorel) są oparte na pojęciu ścieżki. Ścieżka jest to wyrażenie regularne, opisujące drogę od korzenia do poszukiwanego wierzchołka (lub wierzchołków).

Przykłady ścieżek:

```
biblio.ksiazka|artykul.autor
biblio._*.autor
```

14.3.1. Przykłady zapytań

Popatrzmy na kilka przykładów zapytań.

Podaj wszystkich autorów książek:

```
Query z1
select autor: x
from biblio.ksiazka.autor x;
```

Podaj wszystkie pozycje, których autorem jest Jeffrey Ullman:

```
Query z2
select pozycja: x
from biblio._ x
where "Jeffrey Ullman" in X.autor;
```

Podaj autorów wszystkich pozycji z bazą danych w tytule:

```
Query z3
select autor: x
from biblio._ x, x.autor y, x.tytul z
where ".*(D|d)atabase.*" ~  z;
```

Podaj autorów i tytuły wszystkich książek:

```
Query z4
select pozycja: tytul: y, autor: z
from biblio.ksiazka x, x.tytul y, x.autor z;
```

14.4. XML

Dzięki gwałtownemu rozwojowi sieci WWW i stron pisanych w HTML odkryto na nowo zalety nawiasowanej reprezentacji danych. Rozwój języka XML (*eXtensible Markup Language*), opartego na SGML, zmierza w kierunku standaryzacji reprezentacji danych zapisywanych tekstowo w plikach i przesyłanych siecią.

Obecnie większość narzędzi do wspomagania budowy oprogramowania (CASE — *Computer-Aided Software Engineering* jest wyposażona w możliwość odczytywania (importu) i zapisywania (eksportu) plików w notacji XML.

Język XML, podobnie jak HTML, służy do pisania dokumentów ze znacznikami (*tags*). W HTML jednak jest ustalony zbiór znaczników. Wybrano je pod kątem „prezentacji" informacji (inaczej mówiąc do jej formatowania), np. `<blockquote>` czy `<pre`.

W XML natomiast znaczniki są badziej semantyczne, np. `<student>`, i mogą być dowolnie definiowane przez piszącego.

Generalnie, XML jest to język do reprezentowania danych posiadających strukturę (najlepiej drzewiastą). Wyrażenie XML jest to w pełni nawiasowana forma zapisu danych. Nawiasy w XML posiadają etykiety (inaczej mówiąc są to *znaczniki frazowe*), na przykład zamiast zapisywać listę liczb 3, 5 i 4 w postaci

```
(3 5 4)
```

w XML napiszemy

```
<nawias>3 5 4</nawias>
```

Jednostki `<nawias>` i `</nawias>` nazywa się *znacznikiem początkowym* i *znacznikiem końcowym*, można jednak traktować je po prostu jako nawiasy otwierające i zamykające. Oczywiście nazwa znacznika w powyższym przykładzie nie została wybrana najlepiej, lepsza byłaby „lista".

14.4.1. Elementy

Używając XML można obudować znacznikami niemal dowolny ciąg znaków. Parę znaczników wraz z tekstem pomiędzy nimi nazywa się *elementem*, np.

```
<actor> ... </actor>
```

Ciąg znaków zawarty w znaczniku stanowi *nazwą* elementu, zaś tekst pomiędzy znacznikami to *zawartość* elementu, ponadto znacznik początkowy wyrażenia XML może zawierać *atrybuty*, na przykład poniższy element

```
<nawias title="oceny" date="2004-10-22">
3 5 4
</nawias>
```

ma dwa atrybuty: `title` i `date`. *Wartościami* tych atrybutów są napisy `"oceny"` oraz `"2004-10-22"`.

Wyjątek stanowią znane nam z HTML elementy proste, np. `<br>`. Nie posiadają one zawartości. W wersji HTML zgodnej z XML (tzw. XHTML) muszą one być zapisywane następująco: `<br />`. Uwaga: odstęp przed ukośnikiem nie jest konieczny, ale niektóre przeglądarki HTML go wymagają.

14.4.2. Zapis listowy w XML

Przypuśćmy, że chcemy zapisać protokół ocen studentów. W językach wysokiego poziomu o bogatej składni literałów (np. w Lispie, Prologu lub Dylanie) moglibyśmy użyć nawiasowanej listy

```
("Technologia produkcji oprogramowania" "Zima/2004"
 ("201" 78 88 69)
 ("202" 88 87 86)
 ("203" 99 88 88)
 ("204" 77 78 77)
 ("205" 90 89 81)
```

```
("206" 67 78 81))
```

Powyższa reprezentacja pochodzi z Lispu, w Prologu użylibyśmy nawiasów kwadratowych, a elementy list oddzielalibyśmy przecinkami.

Tę samą informację można zapisać w postaci wyrażenia XML. Warto zauważyć, że w tym zapisie informacja dotycząca przedmiotu i semestru podana jest w atrybutach elementu `<egzamin>`, co ułatwia wymianę informacji. Podobnie nazwisko studenta i indeks podane są jako atrybuty elementu `<wyniki>`. Dodatkowo ocena za każde zadanie została wydzielona w osobny element.

```
<egzamin przedmiot="TPO ZSI" semestr="Jesień 2004">
  <oceny indeks="201">
    <pkt>78</pkt> <pkt>88</pkt> <pkt>69</pkt>
  </oceny>
  <oceny indeks="202">
    <pkt>88</pkt> <pkt>87</pkt> <pkt>86</pkt>
  </oceny>
  <oceny indeks="203">
    <pkt>99</pkt> <pkt>88</pkt> <pkt>88</pkt>
  </oceny>
  <oceny indeks="204">
    <pkt>77</pkt> <pkt>78</pkt> <pkt>77</pkt>
  </oceny>
  <oceny indeks="205">
    <pkt>90</pkt> <pkt>89</pkt> <pkt>81</pkt>
  </oceny>
  <oceny indeks="206">
    <pkt>67</pkt> <pkt>78</pkt> <pkt>81</pkt>
  </oceny>
</egzamin>
```

Widać tu wyraźnie, że XML jest uogólnieniem nawiasowanego zapisu list. Nawiasy są nazywane (etykietowane), a każdy nawiasowany element może mieć dodatkowe atrybuty.

Jakkolwiek XML dopuszcza używanie dowolnych znaczników, w praktyce każdy dokument XML powinien posiadać uprzednio zdefiniowany *schemat*. Schemat dokumentu XML opisuje hierarchiczną strukturę dokumentu oraz dozwolone znaczniki używane dla elementów i ich atrybuty.

Używa się do tego osobnych dokumentów o specjalnej budowie. Początkowo było to DTD (Document Type Description), nowsze rozwiązanie to XML Schema.

Na opis schematu dokumentu można patrzeć jak na definicję używanego słownika.

14.4.3. Dokument

Z formalnego punktu widzenia *dokument XML* jest pojedynczym elementem (oczywiście zapewne zagnieżdżonym). Może być poprzedzony opcjonalnym prologiem zawierającym

— deklarację wersji XML
  `<?xml version="1.0" ?>`
— definicję typu dokumentu (DTD), najczęściej zawierającą odwołanie do osobnego pliku
  `<!DOCTYPE` *nazwa głównego elementu* `SYSTEM "`*plik*`.dtd">`

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Student SYSTEM "student.dtd">
<Student>
...
</Student>
```

Poprawność dokumentu XML można określić na dwóch poziomach. Pierwszy poziom to dokument *poprawnie zbudowany* (*well-formed*). Wymagana jest poprawność syntaktyczna, znaczniki muszą być „sparowane”: każdemu znacznikowi otwierającemu (np. `<student>` musi odpowiadać znacznik zamykający (np. `</student>`). Wartości atrybutów zawsze muszą być w cudzysłowach. Nie wymaga się natomiast określania DTD.

Poziom drugi określa się terminem *Valid*: dokument musi być dodatkowo opisany zadeklarowanym schematem DTD (*Document Type Definition*) i zgodny z nim ;-)

DTD

DTD powinien zawierać definicje wszystkich używanych w dokumencie elementów (czyli znaczników). Każdy element deklarujemy znacznikiem `!ELEMENT`, w którym podajemy nazwę elementu i jego składniki (czyli zawartość).

— `<!DOCTYPE` *element-główny* `[` *element* `... ]>`
— `<!ELEMENT` *element* `(`*składnik*`,...)>`

Przykład

```
<!DOCTYPE Studenci [
  <!ELEMENT Studenci (Student*)>
  <!ELEMENT Student (imie,nazwisko,adres,rok)>
  <!ELEMENT imie (#PCDATA)>
  <!ELEMENT nazwisko (#PCDATA)>
  ...
]>
```

Przy deklarowaniu składników elementu w znacznikach DTD używa się odmiany wyrażeń regularnych. Oprócz nazwanych elementów mogą tam wystąpić specjalne elementy, oznaczające zwykłe teksty

— `#PCDATA` to dowolny tekst nie zawierający znaczników
— `CDATA` to dowolny tekst

Po składnikach i pomiędzy nimi mogą wystąpić operatory oznaczające

`?`	element opcjonalny
`*`	0 lub więcej powtórzen
`+`	1 lub więcej powtórzen
`|`	alternatywa

Popatrzmy na przykład użycia DTD

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Studenci SYSTEM "student.dtd">
<Studenci>
  <Student>
  <imie>Onufry</imie>
  <nazwisko>Zagłoba</nazwisko>
  <adres>Dzikie Pola</adres>
  <rok>1648</rok>
  </Student>
  <Student>
  ...
  </Student>
  ...
]>
</Studenci>
```

Oprócz znaczników opisujących elementy w DTD mogą wystąpić znaczniki opisujące atrybuty elementów. Mają one postać

— `<!ATTLIST` *element atrybut zbiór wartości opcje*`>`

Opcje mogą zawierać

— wartość domyślną (ze zbioru wartości)
— `#REQUIRED` oznaczające atrybut wymagany

Przykład DTD z atrybutami

```
<!DOCTYPE Giełda [
<!ELEMENT giełda (tytuł?, kurs*)>
<!ELEMENT kurs (#PCDATA)>
<!ATTLIST kurs
  waluta CDATA #REQUIRED
  typ (sprzedaż|kupno|średni) "średni">
...
]>
```

i jego użycie

```
<?xml version="1.0" ?>
<giełda>
<tytuł>Kursy walut</tytuł>
<kurs waluta="USD">4,235</kurs>
...
</giełda>
```

DTD — atrybuty

Atrybuty umieszcza się w znaczniku otwierającym, aby sprecyzować pewne jego cechy. Każdy atrybut ma postać: *atrybut*=”*wartość*”.

Oprócz określania drugorzędnych cech znacznika atrybuty są także używane do łączenia elementów jako łączniki (*links*).

Atrybuty deklaruje się w DTD konstrukcją

```
<!ATTLIST element
  atrybut typ
  ...>
```

A oto przykład DTD z atrybutami

```
<!DOCTYPE Studenci [
  <!ELEMENT Studenci (Student*)>
  <!ELEMENT Student (imie,nazwisko,adres,rok)>
    <!ATTLIST Student
      studentID ID
      chodziNa IDREFS>
  <!ELEMENT nazwisko (#PCDATA)>
  ...
]>
```

i sposób jego użycia

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Studenci SYSTEM "student.dtd">
<Studenci>
  <Student studentID="OZ" chodziNa="ms,gpp">
  <imie>Onufry</imie>
  <nazwisko>Zagłoba</nazwisko>
  <adres>Dzikie Pola</adres>
  <rok>1648</rok>
  </Student>
  <Student>
  ...
  </Student>
  ...
```

```
]>
</Studenci>
```

14.4.4. Powiązania między dokumentami

Atrybuty łączące

Atrybuty łączące mają ustalone, standardowe nazwy.

Atrybut ID to podaje identyfikator znacznika z myślą o wykorzystaniu tego identyfikatora w innych elementach i dokumentach.

Atrybut IDREF jest to właśnie referencja do wartości atrybutu ID w innym elemencie.

Wadą atrybutów łączących jest brak „kontroli typów"! Jest to bowiem najprostszy mechanizm, można użyć bogatszych: XLink i XPointer.

Podjęzyk XLL (*eXtensible Link Language*), inaczej XLink, daje więcej możliwości opisu niż klasyczne odsyłacze HTML. Można tworzyć odsyłacz do wielu dokumentów (z wyborem przez użytkownika) oraz do grup dokumentów (odpowiednik ramki z wykazem pozostałych).

14.4.5. Wyświetlanie

Dokument zapisany w XML można przekształcić na inną postać, a także umieścić jako stronę na serwerze WWW. Trzeba jednak wtedy określić sposób wyświetlania.

Do prostych zastosowań wystarczą kaskadowe arkusze stylów (CSS, *Cascading Style Sheets*) z HTML. Arkusz CSS deklaruje się zwykle w nagłówku dokumentu konstrukcją

```
<link rel="stylesheet" type="text/css" href="nazwa.css">
```

Elementy stylu można też umieszczać jako atrybuty elementów

```
<li style="color: red">
```

W SGML do opisu sposobu prezentacji używa się DSSSL — języka zbliżonego do języka programowania Scheme.

W XML możemy określić sposób wyświetlania znaczników używając XSL (*eXtensible Stylesheet Language*), pozwalającego określić np. że
— znacznik `<giełda>` ma być wyświetlany jako tabela (`<table>`).
— znacznik `<kurs>` ma być wyświetlany jako wiersz tabeli (`<tr>`).

Istnieją parsery XML (i ogólnie SGML) pozwalające dokonać konwersji na dowolną inną postać (np. TEX).

14.4.6. Wymiana informacji a XML

Jedno z popularnych zastosowań dokumentów XML to wymiana informacji między różnymi narzędziami CASE.

Strukturę aplikacji wymodelowaną w UML w takich narzędziach jak Rational Rose, Umbrello czy Visual Paradigm można zapisać jako dokument XML i następnie wczytać do innego narzędzia, np. generatora specjalizowanej aplikacji, lub umieścić na stronie WWW.

Zasady odwzorowania:
— obiekty ↔ dokumenty XML
— klasy ↔ schematy XML

XMI

Do tych celów OMG (Object Management Group) [www.omg.org] zaproponowała jako standardowy format wymiany schemat XMI (XML Metadata Interchange) [http://cgi.omg.org/

cgi-bin/doc?ad/01-06-12]. Wiele ciekawych informacji na ten temat mozna znaleźć na stronie [http://XMLmodeling.com].

15. Wydajność

15.1. Strojenie bazy danych

Strojenie bazy danych obejmuje dwa aspekty:
— optymalizację wykorzystania procesora, pamięci i przestrzeni dyskowej przez operacje na bazie danych;
— optymalizację wykonywania zapytań (np. w Postgresie polecenia `CREATE INDEX`, `VACUUM`, `VACUUM ANALYSE`, `CLUSTER` i `EXPLAIN`).

15.2. Indeksy nad kolumnami tabeli

Potrzebne są co najmniej indeksy dla klucza głównego tabeli oraz każdego klucza obcego.

Do dużych zapytań często potrzebne są indeksy złożone, których klucze składają się z kilku kolumn. Sztuka korzystania z indeksów złożonych polega na odpowiednim wyborze kolejności kolumn w indeksie. Zamiast korzystać z intuicyjnie „naturalnej" kolejności kolumn należy zacząć od kolumny dającej największą redukcję.

Rozważmy na przykład indeks obejmujący kolumny kodu firmy, numeru konta oraz typu transakcji w tabeli *pozycji dziennika* bazy danych dla księgowości finansowej. Jeśli istnieją tylko dwie firmy, kolumna numeru konta redukuje liczbę wierszy wynikowych znacznie bardziej niż kolumna kodu firmy.

Podobnie typ transakcji redukuje dalej wiersze wynikowe bardziej niż kod firmy. Tak więc właściwa kolejność kolumn w indeksie to kod konta, typ transakcji i na końcu kod firmy.

Budowa optymalnych indeksów może o rząd wielkości zmienić czas wykonania zapytania.

15.3. Wybór indeksów

Podczas pisania zdań SQL w raportach warto mieć w pamięci kilka wskazówek:
— Upewnij się, że istnieje indeks dla każdej kolumny używanej w złączeniu. Indeks może obejmować inne kolumny z tabeli pod warunkiem, że kolumna złączenia występuje na pierwszym miejscu.
— Co najmniej jedna z kolumn złączenia powinna mieć unikalny indeks. Jest to naturalna reguła przy złączeniach opartych na związkach z diagramu związków encji. Jedna z kolumn w złączeniu będzie wtedy jednoznacznym identyfikatorem (kluczem głównym) tabeli.
— Zredukuj korzystanie z podzapytań skorelowanych w SQL — staraj się używać złączeń. Optymalizator SQL jest bowiem zwykle nastawiony na złączenia, zwłaszcza gdy istnieją odpowiednie indeksy. Zależy to jednak od konkretnej implementacji.

15.3.1. Rozmieszczenie wierszy tabeli

Poleceniem `CLUSTER` można zmienić kolejność wierszy w tabeli na zgodną z podanym indeksem. Ma to sens głównie dla indeksów nieunikalnych, gdy występuje wiele wierszy o tej

samej wartości indeksu, lub też w sytuacjach, gdy indeks jest używany do wybierania wierszy z pewnego zakresu klucza indeksowego.

15.4. Denormalizacja

Normalizacja upraszcza zachowanie integralności danych dzięki wyeliminowaniu ich dublowania. Prowadzi to jednak zwykle do zwiększenia liczby tabel używanych w pojedynczej funkcji, a co za tym idzie liczby złączeń — najbardziej kosztownej operacji.

Weźmy na przykład typowy dla komputeryzacji zwyczaj kodowania wszystkiego. Używa się kodów dla kolorów, kategorii, klas, stopni jakości — czyli każdej cechy, która może być opisana jedną z listy predefiniowanych wartości.

W znormalizowanej strukturze bazy danych dla każdego rozdzaju kodu jest potrzebna osobna tabela *danych słownikowych*, zawierająca opis tego kodu. Normalizacji dokonuje się po to, aby w razie zmiany opisu modyfikacja była wykonywana tylko w jednym miejscu.

Nie zawsze jednak elementy takich list ulegają częstym zmianom. Ponadto jeśli producent samochodów wprowadza nową barwę, np. cynober, to inne kolory dalej są aktualne. W praktyce więc do takich list dodaje się nowe elementy, ale nie modyfikuje istniejących. Opisy kodów można by więc przechowywać zarówno w głównej tabeli, jak i w tabeli słownikowej.

Tabela z danymi słownikowymi jest wciąż przydatna do weryfikacji danych przy wstawianiu lub modyfikacji. A przy okazji, często używany argument o możliwości błędów literowych staje się przestarzały, gdyż do wprowadzania opisów będziemy stosować wybór z listy. Tabela słownikowa przydaje się wtedy do wyświetlania listy wyborów.

Zaprojektowanie wydajnej bazy danych wymaga więc czasem kompromisu między szybkim dostępem a dublowaniem danych. Zdublowane dane wymagają bardziej złożonych algorytmów zachowania integralności, a to powoduje większą złożoność programów.

Przed przyspieszeniem dostępu należy zbadać rodzaje dostępu wymagane przez kluczowe funkcje i skutki korzystania wyłącznie ze znormalizowanych struktur.

15.5. Pamięć buforowa (cache)

Podstawowe cele przy optymalizacji wykorzystania zasobów serwera bazy danych to trzymanie potrzebnej informacji w pamięci operacyjnej (RAM) i unikanie niepotrzebnych dostępów do dysku.

W PostgreSQL używa się dzielonej pamięci zawierającej bufory. Standardowo przydziela się 64 bufory po 8kB każdy. Pamięć ta jest przydzielana podczas startu programu `postmaster` — serwera Postgresa.

Backend DBMS najpierw szuka żądanej informacji w pamięci buforowej. Dopiero gdy nie znajdzie zgłasza żądanie do systemu operacyjnego. Informacja jest wtedy ładowana z pamięci buforowej jądra lub z dysku.

Dlaczego rozmiar pamięci buforowej jest ważny?

— Jeśli cała tabela zmieści się w pamięci buforowej, to przeglądanie sekwencyjne jest bardzo szybkie.
— Natomiast jeśli zabraknie miejsca choćby na jeden blok dyskowy, przy każdym przeglądzie następuje wymiana — wczytanie brakujących bloków z dysku.
— Jeśli jako strategii wymiany używa się LRU (*least recently used*, to podczas pierwszego przeglądania usunięty zostanie z pamięci buforowej tylko pierwszy blok (żeby zrobić miejsce dla ostatniego).

— Jednak już przy drugim przeglądaniu aby ściągnąć do pamięci pierwszy blok zostanie z niej usunięty drugi (bo jest „najstarszy") itd.

Pozornie najlepiej byłoby przydzielić jak największą pamięć buforową. Nie jest to jednak prawda, bo wtedy zaczyna brakować miejsca dla programów:
— często występuje wymiatanie do pamięci wymiany (*swap*),
— czyli pojawiają się częstsze dostępy do dysku!
Należy to więc kontrolować, np. poleceniami `vmstat` i `sar` Unixa.

15.6. Dostęp do dysku

Dostęp do bloków dyskowych jest najszybszy wtedy, kiedy znajdują się one blisko aktualnego położenia głowicy. Kierując się tym, systemy operacyjne takie jak Unix próbują odpowiednio rozmieszczać pliki kierując się założeniem, że najczęściej stosowaną metodą dostępu jest sekwencyjne czytanie kolejnych bloków.

PostgreSQL przy czytaniu dużych ilości danych preferuje więc dostęp sekwencyjny, starając się wtedy nie korzystać z indeksów.

Należy jednak pamiętać, że głowica jest używana *nie tylko* dla plików bazy danych. Dlatego warto pliki bazy danych trzymać na osobnym dysku (rzecz jasna fizycznym, a nie na osobnej partycji dyskowej, bo to nic nie da).

Poleceniem `initlocation` Postgresa można rozmieszczać bazę danych na różnych dyskach. Można też stosować linki symboliczne ale wtedy są kłopoty z usuwaniem tabel w Postgresie
— Pliki `pg_database.oid` i `pg_class.relfilenode` odwzorowują bazy danych, tabele i indeksy na numeryczne końcówki plików.

Przy dużych bazach danych można też
— trzymać indeksy na innych dyskach niż ich tabele,
— rozdzielać często łączone tabele na osobne dyski.

15.7. Dziennik

Oczywiście warto (i powinno się) na osobnym dysku trzymać dziennik: katalog `pg_xlog` w Postgresie. Dziennik *nie* używa pamięci buforowej, ponieważ korzysta z zapisu natychmiastowego (*immediate write*). Transakcja jest kontynuowana dopiero po zakończeniu zapisu do dziennika.

15.8. Oczyszczanie bazy

Czasem sposób organizacji danych na dysku pociąga za sobą konieczność okresowego „sprzątania" bazy danych.

Przykładowo polecenie `VACUUM` w Postgresie służy do usuwania przeterminowanych wierszy z tabel bazy danych.
— Gdy Postgres modyfikuje wiersz, tworzy jego nową kopię, a starą zaznacza jako „przeterminowaną". Podobnie działa usuwanie.
— Dzięki temu inne transakcje mogą widzieć bazę danych w „starym" stanie.
— Po zakończeniu transakcji takie przeterminowane wiersze są już zbędne, ale nie są usuwane z tabel, ponieważ zakłada się, że miejsce po nich zostanie wykorzystane w przyszłości na nowo wstawiane wiersze.
— Po dużych zmainach w tabeli warto takie puste miejsca usunąć. Należy to robić w okresach małego obciążenia, ponieważ tabele muszą zostać w całości zablokowane.

Opcja `ANALYZE` polecenia `VACUUM` dodatkowo uaktualnia statystyki używane przez optymalizator zapytań.

15.9. Optymalizacja wykonania zapytań

Polecenie SQL `EXPLAIN` otrzymuje jako argument zapytanie SQL, lecz zamiast wykonać je, pokazuje plan wykonania zapytania. Można w ten sposób obserwować, czy utworzony indeks jest rzeczywiście wykorzystywany przez optymalizator Postgresa.

Plan pokazuje rodzaj przeszukiwaniu (*skanu*) dla poszczególnych tabel (skan sekwencyjny, skan indeksowy, ...). Jeśli zapytanie używa wielu tabel, to pokazuje się także użyte algorytmy złączenia.

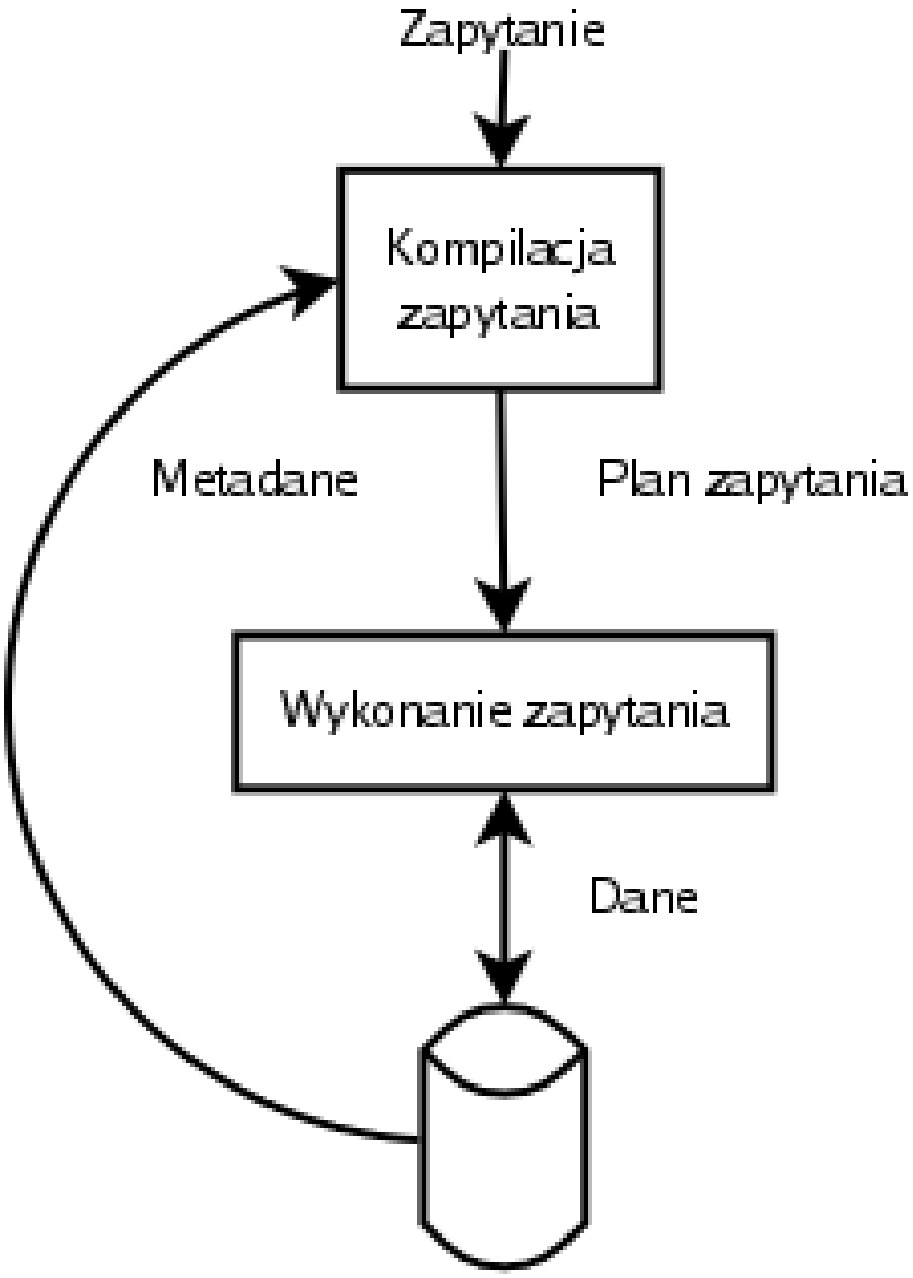
Popatrzmy na przykłady

```
bd=# explain select * from zwierz;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on zwierz  (cost=0.00..18.50 rows=850 width=61)
(1 row)

bd=# explain select * from zwierz where waga < 1000;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on zwierz  (cost=0.00..20.62 rows=283 width=61)
(1 row)

bd=# explain select * from zwierz where waga < 3;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using zwierz_waga on zwierz
 (cost=0.00..3.17 rows=10 width=61)
(1 row)

bd=# explain select * from zwierz, gatunki
bd-# where zwierz.gatunek = gatunki.gatunek
bd-#   and waga < 500;
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=29.58..54.09 rows=283 width=120)
   Hash Cond: ((zwierz.gatunek)::text = (gatunki.gatunek)::text)
   ->  Seq Scan on zwierz  (cost=0.00..20.62 rows=283 width=61)
         Filter: (waga < 500)
   ->  Hash  (cost=18.70..18.70 rows=870 width=59)
         ->  Seq Scan on gatunki  (cost=0.00..18.70 rows=870 width=59)
(6 rows)
```

Opis planu wykonania (*QUERY PLAN*) używa operatorów (*iteratorów*):

— Przeglądanie (skanowanie) tabel:
  — *table-scan*
  — *index-scan*
— Sortowanie podczas skanowania
  — *sort-scan*

15.10. Przetwarzanie zapytania



Rysunek 15.1. Procesor zapytań.

Zapytanie jest przetwarzane w dwóch etapach. Kompilacja zapytania składa się z następujących czynności:

1. Analiza składniowa zapytania (parsing). Sprawdzenie poprawności syntaktycznej i transformacja zapytania na drzewo wyrażeń.
2. Preprocessing: kontrola semantyczna, transformacje drzewa np. na wyrażenia algebry relacji.
3. Optymalizacja: najpierw wybór wybór *planu logicznego*, a następnie *planu fizycznego*, sformułowanego w operacjach sprzętowych.

Tak wykonany plan może być wykonany natychmiast lub przechowany w celu późniejszego (być może wielokrotnego) wykonania. Standard SQL obejmuje tę możliwość w postaci instrukcji PREPARE i EXECUTE.

Literatura

[1] Sikha Bagui. Achievements and weaknesses of object-oriented databases. *Journal of Object Technology*, 2(4):29–41, July-August 2003. http://www.jot.fm/issues/issue_2003_07/column2.

[2] R. Elmasri, S.B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley.

[3] H. Garcia-Molina, J.D. Ullman, J.Widom. *Systemy baz danych. Pełny wykład.* WNT.

[4] Date C. J. *An Introduction to Database System, vol. II.* Adison-Wesley Pub. Comp., tłum. polskie WNT – Warszawa (seria: Klasyka Informatyki), 2000.

[5] J. D. Ullman, J. Widom. *Podstawowy wykład z systemów baz danych.* WNT.