**Wykład 1**

***SQL - język relacyjnych i obiektowo-relacyjnych baz danych – podstawowe konstrukcje***

**Streszczenie**

W tym wykładzie rozpoczynamy naukę podstawowego języka baz danych - SQL. Warto przypomnieć, że wprowadzenie do języka SQL było już przedstawione na wykładach z "Relacyjnych Baz Danych". Część materiału będzie więc teraz przypomnieniem.

Rozpoczniemy od krótkiego wprowadzenia w dziedzinę baz danych w tym od przypomnienia definicji relacyjnego modelu danych (z [pierwszego wykładu](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/rW0.htm) z przedmiotu "Relacyjne bazy danych"). Następnie przejdziemy do nauki języka SQL przedstawiając podstawowe informacje na temat typów danych, instrukcji tworzących, modyfikujących i usuwających tabele bazy danych oraz instrukcji, które wstawiają, usuwają i modyfikują dane w tabelach w bazie danych. Na temat wymienionych instrukcji będzie też mowa w kolejnych wykładach. Na tym wykładzie rozpoczniemy też dyskusję centralnego tematu w dziedzinie baz danych, a mianowicie jak wydobywać dane z tabel w bazie danych. Wspomnimy także o programie SQL\*Plus, który jest standardowym, tekstowym interfejsem do baz danych Oracle.

**1.1** **Bazy danych**

Informacje (dane) są takim samym zasobem firmy, jak każdy inny (np. pracownicy, materiały, urządzenia) i wymagającym, tak samo jak one, zarządzania. Informacja może mieć wartość dla firmy tylko wtedy, gdy jest dokładna i dostępna wtedy, kiedy trzeba. *Baza danych* stała się standardową metodą strukturalizacji zarządzania informacją w większości organizacji.

Baza danych dotyczy zawsze pewnego fragmentu rzeczywistości związanego z firmą, organizacją lub innym działającym lub planowanym systemem. Stanowi kolekcję danych, których zadaniem jest reprezentowanie tego fragmentu rzeczywistości. Baza danych jest na ogół częścią *systemu informacyjnego*, obsługującego zapotrzebowania informacyjne związane z tym fragmentem rzeczywistości.

W kontekście systemów informacyjnych warto podkreślić wszechobecność informacji elektronicznej w codziennym życiu, dostępnej zaczynając od serwisów działających na przeglądarkach internetowych, a kończąc na skomplikowanych aplikacjach naukowych, jak projekty badania genomu człowieka i projekt obserwacji Ziemi prowadzony przez NASA. Warto tu też wspomnieć o zastosowaniach, które obejmują duże zbiory danych, takie jak biblioteki elektroniczne, multimedialne bazy danych i interaktywne wideo.

Każda baza danych jest oparta na ustalonym modelu danych określającym podstawowe zasady strukturalizacji i posługiwania się zbiorami danych. Obecnie najpopularniejszym modelem jest *relacyjny model danych*, który za chwilę przypomnimy. W wykładzie 7 będzie mowa o rozszerzeniu modelu relacyjnego do obiektowo-relacyjnego modelu danych. W dodatkowym wykładzie przedstawimy najnowszy model danych – oparty na języku XML.

Każda baza danych jest zarządzana przez wyspecjalizowany system, który zarządza zapisem i przechowywaniem danych w pamięci komputera (przede wszystkim na dyskach), zabezpiecza dane przed utratą i niepowołanym lub nieodpowiednim użyciem oraz udostępnia dane zapisane w bazie danych przy użyciu języka wysokiego poziomu opartego na przyjętym modelu danych.

Począwszy od tego wykładu aż do wykładu 7 będziemy prezentować język SQL służący do komunikacji z bazami danych (oparty na relacyjnym i obiektowo-relacyjnym modelu danych).

W wykładach od 8 do 13 będziemy studiować architekturę i zasady działania systemu zarządzania bazą danych.

Natomiast wykłady od 14 do 15 i dodatkowe 2 i 3 zawierają materiał dotyczący trzech specyficznych rodzajów systemów baz danych, odpowiednio systemów rozproszonych, hurtowni danych oraz baz danych opartych na języku XML.

**1.2** **Relacyjny model danych**

Relacyjny model danych pojawił się po raz pierwszy w artykule naukowym Edgara Codda w 1970 roku. W terminologii matematycznej *baza danych* jest zbiorem *relacji.*  Stąd historycznie pochodzi nazwa *relacyjny model danych* i *relacyjna baza danych*. W matematyce definiuje się *relację* jako podzbiór iloczynu kartezjańskiego zbiorów wartości. Reprezentacją relacji jest *dwuwymiarowa tabela* złożona z *kolumn* i *wierszy*.

Przyjmuje się następujące założenia o tabeli jako podstawowym elemencie relacyjnego modelu danych:

* Liczba kolumn jest z góry ustalona.
* Z każdą kolumną jest związana jej *nazwa* oraz *dziedzina*, określająca zbiór wartości, jakie mogą wystąpić w kolumnie.
* Na przecięciu wiersza i kolumny znajduje się pojedyncza (atomowa) wartość należąca do dziedziny kolumny.
* Wiersz reprezentuje jeden rekord informacji np. informacje o osobie.
* W modelu relacyjnym abstrahujemy od kolejności wierszy (rekordów) i kolumn (pól w rekordzie).

Przedstawimy teraz trzy tabele, które będą nam towarzyszyć w przykładach w większości wykładów.

W tabeli EMP zapisujemy informacje o pracownikach hipotetycznej firmy. Jeden wiersz reprezentuje dane jednego pracownika. W szczególności wynika z niego, że pracownik o nazwisku "King" pracuje w departamencie o numerze 10, piastuje stanowisko "PRESIDENT" oraz jego zarobki wynoszą 5000.

**EMP**

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

----- ------- --------- ---- --------- ----- ---- ------

7839 KING PRESIDENT NULL 17-NOV-81 5000 NULL 10

7698 BLAKE MANAGER 7839 01-MAY-81 2850 NULL 30

7782 CLARK MANAGER 7839 09-JUN-81 2450 NULL 10

7566 JONES MANAGER 7839 02-APR-81 2975 NULL 20

7654 MARTIN SALESMAN 7698 28-SEP-81 1250 1400 30

7499 ALLEN SALESMAN 7698 20-FEB-81 1600 300 30

7844 TURNER SALESMAN 7698 08-SEP-81 1500 0 30

7900 JAMES CLERK 7698 03-DEC-81 950 NULL 30

7521 WARD SALESMAN 7698 22-FEB-81 1250 500 30

7902 FORD ANALYST 7566 03-DEC-81 3000 NULL 20

7369 SMITH CLERK 7902 17-DEC-80 800 NULL 20

7788 SCOTT ANALYST 7566 09-DEC-82 3000 NULL 20

7876 ADAMS CLERK 7788 12-JAN-83 1100 NULL 20

7934 MILLER CLERK 7782 23-JAN-82 1300 NULL 10

Wśród wartości występujących w tabeli EMP znajduje się charakterystyczna dla baz danych tzw. *pseudo-wartość* NULL reprezentująca "brak wartości" w danym polu. Jest to coś innego niż po prostu puste miejsce lub zero. W szczególności możemy więc z tabeli EMP odczytać, że pracownik o nazwisku "King" nie ma przełożonego oraz nie dotyczy go prowizja (COMM).

W tabeli DEPT zapisujemy informacje o departamentach (działach) hipotetycznej firmy.

**DEPT**

DEPTNO DNAME LOC

------ ----------- --------

10 ACCOUNTING NEW YORK

20 RESEARCH DALLAS

30 SALES CHICAGO

40 OPERATIONS BOSTON

W tabeli SALGRADE zapisujemy informacje o klasach zaszeregowania zarobków pracowników hipotetycznej firmy.

**SALGRADE**

GRADE LOSAL HISAL

----- ----- -----

1 700 1200

2 1201 1400

3 1401 2000

4 2001 3000

5 3001 9999

W szczególności możemy z tabeli SALGRADE odczytać, że grupa zaszeregowania 1 obejmuje zarobki od 700 do 1200.

Tabele EMP, DEPT i SALGRADE stanowią przykładową bazę danych używaną do nauki SQL w materiałach firmy Oracle.

|  |
| --- |
| **Znaczenie kolumny *Deptno* w tabeli *Emp***   * Jej wartości nie opisują cechy pracownika. * Reprezentują *związek* danego pracownika z departamentem, o którym informacja znajduje się w innej tabeli i tylko korzystając z identyfikatora możemy rozpoznać w innej tabeli wiersz właściwego departamentu i odczytać o nim informacje. * Istotne jest więc, aby identyfikator ten jednoznacznie określał dany departament. W modelu relacyjnym nie ma innej możliwości identyfikacji wiersza tylko poprzez wartości kolumn, które jednoznacznie identyfikują wiersz. |

**Klucz główny i jednoznaczny**

* Dla każdej tabeli musi być określony jednoznaczny identyfikator nazywany *kluczem głównym* - zbiór złożony z jednej lub więcej kolumn, w których wartości jednoznacznie identyfikują cały wiersz.
* *Klucz jednoznaczny* (nazywany też *kluczem alternatywnym* lub w skrócie *kluczem*) ma tę samą własność co klucz główny przy czym klucz główny jest tylko jeden, kluczy jednoznacznych w tabeli może być więcej niż jeden.

|  |
| --- |
| * W tabeli *Emp* kluczem głównym jest jednoznaczny numer pracownika *Empno*. *Ename* nie musi być kluczem! * W tabeli *Dept* kluczem głównym jest jednoznaczny numer departamentu *Deptno*. * W tabeli *Salgrade* kluczem głównym jest jednoznaczny numer zaszeregowania *Grade*. |

**Klucz obcy**

* *Klucz obcy* jest to zbiór złożony z jednej lub więcej kolumn, których wartości występują jako wartości ustalonego klucza głównego lub jednoznacznego w tej samej lub innej tabeli i są interpretowane jako wskaźniki do wierszy w tej drugiej tabeli.

|  |
| --- |
| * W tabeli *Emp* kluczem obcym jest numer *Deptno*, którego wartości pochodzą z kolumny *Deptno* w tabeli *Dept*.  Na przykład, wartość 10 występująca w wierszu pracownika o nazwisku "King" tabeli *Emp* stanowi odwołanie do wiersza w tabeli *Dept*, w którym są zapisane informacje o departamencie o numerze 10 i nazwie "Accounting":   *"Pracownik King pracuje w departamencie Accounting"*   * W tabeli *Emp* kluczem obcym jest też numer *Mgr*, którego wartości pochodzą z kolumny *Empno* w tej samej tabeli *Emp*.  Na przykład, wartość 7839 występująca w wierszu pracownika o nazwisku "Blake" tabeli *Emp* stanowi odwołanie do wiersza w tej samej tabeli *Emp*, w którym są zapisane informacje o pracowniku o numerze 7839 i nazwisku "King":   *"Przełożonym pracownika Blake jest pracownik King"* |

Każdy schemat bazy danych można przedstawić graficznie. Tabele są reprezentowane jako ramki, powiązania między tabelami typu klucz obcy -> klucz główny są reprezentowane za pomocą strzałek. Na rysunku 1.1 są przedstawione tabele naszej przykładowej bazy danych oraz związki między nimi. Związek tabeli SALGRADE z tabelą EMP nie jest typu klucz obcy -> klucz główny – został opisany tekstowo jako warunek między wartościami w kolumnach w odpowiadających sobie wierszach obu tabel.

|  |
| --- |
|  |

Rys. 1.1 Diagram tabel EMP, DEPT i SALGRADE. Oznaczenie PK wskazuje na klucz główny. Oznaczenie FK z numerem oznacza klucz obcy. Diagram został utworzony przy użyciu narzędzia do tworzenia diagramów informatycznych o nazwie Microsoft Visio.

**1.3** **Historia języka SQL**

Modele danych poprzedzające model relacyjny, to jest model sieciowy i hierarchiczny,  opierały się na pojęciu pliku jako zbioru rekordów, przy czym rekordy były rozszerzane o pola wskaźnikowe kierujące do rekordów w tym samym lub innych plikach. Nie było potrzeby wprowadzania nowego języka programowania – wystarczały tradycyjne języki programowania jak Cobol i PL/I.

Dla baz danych opartych na relacyjnym modelu danych opracowano specjalny język o nazwie SQL (Structured Query Language - Strukturalny Język Zapytań), umożliwiający dostęp i przetwarzanie danych w bazie danych na poziomie obiektów modelu relacyjnego tj. tabel.

Ponadto zaistniała potrzeba określenia sposobu użycia instrukcji tego języka w programach konwencjonalnych języków programowania jak C, C++, Java, Visual Basic. Kwestia ta została również objęta Standardem SQL.

Znamienne też jest powszechne stosowanie narzędzi do generowania aplikacji klienckich jak Oracle Forms, Visual C++, Visual Basic, MS Access bez potrzeby sięgania do tradycyjnego sposobu programowania.

Język SQL został przyjęty jako standard przez społeczność informatyczną. Jego historia sięga początku lat siedemdziesiątych, kiedy to w laboratorium badawczym IBM w San Jose powstał język o nazwie *Sequel*.

Pierwszy standard międzynarodowej organizacji ISO języka SQL powstał w 1987 roku, drugi w 1989, a trzeci w 1992 roku (SQL2), czwarty w 1999 roku (SQL’1999); nad następnym standardem uwzględniającym między innymi cechy obiektowe trwają prace. Oprócz samego Standardu SQL załączamy także konstrukcje z najbardziej znanego obecnie systemu relacyjnych baz danych - Oracle9i.

Byłoby dobrze, chociaż nie jest to konieczne, aby Czytelnik miał dostęp do komputera z zainstalowanym systemem Oracle. Oprogramowanie serwera Oracle w postaci wersji bezpłatnej, ważnej przez pewien określony czas, można uzyskać pod internetowym adresem: <http://technet.oracle.com/>.

PJWSTK zapewnia dwa interfejsy do łączenia się z bazą danych Oracle z zewnątrz. Jednym z nich jest iSQL\*Plus (interfejs WWW), a drugim SQL\*Plus (standardowy, interakcyjny interfejs do bazy danych Oracle). Oczywiście zamiast SQL\*Plus możemy użyć innego oprogramowania klienckiego np. TOAD, TOra. Obydwa sposoby dostępu zapewniają podobne możliwości, przy czym do korzystania z iSQL potrzebna jest tylko przeglądarka natomiast do korzystania z SQL\*Plus konieczne jest jego zainstalowanie na swoim komputerze.

W naszym kursie używamy przykładowej bazy danych Oracle. Można ją założyć wykonując skrypt z poleceniami SQL z pliku o nazwie *demobld.sql*.

**1.4** **Definiowanie danych**

W tym punkcie przedstawimy podstawowe informacje na temat typów danych, instrukcji tworzących, modyfikujących i usuwających tabele w bazie danych. Wspomnimy także o programie SQL\*Plus będącym standardowym tekstowym interfejsem do bazy danych Oracle.

**Standardowe typy danych**

W zakresie typów danych nie ma, niestety, zgodności wśród firm dostarczających systemy baz danych. Poniżej wyliczamy typy danych, które występują w Standardzie, a oprócz tego zamieszczamy informacje o typach danych występujących w Oracle.

Z punktu widzenia tego wykładu wchodzenie w szczegóły implementacyjne typów danych nie jest istotne – do przedstawienia instrukcji języka SQL używać będziemy trzech typów: napisowego VARCHAR2, liczbowego NUMBER i datowego DATE według składni systemu Oracle. Dla pełności obrazu wyliczamy pełny zestaw standardowych typów danych.

|  |
| --- |
| 1. **Typy napisowe** (stałe są zapisywane w pojedynczych apostrofach, np.: 'Jan III', 'Kowalski '):  · CHARACTER(N), CHAR(N) - napis znakowy stałej długości N (w Oracle CHAR(N)).  · NATIONAL CHARACTER(N), NATIONAL CHAR(N) - napis znakowy stałej długości N zapisany w alfabecie narodowym (w Oracle NCHAR(N)).  · CHARACTER VARYING(N), VARCHAR(N) - napis znakowy zmiennej długości (w Oracle VARCHAR(N) i zalecane VARCHAR2(N)).  · NATIONAL CHARACTER VARYING(N), NATIONAL CHAR VARYING(N), NCHAR VARYING(N) - napis znakowy zmiennej długości zapisany w alfabecie narodowym (w Oracle NVARCHAR2(N)).  · BIT(N) - ciąg bitów długości N. Ciągi bitów są głównie używane do reprezentowania danych graficznych i dźwiękowych.  · BIT VARYING(N) - ciąg bitów zmiennej długości (w Oracle RAW(n) i LONG RAW).  2. **Typy numeryczne**:  · NUMERIC, DECIMAL, NUMERIC(p,s) - typ dziesiętny (W Oracle NUMBER(p,s)).  · FLOAT, REAL - typ zmiennopozycyjny.  · INTEGER - typ całkowity.  3. **Typy dat**:  · DATE (data), TIME (godzina), TIMESTAMP (data i godzina) - typ rozszerzający dane zawarte w typie danych  DATE (W Oracle nie ma typu TIME).  4. **Typy przedziału czasu** - zaczynające się od słowa INTERVAL.  5. **W Oracle** są jeszcze typy:  · LONG - dokumenty tekstowe (w jednej tabeli może być tylko jedna kolumna tego typu).  · BLOB – duże obiekty binarne.  · BFILE - duże obiekty binarne zapisywane w plikach.  · CLOB – duże obiekty tekstowe.  · NCLOB – duże obiekty tekstowe zapisane w alfabecie narodowym.  · ROWID - identyfikatory (adresy) wierszy tabel.  6. **Nowa wersja standardu SQL’1999** wprowadza cztery nowe typy danych:  · LOB – duże obiekty albo binarne, albo tekstowe.  · BLOB - duże obiekty binarne – podtyp typu LOB.  · CLOB – duże obiekty tekstowe – podtyp typu LOB.  · BOOLEAN – wartości logiczne true, false i unknown. |

Tak jak to podaliśmy powyżej, do ilustracji omawianych konstrukcji języka SQL będziemy używać przykładowej bazy danych dostarczanej przez firmę Oracle. Tabela *Emp* zawiera informacje o pracownikach, tabela *Dept* informacje o działach, w których pracują pracownicy a tabela *Salgrade* informacje o klasach zaszeregowania zarobków pracowników. Wyniki przykładowych zapytań będą obliczane względem ich zawartości podanych powyżej.

Zauważmy, że związek atrybutu *Sal* w tabeli *Emp* z wartościami w tabeli *Salgrade* nie jest standardowy, tzn. jest innego typu niż *klucz-obcy*->*klucz-główny*. Mianowicie, zarobki pracownika *Emp.Sal* są zaliczane do grupy *Salgrade.Grade* takiej, że:

|  |
| --- |
| *Emp.Sal* należy do przedziału: [*Salgrade.Losal*, *Salgrade.Hisal*] |

**Tworzenie tabeli**

Tabela jest tworzona za pomocą następującej instrukcji:

|  |
| --- |
| CREATE TABLE *nazwa\_tabeli*( *nazwa\_kolumny typ\_danych więzy\_spójności*,...); |

Rodzaje więzów spójności omówimy po przedstawieniu przykładu. Przypomnijmy najpierw  podstawowe zasady notacji definiowania składni instrukcji i wyrażeń.

1. Zapis: *nazwa*,...  daje możliwość użycia jednej lub więcej nazw rozdzielonych przecinkami.
2. Meta-nawiasy [...] oznaczają opcjonalne fragmenty.

Tabele przykładowej bazy danych można utworzyć za pomocą następujących instrukcji:

|  |
| --- |
| **CREATE TABLE** Dept(   Deptno NUMBER(2) PRIMARY KEY,   Dname VARCHAR2(14),   Loc VARCHAR2(13) );  **CREATE TABLE** Emp(   Empno NUMBER(4) PRIMARY KEY,   Ename VARCHAR2(10),   Job VARCHAR2(9),   Mgr NUMBER(4) REFERENCES Emp,   Hiredate DATE,   Sal NUMBER(7,2),   Comm NUMBER(7,2),   Deptno NUMBER(2) NOT NULL REFERENCES Dept );  **CREATE TABLE** Salgrade(   Grade NUMBER(2) PRIMARY KEY,   Losal NUMBER(7,2) NOT NULL,   Hisal NUMBER(7,2) NOT NULL, ); |

Oto znaczenie użytych powyżej, podstawowych więzów spójności:

NOT NULL – pseudo-wartość *Null* nie jest dozwolona w danej kolumnie,

PRIMARY KEY – dana kolumna stanowi klucz główny,

REFERENCES *nazwa\_tabeli* – dana kolumna stanowi klucz obcy odwołujący się do klucza głównego podanej tabeli.

Używając instrukcji programu SQL\*Plus:

|  |
| --- |
| **describe** *tabela* |

można przekonać się, jaki jest schemat utworzonej tabeli. Np. wykonanie instrukcji describe Dept spowoduje wyświetlenie na ekranie:

|  |
| --- |
| SQL> describe Dept  Name Null? Type  ------------------ ----------- -----------  DEPTNO NOT NULL NUMBER(2)  DNAME VARCHAR2(14)  LOC VARCHAR2(13) |

Natomiast wyświetlenie zawartości tabeli *Dept* odbywa się za pomocą instrukcji:

SELECT \* FROM Dept;

co po wprowadzeniu przykładowych danych dałoby (w programie SQL\*Plus) następujący rezultat:

|  |
| --- |
| SQL> SELECT \* FROM Dept;  DEPTNO DNAME LOC  ---------- -------------- -------------  10 ACCOUNTING NEW YORK  20 RESEARCH DALLAS  30 SALES CHICAGO  40 OPERATIONS BOSTON |

Zwróćmy uwagę na przedrostek SQL>, który sygnalizuje linię poleceń w programie SQL\*Plus. Będziemy go dalej pomijać.

**Usuwanie tabeli**

Usunięcie tabeli jest prostsze niż jej założenie:

|  |
| --- |
| DROP TABLE *nazwa\_tabeli*; |

np.

DROP TABLE Emp;

Jednak gdy inne tabele mają klucze obce odwołujące się do danej tabeli, jej usunięcie nie powiedzie się. Np. w naszej przykładowej bazie danych nie powiodłaby się instrukcja:

DROP TABLE Dept;

gdyż w tabeli *Emp* jest klucz obcy *Deptno* odwołujący się do klucza głównego tabeli *Dept*. Dopiero użycie dodatkowej klauzuli CASCADE w Standardzie i CASCADE CONSTRAINTS w Oracle spowoduje usunięcie tabeli *Dept* oczywiście razem z więzami spójności referencyjnej dla kolumny *Deptno*.

DROP TABLE Dept CASCADE CONSTRAINTS;

**Zmiana schematu tabeli**

Schemat istniejącej tabeli może być zmieniony za pomocą instrukcji ALTER TABLE.

|  |
| --- |
| ALTER TABLE *nazwa\_tabeli*  *klauzula\_zmiany\_schematu*; |

Może to być:

* dodanie nowej kolumny (za pomocą klauzuli ADD),
* rozszerzenie typu danych istniejącej kolumny (w standardzie za pomocą klauzuli ALTER COLUMN, a w Oracle za pomocą klauzuli MODIFY),
* usunięcie kolumny (w Standardzie za pomocą klauzuli DROP COLUMN, a w Oracle za pomocą klauzuli DROP),
* a także dodanie nowych więzów spójności.

Oto przykład dodania nowej kolumny w Standardzie

ALTER TABLE Emp ADD COLUMN Pesel CHAR(11);

i w Oracle

ALTER TABLE Emp ADD Pesel CHAR(11);

Nowa kolumna zostaje wypełniona pseudo-wartościami NULL.

Oto przykład zmiany typu danych kolumny w Oracle

ALTER TABLE Emp MODIFY Pesel VARCHAR2(20);

Jest też możliwe usunięcia kolumny:

ALTER TABLE Emp DROP Pesel;

Używając polecenia ALTER TABLE można definiować tabele z cyklicznymi powiązaniami REFERENCES między sobą, co nie jest możliwe za pomocą samego CREATE TABLE. Na przykład, jeśli klucz obcy w tabeli *A* odwołuje się do klucza głównego w tabeli *B*, a klucz obcy w tabeli *B* odwołuje się do klucza głównego w tabeli *A*, to mamy do czynienia z cyklicznymi powiązaniami REFERENCES. W takim przypadku najpierw wykonujemy instrukcje CREATE TABLE nie definiując kluczy obcych. Następnie, za pomocą instrukcji ALTER TABLE wprowadzamy więzy REFERENCES odwołujące się do istniejących już w tym momencie tabel.

Na przykład, zakładając, że utworzyliśmy tabelę *Emp* bez definiowania klucza obcego do tabeli *Dept*, moglibyśmy ten klucz obcy wprowadzić wykonując instrukcję:

ALTER TABLE Emp  
ADD CONSTRAINT fk\_Emp\_dept  
FOREIGN KEY (Deptno) REFERENCES Dept;

**Zakładanie indeksów (Oracle)**

Indeksy stanowią podstawową strukturę danych w bazach danych umożliwiającą:

· sprawdzanie warunku jednoznaczności klucza (głównego, jednoznacznego);

· szybkie wyszukiwanie informacji w tabelach.

Jednakże Standard języka SQL, nie określa jak zakładać indeks. Pozostawia opracowanie składni producentom systemów baz danych. Definiując indeks podajemy jego nazwę, nazwę tabeli, do której odnosi się oraz wyróżnione kolumny tej tabeli nazywane *pozycjami indeksu*. W zamierzeniu indeks jest strukturą danych, która ma umożliwiać szybkie wyszukiwanie wierszy tabeli w oparciu o wartości pozycji indeksu.

Poniżej przedstawiamy składnię stosowaną w systemie Oracle.

|  |
| --- |
| CREATE [UNIQUE] INDEX *nazwa\_indeksu* ON *nazwa\_tabeli*(*kolumna* [ASC|DESC] ,...); |

Dwie opcje występujące w składni to:

· UNIQUE – wartości pozycji indeksu muszą jednoznacznie określać wiersz tabeli; indeks taki nazywa się *indeksem jednoznacznym*; opcja ta umożliwia zatem realizację więzów spójności UNIQUE;

· ASC (domyślne) i DESC – w jakiej kolejności utrzymywać pozycje indeksu: rosnącej czy malejącej. Odpowiednie uporządkowanie pozycji indeksu może przyśpieszyć wypisywanie wierszy tabeli w wymaganym porządku.

Przykład

Instrukcja

CREATE UNIQUE INDEX Ind\_Emp\_num ON Emp(Empno);

tworzy indeks jednoznaczny na kolumnie *Empno* tabeli *Emp*. To znaczy, system będzie sprawować kontrolę nad tym, aby wartości w kolumnie *Empno* nie powtarzały się.

Natomiast, instrukcja

CREATE INDEX Ind\_Emp\_naz ON Emp(Ename DESC);

tworzy zwykły indeks na kolumnie *Ename* tabeli *Emp*, tzn. wartości w kolumnie *Ename* mogą się powtarzać. Pozycje tego indeksu są trzymane w odwrotnym porządku leksykograficznym.

Indeks można usunąć z bazy danych za pomocą następującej instrukcji:

|  |
| --- |
| DROP INDEX *nazwa\_indeksu*; |

System zarządzania bazą danych zwykle sam zakłada indeksy dla kluczy głównych i jednoznacznych.

**1.5** **Wprowadzanie zmian w tabelach bazy danych**

**Wstawianie danych**

Do tabeli w bazie danych poszczególne wiersze są wstawiane za pomocą instrukcji INSERT. Jej podstawowa postać jest następująca:

|  |
| --- |
| INSERT INTO *nazwa\_tabeli* VALUES (*wartość*,...); |

Przykład

|  |
| --- |
| Wstaw do bazy nowego pracownika o nazwisku 'Kowalski'. |

Wykonujemy instrukcję:

INSERT INTO Emp VALUES (9901, 'Kowalski', 'SALESMAN', 9345, '23-JAN-90',300,100,12);

Można też dokonywać wstawienia ograniczając się do kolumn o podanych nazwach (do nie wymienionych system sam wstawia pseudo-wartość *Null*):

|  |
| --- |
| INSERT INTO *nazwa\_tabeli*(*nazwa\_kolumny*,...) VALUES (*wartość*,...); |

Przykład

INSERT INTO Emp(Empno, Ename, Deptno)  
VALUES (9901, 'Kowalski', 10);

**Modyfikowanie danych**

System zarządzania bazą danych dokonuje modyfikacji wartości zapisanych w bazie danych, jeśli otrzyma do wykonania instrukcję UPDATE:

|  |
| --- |
| UPDATE *nazwa\_tabeli* SET *nazwa\_kolumny* = *wyrażenie*,...  [WHERE *warunek*]; |

Przykład

|  |
| --- |
| Podnieś wszystkim sprzedawcom (SALESMAN) zarobki *Sal* o 10%. |

Wykonujemy instrukcję:

UPDATE Emp SET Sal = Sal\*1.1  
WHERE Job = 'SALESMAN';

**Usuwanie danych**

Dane (wiersze) można usuwać z tabel w bazie danych kierując do systemu następującą instrukcję:

|  |
| --- |
| DELETE FROM *nazwa\_tabeli* [WHERE *warunek*]; |

Przykład

|  |
| --- |
| Usuń wszystkich pracowników, którzy nie mają określonego stanowiska (Job IS NULL). |

Wykonujemy instrukcję:

DELETE FROM Emp  
WHERE Job IS NULL;

**Zatwierdzanie zmian w bazie danych**

Instrukcje INSERT, DELETE i UPDATE nie dokonują same trwałych zmian w bazie danych. Aby zmiany wprowadzone przez nie utrwalić, należy wykonać instrukcję:

|  |
| --- |
| COMMIT [WORK]; |

Można też zrezygnować z wprowadzenia zmian do bazy danych, wycofując je za pomocą instrukcji:

|  |
| --- |
| ROLLBACK [WORK]; |

Więcej na temat tych instrukcji znajdzie się na trzecim wykładzie w ramach omawiania tematu transakcji na bazie danych.

**1.6** **Proste zapytania**

Centralną instrukcją języka SQL jest instrukcja służąca do wydobywania danych z bazy danych. Jest nią instrukcja SELECT, określająca, z jakich tabel w bazie danych mają być sprowadzone dane, jakie warunki mają spełniać i w jakiej dokładnie postaci mają się pojawić przed użytkownikiem (aplikacją użytkownika). Instrukcja SELECT składa się z kilku części nazywanych *klauzulam*i. Omówimy je po kolei.

Najprostsza postać instrukcji SELECT jest następująca:

|  |
| --- |
| SELECT [DISTINCT] *nazwa\_kolumn*y,... FROM *nazwa\_tabeli* [WHERE *warunek*]; |

Obie klauzule SELECT i FROM są wymagane w każdym zapytaniu – natomiast klauzule DISTINCT i WHERE są opcjonalne.

Przykład

|  |
| --- |
| Wypisz dane o nazwiskach, zarobkach i stanowiskach pracowników firmy. |

Piszemy:

SELECT Ename, Sal, Job  
FROM Emp;

Oto wynik tego zapytania na ekranie:

ENAME SAL JOB

---------- ---------- ---------

SMITH 800 CLERK

ALLEN 1600 SALESMAN

WARD 1250 SALESMAN

JONES 2975 MANAGER

MARTIN 1250 SALESMAN

BLAKE 2850 MANAGER

CLARK 2450 MANAGER

SCOTT 3000 ANALYST

KING 5000 PRESIDENT

TURNER 1500 SALESMAN

ADAMS 1100 CLERK

JAMES 950 CLERK

FORD 3000 ANALYST

MILLER 1300 CLERK

Istnieje także skrócona postać listy SELECT ze znakiem \*, co oznacza wypisanie danych z wszystkich kolumn tabeli, a więc instrukcja:

SELECT \*  
FROM Emp;

wypisze całą zawartość tabeli, taką jak podaliśmy na początku wykładu.

W Standardzie jest jeszcze krótsza postać wypisania zawartości tabeli, nie występująca w Oracle, a równoważna powyższej instrukcji SELECT:

|  |
| --- |
| TABLE Emp; |

|  |
| --- |
| Uwaga: Chociaż nie jest to formalnie wymagane, jednak ze względu na czytelność zapisu zapytania, wskazane jest poprzedzanie każdej nazwy kolumny nazwą tabeli, z której dana kolumna pochodzi, lub ewentualnie nazwą jej aliasu nadanego w klazuli FROM. |

Tak więc nasze pierwsze zapytanie moglibyśmy zapisać na dwa sposoby:

SELECT Emp.Ename, Emp.Sal, Emp.Job  
FROM Emp;

albo

SELECT e.Ename, e.Sal, e.Job  
FROM Emp e;

Będziemy to traktować jako standard zapisu zapytania i będziemy się do niego stosować na wykładach. Jako przykłady innych użytecznych standardów mogą służyć: pisanie słów kluczowych dużymi literami a nazw kolumn i tabel małymi ale rozpoczynając od dużej.

Przykład

|  |
| --- |
| Wypisz dane o nazwiskach, zarobkach i stanowiskach pracowników firmy, którzy pracują w dziale o numerze 10. |

W tym przykładzie chodzi o ograniczenie wypisywanych wyników lub, wyrażając się inaczej, o ich selekcję za pomocą pewnego warunku. Warunek ten zapisujemy w klauzuli WHERE:

SELECT Emp.Ename, Emp.Sal, Emp.Job  
FROM Emp  
WHERE Emp.Deptno = 10;

Oto wynik tego zapytania:

ENAME SAL JOB

---------- ---------- ---------

CLARK 2450 MANAGER

KING 5000 PRESIDENT

MILLER 1300 CLERK

Ze względu na możliwość wystąpienia pseudo-wartości *Null*, warunki logiczne oprócz wartości *True* i *False* mogą także dawać *Null* (pseudo-wartość *Null* omówiono na pierwszym [wykładzie z RBD](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w0.htm#NULL)). Przyjmuje się, że jeśli w predykacie porównania występuje pseudo-wartość *Null*, jego wartość też jest *Null*. A więc wartością warunków zarówno Sal = Null jak i Null = Null jest *Null*. Przy SELECT wypisywane są tylko te wiersze, dla których warunek WHERE przyjmuje wartość *True*.

Warunek w klauzuli WHERE może mieć bardziej skomplikowaną postać.

W szczególności może zawierać porównanie wartości z dwóch różnych kolumn np. zarobków pracownika z jego prowizją:

SELECT Emp.Ename, Emp.Sal, Emp.Comm  
FROM Emp  
WHERE Emp.Sal <= Emp.Comm;

Oto wynik tego zapytania:

ENAME SAL COMM

---------- ---------- ----------

MARTIN 1250 1400

Warunek WHERE może też być koniunkcją (AND), alternatywą (OR) bądź negacją (NOT) innych warunków logicznych.

Oto przykład koniunkcji dwóch warunków - w wypisywanych wierszach zarówno zarobki mają być większe niż 1100, jak i stanowisko ma być równe 'CLERK':

SELECT Emp.Empno, Emp.Ename, Emp.Job, Emp.Sal  
FROM Emp  
WHERE Emp.Sal >= 1100 AND Emp.Job='CLERK';

EMPNO ENAME JOB SAL

---------- ---------- --------- ----------

7876 ADAMS CLERK 1100

7934 MILLER CLERK 1300

Należy pamiętać, że przy obliczaniu wartości wyrażeń obowiązuje ustalone pierwszeństwo stosowania operatorów. W przypadku operatorów logicznych obowiązuje następująca kolejność:

|  |
| --- |
| 1.  NOT 2.  AND 3.  OR |

Kolejność tę można zmienić za pomocą nawiasów.

Powtarzające się wiersze nie są automatycznie eliminowane z wyników zapytania. Słowo DISTINCT oznacza eliminację powtarzających się wierszy.

Przykład

|  |
| --- |
| Wypisz identyfikatory osób, które mają podwładnych, tzn. które są kierownikami pewnych pracowników. |

SELECT DISTINCT Emp.Mgr  
FROM Emp;

MGR

----------

7566

7698

7782

7788

7839

7902

Gdybyśmy nie zamieścili słowa DISTINCT, dla każdego pracownika zostałby wypisany identyfikator jego kierownika, a więc wiele identyfikatorów powtarzałoby się.

W miejscu nazwy kolumny może wystąpić wyrażenie. Wyrażeniom na liście SELECT mogą zostać nadane nazwy czyli *aliasy*. Alias może mieć postać *prostego identyfikatora*, czyli napisu złożonego z liter, cyfr i znaków podkreślenia albo tzw. *ograniczonego identyfikatora*, czyli dowolnego napisu ograniczonego podwójnymi cudzysłowami, np. "Zarobki pracownika:".

W szczególności w ograniczonym identyfikatorze mogą występować spacje, które są niedozwolone w prostym identyfikatorze.

Wyniki zapytania mogą zostać posortowane w porządku rosnącym ASC (domyślnie) lub malejącym – wtedy używamy słowa kluczowego DESC.

|  |
| --- |
| SELECT [DISTINCT] *wyrażenie* [[AS] *alias*],... FROM *nazwa\_tabeli* [WHERE *warunek*] [ORDER BY *wyrażenie* [ASC|DESC],...]; |

np. używając w wyrażeniu standardowych funkcji Oracle, w celu obliczenia okresu zatrudnienia pracownika od daty jego zatrudnienia do dnia dzisiejszego – w pełnych latach:

SELECT Emp.Empno, Emp.Ename,  
Trunc(Months\_Between(Sysdate,Emp.Hiredate)/12) Zatrudnienie  
FROM Emp  
ORDER BY Zatrudnienie DESC;

EMPNO ENAME ZATRUDNIENIE

---------- ---------- ------------

7369 SMITH 19

7499 ALLEN 19

7521 WARD 19

7698 BLAKE 19

7782 CLARK 19

7566 JONES 19

7654 MARTIN 18

7934 MILLER 18

7902 FORD 18

7839 KING 18

7844 TURNER 18

7900 JAMES 18

7788 SCOTT 17

7876 ADAMS 17

Przy wypisywaniu wyniku, jako etykiety kolumn są używane aliasy kolumn. Mogą one też wystąpić w klauzuli ORDER BY, jak w powyższym przykładzie, gdzie sortujemy dane o pracownikach według stażu pracy. W klauzuli WHERE, jak również w klauzulach GROUP BY i HAVING, które zostaną opisane później, nie można ich używać.

Aby zrozumieć, jak działa instrukcja SELECT, warto pamiętać o jej semantyce określonej przez zasady jej wykonywania. Zasady te są przydatne do zrozumienia działania instrukcji SELECT, natomiast system bazy danych wykonuje na ogół zapytanie w inny, szybszy sposób, korzystając przede wszystkim z założonych indeksów. Temat ten zostanie przedstawiony na dalszych wykładach.

|  |
| --- |
| **Zasady wykonywania prostego zapytania (semantyka)**  1. Weź tabelę podaną w klauzuli FROM.  2. Jeśli występuje klauzula WHERE, do każdego wiersza danej tabeli zastosuj warunek WHERE. Pozostaw wiersze dające wartość *True* usuwając wiersze dające *False* lub *Null*.  3. Dla każdego pozostającego wiersza oblicz wartości wyrażeń na liście SELECT.  4. Jeśli po SELECT występuje DISTINCT, usuń duplikaty wśród wynikowych wierszy.  5. Jeśli występuje klauzula ORDER BY, wykonaj sortowanie wynikowych wierszy zgodnie ze specyfikacją. |

Zauważmy, że każda instrukcja SELECT określa funkcję, która wejściowej tabeli (jednej lub więcej jak okaże się na następnym wykładzie), znajdującej się na liście FROM przyporządkowuje nową tabelę (bez nazwy) o kolumnach zdefiniowanych przez elementy listy SELECT - zobacz Rys. 1.2. Oznacza to jednorodny charakter danych i wyników. Wyniki jednej instrukcji SELECT mogą posłużyć jako dane do innej instrukcji SELECT. Przy opisie semantyki języka SQL nie musimy sięgać do pojęć znajdujących się poza relacyjnym modelem danych – w szczególności nie musimy odwoływać się ani do pojęć języka programowania ani do pojęć systemu komputerowego. Właśnie ta własność łącznie z prostotą i naturalnością struktury danych jaką jest tabela, stała się podstawą niezmiernej popularności relacyjnego modelu danych.

Rys. 1.2 Zapytanie przekształca wejściowe tabele *A* i *B* na tabelę wyjściową *C*

**1.7** **Wyrażenia**

W wyrażeniach mogą się pojawiać operatory wykonujące operacje na danych. Są to:

1. **Operatory arytmetyczne**: +, -, \*, /

2. **Operator łączenia** napisów ||  
Np. zestawienie nazwiska pracownika z zajmowanym przez niego stanowiskiem

'Osoba '|| Ename || ' pracuje na stanowisku ' || Job

3. **Operatory porównań** =, <>, <, <=, >, >=  
W Oracle argumenty operatorów porównań muszą być wyrażeniami, w Standardzie mogą być listami wyrażeń tej samej długości (wówczas porównania odbywają się po składowych).

4. **Operator przynależności do listy wartości** x [NOT] IN (x1,....)  
Np. Kolor IN ('Czarny', 'Biały', 'Czerwony')

5. **Operator należenia do przedziału**  x [NOT] BETWEEN z AND y  
Np. Sal BETWEEN 1000 AND 2000 W Oracle *x*,*y*,*z* muszą być wyrażeniami, w Standardzie mogą być listami wyrażeń tej samej długości (porównania odbywają się po składowych).

6. **Operator wzorca w tekście** x [NOT] LIKE y  
W argumencie *y* mogą występować symbole uniwersalne sygnalizujące możliwość wystąpienia jednego znaku lub ciągu znaków; są dwa rodzaje znaków uniwersalnych: znak podkreślenia *\_* oznaczający dowolny jeden znak oraz znak *%* oznaczający dowolny ciąg znaków. Gdy *Ename* = 'Kowalski', to oba predykaty Ename LIKE 'Kowal%' i Ename LIKE 'Kowalsk\_' są spełnione.

7. **Operator testujący *Null*** x IS [NOT] NULL

8. **Operatory logiczne** NOT, AND, OR

Każdy konkretny system dostarcza dużej liczby pomocniczych funkcji na danych. Kilka z nich, występujących w Standardzie i w Oracle, przedstawimy poniżej.

**CASE, DECODE i NVL**

W Standardzie występuje następujący schemat funkcji warunkowych:

|  |
| --- |
| CASE *wyrażenie1*   WHEN *wyrażenie2* THEN *wyrażenie3*   [WHEN *wyrażenie4* THEN *wyrażenie5* …]    [ELSE *wyrażenie*] END |

który ma następującą interpretację:  
jeśli *wyrażenie1* = *wyrażenie2*, to wartością wyrażenia jest *wyrażenie3*, w przeciwnym razie, jeśli *wyrażenie1* = *wyrażenie4*, to wartością wyrażenia jest *wyrażenie5* itd…

W Oracle występuje taka sama funkcja, tylko ma nieco odmienną składnię:

|  |
| --- |
| DECODE(*wyrażenie1*,   *wyrażenie2*, *wyrażenie3*,   *wyrażenie4*, *wyrażenie5*...   [,*wyrażenie*] ) |

W Standardzie funkcja CASE zamiast z wyrażeniem może występować z predykatem:

|  |
| --- |
| CASE   WHEN *predykat* THEN *wyrażenie*      ...   [ELSE *wyrażenie*] END |

Przy prezentacji danych użyteczną operacją jest reprezentowanie pseudo-wartości *Null* jako konkretnej wartości. W Standardzie można to wykonać stosując uprzednio zdefiniowane funkcje:

CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2

Natomiast w Oracle używamy specjalnej funkcji:

|  |
| --- |
| NVL(V1,V2) |

której wartością jest *V2*, jeśli *V1* jest *Null*, w przeciwnym razie *V1*

np. funkcje NVL(Sal,0) i NVL(Stopień, 'Brak') umożliwiają zinterpretowanie wartości *Null* jako odpowiednio 0 bądź 'Brak'.

**Konwersja typu**

Kolejną funkcją (grupą funkcji) jest funkcja służąca do zmiany typu wyrażenia. W Standardzie występuje konstrukcja:

|  |
| --- |
| CAST(*wyrażenie* AS *typ\_danych*) |

oznaczająca zmianę typu danych wyrażenie na *typ\_danych*. A więc na przykład, aby dokonać konwersji napisu na liczbę, piszemy

CAST('123' AS INTEGER)

lub liczbę na napis

CAST(123 AS VARCHAR(4))

W Oracle natomiast jest kilka osobnych funkcji określających w sposób jawny, na jaki typ danych ma być dokonana konwersja TO\_CHAR, TO\_DATE, TO\_NUMBER,... Na przykład,

TO\_DATE('11-12-97', 'mm-dd-yy')

zmienia typ napisowy na datowy, przy czym informacja o dacie ma być odczytywana zgodnie z formatem podanym jako drugi argument, czyli z numerem miesiąca zamieszczonym na początku, potem następującym dniem i rokiem reprezentowanym przez dwie cyfry.

**Konkretne wartości bieżące**

Jest też szereg funkcji zwracających konkretne wartości, np. bieżącą datę:

w Standardzie

Current\_date

w Oracle

Sysdate

lub aktualnego użytkownika (programu, w którym jest stosowana ta funkcja, może używać wielu użytkowników):

User

zarówno w Standardzie, jak i w Oracle.

**1.8** **Podsumowanie**

Na tym wykładzie przedstawiliśmy podstawowe informacje na temat typów danych, instrukcji tworzących, modyfikujących i usuwających tabele bazy danych oraz instrukcji, które wstawiają, usuwają i modyfikują dane w tabelach bazy danych.

Rozpoczęliśmy dyskusję najważniejszego tematu w dziedzinie baz danych, a mianowicie jak wydobywać dane z bazy danych.

**1.9** **Słownik pojęć**

[CREATE TABLE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#CREATE TABLE) – instrukcja SQL powodująca utworzenie tabeli o podanej nazwie i schemacie.

[DROP TABLE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#DROP TABLE) - instrukcja SQL powodująca usunięcie tabeli o podanej nazwie.

[ALTER TABLE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#ALTER TABLE) - instrukcja SQL powodująca zmianę schematu tabeli o podanej nazwie.

[CREATE INDEX](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#CREATE INDEX) - instrukcja SQL powodująca utworzenie indeksu dla podanej tabeli.

[DROP INDEX](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#DROP INDEX) - instrukcja SQL powodująca usunięcie indeksu o podanej nazwie.

[INSERT](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Wprowadzaniez) - instrukcja SQL powodująca wstawieniego jednego lub więcej wierszy do tabeli w bazie danych.

[UPDATE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Modyfikowanied) - instrukcja SQL powodująca modyfikację wartości w jednym lub więcej wierszach tabeli bazy danych.

[DELETE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Usuwanied) - instrukcja SQL powodująca usunięcie jednego lub więcej wierszy z tabeli bazy danych.

[COMMIT](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Zatwierdzanie) - instrukcja SQL powodująca zatwierdzenie zmian dokonanych w bazie danych.

[ROLLBACK](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Zatwierdzanie) - instrukcja SQL powodująca wycofanie zmian dokonanych w bazie danych.

[SELECT](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Proste zapytania) - instrukcja SQL powodująca sprowadzenie danych z bazy danych.

[Operatory SQL](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w1.htm#Wyrazenia) - operatory ułatwiające układanie zapytań. Takie jak BETWEEN, IN, LIKE.

**Wykład 2**

***SQL - instrukcja SELECT***

**Streszczenie**

Będziemy kontynuować studiowanie języka SQL. Mianowicie, dokończymy prezentację zapytań rozpoczętą w wykładzie 1. Rozważymy kolejno następujące konstrukcje:

1. operatory algebraiczne na zapytaniach,
2. operatory złączenia,
3. zapytania sumaryczne i grupujące,
4. podzapytania,
5. operatory złączenia zewnętrznego.

**2.1** **Operatory algebraiczne na zapytaniach**

Do określenia danych, które chcemy wydobyć z bazy danych, można użyć kilku zapytań połączonych ze sobą operatorami algebraicznymi zgodnie ze składnią:

|  |
| --- |
| *instrukcja\_SELECT operator instrukcja\_SELECT* |

Są trzy takie operatory:

1. UNION, UNION ALL - sumowanie zbiorów wyników (odpowiednio, z eliminacją bądź nie - powtórzeń wierszy),

2. INTERSECT - przecięcie zbiorów wyników,

3. EXCEPT - różnica zbiorów wyników (w Oracle MINUS).

Przykład

|  |
| --- |
| Wypisz numery działów, w których w danej chwili nie są zatrudnieni żadni pracownicy. |

Naturalne jest użycie operatora MINUS:

SELECT Dept.Deptno FROM Dept  
MINUS  
SELECT Emp.Deptno FROM Emp;

DEPTNO

----------

40

Przykład

|  |
| --- |
| Wypisz nazwiska pracowników, zamieszczając przy pracownikach działu numer 10 gwiazdkę. |

Naturalne jest użycie operatora UNION:

SELECT Emp.Ename||'\*' FROM Emp  
WHERE Emp.Deptno = 10  
UNION  
SELECT Emp.Ename FROM Emp  
WHERE Emp.Deptno <> 10  
ORDER BY 1;

EMP.ENAME||'\*'

--------------

ADAMS

ALLEN

BLAKE

CLARK\*

FORD

JAMES

JONES

KING\*

MARTIN

MILLER\*

SCOTT

SMITH

TURNER

WARD

Aby móc zastosować operator algebraiczny, liczba i typy kolumn w składowych zapytaniach muszą być takie same.

Według Standardu, tabela wynikowa nie posiada nazw kolumn. Natomiast Oracle, przy wypisywaniu zawartości tabeli wynikowej, jako etykiet kolumn używa wyrażeń z pierwszej instrukcji SELECT.

Klauzula ORDER BY może wystąpić tylko na końcu całego zapytania. W klauzuli ORDER BY do wynikowych kolumn odwołujemy się używając ich kolejnych numerów 1, 2,....

**2.2** **Zapytania dotyczące kilku tabel (złączenia tabel)**

Informacja wypisywana z bazy danych może dotyczyć kilku tabel. Dane z kilku tabel są na ogół złączane korzystając z naturalnych powiązań między wierszami tabel opartych na związkach *klucz obcy* -> *klucz główny*. W takich przypadkach najczęściej klucz obcy i klucz główny mają tę samą nazwę. W celu ich odróżnienia konieczne jest użycie konstrukcji poprzedzania nazwy kolumny nazwą tabeli np. *Emp.Deptn*o - czyli kwalifikowania nazwy kolumny nazwą tabeli.

Przykład

|  |
| --- |
| Wypisz wszystkich pracowników i dla każdego z nich podaj nazwę jego działu. |

SELECT Emp.Empno, Emp.Ename, Dept.Dname  
FROM Emp, Dept  
WHERE Emp.Deptno= Dept.Deptno;

EMPNO ENAME DNAME

---------- ---------- --------------

7369 SMITH RESEARCH

7499 ALLEN SALES

7521 WARD SALES

7566 JONES RESEARCH

7654 MARTIN SALES

7698 BLAKE SALES

7782 CLARK ACCOUNTING

7788 SCOTT RESEARCH

7839 KING ACCOUNTING

7844 TURNER SALES

7876 ADAMS RESEARCH

7900 JAMES SALES

7902 FORD RESEARCH

7934 MILLER ACCOUNTING

Wiersz każdego pracownika zostaje złączony z dokładnie jednym, odpowiadającym mu przez wartość *Deptno*, wierszem z tabeli *Dept*.

Predykat Emp.Deptno=Dept.Deptno występujący w powyższej instrukcji SELECT nazywa się *predykatem złączenia*, w odróżnieniu od innych predykatów nazywanych *predykatami ograniczającymi* np.

   Sal>1000

lub

   Loc='Warszawa'

Zwróćmy uwagę na istotność podania warunku złączenia. Gdybyśmy go opuścili, otrzymalibyśmy zbiór wszystkich możliwych kombinacji wierszy ze złączanych tabel – nie tylko tych, które są ze sobą powiązane wspólną cechą, taką jak numer działu w powyższym zapytaniu. Wynik takiego pełnego połączenia tabel nosi nazwę *iloczynu kartezjańskiego* tych tabel.

**Operatory złączenia**

W Standardzie SQL jak i w Oracle od wersji 9i warunek złączenia można także zapisać w klauzuli FROM jako część jednego z operatorów  złączenia i to w różny sposób. Dla powyższego zapytania są możliwe następujące zapisy:

SELECT Emp.Empno, Emp.Ename, Dept.Dname  
FROM Emp JOIN Dept ON Emp.Deptno = Dept.Deptno;

(tej postaci z operatorem złączenia JOIN będziemy używać zazwyczaj od tej pory)

albo gdy nazwy kolumn złączenia są takie same

SELECT Emp.Empno, Emp.Ename, Dname  
FROM Emp JOIN Dept USING (Deptno);

albo jeszcze krócej

SELECT Empno, Ename, Dname  
FROM Emp NATURAL JOIN Dept;

W ostatnim przypadku za kolumy złączenia przyjmuje się wszystkie wspólne, tzn. mające takie same nazwy, kolumny.

**Samozłączenia**

Korzystając ze związku klucz obcy -> klucz główny można dokonać złączenia tabeli z nią samą. Wówczas ta sama tabela występuje w dwóch (lub więcej) rolach wskazywanych przez aliasy dołączane do nazwy tabeli w klauzuli FROM.

Przykład

|  |
| --- |
| Wypisz nazwiska wszystkich pracowników i nazwiska ich kierowników. |

SELECT Prac.Ename, Kier.Ename AS Mgr  
FROM Emp Prac JOIN Emp Kier ON Prac.Mgr = Kier.Empno;

ENAME MGR

---------- ----------

SMITH FORD

ALLEN BLAKE

WARD BLAKE

JONES KING

MARTIN BLAKE

BLAKE KING

CLARK KING

SCOTT JONES

TURNER BLAKE

ADAMS SCOTT

JAMES BLAKE

FORD JONES

MILLER CLARK

Alias *Prac* reprezentuje tu wiersz pracownika, a alias *Kier* wiersz jego kierownika – zapewnienie zachodzenia związku między nimi uzyskuje się dzięki warunkowi, który ma postać równości identyfikatorów:

Prac.Mgr = Kier.Empno

Dzięki temu, wiersz z nazwiskiem pracownika zostaje złączony z dokładnie jednym wierszem pochodzącym z tej samej tabeli – mianowicie wierszem z nazwiskiem jego kierownika, określonym przez identyfikator *Prac.Mgr*.

Najbardziej naturalnym rodzajem złączenia jest złączenie przez związek *klucz obcy -> klucz główny*. Jednak w ogólności warunek złączenia dwóch (lub więcej) tabel może być zupełnie dowolny. W przypadku naszej przykładowej bazy danych jest jeszcze jedno złączenie mające naturalny charakter, które przedstawimy w następującym zadaniu.

Przykład

|  |
| --- |
| Wypisz wszystkich pracowników podając grupę zaszeregowania ich zarobków. |

SELECT e.Empno, e.Ename, s.Grade  
FROM Emp e, Salgrade s   
WHERE e.Sal BETWEEN s.Losal AND s.Hisal;

EMPNO ENAME GRADE

---------- ---------- ----------

7369 SMITH 1

7876 ADAMS 1

7900 JAMES 1

7521 WARD 2

7654 MARTIN 2

7934 MILLER 2

7499 ALLEN 3

7844 TURNER 3

7566 JONES 4

7698 BLAKE 4

7782 CLARK 4

7788 SCOTT 4

7902 FORD 4

7839 KING 5

Zauważmy, że *Sal* to kolumna z tabeli *Emp*, natomiast *Losal* i *Hisal* to kolumny z tabeli *Salgrade*. Zakładamy tutaj, że przedziały określone przez pary wartości (*Losal*, *Hisal*) pokrywają cały dopuszczalny zakres zarobków pracowników. A więc, dla każdej wartości zarobków jest określona dokładnie jedna taka para zawierająca tę wartość *Sal*. Wiersz każdego pracownika zostaje złączony z dokładnie jednym wierszem z tabeli *Salgrade*, mianowicie z tym, do którego przedziału [*Losal*, *Hisal*] wpadają zarobki danego pracownika. (Jest pewna niezgodność między wymaganiem pokrycia przestrzeni możliwych wartości przez przedziały a zastosowaniem predykatu BETWEEN, który sugeruje przedziały domknięte, ale wtedy punkty graniczne między przedziałami wpadałyby do dwóch przedziałów i byłyby dwukrotnie zwracane przez warunek BETWEEN. W naszym przykładzie rozwiązaliśmy ten problem kładąc prawy koniec przedziału o jeden mniejszym niż lewy koniec następnego przedziału i zakładając, że zarobki pracowników są liczbami całkowitymi. W ten sposób uzyskujemy rozłączne pokrycie zbioru wszystkich możliwych zarobków.

Oto semantyka zapytania ze złączeniami tabel.

|  |
| --- |
| **Zasady wykonywania zapytania z listą tabel w klauzuli FROM (semantyka)**  1. Jeśli występuje operator algebraiczny UNION, UNION ALL, INTERSECT lub  EXCEPT, to powtórz kroki 2-5 dla każdego składnika.  2. Jeśli występują operatory algebraiczne takie jak JOIN, NATURAL JOIN wykonaj je i wyniki zapisz w tymczasowych tabelach.  3. Rozważ kolejno wszystkie kombinacje wierszy tabel występujących w klauzuli FROM (w tym obliczonych tabel tymczasowych). Kombinacje wierszy traktujemy od tej chwili tak jak zwykłe wiersze tabel.  4. Do każdego rozpatrywanego wiersza zastosuj warunek WHERE. Pozostaw tylko wiersze dające wartość *True* (usuwając wiersze dające *False* lub *Null*).  5. Dla każdego pozostającego wiersza oblicz wartości wyrażeń na liście SELECT.  6. Jeśli po SELECT występuje DISTINCT, usuń duplikaty wśród wynikowych wierszy.  7. Wykonaj operacje algebraiczne na wynikach zapytań.  8. Jeśli występuje klauzula ORDER BY, wykonaj sortowanie wynikowych wierszy zgodnie ze specyfikacją. |

**2.3** **Zapytania sumaryczne (podsumowujące)**

Dane w jednej lub więcej tabel mogą zostać podsumowane przy użyciu jednej z funkcji sumarycznych. W efekcie uzyskujemy jeden wiersz wynikowy. Są następujące jedno-argumentowe funkcje sumaryczne (agregujące):

|  |
| --- |
| * COUNT * AVG * SUM * MAX * MIN |

W Oracle dodatkowo są jeszcze STDDEV – standardowe odchylenie i VARIANCE - wariancja. Argumentem tych funkcji może być *wyrażenie* (odpowiedniego typu) lub DISTINCT *wyrażenie*

W przypadku COUNT argumentem może być też symbol \*. COUNT(\*) oznacza liczbę wszystkich wierszy spełniających warunek WHERE.

Jeśli wartością wyrażenia jest *Null*, to ta konkretna wartość nie jest brana pod uwagę przy obliczaniu wartości funkcji. A więc np. przy obliczaniu SUM(Sal), nie bierzemy pod uwagę wartości *Null*, chyba że kolumna *Sal* zawiera same wartości *Null* – w takim przypadku SUM(Sal)jest *Null*.

Przykład

|  |
| --- |
| Wyznacz liczbę pracowników firmy. |

SELECT COUNT(\*) AS "Liczba pracowników firmy"  
FROM Emp;

Liczba pracowników firmy

------------------------

14

Przykład

|  |
| --- |
| Wyznacz statystyki zarobków pracowników pracujących w dziale *Sprzedaży*. |

SELECT MIN(e.Sal) AS "Min zarobki",  
       MAX(e.Sal) AS "Max zarobki",  
       MAX(e.Sal)- MIN(e.Sal) AS "Rozp zarobków",  
       AVG(e.Sal) AS "Śred zarobki"  
FROM Emp e JOIN Dept d ON e.Deptno = d.Deptno  
WHERE d.Dname = 'SALES';

Min zarobki Max zarobki Rozp zarobków Śred zarobki

----------- ----------- ------------- ------------

950 2850 1900 1566.66667

Gdy w obliczeniach wartości podsumowujących chcemy uwzględnić pseudo-wartości *NULL*, używamy funkcji NVL. Następujące zapytanie liczy średnią wartość prowizji interpretując *NULL* jako 0.

SELECT AVG(NVL(e.comm,0)) "Średnia prowizja"  
FROM Emp e;

Średnia prowizja

----------------

157.142857

**2.4** **Zapytania grupujące**

W języku SQL jest możliwość podziału wynikowych wierszy zapytania na grupy i wykonania funkcji sumarycznych na wartościach należących do poszczególnych grup.

|  |
| --- |
| SELECT … FROM … WHERE … GROUP BY wyrażenie,... [HAVING warunek] |

Przykład

|  |
| --- |
| Podziel pracowników na grupy zaliczając do jednej grupy pracowników pracujących w jednym dziale, następnie dla każdej grupy oblicz ich liczbę i sumaryczne zarobki. |

Wersja 1 (bez załączania nazwy działu)

SELECT e.Deptno Id, COUNT(\*) Liczba, SUM(e.Sal) Suma  
FROM Emp e  
GROUP BY e.Deptno;

ID LICZBA SUMA

---------- ---------- ----------

10 3 8750

20 5 10875

30 6 9400

Wersja 2 (z załączeniem nazwy działu)

SELECT d.Deptno Id, d.Dname Nazwa\_działu, COUNT(\*) Liczba, SUM(e.Sal) Suma  
FROM Dept d JOIN Emp e ON d.Deptno = e.Deptno  
GROUP BY d.Deptno, d.Dname;

ID NAZWA\_DZIAŁU LICZBA SUMA

---------- -------------- ---------- ----------

10 ACCOUNTING 3 8750

20 RESEARCH 5 10875

30 SALES 6 9400

W drugim przypadku grupowanie odbywa się po złączeniu obu tabel *Dept* i *Emp*. Kolumny, względem których następuje podział na grupy, nazywają się *kolumnami grupującymi*. W powyższych przykładach za pierwszym razem jest to tylko *Dept.Deptno*, za drugim razem *Dept.Deptno* i *Dept.Dname*.

Kolumna *Dept.Dname*, chociaż o wartościach zdeterminowanych przez wartości w kolumnie *Dept.Deptno*, musi zostać wpisana na listę GROUP BY, ponieważ kolumna *Dept.Dname* znajduje się na liście SELECT – zwróć uwagę na podpunkt poniżej "Ograniczenia".

W kolejnym przykładzie na liście GROUP BY znajdują się dwie kolumny *Deptno* i *Job*. W pierwszej kolejności wiersze są grupowane według wartości *Deptno*, a w ramach grupy o tych samych wartościach *Deptno* dokonujemy jeszcze podziału na grupy względem wartości *Job* – uzyskując ostateczny podział na grupy. Dla każdej ostatecznej grupy dokonujemy sumowania zarobków pracowników zaliczonych do tej grupy o tych samych wartościach *Deptn*o i *Job*

SELECT e.Deptno, e.Job, SUM(e.Sal)  
FROM Emp e  
GROUP BY e.Deptno, e.Job;

DEPTNO JOB SUM(SAL)

---------- --------- ----------

10 CLERK 1300

10 MANAGER 2450

10 PRESIDENT 5000

20 ANALYST 6000

20 CLERK 1900

20 MANAGER 2975

30 CLERK 950

30 MANAGER 2850

30 SALESMAN 5600

Zwróćmy uwagę na to, że podczas grupowania nastąpiło sortowanie względem wartości kolumn grupujących. Nie należy jednak tego zakładać, ponieważ grupowanie może być także zrealizowane poprzez haszowanie. W następnej wersji serwera, wynik grupowania wcale nie musi być więc posortowany.

Można ograniczyć wypisywane grupy formułując warunek dla grup. Np. aby ograniczyć się do działów zatrudniających co najmniej 5 pracowników, dołączylibyśmy warunek HAVING:

HAVING COUNT(\*)>=5

otrzymując instrukcję:

SELECT e.Deptno Id, COUNT(\*) Liczba, SUM(e.Sal) Suma  
FROM Emp e  
GROUP BY e.Deptno  
HAVING COUNT(\*)>=5;

ID LICZBA SUMA

---------- ---------- ----------

20 5 10875

30 6 9400

W wyniku zabrakło jednego wiersza o wartości LICZBA = 3, który występował w wersji bez klauzuli HAVING:

ID LICZBA SUMA

---------- ---------- ----------

10 3 8750

Należy pamiętać, że najpierw jest stosowany warunek WHERE ograniczający rozpatrywane wiersze z połączonych tabel. Otrzymane wiersze są następnie grupowane, po czym jest stosowany warunek HAVING ograniczający wypisywane grupy.

Należy pamiętać o następujących zasadach dotyczących grupowania.

|  |
| --- |
| Ograniczenia dla klauzuli GROUP BY   * Na liście GROUP BY mogą być tylko nazwy kolumn (w Oracle dowolne wyrażenia). * Elementami listy SELECT, klauzuli HAVING i ORDER BY mogą być tylko:   1. stała,   2. funkcja sumaryczna,   3. kolumna grupująca (występująca w klauzuli GROUP BY),   4. wyrażenie zawierające (1)–(3), przy czym każde wystąpienie kolumny nie-grupującej musi się znajdować w zasięgu funkcji sumarycznej. * SQL robi wyjątek dla pseudowartości *Null*: dwa wiersze mające te same wartości w kolumnach grupujących wliczając w to *Null*, idą do tej samej grupy (w przeciwnym przypadku do różnych). |

Oto semantyka zapytania z klauzulami GROUP BY i HAVING.

|  |
| --- |
| **Zasady wykonywania zapytania grupującego (semantyka)**  1. Jeśli występuje operator algebraiczny UNION, UNION ALL, INTERSECT lub  EXCEPT, to powtórz kroki 2-7 dla każdego składnika.  2. Jeśli występują operatory algebraiczne takie jak JOIN, NATURAL JOIN wykonaj je i wyniki zapisz w tymczasowych tabelach.  3. Rozważ kolejno wszystkie kombinacje wierszy tabel występujących w klauzuli FROM (w tym obliczonych tabel tymczasowych). Kombinacje wierszy traktujemy od tej chwili tak jak zwykłe wiersze tabel.  4. Do każdego rozpatrywanego wiersza zastosuj warunek WHERE. Pozostaw tylko wiersze dające wartość *True* (usuwając wiersze dające *False* lub *Null*).  5. Podziel pozostające wiersze na grupy.  6. Do każdej grupy zastosuj warunek w klauzuli HAVING. Pozostaw tylko grupy, dla których wartość warunku jest *True*.  7. Dla każdego pozostającego wiersza reprezentującego grupę oblicz wartości wyrażeń na liście SELECT.  8. Jeśli po SELECT występuje DISTINCT, usuń duplikaty wśród wynikowych wierszy.  9. Jeśli trzeba, zastosuj odpowiedni operator algebraiczny.  10. Jeśli występuje klauzula ORDER BY, wykonaj sortowanie wynikowych wierszy zgodnie ze specyfikacją. |

**2.5** **Podzapytania**

Wewnątrz klauzul WHERE i HAVING, a także SELECT i FROM, mogą wystąpić podzapytania, mające taką samą postać jak zapytania (tylko są ujęte w nawiasy). Podzapytanie może wystąpić jako argument predykatów =, <, <=, >, >=, <>, IN, NOT IN, przy czym w przypadku predykatów =, <, <=, >, >=, <>, powinno określać jedną wartość, a w przypadku wystąpienia po prawej stronie  predykatów IN oraz NOT IN listę wartości. (W poprzednich wersjach zarówno Standardu jak i systemu Oracle istniało ograniczenie podzapytania tylko do prawego argumentu.)

W  Standardzie SQL w podzapytaniu nie można używać klauzul ORDER BY ani UNION. W przypadku podzapytań w Oracle mogą występować operatory UNION, MINUS i INTERSECT. W podzapytaniu dostępne są nazwy kolumn wprowadzone w głównym zapytaniu.

Wyróżniamy dwa rodzaje podzapytań: *zwykłe* i *skorelowane*.

Podzapytanie nazywamy *zwykłym* jeśli nie zawiera odwołań do kolumn tabel określonych w głównym zapytaniu. Podzapytanie nazywamy *skorelowanym* jeśli zawiera odwołania do kolumn tabel określonych w głównym zapytaniu.

W podzapytaniu zwykłym zbiór wynikowych wierszy nie zmienia się i nie zależy od wierszy w głównym zapytaniu.

Przykład

|  |
| --- |
| Wypisz osoby, które zarabiają najwięcej ze wszystkich pracowników. |

Rozwiązując to zadanie najpierw liczymy największe *Sal* za pomocą zapytania:

SELECT Max(e.Sal) FROM Emp e;

Zapytanie to można z kolei użyć jako podzapytanie (bez średnika) w warunku WHERE, wtedy kiedy trzeba przyrównać zarobki pracownika do maksymalnych zarobków. W efekcie uzyskujemy możliwość wyszukania pracowników, których zarobki są równe tym maksymalnym.

SELECT e.Ename, e.Sal  
FROM Emp e  
WHERE e.Sal = (SELECT Max(f.Sal) FROM Emp f);

ENAME           SAL  
---------- ----------  
KING           5000

W klauzuli WHERE może być więcej niż jedno podzapytanie.

Przykład

|  |
| --- |
| Wypisz pracowników, którzy pracują na tym samym stanowisku co pracownik o numerze 7369 i których zarobki są większe niż zarobki pracownika o numerze 7876. |

Oto zapytanie:

SELECT e.Ename, e.Job  
FROM Emp e  
WHERE e.Job = (SELECT f.Job FROM Emp f WHERE f.Empno = 7369)   
  AND e.Sal > (SELECT g.Sal FROM Emp g WHERE g.Empno = 7876);

ENAME JOB

---------- ---------

MILLER CLERK

Gdy podzapytanie zwraca więcej niż jedną wartość, zamiast operatora = stosuje się operator IN.

Przykład

|  |
| --- |
| Wypisz działy, w których pracują urzędnicy. |

Oto zapytanie:

SELECT d.Dname  
FROM Dept d  
WHERE d.Deptno IN (SELECT e.Deptno FROM Emp e WHERE e.Job = 'CLERK');

DNAME

--------------

ACCOUNTING

RESEARCH

SALES

Pewne problemy mogą się pojawić przy stosowaniu operatora NOT IN w sytuacji, gdy w wyniku podzapytania występuje pseudo-wartość NULL, gdyż o żadnej wartości nie da się stwierdzić przez porównanie, że jest różna od NULL. Z tego powodu następujące zapytanie, które pozornie liczy wszystkich pracowników, którzy nie są kierownikami,

SELECT Prac.Ename   
FROM Emp Prac   
WHERE Prac.Empno NOT IN (SELECT Podw.Mgr FROM Emp Podw);

daje w rzeczywistości wynik będący pustą tabelą! Oczywiście, aby uniknąć w podzapytaniu rozważania wartości *Null*, wystarczy podać odpowiedni warunek w klauzuli WHERE.

SELECT Prac.Ename   
FROM Emp Prac   
WHERE Prac.Empno NOT IN (SELECT Podw.Mgr FROM Emp Podw  
                         WHERE Podw.Mgr IS NOT NULL);

W przeciwieństwie do podzapytania zwykłego w podzapytaniu skorelowanym zbiór wyników podzapytania zależy od wartości występujących w wierszach w głównym zapytaniu.

Przykład

|  |
| --- |
| Dla każdego działu wypisz osobę, która zarabia najwięcej w tym dziale. |

W głównym zapytaniu używamy podzapytania liczącego maksymalne zarobki w danym dziale:

SELECT Max(e.Sal)  
FROM Emp e  
WHERE e.Deptno= <*Deptno określone w głównym zapytaniu*>

A oto całe zapytanie:

SELECT e.Deptno, e.Ename, e.Sal  
FROM Emp e  
WHERE e.Sal = (SELECT Max(f.Sal)FROM Emp f WHERE f.Deptno= e.Deptno);

DEPTNO ENAME SAL

---------- ---------- ----------

30 BLAKE 2850

20 SCOTT 3000

10 KING 5000

20 FORD 3000

**Operatory SOME, ANY i ALL**

Predykaty porównań można łączyć ze słowami kluczowymi oznaczającymi kwantyfikatory: SOME (równoważne ANY) - odczytywane "dla pewnego" i ALL - odczytywane "dla wszystkich" lub "dla każdego", otrzymując predykaty, których prawym argumentem może być lista wyrażeń lub podzapytanie - objęte nawiasami. Składnia jest następująca:

|  |
| --- |
| *wyrażenie* *operator\_porównania* [ANY|SOME|ALL][*lista\_wyrażeń*|(*podzapytanie*)] |

Przykład

10000 >= ALL (SELECT Sal FROM Emp)

co można odczytać: "10000 większe lub równe od zarobków każdego z pracowników"

1000 >= SOME (SELECT Sal FROM Emp)

co można odczytać: "1000 większe lub równe od zarobków jakiegoś pracownika"

**Operatory EXISTS i NOT EXISTS**

Są też dwa predykaty EXISTS i NOT EXISTS sprawdzające czy podzapytanie daje pusty zbiór wyników czy nie, np.

EXISTS(SELECT 'x' FROM Emp WHERE Deptno= 10)

co można odczytać "istnieje co najmniej jeden pracownik zatrudniony w dziale o numerze 10". Zauważmy, że dla wyniku nie jest istotne, co napiszemy na liście SELECT w ramach predykatu EXISTS – najprostsza obliczeniowo jest wartość stała taka jak 'x'.

Przykład

|  |
| --- |
| Wypisz działy, w których aktualnie nie ma zatrudnionych pracowników. |

Używamy predykatu NOT EXISTS i podzapytania skorelowanego:

SELECT DISTINCT d.Dname   
FROM Dept d  
WHERE NOT EXISTS (SELECT 'x' FROM Emp e WHERE e.Deptno = d.Deptno);

DNAME

--------------

OPERATIONS

Za pomocą predykatu NOT EXISTS można rozwiązać problem wyznaczenia wszystkich pracowników, którzy nie są kierownikami - nie ma tu problemu (jak w przypadku  predykatu [NOT IN](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#Pewne problemy)) z wystąpieniem pseudo-wartości NULL w wyniku podzapytania.

SELECT e.Empno, e.Ename, e.Job, e.Deptno  
FROM Emp e  
WHERE NOT EXISTS (SELECT 'x'  
                  FROM Emp f  
                  WHERE f.Mgr = e.Empno);

EMPNO ENAME JOB DEPTNO

---------- ---------- --------- ----------

7369 SMITH CLERK 20

7499 ALLEN SALESMAN 30

7521 WARD SALESMAN 30

7654 MARTIN SALESMAN 30

7844 TURNER SALESMAN 30

7876 ADAMS CLERK 20

7900 JAMES CLERK 30

7934 MILLER CLERK 10

Predykaty EXISTS i NOT EXISTS odpowiadają bezpośrednio odpowiednio kwantyfikatorowi egzystencjalnemu i kwantyfikatorowi uniwersalnemu z negacją objętej nim formuły, gdzie zmienna związana kwantyfikatora jest zmienną wiersza zapytania. Daje to silne narzędzie do rozwiązywania skomplikowanych zadań za pomocą zapytań SQL.

W Standardzie występuje jeszcze predykat:

|  |
| --- |
| UNIQUE (*podzapytanie*) |

sprawdzający czy w wyniku podzapytania nie występują powtarzające się wiersze.

Przykład

Predykat

UNIQUE (SELECT Ename FROM Emp)

daje wartość *True*, gdy nazwiska pracowników nie powtarzają się, w przeciwnym razie *False*.

**Podzapytania w instrukcjach UPDATE i DELETE**

Podzapytania są na ogół częścią składową warunków w klauzulach WHERE i HAVING. Przypominamy, że klauzula WHERE występuje również w instrukcjach UPDATE i DELETE. Rozważymy przykład podzapytania skorelowanego występującego w instrukcji UPDATE.

Przykład

|  |
| --- |
| Zwiększmy zarobki wszystkim pracownikom pracującym w Dallas. |

UPDATE Emp e  
SET e.Sal = e.Sal \* 1.1  
WHERE 'DALLAS' = (SELECT d.Loc  
                  FROM Dept d  
                  WHERE e.Deptno = d.Deptno);

Podzapytania mogą również występować w innych miejscach niż warunki. Oto te sytuacje.

**Podzapytania w klauzuli FROM instrukcji SELECT**

Przykład

|  |
| --- |
| Oblicz procentowy udział poszczególnych działów w liczbie pracowników i zarobkach w firmie. |

SELECT a.Deptno "Dział",  
       Trunc(100\*a.Liczba\_Prac/b.Liczba\_Prac,1)AS "%Pracowników",  
       Trunc(100\*a.Suma\_zarob/b.Suma\_zarob,1) AS "%Zarobków"  
FROM (SELECT Deptno, COUNT(\*) AS Liczba\_Prac,  
      SUM(Sal) AS Suma\_zarob  
      FROM Emp  
      GROUP BY Deptno) a,  
     (SELECT COUNT(\*) AS Liczba\_Prac,  
             SUM(Sal) AS Suma\_zarob  
      FROM Emp) b;

Dzial %Pracowników %Zarobków

---------- ------------ ----------

10 21.4 30.1

20 35.7 37.4

30 42.8 32.3

Konstrukcja ta umożliwia niezależne przeprowadzenie kilku obliczeń, a następnie umieszczenie otrzymanych wyników w głównym zapytaniu!

**Podzapytania w klauzuli SELECT instrukcji SELECT**

Przykład

|  |
| --- |
| Dla każdego działu policz ilu pracowników jest w nim zatrudnionych. |

SELECT d.Deptno "Dział",  
      (SELECT COUNT(\*) FROM Emp e WHERE e.Deptno=d.Deptno) AS "#Pracowników"  
FROM Dept d;

Dział #Pracowników

---------- ------------

10 3

20 5

30 6

40 0

Konstrukcja ta umożliwia zastosowanie strukturalnego podejścia przy rozwiązywaniu problemów. Podproblem "wyznacz liczbę pracowników w danym dziale" jest rozwiązywany niezależnie od głównego problemu "wypisz działy".

**Podzapytania w klauzuli AS instrukcji CREATE TABLE (tylko w Oracle)**

Przykład

|  |
| --- |
| Skopiuj informacje o urzędnikach i umieść je w nowej tabeli *Urzędnicy*. |

CREATE TABLE Urzędnicy (Empno, Ename, Sal)  
AS SELECT Empno, Ename, Sal  
   FROM Emp  
   WHERE Job = 'CLERK';

Używając tej konstrukcji, na liście kolumn tabeli nie podajemy nazw typów danych. Możemy natomiast określać więzy spójności i wartości domyślne. System sam wyprowadza informację o typach danych kolumn i ich rozmiarach z wyrażeń występujących na liście SELECT w podzapytaniu.

**Podzapytania w instrukcji INSERT**

Przykład

|  |
| --- |
| Wstaw do pomocniczej tabeli wszystkich pracowników zatrudnionych w okresie ostatnich 10 dni. |

INSERT INTO Emp\_new   
      SELECT \*  
      FROM Emp  
      WHERE Sysdate - Hiredate < 10;

W Standardzie podzapytanie jest zawsze ujmowane w nawiasy (w systemie Oracle nie zawsze).

**2.6** **Złączenia zewnętrzne**

Złączenie zewnętrzne rozszerza rezultat prostego złączenia (nazywanego *wewnętrznym*) o te wiersze z jednej z tabel, dla których w trakcie złączania nie znaleziono odpowiadających im wierszy w drugiej tabeli.

Przykład

|  |
| --- |
| Wypisz wszystkich pracowników podając dla każdego z nich nazwę działu, w którym pracuje. |

Załóżmy dla potrzeb tego przykładu, że kolumna *Emp.Deptno* może zawierać pseudo-wartość *NULL* i że do tabeli *Emp* wstawiliśmy nowego pracownika o nazwisku *Kowalski* bez przypisania go do żadnego działu.

Zapytanie:

SELECT e.Empno, e.Ename, d.Dname   
FROM Emp e LEFT JOIN Dept d ON e.Deptno = d.Deptno;

zwraca informację o pracownikach i ich działach, a w przypadku, gdy pracownik nie ma przyporządkowanego działu, podawana jest wartość *Null* reprezentowana na wydruku w Oracle przez pusty ciąg znaków.

EMPNO ENAME DNAME

---------- ---------- --------------

7369 SMITH RESEARCH

7499 ALLEN SALES

7521 WARD SALES

7566 JONES RESEARCH

7654 MARTIN SALES

7698 BLAKE SALES

7782 CLARK ACCOUNTING

7788 SCOTT RESEARCH

7839 KING ACCOUNTING

7844 TURNER SALES

7876 ADAMS RESEARCH

7900 JAMES SALES

7902 FORD RESEARCH

7934 MILLER ACCOUNTING

7799 KOWALSKI

Zwróćmy uwagę na pojawienie się wiersza z nowym pracownikiem, który jako nie zaliczony do żadnego działu, nie pojawiłby się w wyniku zwykłego złączenia wewnętrznego JOIN a nie LEFT JOIN.

Przykład

|  |
| --- |
| Wypisz wszystkie działy dla każdego z nich podając sumaryczne zarobki zatrudnionych w nim pracowników. |

Użyjemy symetrycznego do LEFT JOIN operatora złączenia zewnętrznego RIGHT JOIN.

SELECT d.Deptno, d.Dname, SUM(e.Sal)  
FROM Emp e RIGHT JOIN Dept d ON e.Deptno = d.Deptno  
GROUP BY d.Deptno, d.Dname;

DEPTNO DNAME SUM(SAL)

------ -------------- --------------

10 ACCOUNTING 8750

20 RESEARCH 1087

30 SALES 9400

40 OPERATIONS

Gdy w dziale nie ma żadnego pracownika, zbiór sumowanych wartości składa się z jednej wartości *Null*. Zgodnie z przyjętymi zasadami, wynikiem sumowania jest wartość *Null*, reprezentowana na wydruku przez puste miejsce. Gdybyśmy chcieli w takim przypadku wypisać wartość 0, powinniśmy wyrażenie SUM(Sal) umieścić jako argument funkcji NVL:

NVL(Sum(Sal),0)

Zauważmy, że podobny, chociaż nie dokładnie ten sam efekt, można w Oracle uzyskać za pomocą operatora UNION:

SELECT d.Deptno, d.Dname, TO\_CHAR(SUM(e.Sal))  
FROM Emp e JOIN Dept d ON e.Deptno = d.Deptno  
GROUP BY d.Deptno, d.Dname  
UNION  
SELECT d.Deptno, d.Dname, NULL  
FROM Dept d  
WHERE NOT EXISTS (SELECT 'x' FROM Emp f WHERE f.Deptno = d.Deptno);

DEPTNO DNAME TO\_CHAR(SUM(SAL))

------ ------------- --------------------

10 ACCOUNTING 8750

20 RESEARCH 10875

30 SALES 9400

40 OPERATIONS

**Inne operatory złączeń**

Oprócz wymienionych już operatrów złączeń istnieją jeszcze następujące (*T*, *U* oznaczają tabele):

1. *Złączenie krzyżowe* T CROSS JOIN U - iloczyn kartezjański - wszystkie kombinacje wierszy:

SELECT \* FROM T CROSS JOIN U;

jest tym samym co

SELECT \* FROM T,U;

2. *Pełne złączenie zewnętrzne* T FULL JOIN U - suma wyników złączenia zewnętrznego lewostronnego i prawostronnego). Na przykład, zapytanie

SELECT e.Ename, d.Dname  
FROM Emp e FULL JOIN Dept d ON e.Deptno=d.Deptno;

zestawia ze sobą nazwiska pracowników z nazwami departamentów, w których są zatrudnieni podając również pracowników, którzy nie zostali przypisani do żadnego departamentu a także nazwy departamentów, w których nie ma zatrudnionych pracowników.

Omawianie konstrukcji języka SQL będziemy kontynuować na następnym wykładzie.

**2.7 Podsumowanie**

W wykładzie 2 dokończyliśmy prezentację zapytań - czyli różnych postaci instrukcji SELECT. Rozważyliśmy kolejno następujące konstrukcje: operatory algebraiczne na zapytaniach, operatory złączenia (wewnętrznego), zapytania sumaryczne, podzapytania, operatory złączenia zewnętrznego.

**2.8 Słownik pojęć**

[UNION, UNION ALL](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#UNION) - operatory sumowania zbiorów wyników zapytań.

[INTERSECT](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#INTERSECT) - operator przecięcie zbiorów wyników zapytań.

[EXCEPT](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#EXCEPT) - operator różnicy zbiorów wyników zapytań (w Oracle także MINUS).

[złączenie tabel](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#Zlaczenia) - połączenie ze sobą danych zawartych w dwóch tabelach. Wynikiem jest też tabela.

[JOIN](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#JOIN) - operator złączenia występujący w klauzuli FROM.

[zapytanie sumaryczne (podsumowujące)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#Zapytania sumaryczne) - zapytanie wykonujące operacje sumaryczne na kolumnach wartości. Wynikiem jest tabela złożona z pojedynczego wiersza podsumowań.

[zapytanie grupujące](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#Grupuj) - zapytanie wykonujące najpierw podział wierszy na grupy a następnie operacje sumaryczne na kolumnach wartości - osobno dla każdej grupy. Wynikiem jest tabela złożona z tylu wierszy podsumowań ile jest grup.

[podzapytanie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#Podzapytania) - zapytanie występujące jako część innego zapytania.

[złączenie zewnętrzne](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w2.htm#Zlaczenie zewn) - specjalny rodzaj złączenia powodujący, że w wyniku zapytania znajdą się wszystkie wiersze z danej tabeli a nie tylko te które można uzgodnić z wierszami drugiej tabeli.

## Wykład 3

***SQL - zaawansowane konstrukcje***

**Streszczenie**

Na tym wykładzie kontynuujemy omawianie kolejnych obiektów i związanych z nimi instrukcji języka SQL. Przedstawiamy więc takie obiekty jak: *perspektywy*, *synonimy*, *schematy*, *dziedziny atrybutów*, *asercje*, *tabele tymczasowe*, *sekwencje* i *klastry*. Dodatkowo omawiamy problem reprezentowania w bazie danych metadanych (danych o danych) za pomocą *słownika danych* (inaczej nazywanego *katalogiem systemowym*) oraz podstawowy dla środowiska współbieżnego wykonywania operacji na bazie danych przez wielu użytkowników temat *transakcji*, *blokad* i *poziomów izolacji*.

**3.1** **Perspektywy**

W sposób oczywisty dobrze byłoby zapamiętywać zapytania w bazie danych, aby je móc później użyć. Do tego celu służy pojęcie *perspektywy*. Perspektywa jest to *wirtualna* tabela. Jest określona przez zapytanie czyli instrukcję SELECT.

|  |
| --- |
| CREATE VIEW *nazwa\_perspektywy* AS *zapytanie*; |

Można jej używać tak jakby była tabelą np. w instrukcji SELECT. Wiersze perspektywy nie są jednak przechowywane w bazie danych - mogą być tylko wyliczone na żądanie.

CREATE VIEW Urzędnicy (Empno, Ename, Sal)  
AS SELECT Emp.Empno, Emp.Ename, Emp.Sal  
   FROM Emp  
   WHERE Emp.Job = 'CLERK';

W Standardzie podzapytanie jest ujmowane w nawiasy.

|  |
| --- |
| Perspektywy służą do dostosowania bazy danych do potrzeb różnych grup użytkowników. Stanowią widok, perspektywę, z jakiej dana klasa użytkowników widzi bazę danych. Różne grupy użytkowników mogą mieć odmienne "perspektywy" na dane w bazie danych. |

Skoro perspektywy mają zastępować użytkownikom tabele, muszą także umożliwiać wykonywanie podobnych operacji, jakie są wykonywane na tabelach, a więc takich jak przeglądanie i wyszukiwanie danych, wstawianie, modyfikowanie i usuwanie danych z bazy danych.

Postulat dotyczący przeglądania i wyszukiwania danych jest w pełni spełniony. W instrukcjach SELECT perspektyw można używać na takich samych zasadach jak tabel, bez żadnych ograniczeń np.

SELECT u.Ename, u.Sal  
FROM Urzędnicy u  
WHERE u.Sal>5000  
ORDER BY u.Ename;

Przedyskutujemy teraz możliwość wykonywania operacji INSERT, DELETE i UPDATE.

Przykład

Rozpatrzmy perspektywę:

CREATE VIEW Prac\_sprzedaży  
AS SELECT \* FROM Emp  
   WHERE Emp.Deptno= 20;

Używając tej perspektywy możemy przeprowadzać modyfikację danych w odniesieniu do pracowników z działu numer 20, np. możemy podnieść ich zarobki *Sal* o 10%:

UPDATE Prac\_sprzedaży  
SET Sal = Sal \* 1.1;

Każdy system baz danych ogranicza zakres perspektyw, przez które można dokonywać zmian w bazie danych. W odniesieniu do Standardu i systemu Oracle mamy do czynienia z następującymi ograniczeniami:

1. w klauzuli SELECT nie może być słowa DISTINCT,

2. w klauzuli FROM może być tylko jedna nazwa tabeli lub jedna nazwa perspektywy - spełniająca definiowane kryteria - dotyczy to Standardu, gdyż Oracle umożliwia użycie więcej niż jednej tabeli,

3. na liście SELECT mogą znajdować się tylko nazwy kolumn,

4. w klauzuli WHERE nie może być podzapytania,

5. w zapytaniu nie mogą występować klauzule GROUP BY i HAVING.

Powyższe warunki zapewniają to, aby jednoznacznie był określony wiersz w tabeli, którego dotyczy zmiana. W Oracle możliwe jest dokonywanie ograniczonych zmian w bazie danych poprzez perspektywy zawierające złączenia. Można mianowicie w Oracle dokonywać zmian po stronie klucza obcego, ale nie głównego. Przy użyciu perspektywy:

CREATE VIEW Ed  
AS SELECT e.Empno, e.Ename, d.Deptno, d.Loc  
   FROM Emp e, Dept d  
   WHERE e.Deptno = d.Deptno;

możliwe są zmiany (INSERT, DELETE, UPDATE) w odniesieniu do tabeli *Emp*, ale nie w odniesieniu do tabeli *Dept.* (Istotne przy tym jest aby kolumna *Empno* była kluczem głównym w tabeli EMP.)

Przykład

Przy użyciu instrukcji:

INSERT INTO Ed(Empno, Ename, Deptno)  
VALUES (5600, 'Nowiński',20);

wstawimy pracownika o nazwisku 'Nowiński' do tabeli *Emp* ustalając jego numer działu na 20.

Istnieje też oczywiście możliwość usunięcia perspektywy z bazy danych. Instrukcja

|  |
| --- |
| DROP VIEW *nazwa\_perspektywy*; |

powoduje usunięcie perspektywy.

**Tworzenie perspektyw z opcją sprawdzania**

W celu zapewnienia ograniczenia danych operowanych przy użyciu perspektywy do danych określonych przez jej warunek WHERE, użyteczna jest dodatkowa opcja definicji perspektywy:

|  |
| --- |
| CREATE VIEW *nazwa\_perspektywy*[(*nazwa\_kolumny*,...)] AS *zapytanie* WITH CHECK OPTION; |

Jeżeli została użyta opcja WITH CHECK OPTION, to przy wykonywaniu INSERT i UPDATE następuje sprawdzenie, czy wstawiany bądź modyfikowany wiersz spełnia warunek określony w klauzuli WHERE. Jeśli spełnia, operacja jest wykonywana. Jeśli nie spełnia, operacja nie zostanie wykonana.

Gwarantuje to dodatkową ochronę danych zapewniając, że zmiany dokonywane przez użytkowników przez perspektywę są wyłącznie ograniczone do zbioru danych, do których oglądania i modyfikowania jest uprawniony dany użytkownik. Jest to zgodne z ideą dopasowywania poziomu zewnętrznego do danej grupy użytkowników.

Przykład

Utwórz perspektywę pracowników będących na urlopie bezpłatnym.

CREATE VIEW Emp\_na\_urlopie\_bezpłatnym  
AS SELECT \*   
   FROM Emp  
   WHERE Emp.Sal = 0 OR Emp.Sal IS NULL  
WITH CHECK OPTION;

Poprzez tę perspektywę modyfikacja danych jest możliwa tylko w stosunku do pracowników, których zarobki są równe 0 albo *Null*, ponieważ tylko tacy pracownicy są widoczni w tej perspektywie. Same zarobki mogą być jednak zmienione tylko na 0 albo *Null*, ponieważ po operacji warunek w klauzuli WHERE perspektywy też musi być spełniony. Oznacza to w praktyce zablokowanie zmiany zarobków poprzez tę perspektywę np. nie uda się zmiana wysokości zarobków wszystkim pracowników na urlopie bezpłatnym:

UPDATE Emp\_na\_urlopie\_bezpłatnym  
SET Sal = 10000;

**Tworzenie perspektyw z opcją tylko do odczytu**

Kolejnym elementem ochrony danych dostarczanym przez perspektywy jest opcja tylko do odczytu. Mianowicie można zabezpieczyć się przed dokonywaniem zmian poprzez perspektywę przez utworzenie jej z opcją WITH READ ONLY.

|  |
| --- |
| CREATE VIEW *nazwa\_perspektywy*[(*nazwa\_kolumny*,...)] AS *zapytanie* WITH READ ONLY; |

Na przykład:

CREATE VIEW Pracownicy  
AS SELECT \* FROM Emp  
   WITH READ ONLY;

Każdy uprawniony użytkownik może tylko oglądać dane o pracownikach bez możliwości wykonania wstawiania, modyfikowania czy usuwania wierszy.

Reasumując, perspektywy są obiektami, jakie udostępnia się konkretnym grupom użytkowników. Określają one widok na bazę danych zaprojektowany dla tej grupy użytkowników. Ułatwia to użycie bazy danych, jak również stanowi element ochrony przed niepowołanym lub nieprawidłowym dostępem do danych. Każdy użytkownik bazy danych ma dostęp tylko do danych dotyczących jego działalności w firmie. Ponadto jeśli w aplikacji korzystamy z perspektyw a nie z tabel to mamy zagwarantowaną niezależność logiczną danych tzn. po zmianie schematu tabel trzeba tylko ewentualnie zmienić definicje perspektyw a same aplikacje nie wymagają już zmiany.

Postulat modyfikowalności bazy danych przez perspektywy nie jest w pełni spełniony przez Standard SQL z 1992 roku. Do pełnej realizacji tego postulatu potrzebna jest dodatkowa konstrukcja - *wyzwalacz INSTEAD OF -* wprowadzona do Standardu SQL'1999 i omawiana w wykładzie 6.

**Perspektywa zmaterializowana**

Ze względu na użycie w rozproszonych bazach danych i w hurtowniach danych, kopię zawartości perspektywy można materializować tzn. zapisywać jako tabelę. Tego rodzaju perspektywa nosi nazwę *perspektywy zmaterializowanej.* Jej użycie zostanie przedstawione w wykładach 14 i 15.

Dokonamy teraz przeglądu najważniejszych dodatkowych konstrukcji języka SQL biorąc pod uwagę zarówno Standard, jak i system Oracle.

**3.2** **Tworzenie synonimów nazw tabel i perspektyw**

Ze względu na możliwość sięgania do obiektów znajdujących się w innych bazach danych (również przez sieć), specyfikacje obiektów mogą być długie. Wygodnie jest więc zdefiniować krótką nazwę i używać jej zamiast długiej specyfikacji. Do tego właśnie celu służą *synonimy*.

|  |
| --- |
| CREATE SYNONYM *nazwa\_synonimu* FOR *nazwa\_tabeli\_lub\_perspektywy*; |

Przykład

Utwórz synonim tabeli *Dept*.

CREATE SYNONYM Dept  
FOR Kadry.Dept@mojafirma.com.pl;

Instrukcja

|  |
| --- |
| DROP SYNONYM *nazwa\_synonimu*; |

powoduje usunięcie synonimu.

Synonimy uniezależniają aplikacje i użytkowników od zmian w ścieżkach do obiektów bazy danych np. gdy tabela z danymi o pracownikach wędruje z jednego konta, lub bazy danych na inne konto, lub do innej bazy danych.

**3.3** **Transakcje**

Z charakteru aplikacji korzystających z baz danych wynika, że elementarną operacją na bazie danych często nie jest pojedyncza instrukcja SQL, ale ciąg takich instrukcji, nazywany *transakcją*. Na przykład, transfer  pieniędzy z jednego konta na drugie jest elementarną operacją z punktu widzenia aplikacji bankowej. Natomiast w SQL, użylibyśmy w tym celu co najmniej dwóch instrukcji UPDATE:

UPDATE Konta  
SET Saldo = Saldo - 1000  
WHERE Id\_klienta = 1001;

UPDATE Konta  
SET Saldo = Saldo + 1000  
WHERE Id\_klienta = 9999;

Załóżmy teraz, że pierwsza instrukcja wykonała się, a druga nie może zostać wykonana, na przykład z powodu tego, że 9999 jest błędnym identyfikatorem klienta albo z powodu awarii komputera. Z punktu widzenia aplikacji dane znalazły się w stanie niespójnym i pozostaje tylko jedna możliwość - wycofać (anulować) wynik pierwszej instrukcji UPDATE. Do tego celu służy instrukcja ROLLBACK.

Symetryczną operacją do ROLLBACK jest COMMIT, która zatwierdza zmiany w bazie danych bez możliwości późniejszego ich wycofania. Reasumując, użytkownik lub aplikacja po wykonaniu instrukcji wchodzących w skład transakcji, albo przekazuje do wykonania instrukcję COMMIT - **zatwierdź zmiany i zakończ transakcję**, albo przekazuje do wykonania instrukcję ROLLBACK - **wycofaj zmiany i zakończ transakcję**.

Istnieje możliwość wycofania w tył części ostatnio wykonywanych instrukcji, pod warunkiem, że zaznaczyliśmy tę część stawiając przed nią tzw. *punkt zachowania* (ang. savepoint):

|  |
| --- |
| SAVEPOINT *punkt*; |

Instrukcja wycofania ma wtedy postać:

|  |
| --- |
| ROLLBACK TO SAVEPOINT *punkt*; |

|  |
| --- |
| Uwaga W Oracle istnieje możliwość zagnieżdżania transakcji. Wymaga to użycia języka PL/SQL i dyrektywy PRAGMA (szczegóły w dalszych wykładach). Zagnieżdżona transakcja nazywa się *autonomiczną*. Oto przykład:  .... DECLARE    PRAGMA autonomous\_transaction; BEGIN    DELETE FROM Emp WHERE Emp.Job='GHOSTBUSTER';    COMMIT; END; .......  Instrukcje SQL znajdujące się w bloku między słowami kluczowymi DECLARE i END są wykonywane jako niezależna, autonomiczna transakcja wewnątrz innej transakcji. |

W trakcie wykonywania transakcji użytkownika SZBD blokuje dostęp innym  użytkownikom do obiektów bazy danych, których dotyczą operacje w transakcji. Szczegóły implementacyjne zostaną przedstawione w dalszych wykładach na temat budowy SZBD.

**3.4** **Blokowanie dostępu do tabel (Oracle)**

Oracle zakłada automatycznie blokady na wierszach, na których są przeprowadzane operacje zmieniające te wiersze (INSERT, DELETE, UPDATE).

Programista może sam założyć blokadę na tabelę ograniczając możliwości jej zmiany przez innych użytkowników. Do tego celu służy następująca instrukcja.

|  |
| --- |
| LOCK TABLE *nazwa\_tabeli* IN [SHARE|EXCLUSIVE] MODE [NOWAIT]; |

Podstawowymi trybami blokowania tabeli w bazie danych są:

* blokada współdzielona SHARE - blokada tabeli przeciwko zmianom dokonywanym przez kogokolwiek; wielu użytkowników może założyć taką blokadę jednocześnie;
* blokada wyłączna EXCLUSIVE - blokada tabeli przeciwko zmianom dokonywanym przez innych użytkowników, samemu można zmieniać; tylko jeden użytkownik może mieć założoną taką blokadę.

Opcja NOWAIT specyfikuje, aby nie czekać na założenie blokady, gdy nie można jej uzyskać w danej chwili.

Przykład założenia wyłącznej blokady na całą tabelę:

LOCK TABLE Klienci  
IN EXCLUSIVE MODE   
-- zablokowanie całej tabeli w trybie wyłącznym  
NOWAIT;

W Oracle jest przyjęta zasada, że każdy użytkownik - mający odpowiednie uprawnienia - może odczytywać aktualne, zatwierdzone dane w tabeli, niezależnie od założonych blokad (dane te mogą być w trakcie zmieniania przez inne transakcje). Zatem może się zdarzyć sytuacja, że gdy za chwilę użytkownik odczyta ponownie te same dane, zobaczy, że w międzyczasie zostały one zmienione.

Zdjęcie blokady następuje przez wykonanie COMMIT lub ROLLBACK.

W systemie Oracle jest możliwe założenie blokady wyłącznej na wiersze będące wynikiem określonego zapytania. Do tego celu służy specjalna, dodatkowa klauzula dołączana na koniec instrukcji SELECT:

|  |
| --- |
| SELECT *nazwa\_kolumny,. . .* . . . FOR UPDATE [NOWAIT]; |

Opcja NOWAIT oznacza kontynuowanie aplikacji gdy nie można założyć blokady na wiersze.

Wykonanie COMMIT lub ROLLBACK kończy blokadę wskazanych wierszy.

Przykład założenia wyłącznej blokady na wybrane wiersze:

SELECT \* FROM Klienci  
WHERE Kraj = 'Polska'  
FOR UPDATE   
-- założenie blokady wyłącznej na klientów z Polski i współdzielonej na tabelę Klienci  
NOWAIT;  
-- gdy nie można założyć blokady, nie czekaj

Jest też możliwe "zamrożenie" widoku bazy danych do jej aktualnego stanu dla następnie wykonywanych w ramach transakcji instrukcji SELECT. Instrukcje SELECT korzystają wówczas jakby ze "zdjęcia migawkowego" bazy danych wykonanego w tym momencie. Zmiany wprowadzane w tym czasie do bazy danych nie są widoczne dla tych instrukcji SELECT. Instrukcja

SET TRANSACTION READ ONLY;

powinna być pierwszą instrukcją w transakcji – złożonej z ciągu instrukcji SELECT i LOCK TABLE. ''Odmrożenie'' bazy danych następuje przez wykonanie polecenia COMMIT.

Dla zachowania symetrii, rozpoczęcie transakcji typu nie READ ONLY można zacząć (nie jest to wymagane) od

SET TRANSACTION READ WRITE;

**Poziomy izolacji transakcji**

Poziom izolacji określa jakie zjawiska związane ze współbieżnym wykonywaniem transakcjami są dopuszczalne:

* *Odczyt niezatwierdzonych danych* (ang. *dirty read*) - transakcja odczytuje dane, które zmieniła druga transakcja ale ich nie zatwierdziła.
* *Niepowtarzalny odczyt* (ang. *unreapetable read*) - transakcja widzi zmiany wprowadzane przez zatwierdzone transakcje. Wynik tego samego zapytania może być różny w czasie w ramach jednej transakcji.
* *Fantom* - wiersz, którego nie było w tabeli na początku wykonywania transakcji, a który został wprowadzony przez zatwierdzoną transakcję w trakcie wykonywania transakcji. Z tego powodu wynik tego samego zapytania może być różny w czasie w ramach jednej transakcji.

Standard ANSI/ISO definiuje cztery poziomy izolacji określające jakie zmiany widzą transakcje - zmiany dokonywane przez inne współbieżnie działające transakcje.

|  |  |  |  |
| --- | --- | --- | --- |
| **Poziom izolacji** | **Niezatwierdzony odczyt** | **Niepowtarzalny odczyt** | **Fantomy** |
| READ UNCOMMITED | TAK | TAK | TAK |
| READ COMMITED | NIE | TAK | TAK |
| REPEATABLE READS | NIE | NIE | TAK |
| SERIALIZABLE | NIE | NIE | NIE |

**Ustawianie poziomu izolacji w SQL**

Programista, w zależności od potrzeb, może określić wymagany *poziom izolacji* transakcji. Składnia instrukcji określającej poziom izolacji jest następująca (są to wspólne opcje występujące zarówno w Standardzie, jak i w Oracle):

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
.......  
COMMIT;

SET TRANSACTION ISOLATION LEVEL READ COMMITED;  
.......  
COMMIT;

Wybierając poziom SERIALIZABLE (*izolowana transakcja*) mamy gwarancję, że transakcja działa na spójnych, niezmiennych na czas działania transakcji, danych. Jest to domyślny poziom w Standardzie.

Na poziomie READ COMMITED (*transakcja z odczytem zatwierdzonych danych*) transakcja działa na zmienianych jednocześnie przez innych użytkowników danych (ale dopiero po zatwierdzeniu przez nich zmian). Zapewnia większą współbieżność działania systemu baz danych. Jest to domyślny poziom w Oracle.

Wybierając drugi poziom, dane którymi zajmuje się transakcja, mogły być, już w czasie gdy transakcja była wykonywana, zmieniane przez inne transakcje (po czym zatwierdzone). Jeśli jest to dopuszczalne z punktu widzenia poprawności aplikacji, to ten drugi sposób, jako zapewniający większą współbieżność działania systemu baz danych, powinien być preferowany.

Nie wszystkie aplikacje będą działać poprawnie w tym trybie. Na przykład, jeśli aplikacja dokonująca rezerwacji biletów, najpierw sprawdza czy jest wolne miejsce w samolocie, następnie powiadamia klienta, że znalazła wolne miejsce, a następnie próbuje dokonać rezerwacji znalezionego miejsca, to w tym momencie miejsce w samolocie może nie być już wolne.

Pracując w trybie READ COMMITED, gdy nie chcemy aby wartości w wierszu zmieniły się, wystarczy przy odczytywaniu wiersza jednocześnie dokonać blokady tego wiersza. Możemy przed instrukcją SELECT postawić punkt zachowania (savepoint), aby w przypadku gdy wartości w wierszu nam nie odpowiadają dokonać wycofania do tego punktu zachowania. Unikniemy w ten sposób blokowania wiersza innym użytkownikom.

**3.5** **Słownik danych - informacja o obiektach bazy danych (w wersji używanej przez Oracle)**

Słownik danych jest zbiorem informacji o obiektach bazy danych. Jest używany zarówno przez system zarządzania bazą danych, jak i przez użytkowników. Użytkownik ma uprawnienia tylko do odczytu informacji ze słownika danych. Słownik danych ma postać zbioru tabel i perspektyw.

Oto przykładowe perspektywy słownika danych Oracle:

1. z przedrostkiem User - informacja o wszystkich obiektach, których dany użytkownikiem jest właścicielem - o tabelach: User\_Tables (Tabs), o tabelach i kolumnach: User\_Tab\_Columns (Cols), o więzach spójności User\_Constraints oraz User\_Cons\_Columns, o indeksach User\_Indexes (Ind) oraz User\_Ind\_Columns, o synonimach User\_Synonyms (Syn), o perspektywach User\_Views;

2. z przedrostkiem All - dotyczy wszystkich obiektów, do których użytkownik ma uprawnienia;

3. z przedrostkiem Dba - informacja o obiektach dostępnych dla administratorów systemu.

Najpowszechniej stosowaną przez użytkowników perspektywą jest perspektywa User\_Tables.

Przykład

Instrukcja:

SELECT u.Table\_Name  
FROM User\_Tables u;

wypisuje nazwy tabel, których właścicielem jest dany użytkownik. Instrukcja

SELECT u.Column\_Name, u.Data\_Type  
FROM User\_Tab\_Columns u  
WHERE u.Table\_Name = 'EMP';

wypisuje nazwy i typy danych kolumn w tabeli *Emp*. To znaczy wypisuje tabelę wartości:

COLUMN\_NAME DATA\_TYPE

------------------------------ ----------------------

EMPNO NUMBER

ENAME VARCHAR2

JOB VARCHAR2

MGR NUMBER

HIREDATE DATE

SAL NUMBER

COMM NUMBER

DEPTNO NUMBER

**3.6** **Przyznawanie i odbieranie uprawnień w bazie danych**

Do podstawowych obowiązków administratora bazy danych należy zakładanie kont użytkowników oraz przyznawanie im uprawnień do korzystania z zasobów i obiektów bazy danych. Zarówno w Standardzie, jak i Oracle mamy instrukcję do przyznawania uprawnień:

|  |
| --- |
| GRANT *uprawnienie*, ...  ON *nazwa\_obiekt*u  TO *użytkownik*, ...|PUBLIC [WITH GRANT OPTION] |

gdzie:

1. *uprawnienie* określa rodzaj operacji, do wykonywania której użytkownik uzyskuje uprawnienie, np. ALL PRIVILEGES, SELECT, DELETE, INSERT, UPDATE, REFERENCES (w przypadku INSERT, UPDATE, REFERENCES można ograniczyć uprawnienia do podzbioru kolumn), a w Oracle także INDEX, ALTER i EXECUTE;
2. *obiektem* może być tabela, perspektywa a w Oracle także inne obiekty jak sekwencja, procedura, funkcja, pakiet;
3. WITH GRANT OPTION oznacza, że obdarowany użytkownik uzyskane uprawnienia może przekazywać innym użytkownikom;
4. PUBLIC oznacza przyznanie uprawnienia wszystkich użytkownikom.

Przykład

|  |
| --- |
| Nadaj księgowej uprawnienia dostępu do tabeli *Emp* w zakresie wykonywania instrukcji SELECT oraz również instrukcji UPDATE, ale tylko w odniesieniu do kolumny *Sal*. |

GRANT SELECT, UPDATE(Sal)   
ON Emp  
TO Księgowa;

Uprawnienia odwołuje się za pomocą instrukcji odwrotnej do instrukcji GRANT.

|  |
| --- |
| REVOKE *uprawnienie*, ...  ON *nazwa\_obiektu*  FROM *użytkownik*, ...|PUBLIC; |

W Oracle jest też postać instrukcji GRANT przyznającej uprawnienia systemowe do wykonywania instrukcji, np. CREATE TABLE, CREATE VIEW, CREATE SESSION, czyli uprawnienie do logowania się do bazy danych.

Administrator bazy danych Oracle tworzy konto użytkownika za pomocą instrukcji (jak to robić, Standard pozostawia tę kwestię nieokreśloną):

|  |
| --- |
| CREATE USER *użytkownik* IDENTIFIED BY *hasło*; |

i w pierwszym kroku nadaje uprawnienia systemowe np.

|  |
| --- |
| GRANT CREATE SESSION, CREATE TABLE TO *użytkownik*; |

Konto użytkownika może zostać usunięte (w Oracle) za pomocą instrukcji

|  |
| --- |
| DROP USER *użytkownik* [CASCADE]; |

Opcja CASCADE jest wymagana, gdy użytkownik jest właścicielem obiektów w bazie danych. Wtedy system usuwa użytkownika razem ze wszystkimi jego obiektami.

W celu ułatwienia przyznawania uprawnień dużej liczbie użytkowników, w Oracle zostało wprowadzone pojęcie *roli*, to jest powiązania grupy użytkowników ze zbiorem uprawnień. Z jednej strony, konkretnym użytkownikom przyznaje się uprawnienie do danej roli (przyznając im tę rolę). Z drugiej strony, roli przyznaje się konkretne uprawnienia do wykonywania operacji w bazie danych.

Przykład

|  |
| --- |
| Utwórz role Dyrektora i Urzędnika, przydziel im uprawnienia i zalicz do nich konkretnych użytkowników: |

CREATE ROLE Dyrektor;

CREATE ROLE Urzędnik;

GRANT SELECT ON Emp TO Dyrektor;

GRANT SELECT, UPDATE ON Emp TO Urzędnik;

GRANT Dyrektor TO King, Leon;

GRANT Urzędnik TO Liza, Ewa, Marta;

Trzy role są z góry zdefiniowane:

* CONNECT - zawierająca uprawnienie CREATE SESSION,
* RESOURCE - uprawnienia do tworzenia obiektów,
* DBA - uprawnienia do wykonywania funkcji administratora systemu.

Przy tworzeniu konta nowego użytkownika, któremu chcemy przyznać swobodę zakładania i operowania obiektami na jego koncie, przydziela się najczęściej role: CONNECT i RESOURCE – umożliwiające logowanie się na konto i pełne korzystanie z instrukcji SQL w odniesieniu do jego własnych obiektów.

Natomiast w przypadku, gdy użytkownik korzysta z gotowej aplikacji klienckiej przyznaje mu się tylko rolę CONNECT, a pozostałe role konieczne do zrealizowania aplikacji włącza się i wyłącza w trakcie realizacji aplikacji bez jego udziału. Mianowicie instrukcja SET ROLE powoduje wyłączenie wszystkich aktualnie dostępnych ról i włączenie ról określonych w instrukcji SET ROLE. Np. wykonanie instrukcji

SET ROLE agent;

spowoduje wyłączenie wszystkich dotychczasowych ról i włączenie tylko jednej o nazwie *agent*.

**3.7** **Schemat**

Transakcje dotyczą wykonywania ciągu instrukcji INSERT, DELETE i UPDATE. Odpowiednikiem transakcji dla instrukcji definiujących obiekty i uprawnienia jest pojęcie *schematu*. Schemat tworzy grupę powiązanych obiektów. Jest realizowany za pomocą instrukcji:

|  |
| --- |
| CREATE SCHEMA *nazwa\_schematu* *ciąg instrukcji* CREATE TABLE, CREATE VIEW i GRANT (bez rozdzielających średników); |

Przy wykonywaniu instrukcji CREATE SCHEMA ciąg składowych instrukcji jest realizowany jako jedna transakcja. To znaczy, albo są wykonywane wszystkie instrukcje albo żadna. W instrukcjach składowych mogą być odwołania cykliczne REFERENCES między tabelami (tabela *A* zawiera odwołanie do tabeli *B*, a tabela *B* zawiera odwołanie do tabeli *A*) - co przy normalnej implementacji nie byłoby możliwe do zrealizowania za pomocą samych instrukcji CREATE TABLE (bez ALTER TABLE).

W Standardzie jest też instrukcja (nie ma jej w Oracle)

|  |
| --- |
| DROP SCHEMA *nazwa\_schematu* CASCADE; |

która usuwa wszystkie obiekty danego schematu z bazy danych.

W Standardzie nie występuje w sposób jawny pojęcie bazy danych, za to obok pojęcia schematu występuje jeszcze pojęcie *katalogu* jako zbioru schematów oraz *klastra* jako zbioru katalogów. Klaster może być traktowany jako rozproszona baza danych składająca się ze zbioru katalogów, do których użytkownik ma dostęp w ramach jednej sesji. Schemat ma jednego właściciela. W skład katalogu mogą wchodzić schematy mające różnych właścicieli. Dla każdego katalogu powinien być określony jeden schemat nazywany *schematem informacyjnym* pełniący rolę słownika danych dla całego katalogu.

**3.8** **Sesje i połączenia**

Zarówno w Standardzie, jak i w Oracle istnieją możliwości zmiany ustawień dla danej sesji użytkownika.

W Standardzie z każdą sesją jest związany zbiór połączeń z różnymi bazami danych, z których tylko jedno jest aktywne. Stosowane są następujące polecenia:

* uzyskiwanie nowego połączenia

|  |
| --- |
| CONNECT TO *nazwa\_serwera*; |

* zmiana istniejącego połączenia

|  |
| --- |
| SET CONNECTION *nazwa\_serwera*; |

* rozłączenie połączenia

|  |
| --- |
| DISCONNECT *nazwa\_serwera*; |

Niektóre własności sesji mogą być zmieniane w trakcie połączenia. W Oracle zmiany wprowadza się za pomocą instrukcji ALTER SESSION, na przykład:

    ALTER SESSION SET ISOLATION\_LEVEL=SERIALIZABLE;

powoduje ustawienie poziomu izolacji transakcji w sesji na SERIALIZABLE.

**3.9** **Dziedziny atrybutów (Standard)**

Dziedziny atrybutów to typy danych definiowane łącznie z więzami spójności.

Przykład

Zdefiniujmy dziedzinę numerów departamentów:

CREATE DOMAIN Dept# CHAR(3)  
CHECK (VALUE IN ('A00','B01','D01','D11','D21', 'XXX'))  
DEFAULT 'XXX';

Następnie możemy dziedzinę Dept# wielokrotnie używać przy określaniu typów danych kolumn w tabelach:

CREATE TABLE Dept  
(DeptNo DOMAIN Dept# PRIMARY KEY,  
...);

Występują także instrukcje ALTER DOMAIN *nazwa\_dziedziny* i DROP DOMAIN *nazwa\_dziedziny*.

**3.10** **Asercje (Standard)**

*Asercje* to więzy spójności definiowane poza instrukcjami CREATE TABLE i ALTER TABLE.

Możliwe jest sformułowanie więzów spójności dotyczących całej tabeli

CREATE ASSERTION maxempl  
CHECK (1000 <= SELECT COUNT (\*)  
               FROM Emp);

Więzy spójności można usunąć za pomocą instrukcji

|  |
| --- |
| DROP ASSERTION *nazwa\_asercji*; |

**3.11** **Tabele tymczasowe**

Zarówno w Standardzie jak i w Oracle występuje konstrukcja *tabeli tymczasowej*, która jest częścią schematu bazy danych, ale której zawartość jest niszczona przy każdym zakończeniu sesji użytkownika (opcja ON COMMIT PRESERVE ROWS) lub już przy każdej operacji COMMIT (opcja ON COMMIT DELETE ROWS).

Przykład

Utwórz tabelę tymczasową

CREATE GLOBAL TEMPORARY TABLE Prac\_zatrudniani\_dziś  
  (Nr\_kolejny INTEGER PRIMARY KEY,  
  Imię VARCHAR(40) NOT NULL,  
  Nazwisko VARCHAR(50) NOT NULL,  
  Informacja VARCHAR(1000))  
ON COMMIT PRESERVE ROWS;

Tabele tymczasowe stosuje się do zapisu wyników, które następnie są używane wielokrotnie w ramach tej samej transakcji lub sesji. Wyniki mogą być wyliczane przez więcej niż jedną instrukcję SELECT.

**3.12** **Generowanie jednoznacznych numerów (Oracle)**

W systemie Oracle jest specjalny obiekt - *sekwencja* (przechowywany w bazie danych), który służy do generowania jednoznacznych numerów dla wartości kluczy głównych i jednoznacznych. Oto składnia tworzenia sekwencji:

|  |
| --- |
| CREATE SEQUENCE *nazwa\_sekwencji* [INCREMENT BY *k*] [START WITH *n*]; |

gdzie *n* jest pierwszą generowaną liczbą naturalną, a *k* jest wartością, o jaką wzrasta kolejno generowany numer.

Sekwencji używa się zgodnie ze składnią:

nazwa\_sekwencji.NextVal - generowanie kolejnej wartości w sekwencji,

nazwa\_sekwencji.CurrVal - ostatnio wygenerowana wartość w sekwencji.

Instrukcja

|  |
| --- |
| DROP SEQUENCE *nazwa\_sekwencji*; |

usuwa sekwencję.

Poniżej podajemy przykład użycia sekwencji do generowania jednoznacznych identyfikatorów przy tworzeniu działu i jego pracowników.

Przykład

|  |
| --- |
| Utwórz dwie sekwencje generujące jednoznaczne numery dla działów i dla pracowników. |

CREATE SEQUENCE Dept\_seq  
INCREMENT BY 10  
START WITH 10;

CREATE SEQUENCE Emp\_seq  
INCREMENT BY 1  
START WITH 1000;

Oto instrukcja tworząca kolejny dział:

INSERT INTO Dept(Deptno, Dname, Loc)  
VALUES (Dept\_seq.NextVal, 'Sprzedaż', 'Warszawa');

Oto instrukcja tworząca kolejnego pracownika z wykorzystaniem wygenerowanego przed chwilą numeru działu:

INSERT INTO Emp VALUES (Emp\_seq.NextVal, 'Stec',   
'MANAGER', NULL, '20-JAN-90', 10000, 0 , Dept\_seq.CurrVal);

**3.13** **Klaster (Oracle)**

*Klaster* to zbiór powiązanych ze sobą tabel, które zwykle w aplikacji są przetwarzane jednocześnie i dlatego wskazane jest aby ich wiersze były przechowywane obok siebie na dysku. Powiązanie tabel odbywa się za pomocą kolumn o wspólnych wartościach. Kolumny te tworzą indeks klastra. Najpierw tworzy się klaster, następnie tabele w klastrze i w końcu indeks klastra. Dopiero w tym momencie można rozpocząć wstawianie wierszy do tabel w klastrze.

Oto przykład klastra dla tabel *Dept* i *Emp*. Indeksem klastra, względem którego gromadzone są obok siebie na dysku wiersze tych tabel, jest numer działu *Deptno*. A więc razem jest trzymany wiersz opisujący dział, jak i wiersze pracowników przypisanych do tego działu.

CREATE CLUSTER Personnel  
(Department\_number NUMBER(2));

CREATE TABLE Dept  
(Deptno NUMBER(2) PRIMARY KEY,  
 Dname VARCHAR2(9),   
 Loc VARCHAR2(9))  
CLUSTER Personnel (Deptno);

CREATE TABLE Emp  
(Empno NUMBER PRIMARY KEY,  
 Ename VARCHAR2(10) NOT NULL,  
 Job VARCHAR2(9),  
 Mgr NUMBER REFERENCES Emp(Empno),  
 Hiredate DATE,  
 Sal NUMBER(10,2),  
 Comm NUMBER(9,0),  
 Deptno NUMBER(2)NOT NULL REFERENCES Dept(Deptno))  
CLUSTER Personnel (Deptno);

Zanim rozpocznie się korzystać z tabel w klastrze należy explicite utworzyć indeks klastra:

CREATE INDEX Idx\_personnel ON CLUSTER Personnel;

W tym momencie można zacząć wstawiać dane do obu tabel znajdujących się w utworzonym klastrze.

**3.14 Podsumowanie**

W kolejnych trzech wykładach przedstawiliśmy podstawowe konstrukcje języka SQL zarówno w wersji Standardu ISO1992, jak i Oracle. Znajomość tych konstrukcji jest nieodzowna przy tworzeniu aplikacji baz danych. Więcej informacji o środowisku tworzenia aplikacji w systemie Oracle znajduje się w następnych wykładach.

**3.15 Słownik pojęć**

[perspektywa](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Perspektywy) - wirtualna tabela czyli tabela, której źródłem wartości są inne tabele w bazie danych. Jest określona przez zapytanie czyli instrukcję SELECT. Można ją używać tak jakby była tabelą (np. w instrukcji SELECT). Wiersze perspektywy nie są jednak przechowywane w bazie danych - mogą być tylko wyliczone na żądanie.

[perspektywa z opcją sprawdzania](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#PSpr) - opcja ta zapewnia ograniczenie danych operowanych przy użyciu perspektywy do danych określonych przez jej warunek WHERE.

[perspektywa z opcją tylko do odczytu](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#POdcz) - opcja ta zapewnia ograniczenie użycia perspektywy tylko do odczytywania danych bez możliwości ich zmiany. Stanowi element ochrony danych w bazie danych.

[synonim nazwy tabeli lub perspektywy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Synonimy) - skrócona nazwa dla obiektu bazy danych.

[transakcja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Transakcje) - ciąg instrukcji SQL odpowiadający jednostkowej operacji biznesowej jak np. przelanie pieniędzy z konta na konto.

[blokada](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Blokady) - ograniczenie użycia obiektu aż do chwili zakończenie transakcji, która założyła tę blokadę. Podstawowe dwa rodzaje blokad to: *wyłączna* (do wykonania zapisu na obiekcie) oraz *współdzielona* (do uniemożliwienia wprowadzania zmian przez innych).

[poziom izolacji](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Poziomy) - stopień dopuszczalnej interakcji między dwiema transakcjami przy współbieżnym korzystaniu z tych samych obiektów.

[słownik danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Slownik danych) - informacje o danych w bazie danych. Są one zapisywane w wydzielonych tabelach bazy danych tak jak inne dane. Są udostępniane użytkownikom za pomocą perspektyw tylko do odczytu.

[uprawnienie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Uprawnienia) - przydzielone użytkownikowi prawo do wykonywania dokładnie określonych operacji na obiektach bazy danych.

[schemat](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Schemat) - tworzy grupę powiązanych obiektów.

[sesja, połączenie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Sesje) - sesja reprezentuje pracę użytkownika ze zbiorem baz danych; z każdą sesją jest związany zbiór połączeń z różnymi bazami danych, z których tylko jedno jest aktywne.

[dziedzina atrybutów](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Dziedziny atrybutow) - typ danych definiowany łącznie z więzami spójności.

[asercja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Asercje) - więzy spójności definiowane poza instrukcjami CREATE TABLE i ALTER TABLE.

[tabele tymczasowa](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Tabele tymczasowe) - tabela, która jest częścią schematu bazy danych, ale której zawartość jest niszczona przy każdym zakończeniu sesji użytkownika (opcja ON COMMIT PRESERVE ROWS) lub już przy każdej operacji COMMIT (opcja ON COMMIT DELETE ROWS).

[sekwencja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Sekwencje) - generator jednoznacznych identyfikatorów w Oracle.

[klaster](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Klaster) - struktura fizycznego przechowywania danych, w której kilka tabel jest zebranych razem według wartości wspólnej kolumny (kolumn) - lub wartości funkcji haszującej.

**Wykład 4**

***Programowanie aplikacji baz danych po stronie serwera***

**Streszczenie**

W tym wykładzie omówimy deklaratywne więzy spójności stanowiące jeden z elementów aplikacji bazy danych po stronie serwera.

Ponadto dla pełności obrazu tworzenia aplikacji w Oracle podamy informacje o interakcyjnym systemie poleceń SQL\*Plus – nieodzownym przy tworzeniu bazy danych Oracle i testowaniu aplikacji.

**4.1** **Aplikacja po stronie serwera bazy danych**

Tworząc aplikację należy podjąć decyzję, jaka jej część ma się znajdować na serwerze bazy danych, a jaka na komputerze użytkownika, czyli po stronie klienta. Nie ma wątpliwości, że dane oraz mechanizm wykonywania instrukcji SQL, powinny się znajdować po stronie serwera, natomiast „przyjazny” interfejs użytkownika – po stronie klienta. Natomiast jeśli chodzi o przetwarzanie danych, opinie są podzielone. Z punktu widzenia „interesu” bazy danych lepiej, żeby odbywało się ono na serwerze, który jest w stanie centralnie sprawować kontrolę nad spójnością danych. Ze względu na możliwość precyzyjniejszej diagnostyki, obsługa błędów dla przetwarzania danych może być lepiej zrobiona na stacji klienckiej.

**4.2** **Deklaratywne więzy spójności**

Więzy spójności mogą być sprawdzane zarówno po stronie serwera, jak i po stronie klienta. Bardziej naturalnym miejscem sprawdzania jest serwer, który obsługuje wszystkie możliwe aplikacje utworzone dla bazy danych. Lepiej jest mieć jeden mechanizm gwarantujący spełnienie więzów spójności niż wiele, potencjalnie niespójnych.

Są dwie podstawowe metody określania więzów spójności na serwerze: albo za pomocą mechanizmu deklaratywnych więzów spójności – w definicjach tabel (instrukcje CREATE TABLE i ALTER TABLE) albo za pomocą mechanizmu wyzwalaczy bazy danych, o których będzie mowa w wykładzie 6.

O więzach spójności wspominaliśmy już w wykładzie 1. Tam były one przedstawiane jako części instrukcji CREATE TABLE i ALTER TABLE, teraz będą omawiane jako część oprogramowania tworzącego aplikację, które umieszczamy po stronie serwera.

Zadaniem więzów spójności jest zagwarantowanie tego, aby dane w bazie danych wiernie odzwierciedlały świat rzeczywisty, dla którego baza danych została zaprojektowana. System bazy danych ma zapewnić, aby więzy spójności pozostawały prawdziwe przy wszystkich operacjach, jakie mogą zostać wykonane na bazie danych w postaci transakcji użytkowników, wyzwalaczy bazy danych, ładowania lub importu danych do bazy danych.

Są dwa typy deklaratywnych więzów spójności:

1. *więzy spójności encji*,
2. *więzy spójności referencyjnej*.

Więzy spójności encji ograniczają możliwe wartości, jakie mogą się pojawić w wierszu tabeli. Oto te więzy:

1. Więzy klucza głównego PRIMARY KEY – wartości w określonych kolumnach jednoznacznie identyfikują wiersz tabeli. W kolumnach klucza głównego nie jest dozwolona pseudo-wartość NULL . Automatycznie jest zakładany indeks na kolumnach tworzących klucz główny. Może być określony tylko jeden klucz główny dla jednej tabeli.
2. Więzy klucza jednoznacznego UNIQUE – wartości w określonych kolumnach jednoznacznie identyfikują wiersz tabeli. W kolumnach klucza jednoznacznego jest dozwolona pseudo-wartość NULL . Automatycznie jest zakładany indeks na kolumnach tworzących klucz jednoznaczny. Może być określony więcej niż jeden klucz jednoznaczny dla jednej tabeli.
3. Więzy NOT NULL – w kolumnie nie jest dozwolona pseudo-wartość NULL.
4. Więzy CHECK – warunek, który ma być prawdziwy dla wszystkich wierszy  w tabeli. Nie może zawierać podzapytania ani funkcji zmiennych w czasie, jak *Sysdate* lub *User*. Może zawierać nazwy jednej lub więcej kolumn.

Więzy spójności referencyjnej zapewniają, że zbiór wartości w kolumnach klucza obcego jest zawsze podzbiorem zbioru wartości odpowiadającego mu klucza głównego lub jednoznacznego. Ponieważ wartości klucza głównego lub jednoznacznego jednoznacznie określają obiekty, więc klucz obcy wskazuje zawsze na istniejący obiekt. Wartością klucza obcego może też być *NULL* – wówczas klucz obcy nie wskazuje na żaden obiekt. System zapewnia, aby obiekt wskazywany przez wartość klucza obcego zawsze istniał, niezależnie od wszystkich możliwych operacji na tabelach, w których biorą udział klucze główne, jednoznaczne i obce.

Przy definiowaniu klucza obcego nie można się odwoływać ani do synonimu tabeli, ani do tabeli znajdującej się w innej (odległej) bazie danych. Samo jego zdefiniowanie nie powoduje automatycznie utworzenia indeksu, jak w przypadku klucza głównego i jednoznacznego. Ze względu na znaczenie klucza obcego przy złączaniu tabel projektant bazy danych powinien sam zaplanować założenie indeksu na kluczu obcym przy użyciu instrukcji CREATE INDEX.

Deklaratywne więzy spójności mogą być definiowane na dwa sposoby: albo razem z definicją kolumny *w-linii*, albo poza definicją kolumny *poza-linią*.

Oto składnia *definicji w-linii*:

|  |
| --- |
| 1. [NOT] NULL 2. {UNIQUE|PRIMARY KEY} 3. CHECK (*warunek*) 4. REFERENCES *tabela*[(*kolumna*)][ON DELETE CASCADE] 5. REFERENCES *tabela*[(*kolumna*)][ON DELETE SET NULL] |

Składnia więzów definiowanych *poza-linią* jest rozszerzeniem składni definicji *w-linii*:

|  |
| --- |
| 1. {UNIQUE|PRIMARY KEY} (*kolumna*,…) 2. CHECK (*warunek*) 3. FOREIGN KEY (*kolumna*,…) REFERENCES *tabela*(*kolumna*,…)[ON DELETE [CASCADE|SET NULL]] |

W każdym z powyższych przypadków można poprzedzić więzy napisem:

|  |
| --- |
| CONSTRAINT *nazwa\_więzów* |

nadając w ten sposób im nazwę, za pomocą której można na nich operować, np. włączać je, wyłączać lub usuwać.

Klucz główny złożony z jednej kolumny można utworzyć zarówno przy użyciu więzów definiowanych *w-linii*, jak i więzów definiowanych *poza-linią*. Natomiast klucz główny złożony z więcej niż jednej kolumny może być utworzony tylko przy użyciu więzów definiowanych *poza-linią*. Oto przykład:

|  |
| --- |
| CREATE TABLE Pożyczki(     Num\_kon NUMBER(6,0),     Num\_poż NUMBER(6,0),     Typ\_poż VARCHAR2(8) CONSTRAINT Typ\_poż\_ck        CHECK (Typ\_pożycz IN ('AUTO', 'DOM', 'OSOBISTE')),     Wielk NUMBER(8,0) CONSTRAINT Wielk\_nn NOT NULL,     Data\_poż DATE DEFAULT Sysdate,     Akcept\_przez VARCHAR2(25)        CONSTRAINT Akc\_ko REFERENCES Kierownicy(Nazwisko\_kier),     CONSTRAINT Poż\_kg PRIMARY KEY(Num\_kon, Num\_poż),     CONSTRAINT Konto\_ko FOREIGN KEY(Num\_kon) REFERENCES Klienci(Num\_kon) ); |

W powyższym przykładzie została użyta konstrukcja określania wartości domyślnej dla kolumny

DEFAULT Sysdate

ustalającą datę pożyczki na dzień dzisiejszy, to jest dzień wprowadzania informacji o pożyczce do bazy danych. W specyfikacji wartości domyślnej mogą występować stałe i funkcje SQL także *User*, *Sysdate*. Nie mogą za to występować ani nazwy kolumn, ani funkcje PL/SQL, ani sekwencje.

|  |
| --- |
| CREATE TABLE Pracownicy(    Id\_prac NUMBER(7,0) PRIMARY KEY,    Pesel NUMBER(11,0) UNIQUE NOT NULL,    Imię VARCHAR2(25) NOT NULL,    Nazwisko VARCHAR2(25) NOT NULL,    Szef NUMBER(7,0)         CONSTRAINT Szef\_ko REFERENCES Pracownicy,    Nazwa\_działu VARCHAR2(35),    Miejsce VARCHAR2(35),    Zarobki NUMBER(8,2),    Premia NUMBER(8,2),    CONSTRAINT Dz\_ko FOREIGN KEY (Nazwa\_działu, Miejsce) REFERENCES Działy(Nazwa, Miasto),    CONSTRAINT P\_ck CHECK(0.1\*Zarobki<=Premia AND Premia<=0.5\*Zarobki) ); |

W przypadku warunku występującego w CHECK akceptowane są wiersze, których wartością jest *True* lub *Null*. System nie pozwala, aby w tabeli pojawił się wiersz dający wartość *False* (przy próbie wprowadzenia sygnalizowany jest błąd).

Zwróćmy uwagę na to, że w powyższej instrukcji wystąpił przykład odwołania się klucza obcego (*Szef*) do klucza głównego (*Id\_pracownika*) w tej samej tabeli.

Akcje referencyjne

Przy zwykłej semantyce instrukcji DELETE, jeśli narusza ona warunek istnienia obiektu wskazywanego przez wartość klucza obcego, podnoszony jest błąd i instrukcja nie zostaje wykonana. Natomiast przy zastosowaniu opcji ON DELETE CASCADE, wiersz tabeli nadrzędnej (wskazywanej przez odwołanie klucza obcego) zostaje nie tylko usunięty, ale powoduje jednocześnie usunięcie wszystkich wierszy tabeli podrzędnej (tj. zawierającej klucz obcy), w których wartości klucza obcego wskazują na usuwany wiersz - oczywiście pod warunkiem, że usunięcie tych wierszy jest możliwe bez naruszenia innych więzów referencyjnych – w przeciwnym razie wykonanie instrukcji DELETE zostaje wstrzymane.

Natomiast przy zastosowaniu opcji ON DELETE SET NULL przy usuwaniu wiersza, do którego są odwołania przez wartości kluczy obcych, następuje wstawienie pseudo-wartości *Null* jako wartości klucza obcego.

W Oracle poprzez akcje referencyjne nie są realizowane inne opcje występujące w Standardzie, jak:

|  |
| --- |
| ON DELETE SET DEFAULT |

oznaczająca wstawienie wartości domyślnej jako wartości klucza obcego; jak również analogiczne opcje dla operacji UPDATE:

|  |
| --- |
| ON UPDATE [CASCADE|SET NULL|SET DEFAULT] |

W Oracle można je zaprogramować za pomocą wyzwalaczy bazy danych (wykład 6).

Klucz obcy a NULL

Gdy klucz obcy składa się z kilku kolumn i w jednej z nich jest wartość *Null*, wtedy system akceptuje taką wartość wiersza bez żadnych dodatkowych sprawdzeń. Gdybyśmy chcieli realizować semantykę polegającą na tym, że albo wszystkie wartości klucza obcego są *Null*, albo żadna z nich nie jest, wówczas należałoby zagwarantować to za pomocą odpowiedniego warunku CHECK, np. aby zagwarantować ten warunek dla kolumn *A* i *B* należałoby użyć następującego warunku:

|  |
| --- |
| CONSTRAINT A\_B\_ck CHECK (((A IS NOT NULL) AND (B IS NOT NULL)) OR ((A IS NULL) AND (B IS NULL))) |

Operacje na więzach spójności

Więzy spójności dodaje się tak jak nowe kolumny używając instrukcji ALTER TABLE, np.

ALTER TABLE Emp  
ADD (CONSTRAINT Sal\_ck CHECK (sal >5000));

Używając instrukcji ALTER TABLE można też włączać i wyłączać więzy. Klauzula ENABLE włącza więzy:

ALTER TABLE Emp ENABLE CONSTRAINT Sal\_ck;

Klauzula DISABLE je wyłącza:

ALTER TABLE Emp DISABLE CONSTRAINT Sal\_ck;

Natomiast klauzula DROP usuwa więzy:

ALTER TABLE emp DROP CONSTRAINT Sal\_ck;

Od tego miejsca do końca wykładu 6 skupimy się na konstrukcjach należących do Oracle. Nie będziemy już tego faktu dalej podkreślać.

**4.3** **System SQL\*Plus**

SQL\*Plus jest językiem i jednocześnie interakcyjnym systemem umożliwiającym wprowadzanie i wykonywanie instrukcji języka SQL, skryptów złożonych z instrukcji języka SQL i SQL\*Plus, bloków języka PL/SQL oraz wywołań procedur. Służy do tworzenia bazy danych, wprowadzania do niej zmian i do testowania procedur i części aplikacji. Możliwe jest wręcz układanie prostych aplikacji w postaci skryptów poleceń języka SQL\*Plus związanych z wprowadzaniem danych, wyszukiwaniem danych i drukowaniem raportów – czynności te oczywiście są realizowane dużo lepiej przez specjalistyczne programy Oracle, jak np. Oracle Forms, które jednak są dużo bardziej skomplikowane i wymagają dużo silniejszego sprzętu. Za to, SQL\*Plus jest prosty i może działać praktycznie na każdym komputerze. Każdy programista i administrator systemu Oracle używa go.

W SQL\*Plus można deklarować zmienne dwóch rodzajów: *wiązania* i *podstawienia*, których można używać w instrukcjach SQL i PL/SQL do wzajemnej komunikacji wartości.

Wyliczymy teraz kilka najważniejszych poleceń języka SQL\*Plus, z których korzysta się tworząc bazę danych i aplikację.

* START *nazwa\_pliku* – wykonaj skrypt (w postaci ciągu) poleceń SQL i SQL\*Plus zapisany w pliku o podanej nazwie (zamiast START można używać skrótu @);
* ED *nazwa\_pliku* – wywołaj standardowy edytor np. *notepad* albo *vi*;
* SPOOL *nazwa\_plik*u – zapisuj kolejne polecenia i ich wyniki w pliku o podanej nazwie;
* SPOOL OFF – zamknięcie poprzednio otwartego przez SPOOL pliku;
* SET PAUSE ON – zatrzymywanie przewijania wyświetlania wyników po wyświetleniu zawartości jednego ekranu;
* SET AUTOCOMMIT ON – wykonywanie operacji zatwierdzania COMMIT po każdej zmianie danych (a nie tylko na koniec pomyślnej transakcji);
* SET SERVEROUTPUT ON - wypisywanie wyników realizacji  kodu PL/SQL na ekran (więcej na ten temat w następnym wykładzie);
* SET PAGESIZE *n* – ustal wielkość jednej strony na *n* linii;
* HOST *polecenie* – wykonaj polecenie systemu operacyjnego;
* EXIT – zakończ sesję SQL\*Plus – wyloguj się z Oracle;
* DESCRIBE *nazwa* – wyświetl schemat obiektu (tabeli, perspektywy, procedury, funkcji) o podanej nazwie;
* EXECUTE *nazwa\_procedury*(...) – wykonaj procedurę;
* COLUMN *nazwa* FORMAT A*n* – ustal szerokość wyświetlania kolumny napisowej o podanej nazwie na *n* znaków;
* COLUMN *nazwa* FORMAT 99999 – ustal szerokość wyświetlania kolumny liczbowej o podanej nazwie (9 reprezentuje jedną cyfrę);
* VARIABLE *X* NUMBER (lub CHAR(n) lub VARCHAR2(n), gdzie *n* jest stałą będącą liczbą całkowitą większą od 0) – deklaracja zmiennej wiązania o nazwie *X*;
* EXECUTE :*X*:= *wyrażenie* – przypisz na zmienną wiązania *X* wartość *wyrażenia*. Po ustaleniu wartości, zmienne wiązania poprzedzone dwukropkiem można używać w instrukcjach SQL i PL/SQL.
* PRINT *X* – wypisz wartość zmiennej wiązania *X*;
* ACCEPT *Zmienna* PROMPT 'Podaj wartość: ' – utwórz zmienną  podstawienia o nazwie *Zmienna* i pobierz od użytkownika jej wartość. Następnie zmienne podstawienia poprzedzone znakiem & można używać w instrukcjach SQL i PL/SQL;
* &zmienna, &&zmienna  - wskazują na zmienne podstawienia w instrukcjach w SQL i w SQL\*Plus. SQL\*Plus wstawia wartość za zmienną podstawienia. Jeśli zmienna nie ma wartości zdefiniowanej to SQL\*Plu prosi o wartość za każdym razem gdy napotka zmienną z jednym „&” bądź tylko za pierwszym razem gdy napotka zmienną z dwoma „&&”;
* SET VERIFY OFF - system nie przypomina o wartościach zmiennych podstawienia przy ich użyciu.

Przykłady zastosowania zmiennych wiązania i podstawienia są przedstawione w następnym wykładzie.

**Uwaga**

Wersja programu SQL\*Plus, o nazwie iSQL\*Plus, działa przez Internet na przeglądarce internetowej. Niestety niektóre użyteczne cechy SQL\*Plus nie są obsługiwane w iSQL\*Plus, między innymi polecenia ACCEPT i SPOOL.

**Podsumowanie**

Na tym wykładzie zajmowaliśmy się dwoma tematami związanymi z tworzeniem aplikacji bazodanowej po stronie serwera. Pierwszy temat to *deklaratywne więzy spójności* - definiowane razem z tabelami. Są one sprawdzane przez serwer bazy danych przy każdej operacji na danych. Jeśli wykonanie operacji powodowałoby naruszenie więzów, serwer generuje wyjątek i nie dopuszcza do wykonania operacji.

Drugi przestudiowany temat to informacja o interakcyjnym systemie poleceń SQL\*Plus – nieodzownym przy tworzeniu bazy danych i testowaniu aplikacji korzystających z bazy danych Oracle.

**Słownik pojęć**

[więzy spójności](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#Deklaratywne wi) - definiowane warunki poprawności danych w bazie danych. Ich zadaniem jest zagwarantowanie tego, aby dane w bazie danych wiernie odzwierciedlały świat rzeczywisty, dla którego baza danych została zaprojektowana. System zarządzania bazą danych ma zapewniać, aby więzy spójności pozostawały zawsze prawdziwe. Są dwie podstawowe metody określania więzów spójności na serwerze: albo za pomocą mechanizmu deklaratywnych więzów spójności – w definicjach tabel (instrukcje CREATE TABLE i ALTER TABLE) albo za pomocą mechanizmu wyzwalaczy bazy danych.

[więzy spójności encji](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#wizye) - więzy spójności dotyczące samych instancji encji bez ich związku z powiązanymi encjami.

[PRIMARY KEY](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#PRIMARY KEY) - więzy klucza głównego oznaczające, że wartości w określonych kolumnach jednoznacznie identyfikują wiersz. Implikują, że nie jest dozwolona pseudo-wartość NULL.

[UNIQUE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#UNIQUE) - więzy klucza jednoznacznego oznaczające, że wartości w określonych kolumnach jednoznacznie identyfikują wiersz. Jest dozwolona pseudo-wartość NULL.

[NOT NULL](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#NOT NULL) – więzy oznaczające, że w kolumnie nie jest dozwolona pseudo-wartość NULL.

[CHECK](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#CHECK) - więzy w postaci warunku, który ma być prawdziwy lub *Null* dla wszystkich wierszy  w tabeli.

[więzy spójności referencyjnej](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#Wizyref) - więzy zapewniające, że zbiór wartości w kolumnach klucza obcego jest zawsze podzbiorem zbioru wartości odpowiadającego mu klucza głównego lub jednoznacznego (z dokładnością do NULL).

[akcja referencyjna](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#Akcjer) - specyfikacja, co robić gdy przy operacjach DELETE i UPDATE dojdzie do naruszenia referencyjnych więzów spójności, na przykład CASCADE.

## [SQL\*Plus](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w4.htm#System) - język i jednocześnie interakcyjny system umożliwiający wprowadzanie i wykonywanie instrukcji języka SQL, skryptów złożonych z instrukcji języków SQL i SQL\*Plus, bloków języka PL/SQL oraz wywołań procedur. Na przykład, describe jest to instrukcja SQL\*Plus powodująca wyświetlenie na ekranie schematu obiektu bazy danych takiego jak tabela, perspektywa czy procedura.  Wykład 5

***Programowanie aplikacji baz danych   
po stronie serwera - PL/SQL***

**Streszczenie**

Na kolejnych dwóch wykładach zostanie przedstawione podstawowe narzędzie używane w Oracle przy tworzeniu oprogramowania aplikacji na serwerze, mianowicie rozszerzenie języka SQL o elementy języka programowania – język PL/SQL. Podobny język jest określony w nowym Standardzie SQL:1999. Nosi on tam nazwę SQL/PSM.

W tym wykładzie omówimy bloki PL/SQL, które stanowią podstawowe składniki kodu używanego w procedurach, funkcjach, pakietach i wyzwalaczach (będące tematami następnego wykładu).

**5.1** **Blok PL/SQL**

Język PL/SQL jest proceduralnym rozszerzeniem języka SQL i stanowi podstawę do pisania aplikacji Oracle – jest mianowicie dostępny w różnych programach narzędziowych ORACLE, jak np. SQL\*Plus, Oracle\*Forms czy Oracle\*Reports, prekompilatory, procedury, wyzwalacze, aplikacje internetowe.  
 Odpowiednikiem konstrukcji bloku (i programu) występujących w językach programowania jest w PL/SQL *blok PL/SQL* nazywany też *blokiem* *anonimowym.* Ma on następującą składnię:

|  |
| --- |
| DECLARE    *deklaracje obiektów PL/SQL jak zmienne, stałe, wyjątki, procedury, funkcje* BEGIN    *ciąg instrukcji do wykonania* EXCEPTION    *obsługa wyjątków (błędów)* END; |

Deklaracje i obsługa wyjątków są opcjonalne. Bloki mogą być zagnieżdżone. Jedynymi instrukcjami języka SQL, które mogą się pojawić w bloku PL/SQL, są instrukcje SELECT, INSERT, UPDATE, DELETE, COMMIT i ROLLBACK.

Wprowadzając blok PL/SQL przy użyciu programu SQL\*Plus, należy:

* blok zakończyć kropką znajdującą się na osobnej linii - w celu wpisania go do bufora bieżącej instrukcji;
* bądź znakiem / znajdującym się na osobnej linii - w celu wpisania go do bufora bieżącej instrukcji i wykonania.

Wykonanie instrukcji w buforze bieżącej instrukcji (instrukcji SQL lub bloku PL/SQL) realizujemy poprzez podanie znaku / na osobnej linii. Instrukcje samego SQL\*Plus nie są wpisywane do bufora.

Przykład

W poniższym bloku jest oprogramowana transakcja sprzedaży samochodu marki *Fiat*. Gdy w magazynie jest brak informacji o takiej marce samochodu, podnoszony jest błąd, który jest następnie obsługiwany w sekcji wyjątków poprzez wpisanie odpowiedniej informacji do tablicy dziennika błędów.

DECLARE  
  ilość NUMBER(5);  
BEGIN  
  SELECT m.Stan INTO ilość FROM Magazyn m  
  WHERE m.Produkt = 'Fiat';  
  /\* Gdy w kolumnie *Produkt* tabeli *Magazyn* nie ma wartości 'Fiat' jest podnoszony wyjątek o nazwie *no\_data\_found*. \*/  
  IF ilość> 0 THEN  
    UPDATE Magazyn SET Stan = Stan - 1  
    WHERE Produkt = 'Fiat';  
    INSERT INTO Zakupy  
    VALUES ('Kupiono Fiata', Sysdate);  
  ELSE  
    INSERT INTO Zakupy  
    VALUES ('Brak Fiatów', Sysdate);  
  END IF;  
  COMMIT;  
EXCEPTION -- Początek sekcji wyjątków  
WHEN no\_data\_found THEN  
   INSERT INTO dziennik\_błędów  
   VALUES ('Nie znaleziono produktu FIAT');  
END;  
/

Proszę zwrócić uwagę na zamieszczanie komentarzy w kodzie:

* albo między nawiasami /\* i \*/  albo
* od dwóch kresek -- do końca bieżącej linii.

**Deklaracje zmiennych i stałych**

Deklaracja zmiennej ma następującą postać:

|  |
| --- |
| *identyfikator* *typ\_danych* [NOT NULL] [ := *wyrażenie*]; |

Opcjonalna część := *wyrażenie* umożliwia inicjalizację wartości zmiennej – w przeciwnym razie zmiennej jest przypisywane *NULL*.

Deklaracja stałej ma następującą postać:

|  |
| --- |
| *identyfikator* CONSTANT *typ\_danych* [NOT NULL] [ := *wyrażenie*]; |

Oto przykłady:

zarobki NUMBER(7,2);

pi CONSTANT NUMBER(7,5) := 3.14159;

nazwisko VARCHAR2(25) := 'Kowalski';

data DATE := Sysdate;

żonaty BOOLEAN := False;

liczba\_dzieci BINARY\_INTEGER :=0;

Do inicjalizacji wartości zmiennej zamiast operatora := można użyć klauzuli DEFAULT – używa się jej, gdy zmienna ma charakterystyczną wartość, np.

Data DATE DEFAULT Sysdate;

Zauważmy, że deklaracji zmiennych tego samego typu nie można łączyć razem jak w innych językach!

W PL/SQL są dostępne typy danych z języka SQL, a ponadto kilka innych, z których najważniejsze to typ BOOLEAN wartości logicznych i typ BINARY\_INTEGER liczb całkowitych – niezależny od podtypów typu NUMBER i przez to wymagający przy zapisie mniej pamięci.

Nie należy nadawać tej samej nazwy zmiennej co kolumnie w tabeli. Zmienne i stałe PL/SQL mogą występować w instrukcjach SQL, tak jak stałe SQL i kolumny.

**Wprowadzanie danych z klawiatury i wypisywanie wyników na ekran**

Oprócz zmiennych deklarowanych w bloku PL/SQL mogą występować jeszcze zmienne z aplikacji korzystającej z bloku PL/SQL - poprzedza się je dwukropkiem (:*zmienna*) i nazywa się *zmiennymi wiązania*. Mogą również występować *zmienne podstawienia* SQL\*Plus (&*zmienna*) - ale tylko w wyrażeniach i nigdy po lewej stronie w instrukcji przypisania.

Przy testowaniu i uruchamianiu aplikacji składających się z kodu PL/SQL wygodnie jest pobierać dane do testowania z klawiatury i wypisywać informacje o przebiegu obliczeń na ekran. Realizuje się to w następujący sposób. Po ustawieniu w SQL\*Plus:

SET SERVEROUTPUT ON

w bloku PL/SQL może wystąpić instrukcja:

DBMS\_OUTPUT.Put\_line(*wyrażenie\_napisowe*);

Oto skrypt SQL\*Plus obejmujący wprowadzanie przez użytkownika wartości do bloku PL/SQL - za pomocą zmiennej podstawienia rocz\_zarob, a następnie wypisywanie wyniku na ekran - za pomocą procedury DBMS\_OUTPUT.Put\_line. Należy pamiętać, ze w celu uruchomienia skryptu należy go najpierw zapisać w pliku, a następnie wywołać go w SQL\*Plus przy użyciu instrukcji *start*.

SET SERVEROUTPUT ON  
ACCEPT rocz\_zarob PROMPT 'Podaj roczne zarobki: '  
DECLARE  
mies NUMBER(9,2) := &rocz\_zarob;  
BEGIN  
  mies := mies/12;  
  DBMS\_OUTPUT.PUT\_LINE ('Miesięczne zarobki = ' ||mies);  
END;  
/

Nie możemy użyć tego skryptu w iSQL\*Plus, bo nie obsługuje on instrukcji ACCEPT. Na szczęście opuszczenie instrukci ACCEPT też prowadzi do satysfakcjonującego rezultatu. Dlaczego?

Zamiast zmiennych podstawienia można użyć zmiennych wiązania z SQL\*Plus, np.

ACCEPT rocz\_zarob PROMPT 'Podaj roczne zarobki: '  
VARIABLE mies NUMBER  
BEGIN  
  :mies := &rocz\_zarob/12;  
END;  
/  
PRINT Mies

**Zmienne systemowe**

Jest pewna liczba zmiennych zadeklarowanych w systemie, z których można korzystać w kodzie PL/SQL (ale nie w SQL) – ich wartości dotyczą ostatnio wykonanej instrukcji SQL.

* SQL%ROWCOUNT           liczba wierszy przetworzonych przez ostatnią instrukcję SQL;
* SQL%FOUND=*True*       jeśli został znaleziony (przetworzony) przynajmniej jeden wiersz;
* SQL%NOTFOUND=*True*  jeśli nie znaleziono (przetworzono) żadnego wiersza;
* SQLERRM                  tekstowa informacja o błędzie;
* SQLCODE                  kod błędu.

Obu zmiennych SQLERRM i SQLCODE można używać tylko w sekcji EXCEPTION.

Przykład użycia zmiennej SQL%ROWCOUNT (do obliczenia liczby usuwanych działów o numerze 50) przedstawia następujący blok:

DECLARE usunięte NUMBER;  
BEGIN  
  DELETE FROM Dept WHERE Deptno = 50;  
  usunięte := SQL%ROWCOUNT;  
  INSERT INTO dziennik VALUES ('Dział', usunięte, Sysdate);  
END;  
/

**5.2** **Instrukcja SELECT w PL/SQL**

Wewnątrz bloku PL/SQL instrukcja SELECT nie wypisuje wyników na ekran i w zasadzie nie można jej użyć jako takiej bez dodatkowych konstrukcji.

Instrukcja SELECT ma w PL/SQL swoją specjalną postać. Wynik zapytania nie jest wyświetlany na ekranie, tylko zostaje zapisany na zmiennych - zamieszczanych w klauzuli INTO, która jest wymaganą klauzulą instrukcji SELECT w PL/SQL. Oto składnia instrukcji SELECT w PL/SQL:

|  |
| --- |
| SELECT *wyrażenie*, *wyrażenie*,... INTO *zmienna*, *zmienna*,... FROM *tabela*, *tabela*,... [WHERE ...][GROUP BY ...][HAVING ...][FOR UPDATE OF ...]; |

Aby instrukcja była poprawna, instrukcja SELECT musi zwracać dokładnie jeden wiersz wyników.

Przykład

SELECT e.Ename  
INTO nazwisko  
FROM Emp e  
WHERE e.Empno = 1030;

Wartości, na których operujemy w bloku PL/SQL, pochodzą na ogół z bazy danych, gdzie został określony ich typ danych. W związku z tym, wygodnie jest w bloku PL/SQL określać typ danych przez odniesienie do kolumny w bazie danych jako ''typ danych wymienionej kolumny'' np.

nazwisko Emp.Ename%TYPE

**Rekordy odpowiadające wierszom z tabel**

W PL/SQL jest możliwość użycia zmiennych "wierszowych", na których można zapamiętać cały wiersz pochodzący z tabeli. Dostęp do poszczególnych pól wiersza odbywa się tak, jak w przypadku rekordów, przez kropkę i podanie nazwy kolumny.

DECLARE  
  rek\_osob Emp%ROWTYPE; /\* Typ wierszowy \*/  
BEGIN  
  SELECT \* INTO rek\_osob FROM Emp e WHERE e.Ename = 'SCOTT';  
  rek\_osob.Sal := 1.1\*rek\_osob.Sal;  
  INSERT INTO Dziennik VALUES (rek\_osob.Ename, rek\_osob.Job, rek\_osob.Sal, SYSDATE);  
END;  
/

Zmiennej wierszowej nie można użyć bezpośrednio po słowie kluczowym VALUES w instrukcji INSERT INTO, to znaczy trzeba między nawiasami dokładnie wyspecyfikować wstawiane wartości jedna po drugiej.

**Rekordy PL/SQL**

Rozszerzeniem typu wierszowego jest typ rekordowy – definiowany explicite ze swoimi polami w sekcji deklaracji. Ma on podobne własności i ograniczenia co typ wierszowy. Najpierw należy zdefiniować typ rekordowy, a następnie zadeklarować zmienne tego typu.

Przykład

Zdefiniujmy typ rekordu pracownika.

TYPE Typ\_rek\_prac IS RECORD  
    (numer\_prac NUMBER(4) NOT NULL,  
     nazwisko VARCHAR2(40) NOT NULL,  
     Zarobki NUMBER(8,2),  
     Num\_działu NUMBER(4));  
rekord\_prac Typ\_rek\_prac;

Typy rekordowe mogą być zagnieżdżone. Dostęp do pól rekordu jest za pomocą notacji kropkowej. Na zmienną rekordową można przyporządkowywać wartość innej zmiennej rekordowej, ale tylko tego samego typu rekordowego. Można ich używać w klauzuli INTO, tak jak zmiennych wierszowych:

SELECT \*  
 INTO rekord\_prac  
FROM Pracownicy p  
WHERE p.Id\_prac = 12;

Nie można ani porównywać wartości typu rekordowego ani nie można zmiennej rekordowej użyć bezpośrednio po słowie kluczowym VALUES w instrukcji INSERT.

**5.3** **Obsługa wyjątków**

Ważnym elementem każdej profesjonalnej aplikacji jest obsługa błędów (wyjątków).  
W PL/SQL standardowe błędy (wyjątki) mają przyporządkowane nazwy. Oto najczęściej używane nazwy błędów (wyjątków):

|  |
| --- |
| * ***dup\_val\_on\_index*** - powtórzenie tej samej wartości w indeksie jednoznacznym, * ***no\_data\_found*** - instrukcja SELECT nie zwróciła wartości dla zmiennych w klauzuli INTO, * ***too\_many\_rows*** - instrukcja SELECT zwróciła więcej niż jeden wiersz wartości dla zmiennych w klauzuli INTO, * ***zero\_divide*** - dzielenie przez zero, * ***timeout\_on\_resource*** - zbyt długie oczekiwanie na zasoby, * ***invalid\_cursor*** - niepoprawna operacja na kursorze, * ***login\_denied*** - niepoprawna nazwa użytkownika/hasło, * ***invalid\_number*** - niepoprawna konwersja na liczbę, * ***storage\_error*** - brak pamięci, * ***value\_error*** - błąd związany z działaniem na wartościach, * ***cursor\_already\_open*** - kursor już został otwarty, * ***program\_error*** - błąd w interpreterze PL/SQL. |

Oto przykład, w którym po wykonaniu instrukcji SELECT, występuje przykładowa obsługa możliwych do wystąpienia błędów (wyjątków).

DECLARE  
  nazwisko Emp.Ename%TYPE;  
  stanowisko Emp.Job%TYPE;  
  komunikat VARCHAR2(100);  
BEGIN  
  SELECT e.Ename, e.Job  
  INTO nazwisko, stanowisko  
  FROM Emp e  
  WHERE e.Hiredate BETWEEN '01-JAN-93' AND '01-JAN-94';  
EXCEPTION  
WHEN no\_data\_found THEN  
  INSERT INTO Dziennik   
  VALUES ('0 zatrudnionych w 93');  
  DBMS\_OUTPUT.Put\_line('0 zatrudnionych w 93');  
WHEN too\_many\_rows THEN  
  INSERT INTO Dziennik VALUES (''Więcej niż 1 w 93');  
  DBMS\_OUTPUT.Put\_line('Więcej niż 1 w 93');  
WHEN OTHERS THEN -- Obsługa pozostałych błędów  
  komunikat := 'Błąd nr.= ' || SQLCODE|| ',komunikat= ' || Substr(SQLERRM,1,100);  
   -- SQLCODE i SQLERRM nie mogą wystąpić w instrukcji SQL!  
  INSERT INTO Dziennik VALUES (komunikat);  
  DBMS\_OUTPUT.Put\_line(‘Wystąpił inny błąd ');  
END;  
/

Jeśli blok, w którym wystąpił błąd, zawiera obsługę tego błędu, to po dokonaniu obsługi, sterowanie jest w zwykły sposób przekazywane do bloku go zawierającego (nadrzędnego). Jeśli nie zawiera, następuje propagacja  błędu do zawierających go bloków i albo tam nastąpi jego obsługa, albo błąd przechodzi do środowiska zewnętrznego.

Błąd, który wystąpił w sekcji wykonawczej bloku (między BEGIN i END) jest obsługiwany w sekcji EXCEPTION tego samego bloku. Błędy, które wystąpią w sekcji deklaracji lub w sekcji wyjątków, są od razu propagowane do bloku zawierającego dany blok.

Dobra praktyka programistyczna wymaga, aby każdy błąd został obsłużony – ewentualnie w klauzuli WHEN OTHERS THEN najbardziej zewnętrznego bloku. Aby móc stwierdzić, która instrukcja SQL spowodowała błąd, można używać podbloków z własną obsługą błędów, albo użyć licznika, zwiększającego się o jeden po wykonaniu każdej instrukcji SQL.

**Wyjątki definiowane przez programistę**

Wyjątki można deklarować samemu (w sekcji DECLARE) używając słowa kluczowego EXCEPTION

|  |
| --- |
| *nazwa\_wyjątku* EXCEPTION; |

powodować ich podniesienie (w sekcji instrukcji)

|  |
| --- |
| RAISE *nazwa\_wyjątku*; |

a następnie je obsługiwać (w sekcji EXCEPTION)

|  |
| --- |
| WHEN *nazwa\_wyjątku* THEN … |

W poniższym przykładzie po wykryciu, że w magazynie nie ma samochodu marki 'Fiat', następuje podniesienie wyjątku *Brak\_w\_magazyni*e, a następnie jego obsługa, przez złożenie zamówienia i podniesienie wyjątku *wyjście*, do obsłużenia w bloku zawierającym dany blok – na przykład poprzez wydrukowanie i wysłanie zamówienia na samochody marki 'Fiat' do fabryki.

DECLARE  
  brak\_w\_magazynie EXCEPTION;  
  stan\_magazynu NUMBER(5);  
BEGIN  
  SELECT m.Liczba  
  INTO stan\_magazynu  
  FROM Magazyn m WHERE m.Marka = 'Fiat';  
  IF stan\_magazynu < 1 THEN  
    RAISE brak\_w\_magazynie;  
  END IF;  
EXCEPTION  
WHEN brak\_w\_magazynie THEN  
  INSERT INTO Zamówienia VALUES ('Fiaty');  
  RAISE wyjście;  
END;  
/

Jest też możliwość bezpośredniego podniesienia wyjątku za pomocą procedury *raise\_application\_error* przypisując mu *numer* i *tekst*. Wyjątek taki może zostać obsłużony albo w tym samym bloku, albo w aplikacji zewnętrznej, w której to wywołanie zostanie wykonane.

BEGIN  
.....  
  Raise\_application\_error(-20100,'Błąd');  
EXCEPTION  
WHEN OTHERS THEN  
  IF SQLCODE = -20100 THEN  
     DBMS\_OUTPUT.Put\_line('Błąd przechwycony!');  
  END IF;  
END;  
/

Zakres numerów od –20000 do –20999 jako pierwszy argument procedury *Raise\_application\_error* jest zarezerwowany dla błędów definiowanych przez programistę.

**Obsługa błędów przechwytywanych przez serwer bazy danych**

Rozważmy obsługę błędu naruszenia więzów klucza obcego o numerze -2292.

DECLARE  
  bl\_klucz\_o EXCEPTION;  
  PRAGMA EXCEPTION\_INIT (bl\_klucz\_o, -2292);  
  v\_deptno Dept.Deptno%TYPE := &p\_deptno;  
BEGIN  
  DELETE FROM Dept   
  WHERE Deptno = v\_deptno;  
  COMMIT;  
EXCEPTION  
WHEN bl\_klucz\_o THEN  
  DBMS\_OUTPUT.Put\_line('Nie można usunąć działu ' || TO\_CHAR(v\_deptno) || ', w którym są pracownicy.');  
END;  
/

PRAGMA oznacza dyrektywę wykonywaną przez kompilator. W tym przypadku dyrektywa o nazwie EXCEPTION\_INIT powoduje przypisanie błędowi o numerze –2292 nazwy wyjątku *bl\_klucz\_o*. Do nazwy tej można następnie odwołać się w sekcji obsługi wyjątków w taki sam sposób, jak do każdego innego nazwanego wyjątku.

**5.4** **Instrukcje sterujące**

Język PL/SQL posiada większość typowych instrukcji sterujących występujących w typowych językach programowania:

|  |
| --- |
| IF *warunek* THEN *ciąg\_instrukcji* END IF;  IF *warunek* THEN *ciąg\_instrukcji* ELSE *ciąg\_instrukcji* END IF; |

Ponieważ wartością warunku może być *True*, *False* lub *Null*, więc istotne jest zrozumienie, że instrukcje po THEN są wykonywane wtedy, gdy wartością warunku jest *True*; instrukcje po ELSE są wykonywane wtedy, gdy wartością warunku jest *False* lub *Null*.

Jest jeszcze jedna postać instrukcji IF.

|  |
| --- |
| IF *warunek* THEN *ciąg\_instrukcji* ELSIF *warunek* THEN *ciąg\_instrukcji* END IF; |

Oto możliwe postacie iteracji:

|  |
| --- |
| LOOP    *ciąg instrukcji* (w tym EXIT lub EXIT WHEN *warunek*) END LOOP;  FOR *zmienna* IN *wartość1* .. *wartość2* LOOP    *ciąg instrukcji*  END LOOP;  WHILE *warunek* LOOP    *ciąg instrukcji*   END LOOP; |

Czasami jest konieczne użycie instrukcji pustej – w języku PL/SQL jest to Null – na przykład, w sytuacji gdy obsługa wyjątku jest pusta.

**5.5** **Kursory: dostęp do obszarów roboczych instrukcji SELECT**

Używając dotychczas wprowadzonych konstrukcji języka PL/SQL nie było możliwe przeglądanie kolejno wszystkich wierszy będących wynikiem zapytania. Do tego celu jest potrzebny obiekt PL/SQL o nazwie *kursor*, który stanowi bufor, do którego są zapisywane, kolejno sprowadzane z bazy danych, wiersze wynikowe zapytania.

W sekcji deklaracji definiujemy kursor przyporządkowując mu instrukcję SELECT:

|  |
| --- |
| CURSOR *nazwa\_kursora*  IS *instrukcja\_SELECT*; -- (bez INTO!) |

Następnie otwieramy go za pomocą instrukcji OPEN:

|  |
| --- |
| OPEN *nazwa\_kursora*; |

co oznacza wykonanie instrukcji SELECT przyporządkowanej kursorowi. Po czym możemy w pętli pobierać, przy użyciu instrukcji

|  |
| --- |
| FETCH *nazwa\_kursora* INTO *zmienna*, …; |

kolejny wiersz wyników zapytania i przypisywać go na zmienne PL/SQL w celu przetworzenia zawartych w nim informacji. Standardowo, instrukcja FETCH jest umieszczana w pętli i towarzyszy jej instrukcja

|  |
| --- |
| EXIT WHEN *nazwa\_kursora*%NOTFOUND; |

powodująca wyjście z pętli po sprowadzeniu wszystkich wierszy wynikowych. Na koniec, należy zamknąć kursor za pomocą instrukcji

|  |
| --- |
| CLOSE *nazwa\_kursora*; |

aby zwolnić zasoby systemu przyporządkowane kursorowi.

Przykład

DECLARE  
  zarobki REAL:=0;  
  CURSOR kursor\_osoba IS  
     SELECT \* FROM Emp;  
  rek\_osoby kursor\_osoba%ROWTYPE;  
BEGIN  
  OPEN kursor\_osoba;  
  LOOP  
    FETCH kursor\_osoba INTO rek\_osoby;  
    EXIT WHEN kursor\_osoba%NOTFOUND;  
    zarobki := zarobki + NVL(rek\_osoby.Sal,0);  
  END LOOP;  
  DBMS\_OUTPUT.Put\_line('W sumie zarobki = '||zarobki);  
  CLOSE kursor\_osoba;  
END;  
/

W powyższym przykładzie najpierw deklarujemy kursor przyporządkowując mu instrukcję SELECT. Następnie otwieramy kursor, po czym w pętli wczytując na zmienną wierszową *Rek\_osoby*, rozpatrujemy kolejne wiersze będące wynikiem zapytania, związanego z kursorem. Zwróćmy uwagę na użycie funkcji NVL gwarantującej poprawne sumowanie zarobków w przypadku, gdy zarobki niektórych pracowników nie zostały określone (nieokreślone zarobki interpretujemy jako równe 0) – w takim przypadku bez zastosowania funkcji NVL otrzymalibyśmy wynik *Null* (reprezentowane pustym miejscem przy wyświetlaniu).

Na liście instrukcji SELECT mogą się znajdować dowolne wyrażenia, przy czym jeśli nie jest to nazwa kolumny, musi być użyty alias kolumny, np.

CURSOR kursor IS  
SELECT Ename, Sal+NVL(Comm,0) AS Sumaryczne  
FROM Emp;

Dostępna też jest iteracja (instrukcje OPEN, FETCH i CLOSE są tu niejawne):

|  |
| --- |
| FOR *rek\_osoby* IN *kursor\_osoba* LOOP ... END LOOP; |

W instrukcji SELECT zawartej w kursorze mogą występować zmienne PL/SQL. Ten sam kursor może być otwarty wielokrotnie – z różnymi wartościami zmiennych. Jest też dostępna wersja kursora z parametrami:

|  |
| --- |
| CURSOR *nazwa\_kursora*(*parametr* *typ\_danych*, ....) IS *instrukcja-SELECT* |

Oto przykład użycia kursora z parametrami.

DECLARE  
  CURSOR emp\_cursor(v\_deptno NUMBER, v\_job VARCHAR2) IS  
     SELECT e.Empno, e.Ename  
     FROM Emp e  
     WHERE e.Deptno = v\_deptno AND e.Job = v\_job;  
BEGIN  
  OPEN emp\_cursor(10, 'CLERK');

**Zmienne stanu kursora**

Analogicznie do standardowych zmiennych opisujących stan obliczeń ostatnio wykonanej instrukcji SQL, tak samo każdemu kursorowi są przyporządkowane zmienne, opisujące stan wykonania instrukcji SELECT z nim związanej. Oto te zmienne:

* kursor%FOUND       czy z bazy danych sprowadzono kolejny wiersz,
* kursor%NOTFOUND   czy koniec sprowadzania wierszy,
* kursor%ROWCOUNT   liczba sprowadzonych dotąd wierszy,
* kursor%ISOPEN      czy kursor jest otwarty.

Oto przykład użycia atrybutu %ISOPEN.

IF NOT prac\_kursor%ISOPEN THEN  
  OPEN prac\_kursor;  
END IF;  
LOOP  
   FETCH prac\_kursor...

**Aktualizacja wierszy za pomocą kursora**

Przy wykonywaniu instrukcji SELECT można zakładać blokady na wiersze w celu ich modyfikacji. Klauzula

|  |
| --- |
| FOR UPDATE OF *kolumna*, *kolumna*, … |

powoduje założenie blokady w celu modyfikacji wartości w wierszach; natomiast stowarzyszona z nią w instrukcji UPDATE lub DELETE klauzula

|  |
| --- |
| WHERE CURRENT OF kursor |

umożliwia modyfikację lub usunięcie sprowadzonego przez kursor wiersza odpowiedniej tabeli.

Przykład

Zastosujmy blokowanie wierszy poprzez kursor do zaprogramowania podwyższenia zarobków o 10% najmniej zarabiającym pracownikom oraz zmniejszenia zarobków o 10% najwięcej zarabiającym pracownikom.

DECLARE  
  CURSOR kursor\_osoba IS  
    SELECT e.Ename, e.Sal FROM Emp e  
    FOR UPDATE OF e.Sal;  
  rek\_osoby kursor\_osoba%ROWTYPE;  
BEGIN  
  OPEN kursor\_osoba;  
  LOOP  
    FETCH kursor\_osoba INTO rek\_osoby;  
    EXIT WHEN kursor\_osoba%NOTFOUND;  
    IF rek\_osoby.Sal < 10000 THEN  
      UPDATE Emp SET Sal = Sal \* 1.1  
      WHERE CURRENT OF kursor\_osoba;  
    ELSIF rek\_osoby.Sal > 100000 THEN  
      UPDATE Emp SET Sal = Sal \* 0.9  
      WHERE CURRENT OF kursor\_osoba;  
    END IF;  
    /\* zamiast modyfikować, możemy też usunąć wiersz, np.  
       DELETE Emp WHERE CURRENT OF kursor\_osoba; \*/  
  END LOOP;  
  CLOSE kursor\_osoba;  
  COMMIT;  
END;  
/

**Typy kursorowe i użycie zmiennych kursorowych**

Są one potrzebne, aby usunąć ograniczenia związane z niemożliwością przekazywania kursora jako parametru procedury/funkcji.

DECLARE  
  TYPE Typ\_produktów IS  
  REF CURSOR RETURN Produkty%ROWTYPE; /\* silny typ \*/  
  kursor1 Typ\_produktów;  
  TYPE Typ\_kursorowy IS REF CURSOR; /\* słaby typ \*/  
  kursor2 Typ\_kursorowy;

Można przekazywać jako parametr zmienną kursorową wraz z jej stanem. Wartość zmiennej kursorowej jest traktowana jak wskaźnik. Instrukcję SELECT wiąże się ze zmienną kursorową w instrukcji OPEN. Poza tym, użycie zmiennych kursorowych jest podobne do użycia kursorów (nie można ich tylko używać w pętli kursorowej FOR):

DECLARE  
  TYPE Typ\_kursorowy IS REF CURSOR;  
  z\_kurs\_klienci Typ\_kursorowy;  
BEGIN  
  OPEN z\_kurs\_klienci FOR  
    SELECT k.Nazwisko, k.Imię, k.Telefon FROM Klienci k;  
  WHILE z\_kurs\_klienci%FOUND LOOP ... END LOOP;  
  CLOSE z\_kurs\_klienci;  
END;  
/

**Podsumowanie**

Przedstawiliśmy podstawowe elementy kodu PL/SQL wykonywanego przez serwer bazy danych Oracle. W następnym wykładzie zostanie pokazane jak używać tego kodu w jednostkach programistycznych zapisywanych w bazie danych razem z innymi obiektami.

**Słownik pojęć**

[blok PL/SQL](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w5.htm#Blok anonimowy) (blok anonimowy) – podstawowa jednostka kodu w PL/SQL złożona z trzech sekcji: *deklaracji*, *instrukcji* i *obsługi wyjątków*.

[SELECT INTO](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w5.htm#Instrukcja SELECT) - instrukcja SELECT, w której dodatkowo występuje klauzula INTO wskazująca, na jakich zmiennych ma być zapisany wynik instrukcji SELECT (musi się on składać z dokładnie jednego wiersza).

[obsługa wyjątków](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w5.htm#Obswyj) - jest realizowana w sekcji wyjątków bloku PL/SQL.

[EXCEPTION](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w5.htm#Obswyj) - słowo kluczowe rozpoczynające sekcję wyjątków w PL/SQL.

[kursor](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w5.htm#Kursory) – struktura danych umożliwiająca sprowadzanie z bazy danych kolejnych wierszy wyniku zapytania.

[FOR UPDATE OF](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w5.htm#Aktualizacjak) - klauzula instrukcji SELECT w definicji kursora umożliwiająca aktualizację wierszy w bazie danych poprzez otworzony kursor.

## Wykład 6

***Programowanie aplikacji baz danych   
po stronie serwera - obiekty programistyczne***

**Streszczenie**

W tym wykładzie omówimy obiekty programistyczne przechowywane w bazie danych Oracle: *procedury*, *funkcje*, *pakiety* i *wyzwalacze bazy danych* oraz ich zastosowanie przy programowaniu aplikacji bazy danych po stronie serwera.

**6.1** **Procedury i funkcje**

Procedury, funkcje i pakiety są to obiekty zapisywane w bazie danych tak jak inne obiekty. Jest także możliwość definiowania funkcji i procedur w blokach PL/SQL; ich użycie jest wtedy ograniczone do tego bloku. Raz zapisane w bazie danych procedury i funkcje mogą być używane w każdej aplikacji bazy danych, przez każdego użytkownika (zakładając posiadanie odpowiednich uprawnień tak samo jak w przypadku dowolnych innych obiektów bazy danych).

Typowe sytuacje ich użycia są następujące:

1. zdefiniowanie podstawowych funkcji używanych przez aplikacje, np. utworzenie zamówienia, usunięcie klienta;

2. przechowywanie w bazie danych zadań typu wsadowego, np. tworzenie tygodniowego zestawienia rachunków;

3. zapisanie w bazie danych transakcji, np. zebranie i przetworzenie informacji z odległych baz danych;

4. utworzenie ogólnego, dzielonego kodu przechowywanego na serwerze.

Definicja procedury ma następującą składnię:

|  |
| --- |
| CREATE [OR REPLACE] PROCEDURE *nazwa* (*lista\_parametrów\_formalnych*) {AS|IS} *blok PL/SQL bez słowa kluczowego* DECLARE |

Słowo REPLACE oznacza, że w sytuacji, gdy procedura o tej samej nazwie już istnieje, to nie należy traktować tego jak  błędu, a tylko zastąpić istniejący obiekt przez nowy. Jest to użyteczne przy testowaniu i uruchamianiu procedur, gdy wielokrotnie trzeba zmieniać ich treść.

Są trzy typy przekazywania parametrów:

1. IN - wartość przychodzi z wywołującej jednostki programu i nie ulega zmianie; w procedurze lub funkcji, parametr typu IN nie może wystąpić po lewej stronie instrukcji przypisania,
2. OUT – żadna wartość nie przychodzi z wywołującej jednostki programu; przy zwykłym zakończeniu (bez błędu) wartość parametru jest przekazywana do wywołującej jednostki programu,
3. IN OUT (domyślny) – wartość przychodzi z wywołującej jednostki programu i przy zwykłym zakończeniu (bez błędu) wartość parametru jest przekazywana do wywołującej jednostki programu.

Specyfikując typ danych parametru formalnego nie podaje się jego rozmiaru, a więc dla parametru liczbowego używany jest typ NUMBER, dla typu napisowego zmiennej długości typ VARCHAR2. Można też używać specyfikacji typu danych odwołującego się do typu danych istniejącej kolumny:

|  |
| --- |
| *tabela.kolumna*%TYPE |

Przykład

Wpłatę na konto można zapisać za pomocą następującej procedury:

CREATE OR REPLACE PROCEDURE Credit (acc\_no IN NUMBER, amount IN NUMBER) AS  
BEGIN  
  UPDATE Accounts a SET a.Balance = a.Balance + amount  
  WHERE a.Account\_id = acc\_no;  
END;  
/

W SQL wywołujemy procedurę w następujący sposób:

CALL Credit(10112, 1000);

W kodzie PL/SQL wywołujemy procedurę w zwykły sposób:

Credit(10112, 1000);

W SQL\*Plus poprzedzamy nazwę procedury słowem kluczowym EXECUTE:

EXECUTE Credit(10112, 1000)

Sprawdzenie salda na koncie można zrealizować albo za pomocą procedury korzystając z parametru OUT albo za pomocą funkcji. Oto sposób korzystający z procedury:

CREATE OR REPLACE PROCEDURE Saldo (acc\_no IN NUMBER, bal OUT NUMBER) AS  
BEGIN  
  SELECT a.Balance INTO bal FROM Accounts a  
  WHERE a.Account\_id = acc\_no;  
END;  
/

Przy wywołaniu procedury *Saldo* drugim argumentem powinna być nazwa zmiennej liczbowej, na którą zostanie przypisany aktualny stan konta. Składnia definicji funkcji jest podobna do definicji procedury:

|  |
| --- |
| CREATE [OR REPLACE] FUNCTION *nazwa* (*lista\_parametrów\_formalnych*)  RETURN typ {AS|IS} *blok PL/SQL bez słowa kluczowego DECLARE z instrukcją* RETURN *wyrażenie* |

Przykład

Oto przykład funkcji zwracającej stan konta.

CREATE OR REPLACE FUNCTION Saldo (acc\_no IN NUMBER)  
RETURN NUMBER IS  
  value NUMBER;  
BEGIN  
  SELECT a.Balance INTO value FROM Accounts a  
  WHERE a.Account\_id = acc\_no;  
  RETURN value;  
END;  
/

W kodzie SQL lub PL/SQL używamy funkcji w zwykły sposób, np.

SELECT Saldo(a.Account\_id) FROM Accounts a;

lub

x := Saldo(10112);

Natomiast w SQL\*Plus możemy wywołać i wypisać wartość funkcji w następujący sposób:

VARIABLE p INTEGER  
EXECUTE :p:= Saldo(10112)  
PRINT p

Parametrom procedur i funkcji można przyporządkowywać wartości domyślne. Umieszcza się je wówczas na końcu listy parametrów formalnych.

CREATE OR REPLACE PROCEDURE Wpisz\_studenta(  
          imię IN Studenci.Imię%TYPE,  
          nazwisko IN Studenci.Nazwisko%TYPE,  
          rok IN Studenci.Rok%TYPE DEFAULT 1)  
AS  
  indeks Studenci.Indeks%TYPE;  
BEGIN  
  SELECT NVL(Max(s.Indeks)+1,1) INTO indeks  
  FROM Studenci s;  
  INSERT INTO Studenci  
  VALUES(indeks,imię,nazwisko,rok);  
END;  
/

Gdy dodajemy studenta pierwszego roku, piszemy:

Wpisz\_studenta('Jaś', 'Fasola');

Gdy student z innej uczelni przenosi się do nas od razu na rok 2, piszemy:

Wpisz\_studenta('Jaś', 'Fasola ',2);

Gdy jest więcej parametrów o wartościach domyślnych, np. w procedurze *Wpisz\_studenta* mógłby być jeszcze parametr data DATE DEFAULT Sysdate, wybór parametrów domyślnych, którym w chwili wywołania procedury przypisuje się wartości, dokonuje się przez wskazanie explicite nazwy parametru, np.

Wpisz\_studenta('Jaś', 'Fasola', Data => '1-SEP-99');

Funkcji zapisanych w bazie danych można używać w instrukcjach SQL w taki sam sposób, jak funkcji standardowych – pod warunkiem, że nie zmieniają stanu bazy danych (nie mogą więc zawierać instrukcji INSERT, DELETE i UPDATE) ani nie mają parametrów wyjściowych. Nie powinny też korzystać ze zmiennych nielokalnych zadeklarowanych w pakietach. Wszystkie parametry muszą zostać wyspecyfikowane i nie wolno używać dla nich notacji typu Data => '1-SEP-99'.

Każdy podprogram może zgłosić błąd i przekazać go aplikacji, która go wywołała:

|  |
| --- |
| Raise\_application\_error(*numer\_błędu*, *tekst\_komunikatu*); |

Zakres numerów od –20000 do –20999 jest zarezerwowany dla błędów definiowanych w procedurach i funkcjach.

Informację o parametrach procedury lub funkcji uzyskujemy od systemu za pomocą instrukcji DESCRIBE, np.

DESCRIBE Aktualizuj\_zarobki  
DESCRIBE Saldo

Do oglądania treści procedury zapisanej w bazie danych służy perspektywa słownika danych o nazwie *User\_source*. Najwygodniej jest używać jej w następujący sposób:

COLUMN Line FORMAT 99999  
COLUMN Text FORMAT A80  
SET PAGESIZE 23  
SET PAUSE ON  
SELECT u.Line, u.Text  
FROM User\_Source u  
WHERE u.Name = 'Nazwa\_procedury';

Nazwę procedury (ewentualnie funkcji lub pakietu) należy wpisać dużymi literami! System wypisze tekst procedury podając numery linii, przy których użyciu łatwiej jest wyznaczyć miejsce wystąpienia błędu przy kompilacji procedury (to jest przy wykonywaniu instrukcji CREATE PROCEDURE). Na etapie testowania i usuwania błędów po każdej instrukcji CREATE PROCEDURE należy wykonywać lub ewentualnie umieszczać w skrypcie, jeśli kompilowane procedury znajdują się w skrypcie, następującą instrukcję SQL\*Plus, która, gdy wystąpią błędy, wypisze je na ekran:

SHOW ERRORS

Przy testowaniu i wykrywaniu błędów semantycznych można używać procedury

DBMS\_OUTPUT.Put\_line(*tekst*)

Wyświetlany na ekranie tekst może zawierać w sobie aktualne wartości zmiennych, np.

DBMS\_OUTPUT.Put\_line('Pracownik: '||nazwisko||' Zarobki: ' ||TO\_CHAR(zarobki));

Gdy zmieniają się obiekty, do których odwołuje się procedura lub funkcja, Oracle automatycznie dokonuje ponownej kompilacji, gdy ta procedura lub funkcja zostanie wywołana. Konieczność ponownej kompilacji można odczytać z perspektywy słownika danych *User\_objects*:

SELECT u.Status   
FROM User\_objects u  
WHERE u.Object\_name = 'OBLICZ\_ZAROBKI';

Konieczność ponownej kompilacji wskazuje wartość *Status* = 'INVALID'. Aby samemu spowodować wykonanie ponownej kompilacji, używamy instrukcji:

ALTER PROCEDURE Oblicz\_zarobki COMPILE;

Uprawnienia do użycia procedury (uprawnienie EXECUTE) nadaje się w standardowy sposób, np.

GRANT EXECUTE ON Oblicz\_zarobki TO Moja\_księgowa;

**6.2** **Pakiety**

Ze względu na dużą liczbę procedur i funkcji, jakie zwykle powstają podczas tworzenia aplikacji, konieczne jest pogrupowanie ich w większe jednostki nazywane *pakietami*. W ramach pakietu możemy globalnie dla niego zdefiniować:

* kursory,
* zmienne i stałe (mające trwały charakter),
* wyjątki.

Każda sesja ma swoją własną kopię wewnętrznych struktur danych pakietu (np. zmiennych lokalnych). Zmienne i stałe zachowują swoje wartości na czas trwania sesji.

Pakiet składa się z dwóch części:

* *części publicznej* (dostępnej z zewnątrz), czyli specyfikacji i
* *części prywatnej* (dostępnej tylko z wewnątrz), czyli części implementacyjnej.

Składnia tworzenia specyfikacji pakietu jest następująca:

|  |
| --- |
| CREATE (OR REPLACE) PACKAGE *nazwa\_pakietu* AS        *Deklaracje obiektów publicznych (w przypadku procedur i funkcji specyfikacja nagłówków)* END *nazwa\_pakietu*; |

Składnia tworzenia części implementacyjnej pakietu jest podobna:

|  |
| --- |
| CREATE (OR REPLACE) PACKAGE BODY *nazwa\_pakietu* AS    *Definicje obiektów publicznych i prywatnych* END *nazwa\_pakietu*; |

Można też na koniec części implementacyjnej dołączyć kod inicjacyjny pakietu.

|  |
| --- |
| CREATE (OR REPLACE) PACKAGE BODY *nazwa\_pakietu* AS    *Definicje obiektów publicznych i prywatnych* BEGIN   *Instrukcje inicjujące* END *nazwa\_pakietu*; |

Przykład

Rozważymy problem przyjmowania i zwalniania pracowników. Najpierw specyfikacja pakietu:

CREATE PACKAGE Obsługa\_prac AS  
  PROCEDURE Zatrudnij(  
    numer\_prac NUMBER,  
    nazwisko VARCHAR2,  
    zarobki NUMBER,  
    numer\_działu NUMBER);  
  PROCEDURE Zwolnij(  
    numer\_prac NUMBER);  
  ile\_przyjętych NUMBER;  
  ile\_zwolnionych NUMBER;  
END Obsługa\_prac;  
/

A następnie część implementacyjna tego pakietu:

CREATE PACKAGE BODY Obsługa\_prac AS  
  /\* Procedura zatrudniania pracownika \*/  
  PROCEDURE Zatrudnij(numer\_prac NUMBER, nazwisko VARCHAR2,  
    zarobki NUMBER, numer\_działu NUMBER) IS  
  BEGIN  
    INSERT INTO Pracownicy VALUES (numer\_prac, nazwisko, zarobki, numer\_działu);  
    COMMIT;  
    ile\_przyjętych := ile\_przyjętych+1;  
  END Zatrudnij;  
  /\* Procedura zwalniania pracownika \*/  
  PROCEDURE Zwolnij(numer\_prac NUMBER) IS  
  BEGIN  
    DELETE FROM Pracownicy WHERE Id\_prac = numer\_prac;  
    COMMIT;  
    ile\_zwolnionych := ile\_zwolnionych+1;  
  END Zwolnij;  
  BEGIN  
     ile\_przyjętych := 0;  
     ile\_zwolnionych := 0;  
END Obsługa\_prac;  
/

Z zewnątrz dostęp do obiektów publicznych pakietu odbywa się przez podanie nazwy pakietu, kropki i nazwy obiektu, np.

Obsługa\_prac.Zatrudnij(1000, 'Kowalski ', 2000, 12);

Zdefiniowany w specyfikacji pakietu globalny kursor może być współużywany przez wiele procedur. Rozważmy przykład:

CREATE OR REPLACE PACKAGE Pak IS  
  CURSOR c1 IS SELECT Id\_pracownika  
               FROM Pracownicy  
               ORDER BY Id\_pracownika DESC;  
  PROCEDURE Pierwsze\_3;  
  PROCEDURE Następne\_3;  
END Pak;  
/  
CREATE OR REPLACE PACKAGE BODY Pak IS  
  v\_id NUMBER;  
  PROCEDURE Pierwsze\_3 IS  
  BEGIN  
    OPEN c1;  
    LOOP   
      FETCH c1 INTO v\_id;  
      DBMS\_OUTPUT.PUT\_LINE('Id :' || v\_id);  
      EXIT WHEN c1%ROWCOUNT >= 3;  
    END LOOP;  
  END Pierwsze\_3;  
  PROCEDURE Następne\_3 IS  
  BEGIN  
     LOOP  
        FETCH c1 INTO v\_id;  
        DBMS\_OUTPUT.Put\_line('Id :' || v\_id);  
        EXIT WHEN c1%ROWCOUNT >= 6;  
     END LOOP;  
     CLOSE c1;  
   END Następne\_3;  
END Pak;  
/

W podobny sposób można współdzielić inne obiekty, jak zmienne i rekordy PL/SQL. Zanotujmy, ze nie można przekazywać kursora przez parametr procedury ani nie można współdzielić zmiennych kursorowych. Można natomiast przekazywać zmienne kursorowe poprzez parametry procedur w pakiecie. Oto przykład w postaci skryptu ilustrujący przekazywanie zmiennej kursorowej poprzez parametry procedur w pakiecie.

VARIABLE krs REFCURSOR  
CREATE OR REPLACE PACKAGE Emp\_data AS  
  TYPE Emp\_cur\_typ IS REF CURSOR RETURN emp%ROWTYPE;  
  PROCEDURE Use\_emp\_cv(emp\_cv IN OUT Emp\_cur\_typ);  
  PROCEDURE Write(emp\_cv IN OUT Emp\_cur\_typ);  
END Emp\_data;  
/  
CREATE OR REPLACE PACKAGE BODY Emp\_data AS  
  PROCEDURE Use\_emp\_cv(emp\_cv IN OUT Emp\_cur\_typ)  
  IS  
  BEGIN  
     OPEN emp\_cv FOR SELECT \* FROM emp;  
  END use\_emp\_cv;  
  PROCEDURE Write(emp\_cv IN OUT Emp\_cur\_typ)  
  IS rek emp\_cv%ROWTYPE;  
  BEGIN  
     Use\_emp\_cv(emp\_cv);  
     DBMS\_OUTPUT.Put\_line('Pracownicy firmy:');  
     LOOP  
        FETCH emp\_cv INTO rek;  
        EXIT WHEN emp\_cv%NOTFOUND;  
        Dbms\_output.Put\_line(rek.ename);  
     END LOOP;  
     CLOSE emp\_cv;  
  END write;  
END Emp\_data;  
/  
execute Emp\_data.Write(:krs)

Nazwy funkcji i procedur w pakiecie mogą być przeładowane, to znaczy tej samej nazwy można użyć dla różnych wersji tej samej procedury lub funkcji. Jest to wygodne w sytuacji, gdy różne, specjalne przypadki tej samej procedury lub funkcji możemy zapisać za pomocą zbioru procedur i funkcji używając tej samej nazwy, np. dodawanie z różną liczbą argumentów i różnymi typami danych. Wersje z tą samą nazwą muszą się różnić albo liczbą parametrów, albo nazwą i typem parametru. W przypadku powyższego pakietu można byłoby określić dwie wersje procedury *Zwolnij*:

PROCEDURE Zwolnij(Numer\_prac NUMBER);  
PROCEDURE Zwolnij(Nazwisko VARCHAR2);

Informacje o pakietach można odczytać w podobny sposób jak w przypadku funkcji i procedur posługując się perspektywą słownika danych *User\_source*.

W przypadku funkcji deklarowanych w specyfikacji pakietu, które mają być używane w instrukcjach SQL, należy zamieszczać dyrektywę dla kompilatora określającą brak efektów obocznych w ich działaniu:

|  |
| --- |
| **PRAGMA RESTRICT\_REFERENCES(*nazwa-funkcji*,  *opcja*, ...)**  określa ewentualne współdziałanie funkcji z bazą danych i zmiennymi pakietu. Oto możliwe opcje:  1. **WNDS** - funkcja nie modyfikuje stanu bazy danych;  2. **WNPS** - funkcja nie modyfikuje wartości zmiennych pakietu;  3. **RNDS** - funkcja nie odczytuje zawartości bazy danych;  4. **RNPS** - funkcja nie odczytuje wartości zmiennych pakietu. |

Na przykład dla funkcji obliczającej maksymalne zarobki pracownika w dziale:

FUNCTION Maks(Id\_dz INTEGER) RETURN Real;  
PRAGMA RESTRICT\_REFERENCES (Maks, WNPS, WNDS, RNPS);

Procedury, funkcje i pakiety można usunąć z bazy danych za pomocą instrukcji:

|  |
| --- |
| DROP PROCEDURE *procedura*;  DROP FUNCTION *funkcja*;  DROP PACKAGE *pakiet*; |

Zauważmy, że nie ma instrukcji DROP PACKAGE BODY! Instrukcja DROP PACKAGE usuwa jednocześnie zarówno specyfikację jak i część implementacyjną pakietu.

Oracle dostarcza pewnej liczby standardowych pakietów, takich jak używany już przez nas, DBMS\_OUTPUT. Będzie jeszcze o niektórych z nich mowa za chwilę jak i w następnych wykładach.

Przy wykonywaniu bloku PL/SQL wiązania między identyfikatorami a obiektami bazodanowymi są statyczne – dokonywane w czasie kompilacji. Używając specjalnego standardowego pakietu DBMS\_SQL można dokonywać wiązania w czasie wykonywania bloku. Oto przykład:

CREATE PROCEDURE Usuń\_tab(nazwa\_tabeli VARCHAR2) AS  
  id\_kursora INTEGER;  
  wart INTEGER;  
  polecenie VARCHAR2(250);  
BEGIN  
  polecenie := 'DROP TABLE '|| nazwa\_tabeli;  
  id\_kursora := DBMS\_SQL.Open\_cursor;  
  DBMS\_SQL.Parse(id\_kursora, polecenie, DBMS\_SQL.v7);  
  wart := DBMS\_SQL.Execute(id\_kursora);  
  DBMS\_SQL.Close\_cursor(id\_kursora);  
END Usuń\_tab;  
/

W ten sposób można wykonywać też instrukcje definiujące dane jak CREATE TABLE.

**6.3** **Wyzwalacze bazy danych**

Wyzwalacze bazy danych są procedurami:

* wiązanymi z jednym z obiektów:
  + tabelą,
  + perspektywą,
  + ze schematem (kontem użytkownika),
  + całą bazą danych,
* wywoływanymi (mówimy też *odpalanymi*) przez system przy zajściu odpowiedniego zdarzenia, które może być albo zdarzeniem systemowym albo jedną z instrukcji INSERT, DELETE lub UPDATE skierowaną do tabeli lub perspektywy w bazie danych. Są to więc innego rodzaju zdarzenia, niż te, z którymi mamy do czynienia przy programowaniu aplikacji po stronie klienta, gdzie zdarzenia dotyczą obiektów interfejsu użytkownika: formularzy, raportów i elementów dialogowych.

Wyzwalacze bazy danych służą głównie do programowania więzów spójności i do programowania stałych czynności, które powinny być wykonywane w każdej aplikacji korzystającej z bazy danych.

Najczęściej używane są wyzwalacze tabelowe. Dla każdej tabeli można określić 12 typów wyzwalaczy (może być więcej niż jeden wyzwalacz jednego typu). Typ wyzwalacza zależy przede wszystkim od tego, czy dotyczy operacji na pojedynczym wierszu (*wyzwalacz wierszowy*) czy dotyczy wykonania całej instrukcji. Ponadto, zależy od tego, czy ma być wykonywany przed operacją (typ BEFORE) czy po (typ AFTER). Wreszcie, wyzwalacz może dotyczyć każdej z trzech instrukcji języka operowania danymi: INSERT, DELETE i UPDATE.

Wyzwalacze zostały wprowadzone do Standardu SQL:1999.

**Wyzwalacze tabelowe**

Oto składnia wyzwalacza tabelowego w Oracle:

|  |
| --- |
| CREATE [OR REPLACE] TRIGGER *nazwa\_wyzwalacza*  {BEFORE|AFTER}    *specyfikacja\_instrukcji* ON *tabela* [FOR EACH ROW]  *blok\_PL/SQL* |

gdzie *specyfikacja\_instrukcji* jest ciągiem do trzech nazw instrukcji INSERT, DELETE i UPDATE połączonych spójnikiem OR. W przypadku UPDATE można dodatkowo podać nazwy kolumn, których aktualizacja, ma powodować odpalanie wyzwalacza – używając składni:

|  |
| --- |
| UPDATE OF *kolumna*,…. |

Kolejność odpalania wyzwalaczy jest następująca:

|  |
| --- |
| 1. wyzwalacz przed instrukcją, 2. wyzwalacz przed pierwszym wierszem, na którym operuje instrukcja, 3. wyzwalacz po pierwszym wierszu, na którym operuje instrukcja, 4. … 5. wyzwalacz przed ostatnim wierszem, 6. wyzwalacz po ostatnim wierszu, 7. wyzwalacz po instrukcji. |

Aby odróżnić w wyzwalaczach wierszowych stare i nowe wartości w wierszu tabeli, używamy następujących oznaczeń, odpowiednio na wiersz przed zmianą i na wiersz po zmianie:

1. :OLD – wiersz przed zmianą;

2. :NEW – wiersz po zmianie.

Przykład

IF :NEW.Zarobki < :OLD.Zarobki

Oto podstawowe sytuacje, w których używa się wyzwalaczy.

1. Do sprawdzania więzów spójności używamy wyzwalaczy typu BEFORE. W przypadku wystąpienia błędu, wykonanie odpowiedniej operacji zostaje anulowane.

Przykład

|  |
| --- |
| Utrzymywanie zarobków pracowników w przedziale wartości, w zależności od stanowiska. |

CREATE TRIGGER Sprawdzenie\_zarobków  
BEFORE INSERT OR UPDATE OF Zarobki, Stanowisko  
ON Pracownicy  
FOR EACH ROW  
DECLARE  
  minimum NUMBER;  
  maksimum NUMBER;  
BEGIN  
  /\* Znalezienie zakresu zarobków na danym stanowisku \*/  
  SELECT w.Min\_zar, w.Mak\_zar INTO minimum, maksimum  
  FROM Widełki w  
  WHERE w.Stanowisko = :NEW.Stanowisko;  
  /\* Jeśli zarobki spoza zakresu, generuj błąd \*/  
  IF :NEW.Zarobki < Minimum OR :NEW.Zarobki > Maksimum THEN  
    Raise\_application\_error(-20500,'Zarobki=' ||:NEW.Zarobki|| 'spoza zakresu');  
  END IF;  
END; /\* Koniec wyzwalacza \*/  
/

2. Do realizacji stałych czynności wymaganych po wykonaniu instrukcji (po zmianie wiersza w tabeli) używamy wyzwalacza typu AFTER. W wyzwalaczu mamy do dyspozycji informację o rodzaju instrukcji, która odpaliła dany wyzwalacz. Do tego celu służą systemowe zmienne logiczne: INSERTING, DELETING i UPDATING.

Przykład

|  |
| --- |
| Zajmiemy się aktualizacją budżetu działu. Zmiany, które mają wpływ na budżet działu, są związane ze zwalnianiem i zatrudnianiem pracowników, zmianą ich zatrudnienia w dziale i zmianą ich zarobków – są to wszystko zmiany w tabeli *Pracownicy*. |

CREATE TRIGGER Budżet\_działu  
AFTER DELETE OR INSERT OR UPDATE OF Zarobki, Num\_działu  
ON Pracownicy  
FOR EACH ROW  
BEGIN   
  /\* Zakładamy, że Zarobki, Num\_działu nie są NULL \*/  
  IF DELETING OR (UPDATING AND :OLD.Num\_działu <> :NEW.Num\_działu) THEN  
    UPDATE Budżet  
    SET Fund\_płac = Fund\_płac - :OLD.Zarobki  
    WHERE Id\_działu = :OLD.Num\_działu;  
  END IF;  
  IF INSERTING OR (UPDATING AND :OLD.Num\_działu <> :NEW.Num\_działu) THEN  
    UPDATE Budżet  
    SET Fund\_płac = Fund\_płac + :NEW.Zarobki  
    WHERE Id\_działu = :NEW.Num\_działu;  
  END IF;  
  IF UPDATING AND (:OLD.Num\_działu = :NEW.Num\_działu) AND (:OLD.Zarobki <> :NEW.Zarobki) THEN  
    UPDATE Budżet  
    SET Fund\_płac=Fund\_płac-:OLD.Zarobki+:NEW.Zarobki  
    WHERE Id\_działu = :OLD.Num\_działu;  
  END IF;  
END;  
/

3. Do wykonywania obliczeń, które trzeba zrealizować przed wstawieniem lub aktualizacją nowego wiersza (jak generowanie jednoznacznych identyfikatorów czy wyliczanie wartości dla pewnych pól), używamy wyzwalaczy typu BEFORE.

Przykład

|  |
| --- |
| Generowanie jednoznacznego identyfikatora, wstawianego do jednego z pól przy wykonywaniu instrukcji INSERT. |

CREATE TRIGGER GenerujIdPrac  
BEFORE INSERT ON Pracownicy  
FOR EACH ROW  
BEGIN  
  SELECT NVL(MAX(p.Id\_prac)+1,1)  
  INTO :NEW.Id\_prac  
  FROM Pracownicy p;  
END;  
/

Do wygenerowania kolejnego jednoznacznego identyfikatora pracownika, zamiast NVL(MAX(p.Id\_prac)+1,1) można użyć sekwencji, np. Prac\_seq.nextval. Mianowicie, należałoby zastąpić powyższą instrukcję SELECT instrukcją:

SELECT Prac\_seq.nextval  
INTO :NEW.Id\_prac  
FROM Dual;

Sięganie do wartości w tabeli wyzwalacza może być zabronione w pewnych przypadkach przez konkretny system zarządzania bazą danych!

4. Utrzymywanie reguł odmiennych od przyjętych w deklaratywnych więzach spójności referencyjnych jest realizowane za pomocą wyzwalaczy typu AFTER.

Przykład

|  |
| --- |
| Realizacja więzów spójności referencyjnej typu "ustaw NULL" dla UPDATE i DELETE. |

CREATE TRIGGER Ustaw\_null\_dept  
AFTER DELETE OR UPDATE OF Deptno ON Dept  
FOR EACH ROW  
BEGIN  
  IF UPDATING AND :OLD.Deptno <> :NEW.Deptno OR DELETING THEN  
    UPDATE Emp  
    SET Emp.Deptno = NULL  
    WHERE Emp.Deptno = :OLD.Deptno;  
  END IF;  
END;

Za pomocą wyzwalaczy można także zaprogramować dopasowany do sytuacji system zabezpieczeń przy wykonywaniu instrukcji – niedopuszczania do ich wykonania, gdy pewne warunki, jak np. aktualna godzina czy dzień tygodnia, nie są poprawne dla danej akcji. Można również zaprogramować system kontroli polegający na sporządzaniu śladu, co który użytkownik robił w bazie danych. Można również zaprogramować mechanizm propagowania wprowadzanych zmian do odległych baz danych.

Istnieje kilka naturalnych ograniczeń na to, co może być wykonane w ramach akcji wyzwalacza. Po pierwsze, w  wyzwalaczach nie wolno używać instrukcji COMMIT ani ROLLBACK.

Załóżmy, że mamy do czynienia z tabelą i wyzwalaczem określonym dla niej. Tabela *zmieniana* (ang. mutating) to albo tabela wyzwalacza albo tabela, w której klucz obcy odwołujący się do tabeli wyzwalacza został utworzony z opcją ON DELETE CASCADE lub ON DELETE SET NULL, czyli tabela potencjalnie zmieniana razem z tabelą wyzwalacza.

Reguła

W wyzwalaczu wierszowym nie wolno odczytywać ani zmieniać wartości w tabeli zmienianej z wyjątkiem wstawiania pojedynczego wiersza za pomocą instrukcji

INSERT INTO ... VALUES ...

A więc przy wykonywaniu instrukcji UPDATE na tabeli *Emp*, w wyzwalaczu związanym z tą tabelą nie można sięgać do żadnej wartości zapisanej w tej tabeli, np. aby obliczyć średnie zarobki wśród wszystkich pracowników i aby wpisać je jako nowe zarobki pracownika. Do wartości zapisanych w tabeli zmienianej można sięgać w wyzwalaczu określanym na poziomie instrukcji. Jeśli więc więzy spójności, które ma sprawdzać wyzwalacz, dotyczą całej tabeli, trzeba te operacje wykonywać w wyzwalaczu dla całej instrukcji – ewentualnie zapisując, w wyzwalaczu wierszowym, wymagane wartości do pomocniczych zmiennych, dla każdego zmienianego wiersza. Pomocnicze zmienne muszą być zadeklarowane w pakiecie – jest to jedyna droga przekazywania wartości między wyzwalaczami oprócz wykorzystania pomocniczej tabeli w bazie danych.

**Wyzwalacze typu INSTEAD OF**

Dla perspektywy mamy możliwość zdefiniowania specjalnego rodzaju wyzwalacza

|  |
| --- |
| CREATE [OR REPLACE] TRIGGER *nazwa\_wyzwalacza* INSTEAD OF *specyfikacja\_instrukcji* ON *perspektywa* *blok\_PL/SQL* |

gdzie *specyfikacja\_instrukcji* jest ciągiem do trzech nazw instrukcji INSERT, DELETE i UPDATE połączonych spójnikiem OR.

Wyzwalacz ten jest odpalany zamiast podanej w definicji instrukcji. Daje to możliwość pełnej realizacji postulatu, aby zmiany w bazie danych można było dokonywać z poziomu użytkowego – czyli perspektyw.

Przykład

Rozważmy perspektywę będącą złączeniem trzech tabel.

CREATE VIEW Manager\_info AS  
SELECT e.Ename, e.Empno, d.Dname, d.Deptno, p.Level, p.Projno  
FROM Emp e, Dept d, Project p  
WHERE e.Empno = d.Mgr\_no  
      AND d.Deptno = p.Resp\_dept;

Następujący wyzwalacz pozwala wstawiać wiersze poprzez tę perspektywę, co w efekcie powoduje wstawienie informacji do trzech tabel. Wyzwalacz ten jest odpalany i zastępuje podaną w definicji instrukcję – w tym przypadku INSERT.

CREATE TRIGGER Manager\_info\_insert  
INSTEAD OF INSERT ON Manager\_info  
FOR EACH ROW  
DECLARE  
  p NUMBER;  
BEGIN   
  SELECT Count(\*) INTO p FROM Emp   
  WHERE Emp.empno = :NEW.Empno;  
  IF p=0 THEN  
    INSERT INTO Emp VALUES(:NEW.Empno, :NEW.Ename);  
  ELSE  
    UPDATE Emp SET Emp.Ename = :NEW.Ename  
    WHERE Emp.Empno = :NEW.Empno;  
  END IF;  
  SELECT Count(\*) INTO p FROM Dept   
  WHERE Dept.Deptno = :NEW.Deptno;  
  IF p=0 THEN  
     INSERT INTO Dept VALUES(:NEW.Deptno, :NEW.Dname);  
  ELSE  
…

**Wyzwalacze systemowe**

Wyzwalacze systemowe mają składnię:

|  |
| --- |
| CREATE [OR REPLACE] TRIGGER *nazwa\_wyzwalacza* [BEFORE|AFTER|INSTEAD OF][*zdarzenie\_bazodanowe*|*zdarzenie\_DDL*]  ON [DATABASE|SCHEMA]  *blok\_PL/SQL* |

Przy czym:

1. *zdarzenie\_bazodanowe* to: SERVERERROR, LOGON, LOGOFF, STARTUP lub SHUTDOWN,
2. *zdarzenie\_DDL* to nazwa instrukcji DDL a więc na przykład: CREATE, ALTER, DROP, GRANT, REVOKE.

Zdarzenia można łączyć ze sobą w jednym wyzwalaczu systemowym za pomocą słowa kluczowego OR.

Przykład

|  |
| --- |
| Rozważmy wyzwalacz, który rejestruje w dzienniku wszystkie pojawiające się błędy serwera bazy danych. Dla błędu nr 1017 oznaczającego "niepoprawne logowanie" jest przewidziana specjalna obsługa. |

CREATE TRIGGER Log\_errors AFTER SERVERERROR ON DATABASE   
BEGIN  
  IF (IS\_SERVERERROR (1017)) THEN  
      <*wykonaj specjalną obsługę w tym przypadku*>  
  ELSE  
     <*zapisz w dzienniku informację o błędzie*>  
  END IF;  
END;

Przykład

|  |
| --- |
| Rozważmy wyzwalacz, który odnotowuje fakt wykonywania na koncie użytkownika instrukcji tworzenia nowego obiektu w bazie danych. |

CREATE TRIGGER Audit\_db\_object AFTER CREATE ON SCHEMA  
<*zapisz w dzienniku informację o utworzeniu obiektu*>

Szczególne znaczenie ma wyzwalacz dla zdarzenia logowania się użytkownika. Umożliwia on sprawdzenie, skąd loguje się użytkownik a także umożliwia utworzenie dla danego użytkownika odpowiedniego kontekstu w postaci zbioru wartości atrybutów – do użycia przez aplikację. Oracle wprowadza nawet w tym celu specjalny rodzaj obiektu w bazie danych o nazwie *kontekst*, z którego procedurę wywołuje wyzwalacz przy logowaniu się użytkownika.

Operacje na wyzwalaczach

Wyzwalacz bazy danych może być włączany i wyłączany za pomocą instrukcji:

|  |
| --- |
| ALTER TRIGGER *wyzwalacz* {ENABLE|DISABLE}; |

Wyzwalacz może zostać usunięty za pomocą instrukcji:

|  |
| --- |
| DROP TRIGGER *wyzwalacz*; |

Informacje o wyzwalaczach znajdują się w perspektywie słownika danych *User\_triggers*. Aby je obejrzeć, piszemy (nazwę wyzwalacza dużymi literami):

|  |
| --- |
| SELECT u.Trigger\_type, u.Table\_name, u.Triggering\_event FROM User\_triggers u WHERE u.Trigger\_name = '*WYZWALACZ*'; |

lub

|  |
| --- |
| SELECT u.Trigger\_body FROM User\_triggers u WHERE u.Trigger\_name = '*WYZWALACZ*'; |

**Podsumowanie**

W wykładach 4, 5 i 6 zostały przedstawione podstawowe konstrukcje wchodzące w skład aplikacji po stronie serwera:

1. deklaratywne więzy spójności;

2. język proceduralny będący rozszerzeniem języka SQL;

3. procedury, funkcje, pakiety;

4. wyzwalacze bazy danych.

Z tego tylko punkt 1 należy do Standardu SQL’92. Natomiast pozostałe elementy 2-4 należą do kolejnej wersji Standardu SQL:1999.

**Słownik pojęć**

[procedura](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Procedury i funkcje) – obiekt bazy danych złożony z nagłówka i bloku kodu PL/SQL. Jest możliwe wykonanie tego bloku kodu przez serwer bazy danych korzystając z instrukcji SQL CALL.

[funkcja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Procedury i funkcje) – obiekt bazy danych złożony z nagłówka i bloku PL/SQL. Jest możliwe wykonanie tego bloku kodu przez serwer bazy danych z przekazaniem wartości wywołania w miejsce, gdzie funkcja została użyta.

[pakiet](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Pakiety) – kolekcja *zmiennych*, *stałych*, *kursorów*, *wyjątków*, *funkcji* i *procedur* traktowana jako całość. Pakiet składa się z dwóch części: *specyfikacji* i *implementacji*.

[wyzwalacz bazy danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Wyzwalacze bazy danych) – obiekt bazy danych w postaci nagłówka i bloku PL/SQL, który jest przypisywany określonemu obiektowi i zdarzeniu zachodzącemu dla tego obiektu lub zdarzeniu zachodzącemu w bazie danych (jak np. logowanie się użytkownika). Kod jest wykonywany przy każdym wystąpieniu tego zdarzenia.

[wyzwalacz tabelowy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Wyzwalacze tabelowe) - wyzwalacz wiązany z tabelą i ze zdarzeniem wykonywania instrukcji INSERT, DELETE lub  UPDATE.

[wyzwalacz INSTEAD OF](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Wyzwalacze typu INSTEAD OF) - wyzwalacz wiązany z perspektywą i ze zdarzeniem wykonywania instrukcji INSERT, DELETE lub  UPDATE.

## [wyzwalacz systemowy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w6.htm#Wyzwalacze systemowe) - wyzwalacz wiązany ze zdarzeniem zachodzącym w bazie danych.   Wykład 7

***Obiektowość w obiektowo-relacyjnych bazach danych na przykładzie Oracle***

**Streszczenie**

Współczesne bazy danych posiadają cechy zarówno relacyjne jak i obiektowe. Do tej pory poznaliśmy cechy relacyjne, teraz czas na poznanie cech obiektowych.

Rozważymy dwa rodzaje obiektów, które można przechowywać w obiektowo-relacyjnej bazie danych:

* obiekty będące instancjami typów obiektowych,
* duże obiekty LOB.

**7.1** **Znaczenie obiektowości w bazach danych**

Od dwudziestu już lat trwają prace nad nowymi koncepcjami w bazach danych polegającymi na dołączaniu cech obiektowych do istniejących relacyjnych baz danych. Otrzymano w ten sposób model *obiektowo-relacyjny* i faktycznie na takim modelu jest w tej chwili oparty nowy Standard języka SQL:1999.

Przypomnimy najpierw ogólne znaczenie typów obiektowych. Realizują one zasadę abstrakcji w dwóch postaciach:

* *abstrakcji proceduralnej* polegającej na ukryciu szczegółów złożonych algorytmów poprzez opakowanie ich w procedury i funkcje; gdy w razie potrzeby zmieniamy procedurę - nie musimy modyfikować aplikacji jej używających;
* *abstrakcji danych* polegającej na ukryciu złożoności struktury danych przed użytkownikiem, który korzysta z tych danych; podobnie jak poprzednio, gdy w razie potrzeby zmieniamy strukturę danych – nie musimy modyfikować aplikacji jej używających.

Oto podsumowanie zalet użycia obu rodzajów abstrakcji w bazach danych:

1. Ułatwione modelowanie rzeczywistych obiektów biznesowych.
2. Zmniejszenie złożoności tworzenia aplikacji przez podział zadania na części. Ułatwienie dokonywania zmian. Ukrycie szczegółów implementacyjnych przed użytkownikiem. Modularność aplikacji i możliwość wielokrotnego użycia komponentów w tej samej lub w różnych aplikacjach.
3. Zgrupowanie używanego kodu po stronie serwera wokół obiektów, na których kod działa. Uzyskanie większej kontroli nad kodem.
4. Zastosowanie obiektowo-relacyjnego modelu danych prowadzi do zmniejszenia rozbieżności w modelach danych samej bazy danych i aplikacji bazodanowej napisanej w obiektowym języku programowania. Oba modele można oprzeć o te same pojęcia: *klasy* (*typu obiektoweg*o) i *instancji klasy* (*obiektu*).

**7.2** **Typ obiektowy**

Typ obiektowy, odpowiadający pojęciu klasy w obiektowych językach programowania, jest to złożony typ danych, który hermetyzuje strukturę danych łącznie z metodami potrzebnymi do operowania na strukturze danych. Definiując typ obiektowy określamy atrybuty i metody obiektów np. typ obiektowy zamówienia.

Rys. 7.1 Struktura obiektu

**Struktura typu obiektowego**

Osobno specyfikujemy *publiczny interfejs* obiektów składający się z deklaracji atrybutów i specyfikacji metod oraz *prywatną implementację*  obiektów składającą się z definicji (ciał) metod.

Rys. 7.2 Struktura typu obiektowego

Oto składnia specyfikacji typu obiektowego (w wersji podstawowej):

|  |
| --- |
| CREATE TYPE *nazwa\_typu* AS OBJECT (*atrybut typ\_danych*, ... MEMBER *sygnatura procedury lub funkcji* [, *klauzula* PRAGMA], ... ); |

Oto składnia implementacji typu obiektowego (w wersji podstawowej):

|  |
| --- |
| CREATE TYPE BODY *nazwa\_typu* AS MEMBER *implementacja procedury lub funkcji*; ... END; |

Przykład

Zdefiniujemy dwa typy obiektowe: typ *nazwisk* Name\_typ oraz typ *pracowników* Emp\_typ:

CREATE TYPE Name\_typ  
AS OBJECT(  
   F\_name VARCHAR2(25),  
   L\_name VARCHAR2(25),  
   Initials VARCHAR2(7),  
   MEMBER FUNCTION Full\_name RETURN VARCHAR2,  
   PRAGMA RESTRICT\_REFERENCES(Full\_name, WNDS, RNDS, WNPS, RNPS ));  
/

CREATE TYPE BODY Name\_typ AS  
   MEMBER FUNCTION Full\_name RETURN VARCHAR2 IS  
   BEGIN  
      RETURN (L\_name || ' ' || F\_name );  
   END Full\_name;  
END;  
/

CREATE TYPE Emp\_typ  
AS OBJECT(  
   Emp\_id NUMBER(7),  
   Name Name\_typ,           -- typ obiektowy  
   Street VARCHAR2(25),  
   City VARCHAR2(15),  
   State CHAR(2),  
   Zip INTEGER,  
   MEMBER FUNCTION Get\_name RETURN VARCHAR2,  
   PRAGMA RESTRICT\_REFERENCES(Get\_name, WNDS, RNDS, WNPS, RNPS),  
   MEMBER PROCEDURE Set\_l\_name (v\_name VARCHAR2));  
/

CREATE TYPE BODY Emp\_typ AS  
   MEMBER FUNCTION Get\_name RETURN VARCHAR2 IS  
   BEGIN  
      RETURN (Name.L\_name ||' '|| Name.F\_name);  
   END;  
   MEMBER PROCEDURE Set\_l\_name (v\_name VARCHAR2) IS  
   BEGIN  
     Name.L\_name := v\_name;  
   END;  
END;  
/

**Metody**

Metody to funkcje lub procedury, które tworzymy w definicji typu obiektowego w celu zaimplementowania zachowania się obiektów danego typu. Aplikacja wywołuje metody aby uzyskać w efekcie to zachowanie. Są trzy typy metod: metody *składowe obiektów* (metody typu MEMBER), metody *konstruktorów obiektów* (metody typu CONSTRUCTOR) oraz metody *statyczne*, czyli odnoszące się do całego typu (metody typu STATIC). W tym wykładzie rozważamy tylko metody składowe obiektów oraz metody konstruktora obiektów implementowane przez system – oprócz tego są jeszcze metody konstruktorów definiowane przez użytkowników.

**Dziedziczenie**

Między typami obiektowymi obowiązuje dziedziczenie tak jak w językach programowania - realizowane przez operator UNDER. Na przykład:

CREATE TYPE Person  
AS OBJECT (  
  First VARCHAR2(50),  
  Last VARCHAR2(50)) NOT FINAL;  
/

CREATE TYPE Emp  
UNDER Person (  
  Salary NUMBER) FINAL;  
/

Obiekt typu *Emp* ma wszystkie cechy obiektu typu *Person* a poza tym jeszcze cechy wprowadzone w specyfikacji typu *Emp*.

DECLARE  
  x Emp:=Emp('Jan','Kowalski',10000);  
BEGIN  
  DBMS\_OUTPUT.Put\_line(x.first||' '||x.last||' '||x.salary);  
END;  
/

Wymagana jest specyfikacja NOT FINAL dla typów obiektowych, które mogą być wzorcami dla podtypów podlegając dalszemu dziedziczeniu czyli nie są one końcowe w hierarchii dziedziczenia. Domyślną specyfikacją jest FINAL - typ obiektowy nie podlega dalszemu dziedziczeniu.

**Przesłanianie**

Tak jak w językach programowania jest możliwość zastosowania przesłaniania metod.

CREATE TYPE Figure  
AS OBJECT (  
  NOT INSTANTIABLE MEMBER FUNCTION area RETURN NUMBER  
) NOT INSTANTIABLE NOT FINAL;  
/

CREATE TYPE Rect  
UNDER Figure  
  (x NUMBER,  
   y NUMBER,  
   OVERRIDING MEMBER FUNCTION area RETURN NUMBER) FINAL;  
/

CREATE TYPE BODY Rect AS  
  OVERRIDING MEMBER FUNCTION area RETURN NUMBER IS  
  BEGIN  
     RETURN x\*y;  
  END;  
END;  
/

Definicja metody *area* w podtypie *Rect* przesłania jej specyfikację w nadtypie *Figure.* Wymagane jest słowo kluczowe OVERRIDING. Domyślną specyfikacją metody jest NON OVERRIDING.

Słowo kluczowe NON INSTANTIABLE dla typu obiektowego oznacza, że nie można tworzyć instancji danego typu obiektowego a dla metody -  brak implementacji. Domyślną specyfikacją w obu przypadkach jest INSTANTIABLE.

**Informacje w słowniku danych Oracle**

Informacje o typach obiektowych znajdują się w perspektywie słownika danych o nazwie:

* USER\_OBJECTS

Informacje o metodach typów obiektowych znajdują się odpowiednio w perspektywach słownika danych:

* USER\_METHOD\_PARAMS
* USER\_METHOD\_RESULTS
* USER\_TYPE\_METHODS

**Usuwanie typów obiektowych**

Oprócz instrukcji usuwania typu obiektowego, która usuwa zarówno implementację jak i specyfikację typu obiektowego, jest też instrukcja usuwania samej implementacji typu obiektowego z pozostawieniem jego specyfikacji. W tym momencie do istniejących obiektów tego typu nie można zastosować żadnej z metod!

|  |
| --- |
| DROP TYPE *typ*;  DROP TYPE BODY *typ\_obiektowy*; |

**7.3 Tabela obiektowa**

Zdefiniowanego typu obiektowego można używać do tworzenia tabel obiektowych tego typu.

|  |
| --- |
| CREATE TABLE *nazwa\_tabeli* OF *nazwa\_typu* [(*więzy spójności*)]; |

Zauważmy, że więzów spójności nie określa się w definicji typu obiektowego, tylko w definicji samej tabeli.

W utworzonej tabeli obiektowej są zapisywane obiekty typu obiektowego na tej samej zasadzie co wiersze tabeli relacyjnej. To znaczy, wartości atrybutów tworzą wiersz, na którym działają metody określone w danym typie obiektowym.

CREATE TABLE Name\_table OF Name\_typ;  -- tabela obiektowa

INSERT INTO Name\_table  
VALUES('Jan','Kowalski','JK');

INSERT INTO Name\_table  
VALUES('Anna','Kowalska','AK');

SELECT nt.F\_name, nt.Full\_name()  
FROM Name\_table nt;

**Metoda konstruktora obiektu**

Dla każdego typu obiektowego automatycznie jest tworzona metoda *konstruktora obiektu* tego typu. Ma ona taką samą nazwę jak typ obiektowy oraz ma argumenty takich typów jak atrybuty typu obiektowego. Rozważmy typ obiektowy:

CREATE TYPE Tv\_type  
AS OBJECT (  
  Tv\_category VARCHAR2(20),  
  Screen\_size NUMBER(4));  
/

Oto przykłady użycia jego konstruktora w bloku PL/SQL:

DECLARE  
  v\_new\_tv tv\_type := Tv\_type('WEB tv', 32);  
  v\_alt\_tv Tv\_type;  
BEGIN  
  v\_alt\_tv := Tv\_type('Big Screen', 72);  
END;  
/

oraz w instrukcji INSERT do tabeli obiektowej:

CREATE TABLE Tv OF Tv\_type;

INSERT INTO Tv VALUES(Tv\_type('Color tv', '28'));

**Wyszukiwanie (SELECT) w tabeli obiektowej**

Wynikiem instrukcji:

SELECT \*  
FROM Name\_table p  
WHERE p.L\_name LIKE 'K%';

jest zbiór wierszy:

F\_NAME L\_NAME INITIAL

-------------- ------------- -------

Jan Kowalski JK

Anna Kowalska AK

Wynikiem instrukcji:

SELECT VALUE(p) AS Osoba  
FROM Name\_table p  
WHERE p.L\_name LIKE 'K%';

jest zbiór obiektów reprezentowanych przy użyciu konstruktora typu obiektowego (VALUE(*p*) - oznacza obiekt wskazywany przez *p*):

OSOBA(F\_NAME, L\_NAME, INITIALS)

-----------------------------------

NAME\_TYP('Jan', 'Kowalski', 'JK')

NAME\_TYP('Anna', 'Kowalska', 'AK')

**Wstawianie (INSERT) do tabeli obiektowej**

Wstawiamy albo bezpośrednio tak jak do tabeli relacyjnej:

INSERT INTO Name\_table  
VALUES('Jan','Kowalski','JK');

albo z użyciem konstruktora:

INSERT INTO Name\_table  
VALUES(Name\_typ('Jan','Kowalski','JK'));

**Modyfikacja (UPDATE) na tabeli obiektowej**

Modyfikujemy albo bezpośrednio, tak jak w tabeli relacyjnej:

UPDATE Name\_table p  
SET p.L\_name = 'Janusz'  
WHERE p.L\_name = 'Jan';

albo z użyciem konstruktora:

UPDATE Name\_table p  
SET p = Name\_typ('Janusz','Kowalski','JK')  
WHERE p.L\_name = 'Jan';

**Usuwanie obiektów (DELETE) z tabeli obiektowej**

Obiekty z tabeli obiektowej usuwamy tak jak wiersze z tabeli relacyjnej:

DELETE FROM Name\_table p  
WHERE p.L\_name = 'Janusz';

**Tabela relacyjna w obiektowo-relacyjnej bazie danych**

Oprócz nowego rodzaju tabel obiektowych możemy także używać tabel relacyjnych z kolumnami typów obiektowych. Wartościami wpisywanymi do kolumn tabeli mogą więc być instancje typów obiektowych. Na przykład:

CREATE TABLE Emp1(  -- tabela relacyjna  
   Empno NUMBER PRIMARY KEY,  
   Name Name\_typ,   -- kolumna typu obiektowego  
   Sal NUMBER,  
   Kier REFERENCES Emp1);

**Typ referencyjny**

Dla każdego typu obiektowego *Type* jest automatycznie definiowany jego typ referencyjny oznaczany przez *REF Type*. Typu tego można używać jako typu atrybutów obiektów bądź kolumn w tabeli relacyjnej. Daje to możliwość wiązania wartości atrybutu bądź wartości w kolumnie z innym obiektem podobnie jak klucz obcy odwołuje się do klucza głównego. Jest to alternatywny sposób tworzenia powiązań.

CREATE TYPE Dept\_Type  
AS OBJECT(  
   Name VARCHAR2(10),  
   Loc VARCHAR2(50));

CREATE TYPE Emp\_Type  
AS OBJECT(  
   Name VARCHAR2(20),  
   Sal NUMBER,  
   Dept\_ref REF Dept\_Type);

CREATE TABLE Obj\_Dept OF Dept\_Type;

CREATE TABLE Obj\_Emp OF Emp\_Type -- tabela obiektowa  
(Dept\_ref SCOPE IS Obj\_Dept);

CREATE TABLE Rel\_emp(            -- tabela relacyjna  
  Id NUMBER PRIMARY KEY,  
  Name VARCHAR2(20),  
  Sal Number,  
  Dept\_ref REF Dept\_Type SCOPE IS Obj\_Dept);

Ze względu na możliwość istnienia wielu tabel obiektowych tego samego typu, definiując typ referencyjny trzeba wskazać do obiektów której tabeli odwołuje się on. Służy do tego klauzula SCOPE IS – proszę zwrócić uwagę na jej użycie w powyższych przykładach.

W instrukcjach SQL można bezpośrednio nawigować przez referencje w taki sam sposób jak to ma miejsce dla obiektów. Gdybyśmy chcieli dla każdego pracownika wypisać nazwę jego departamentu, użylibyśmy następującej instrukcji SELECT:

SELECT e.Name, e.Dept\_ref.Name  
FROM Obj\_emp e;

Dla tabeli relacyjnej instrukcja SELECT jest taka sama.

SELECT e.Name, e.Dept\_ref.Name  
FROM Rel\_emp e;

Natomiast w kodzie PL/SQL nawigacja przez referencje jest niedozwolona i wymaga użycia specjalnej funkcji SQL:

|  |
| --- |
| DEREF(*p*) - obiekt na który wskazuje referencja *p* |

Jest też funkcja odwrotna:

|  |
| --- |
| REF(*o*) - referencja do obiektu o identyfikatorze *o* |

Reasumując, w obiektowo-relacyjnej bazie danych pojawiają się dwa nowe rodzaje wartości związane z zapisem danych na dysku: referencje do obiektów oraz jednoznaczne identyfikatory obiektów. Każda implementacja obiektowo-relacyjnej bazy danych może używać innych zbiorów tych wartości, co powoduje, że obiektowo-relacyjna baza danych przestaje być przenaszalna (jako zbiór tabel z atomowymi, skalarnymi wartościami) między różnymi systemami. Tracimy więc wielką zaletę relacyjnych baz danych.

**Kolekcje**

Atrybut typu obiektowego lub kolumna tabeli nie koniecznie musi być typu prostego ale może być typu zbiorowego (kolekcji). Są dwa rodzaje kolekcji:

1. VARRAY – typ tablicy jednowymiarowej (jak wektor);
2. TABLE – typ tabeli zagnieżdżonej. A więc wartością atrybutu może być cała tabela! Tego typu nie będziemy rozważać na wykładzie.

Oto przykład zastosowania typu VARRAY. Najpierw definicja typu obiektowego o nazwie *Projekt*.

CREATE TYPE Projekt AS OBJECT (

Num\_proj NUMBER(3),

Tytuł VARCHAR2(35),

Koszt NUMBER(7,2));

Teraz zdefiniujemy typ kolekcji o nazwie *Lista* – mianowicie tablicę VARRAY projektów. Jako argument VARRAY podajemy górne ograniczenie liczby projektów - w naszym przykładzie będzie to liczba 3.

CREATE TYPE Lista AS

VARRAY(3) OF Projekt;

Oto definicja tabeli, w której jeden z atrybutów o nazwie *Projekty* jest typu tablicowego *Lista*.

CREATE TABLE Wydziały (

Id\_wydz NUMBER(2),

Nazwa VARCHAR2(15),

Budżet NUMBER(11,2),

Projekty Lista);

Do inicjacji wartości kolekcji służy funkcja konstruktora o tej samej nazwie co typ kolekcji. A więc do tabeli *Wydziały* wstawiamy wartości atrybutów kolekcji w następujący sposób – należy tylko pamiętać aby zgodnie ze specyfikacją tablica *Projekty* nie miała więcej niż trzy elementy (ale może mieć mniej).

INSERT INTO Wydziały

VALUES(1,'Informatyka',100000, Lista(Projekt(1,'Analiza',123), Projekt(2,'Projekt',456)));

Również w takiej postaci system wypisuje wartości kolekcji przy wykonywaniu instrukcji SELECT. Wynik instrukcji

SELECT w.Nazwa, w.Projekty

FROM Wydziały w;

jest wypisywany w postaci:

NAZWA PROJEKTY(NUM\_PROJ, TYTUŁ, KOSZT)

----------- --------------------------------------------------------

Informatyka LISTA(PROJEKT(1,'Analiza',123),PROJEKT(2,'Projekt',456))

Natomiast tylko w kodzie PL/SQL do poszczególnych elementów kolekcji można dostawać się używając naturalnej dla tablic składni:

|  |
| --- |
| *Nazwa\_kolekcji*(*wskaźnik*) |

Przykład

Przy reprezentowaniu departamentów i pracowników za pomocą dwóch tabel obiektowych *Obj\_Dept* i *Obj\_Emp* odpowiednio typów *Dept\_Type* i *Emp\_Type*, łatwo jest wyznaczyć dla pracownika jego departament (poprzez atrybut Dept\_ref). Natomiast wobec braku klucza obcego i indeksu na nim, trudno wyznaczyć dla danego departamentu wszystkich pracowników zatrudnionych w tym departamencie. Rozwiązaniem tego problemu jest zastosowanie kolekcji. Definiujemy typ kolekcji

CREATE TYPE Pracownicy AS VARRAY(100) OF REF Emp\_Type;

i do typu *Dept\_Type* dodajemy atrybut Kadra typu tej kolekcji. Oto pełne rozwiązanie.

CREATE TYPE Dept\_Type  
AS OBJECT(  
   Name VARCHAR2(10),  
   Loc VARCHAR2(50),  
   Kadra Pracownicy);        --  pracownicy w departamencie  
/

CREATE TYPE Emp\_Type  
AS OBJECT(  
   Name VARCHAR2(20),  
   Sal NUMBER,  
   Dept\_ref REF Dept\_Type);  --  departament pracownika  
/

CREATE TABLE Obj\_Dept OF Dept\_Type;

CREATE TABLE Obj\_Emp OF Emp\_Type  
(Dept\_ref SCOPE IS Obj\_Dept);

Przykład

Zaprezentujemy przetwarzanie wartości w kolekcji przy użyciu kodu PL/SQL. Wypiszemy wydziały razem z projektami realizowanymi w tych wydziałach. Użyjemy metody *Count* zdefiniowanej dla kolekcji. Wartością Nazwa\_kolekcji.Count jest liczba elementów w kolekcji. Metoda ta pozwala przejrzeć wszystkie elementy kolekcji.

CREATE OR REPLACE PROCEDURE Wypisz AS  
   lis Lista;  
BEGIN  
   FOR z IN (SELECT \* FROM Wydzialy) LOOP  
      DBMS\_OUTPUT.Put\_line('Projekty Wydzialu: '||z.Nazwa);  
      lis:=z.Projekty;  
      FOR i IN 1..lis.Count LOOP  
        DBMS\_OUTPUT.Put\_line(lis(i).Num\_proj||' '||lis(i).Tytul);  
      END LOOP;  
   END LOOP;  
END;

Realizacja polecenia:

    Exec Wypisz

spowoduje wyświetlenie (należy pamiętać o włączeniu opcji SET SERVEROUTPUT ON):

Projekty Wydzialu: Informatyka  
1 Analiza  
2 Projekt

**7.4** **Duże obiekty LOB**

Język SQL i jego rozszerzenie PL/SQL w dotychczas omawianej postaci umożliwiają dostęp do standardowych typów danych zapisanych w bazie danych i umożliwiają wykonywanie na nich operacji. Są jednak sytuacje, gdy może się to okazać niewystarczające.

Z taką sytuacją mamy do czynienia w przypadku danych, których format wykracza poza format standardowych typów danych. Przykładem takich danych są duże obiekty, jak rysunki (grafika), fotografie, zdjęcia satelitarne i rentgenowskie (zdjęcia binarne), video, animacje, wywiady (audio/video), muzyka, odgłosy dźwiękowe (sound waveforms), skrypty, dokumenty tekstowe, książki. Wprowadzono więc, zarówno do Standardu SQL:1999, jak i do Oracle nowe typy danych dla dużych obiektów tzw. *LOB* (ang. *large objects*). Oprócz samych dużych obiektów wspomnimy również krótko o narzędziach do operowania na nich dostarczonych w standardowym pakiecie Oracle o nazwie DBMS\_LOB.

**Operowanie dużymi obiektami w Oracle (pakiet DBMS\_LOB)**

W Oracle są cztery rodzaje dużych obiektów *LOB*:

|  |
| --- |
| 1. BLOB – binarny duży obiekt - strumień bitów jak w przypadku LONG RAW. 2. CLOB – znakowy duży obiekt - strumień znaków (pojedynczych bajtów). 3. NCLOB – uogólniony (dla języków narodowych wielobajtowych) znakowy duży obiekt. 4. BFILE – plik binarny przechowywany poza bazą danych. |

Rys. 7.3 Rodzaje dużych obiektów

Duże obiekty są przechowywane albo *wewnętrznie* - wewnątrz bazy danych – dotyczy to wartości typów CLOB, NCLOB, BLOB, albo *zewnętrznie* w plikach systemu operacyjnego – dotyczy to wartości typu BFILE, które są dostępne z serwera Oracle tylko w trybie odczytu. Serwer Oracle nie potrafi automatycznie dokonywać konwersji danych między różnymi typami danych, np. z wartości typu CLOB do wartości typu BLOB.

Z każdym dużym obiektem *LOB* jest związana:

1. *wartość LOB*: przechowywana w bazie danych zawartość dużego obiektu,
2. *lokalizator LOB*: wskaźnik do wartości *LOB* przechowywanej w bazie danych.

Zasada jest taka, że w kolumnie typu *LOB* jest zapisywany lokalizator *LOB* – nie cała jego zawartość. Interfejsy programistyczne operujące wartościami *LOB* używają tych lokalizatorów w podobny sposób, jak używa się uchwytów do plików w systemie operacyjnym. Przypisując lokalizator do nowej zmiennej lub nowego wiersza następuje skopiowanie wartości dużego obiektu i wygenerowanie nowego lokalizatora. W przypadku zewnętrznych wartości *LOB* nie są one kopiowane (a więc wskaźnik do tego samego pliku BFILE może występować w różnych wierszach w tabeli).

Poniższy rysunek ilustruje fakt, że wartością atrybutu, kolumny lub zmiennej jest *lokalizator* (wskaźnik) do dużego obiektu *LOB*, który jest zapisany albo w samej bazie danych albo poza nią (w przypadku wartości *BFILE*).

Rys. 7.4 Reprezentacja dużego obiektu

Przykład

Rozważmy tabelę pracowników, której kolumny są typów LOB. Dla każdego pracownika zamieszczamy jego życiorys (Resume), zdjęcie (Picture) oraz film video (Video).

CREATE TABLE Employee(  
   Empno NUMBER,  
   Ename VARCHAR2(35),  
   Resume CLOB,  
   Picture BLOB,  
   Video BFILE);

**Zewnętrzne obiekty LOB typu BFILE**

Zewnętrzne obiekty LOB nie są zapisywane w bazie danych lecz każdy w osobnym pliku systemu operacyjnego.

W systemie Oracle został wprowadzony nowy rodzaj obiektu bazodanowego o nazwie *katalog* związany z typem danych BFILE. Mianowicie, *katalog* jest to obiekt bazodanowy reprezentujący katalog systemu operacyjnego. Służy do administrowania dostępem do obiektów bazy danych typu BFILE przechowywanych w plikach poza bazą danych. Fizyczny katalog jest tworzony pod systemem operacyjnym z uprawnieniami odczytu dla procesów Oracle. Pliki w tym katalogu nie mogą być ani zmieniane ani usuwane przez system Oracle.

Na przykład, instrukcja:

CREATE DIRECTORY Videos AS '/oracle/lob/';

tworzy obiekt bazodanowy reprezentujący katalog /oracle/lob

Rys. 7.5 Obiekt bazy danych typu katalog

Zilustrujemy użycie obiektów LOB w instrukcjach SQL na kilku przykładach.

**Wstawianie obiektów LOB**

INSERT INTO Employee VALUES  
(7897,'Jan Kowalski','Znakomity aktor', NULL, NULL);

INSERT INTO Employee VALUES  
(7898,'Piotr Jankowski', Empty\_CLOB(), Empty\_BLOB(), Bfilename('VIDEOS','J.IMG'));

UPDATE Employee e  
  SET e.Resume =(SELECT f.Resume FROM Employee1 f  
                 WHERE f.Ename='Kowalski')  
WHERE e.Empno = 4508;

W powyższym przykładzie NULL oznacza brak lokalizatora obiektu LOB, Empty\_BLOB() i Empty\_CLOB() oznaczają lokalizator do pustego obiektu. Stałe tekstowe zostają automatycznie przekształcone do typu CLOB. Natomiast, funkcja Bfilename('VIDEOS','J.IMG')) zwraca lokalizator dużego obiektu przechowywanego w pliku J.IMG.

**Dopisywanie do obiektów LOB w PL/SQL**

Na zawartościach obiektów BLOB i CLOB można operować za pomocą procedur i funkcji specjalnego pakietu DBMS\_LOB w podobny sposób jak to się robi na zawartościach plików binarnych.

DECLARE  
  lobloc CLOB; -- lokalizator LOB  
  text VARCHAR2(2000);  
  amount NUMBER;  
  offset INTEGER;  
BEGIN  
  text := 'tekst do wpisania do LOB';  
  SELECT e.Resume INTO lobloc  
    FROM Employee e WHERE e.Empno = 5887  
  FOR UPDATE; -- konieczna blokada wiersza  
  offset := DBMS\_LOB.Getlength(lobloc) + 1;  
  amount := Length(text);  
  DBMS\_LOB.Write(lobloc, amount, offset, text);  
  COMMIT;  
  DBMS\_OUTPUT.Put\_line('Wpisano ' || To\_char(amount) ||' znaków');  
END;  
/

W szczególności, procedura DBMS\_LOB.Write(lobloc, amount, offset, text) dopisuje do istniejącej zawartości obiektu o lokalizatorze *lobloc* zawartość *text* o długości *amount* począwszy od miejsca *offset*. Oto jej sygnatura:

PROCEDURE Write(

Lobdst IN OUT BLOB|CLOB, -- dwie możliwości

Amount IN OUT BINARY\_INTEGER,

Offset IN INTEGER := 1,

Buffer IN RAW|VARCHAR2) -- RAW dla BLOB, VARCHAR2 dla CLOB

Jak widać, nie jest przewidziana możliwość wpisywania danych do obiektu typu BFILE!

Oto sygnatura funkcji *Read* odczytującej określoną przez *amount* ilość danych dużego obiektu BLOB, CLOB lub BFILE od jego pozycji *offset* i zapisującą je do bufora *buffer*:

PROCEDURE Read(

Lobsrc IN BFILE|BLOB|CLOB , -- trzy możliwości!

Amount IN OUT BINARY\_INTEGER,

Offset IN INTEGER,

Buffer OUT RAW|VARCHAR2)

-- RAW dla BLOB i BFILE, VARCHAR2 dla CLOB

Jest też możliwość załadowania zawartości obiektu BFILE (czyli pliku binarnego) do bazy danych w postaci obiektu BLOB lub CLOB. Służy do tego procedura *Loadfromfile* o następującej sygnaturze:

PROCEDURE Loadfromfile(

Lobdst IN OUT BLOB|CLOB,

Src\_file IN BFILE,

Amount IN INTEGER,

Dest\_offset IN INTEGER := 1,

Src\_offset IN INTEGER := 1)

Oto przykład użycia procedury *Loadfromfile* dowstawienia zdjęcia osoby dostępnego jako obiekt typu BFILE - do tabeli *Emp* w postaci obiektu wewnętrznego typu BLOB:

CREATE TABLE Emp(

Empno INTEGER(5);

Ename VARCHAR2(50);

Zdjecie BLOB);

DECLARE

b BLOB;

f BFILE;

BEGIN

INSERT INTO Emp VALUES(1, 'Kowalski', Empty\_BLOB());

SELECT Zdjecie INTO b FROM Emp WHERE Empno=1 FOR UPDATE;

f:=Bfilename('KATALOG','Kowalski.gif');

DBMS\_LOB.Loadfromfile(b,f,DBMS\_LOB.Lobmaxsize);

END;

/

**Usuwanie obiektów LOB**

Nie ma bezpośredniej operacji usuwania obiektu LOB. W celu usunięcia go, można albo usunąć cały wiersz z obiektem, albo zastąpić go w wierszu innym. Najpierw, usunięcie wiersza z obiektem LOB:

DELETE FROM Employee e  
WHERE e.Ename = 'Jan Kowalski';

Z kolei, usunięcie obiektu LOB w wierszu:

UPDATE Employee e SET e.Resume = Empty\_CLOB()  
  WHERE e.Ename = 'Jan Kowalski';

**Podsumowanie**

Zapoznaliśmy się ze sposobem przechowywania w bazie danych i posługiwania się w języku SQL:

* obiektami typów obiektowych definiowanych przez użytkownika oraz
* dużymi obiektami tekstowymi i binarnymi.

Wykład stanowi tylko wprowadzenie do pełnego opanowania przez czytelnika posługiwania się obiektami. Zachęcamy czytelnika do wykonania podanych poniżej zadań domowych oraz przestudiowania dokumentacji Oracle: *Application Developer's Guide - Object-Relational Features* i *Application Developer's Guide - Large Objects (LOBs).*

Rodzi się naturalne pytanie. Czy lepiej konstruować relacyjne czy relacyjno-obiektowe bazy danych?

* Po pierwsze, relacyjne bazy danych są łatwiejsze w użyciu i bardziej efektywne w działaniu bo są prostsze oraz bardziej pasują do tabelkowego interfejsu większości aplikacji biurowych. Po drugie, relacyjne bazy danych są łatwiej przenaszalne jak zbiory tabel wartości – mają  reprezentację niezależną od bazy danych.
* Z kolei obiektowo-relacyjne (i czysto obiektowe) bazy danych dają możliwość bardziej naturalnego modelowania obiektów aplikacji tak, że przypominają one rzeczywiste obiekty jak *osoby*, *faktury*, *towary*. Po drugie, zmniejszają one dystans między reprezentacją obiektów w bazie danych a reprezentacją obiektów w językach obiektowych, które są używane do pisania bardziej skomplikowanych aplikacji – ułatwia to posługiwanie się obiektami zarówno na poziomie koncepcyjnym jak i implementacyjnym.

Skoro więc trapią nas wątpliwości, który rodzaj baz danych wybrać, może wybierzemy trzeci rodzaj baz danych, które właśnie na naszych oczach powstają – tzw. *bazy XMLowe*. Wprowadzenie do nich jest tematem osobnego wykładu.

**Słownik pojęć**

[typ obiektowy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w7.htm#Typ obiektowy) - definiowany przez użytkownika typ danych, który hermetyzuje strukturę danych łącznie z metodami potrzebnymi do operowania na strukturze danych. Można stosować dziedziczenie własności między typami obiektowymi i ich przesłanianie (powtórne zdefiniowanie w podtypie).

[tabela obiektowa](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w7.htm#Tworzenieto) - tabela, w której są zapisywane obiekty typu obiektowego na tej samej zasadzie co wiersze tabeli relacyjnej. To znaczy, wartości atrybutów tworzą wiersz, na którym działają metody określone w typie obiektowym.

[typ referencyjny](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w7.htm#Typ referencyjny) - typ wartości reprezentujący odwołania do obiektów w bazie danych. Jest obiektową alternatywą do wiązania tabel przez wartość klucza obcego.

[kolekcja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w7.htm#Kolekcje) - typ wartości w bazie danych reprezentujący zbiory wartości prostych lub obiektów.

[duży obiekt LOB](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w7.htm#Duze obiekty LOB) - typ wartości w bazie danych reprezentujący duże obiekty w rodzaju dokumentów tekstowych, plików z obrazem, dźwiękiem lub video.

[DBMS\_LOB](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w7.htm#Dopisywanie) - pakiet z operacjami na dużych obiektach LOB.

## Wykład 8

***Fizyczna organizacja danych w bazie danych.***

**Streszczenie**

Tematem wykładu jest *model fizyczny* bazy danych - w odróżnieniu od *modelu logicznego* rozważanego do tej pory. Dla większości użytkowników baz danych znajomość szczegółów technicznych organizacji baz danych w komputerze może nie jest istotna, po to został zresztą wprowadzony poziom logiczny bazy danych, jednakże ogólna orientacja w tym temacie jest pożądana. Umożliwia ona zrozumienie działania baz danych i ich uwarunkowań a co za tym idzie daje możliwość dostrojenia bazy danych w celu efektywniejszego jej działania. Jest więc nieodzowna dla osób administrujących systemami baz danych nie mówiąc już o osobach planujących zbudowanie własnego SZBD.

Zostaną rozważone takie zagadnienia jak: zarządzanie miejscem na dysku, zarządzanie obszarem buforów danych (w RAM) oraz organizacja zapisu rekordów na dysku. Inaczej mówiąc, zostanie opisane działanie modułów SZBD najniższego poziomu jak i wymagane struktury danych.

Zagadnienia budowy indeksów i wykonywania instrukcji SQL zostaną przedstawione na kolejnych wykładach.

**8.1** **Model fizyczny bazy danych**

Do tej pory zajmowaliśmy się głównie problemami modelu logicznego bazy danych. Teraz przyszła pora poznać szczegóły implementacji modelu logicznego w komputerze czyli szczegóły *modelu fizycznego bazy danych*. Model ten ściśle zależy od bieżącej architektury systemów komputerowych. Wraz ze zmianami w architekturze model ten ulega i także w przyszłości będzie ulegał zmianom. Zakładamy, że czytelnik zna podstawy budowy współczesnych systemów komputerów w tym nośników przechowywania danych takich jak magnetyczne dyski. Szczegółami technicznymi budowy dysków nie będziemy się zajmować na tym wykładzie.

*Model fizyczny bazy danych* jest oparty na pojęciach *pliku* i *rekordu*. *Plik obiektu bazy danych* (nazywany też *segmentem* tego obiektu)składa się z *rekordów* posiadających ten sam *format*. *Format* rekordu jest listą *nazw pól* z określeniem ich typ*ów danych*. *Rekord* składa się z wartości poszczególnych *pól*. Niektóre pola są wyróżnione jako *klucz wyszukiwania rekordu.* Może być więcej niż jeden klucz wyszukiwania rekordu. Na poziomie fizycznym nie wymaga się aby był określony jednoznaczny identyfikator (klucz) rekordu w pliku.

Podstawowymi operacjami na pliku, inicjowanymi przez programy aplikacyjne, są:

1. *Wstawianie* - wstaw rekord do pliku.
2. *Usuwanie* - usuń rekord z pliku.
3. *Modyfikacja* - zmodyfikuj zawartość pól w rekordzie w pliku.
4. *Wyszukiwanie* - znajdź w pliku rekord(y) z podaną wartością  klucza wyszukiwania lub spełniające podane warunki.

Zatem duża różnorodność operacji na bazie danych dostępnych za pomocą instrukcji języka SQL, sprowadza się do tylko czterech na poziomie fizycznym. Oczywiście operacje na słowniku danych również są wykonywane w ten sposób.

Trzeba tu podkreślić, że plik modelu fizycznego bazy danych może być reprezentowany przez plik dostępny w systemie plików komputera ale nie musi. Tak więc pojęcia pliku rekordów z danymi i pliku dyskowego nie są tożsame.

W specjalnym przypadku plik dyskowy może obejmować kilka plików bazy danych (np. w przypadku klastra kilku tabel).

Również w specjalnych przypadkach tabela jest zapisywana w więcej niż jednym pliku dyskowym np. gdy tabela jest dzielona na partycje, które mogą być przetwarzane równolegle przez osobne procesy. Wtedy jednemu plikowi modelu fizycznego bazy danych odpowiada kilka plików dyskowych.

**Dyski i pliki**

Współczesne systemy zarządzania bazą danych przechowują dane na twardych (magnetycznych) dyskach. Aby wykonać operacje na danych, trzeba je najpierw zapisać w pamięci wewnętrznej RAM, wykonać na nich operacje a następnie przepisać wyniki na dysk. Stąd nieodzowność stosowania operacji We/Wy:

1. *Odczyt (READ)*: przesłanie danych z dysku do pamięci RAM.
2. *Zapis (WRITE)*: przesłanie danych z pamięci RAM na dysk.

Obie operacje są o rząd wielkości wolniejsze niż operacje w pamięci RAM – muszą być stosowane przez SZBD umiejętnie! Koszt operacji na bazie danych jest przedstawiany jako liczba operacji We/Wy.

|  |
| --- |
| Naturalne wydaje się pytanie dlaczego danych nie można przechowywać w pamięci RAM?  Pierwszym argumentem jest fakt, że pamięć RAM jest chwilowa. Po wyłączeniu komputera z prądu tracimy wszystkie dane. Można tu jednak argumentować, że nie trzeba komputera wyłączać z prądu a wszelkie awarie zasilania rozwiązywać za pomocą urządzenia UPS, które w chwili awarii może przepisać dane na trwały nośnik.  Drugim, poważnym argumentem jest istotnie wyższy koszt pamięci RAM niż dysku (rzędu sto razy wyższy).  Trzecim, decydującym argumentem było do tej pory to, że stosowane we współczesnych komputerach 32 bitowe adresowanie ogranicza ilość danych, które można zapisać w pamięci RAM. Jednakże znajdujemy się już w trakcie przechodzenia na adresowanie 64 bitowe.  Powyższa dyskusja pokazuje, że w niedalekiej przyszłości możemy mieć do czynienia z nowym, alternatywnym modelem fizycznym bazy danych. Na razie jednak jest faktem, że podstawowym nośnikiem przechowywania danych w bazie danych jest dysk magnetyczny. |

Oprócz dysków używa się też obszerniejszych ale za to wolniejszych w dostępie nośników danych takich jak taśmy magnetyczne, dyski optyczne, CD ROMy. Służą one do przechowywania: kopii zabezpieczających bieżącej bazy danych (backup), dzienników transakcyjnych off-line oraz kopii archiwalnych danych.

Charakterystyka używania danych przechowywanych na dysku

1. Dostęp swobodny (ang. *random access*): w oparciu o adres rekordu na dysku możemy go przesłać do pamięci RAM w jednej operacji We-Wy.
2. Dane są przechowywane i przekazywane w jednostkach nazywanych *blokami dyskowymi* lub *stronami*.
3. Inaczej niż w przypadku pamięci RAM, czas dostępu do danych na dysku zależy od ich położenia na dysku. Dlatego wzajemne rozmieszczenie stron na dysku może mieć  zasadniczy wpływ na szybkość działania SZBD! Najlepsze wyniki uzyskujemy przy operowaniu ciągami sąsiadujących ze sobą stron.
4. Dąży się do tego, aby dane, które są często wykorzystywane przez programy aplikacyjne, na stałe przebywały w buforach pamięci RAM (tzw. c*achowanie*). Wraz z obniżaniem się kosztu pamięci RAM coraz więcej danych bazy danych rezyduje w pamięci RAM. Dostęp do nich jest wtedy bardzo szybki.
5. Operacje odczytu i zapisu bloków na dysku mogą być realizowane współbieżnie. Stąd opłaca się aby transakcje użytkowników były realizowane przez system współbieżnie a nie sekwencyjnie. Również w ramach jednej transakcji rozkładając dane do różnych dysków można istotnie przyśpieszyć jej wykonanie.

Natomiast w przypadku taśm mamy do czynienia z  dostępem sekwencyjnym - dane są odczytywane jedna po drugiej od początku do końca taśmy jako jedna, długa operacja We-Wy. Zaletami taśm są duża pojemność i niski koszt. Są używane do archiwizowania i tworzenia kopii zabezpieczających (backup-ów).

**8.2** **Dyskowy model fizyczny bazy danych**

W przypadku dysku jako podstawowego nośnika danych w bazie danych nasz ogólny model fizyczny ulega uzupełnieniu o strony dyskowe. Oto jego podsumowanie.

*Tabela (relacja)* jest reprezentowana przez *plik* (dyskowy). *Plik* składa się ze *stron*. *Strona* składa się z *rekordów*. *Rekord* składa się z *pól*.

Rys. 8.1 Porównanie pojęć poziomu logicznego i fizycznego

Ponadto:

1. Gdy rozmiar rekordu jest większy niż rozmiar strony, rekord jest dzielony na części przechowywane na osobnych stronach - najlepiej sąsiadujących na dysku. (Ta cecha nie jest implementowana w każdym systemie SZBD. Jeśli nie jest implementowana projektant bazy danych musi to uwzględnić, dzieląc długie wiersze na części przechowywane albo w tej samej tabeli albo w różnych - np. część atrybutów umieszczamy w jednej tabeli część w drugiej.)
2. Duże obiekty LOB są trzymane w osobnym obszarze przeznaczonym do ich przechowywania w bazie danych, zwykle jako ciąg sąsiednich stron. W rekordach z danymi znajdują się tylko ich lokalizatory.
3. Gdy schemat dostępu do danych polega na użyciu powiązanych danych z dwóch lub więcej tabel (np. departamenty i ich pracownicy; zamówienia i pozycje zamówień), w jednym pliku są zbierane dane z kilku tabel w oparciu o wspólny klucz (np. numer departamentu czy identyfikator klienta) - mamy wtedy do czynienia z *klastrem* tabel.

Hierarchia nośników przechowywania danych w bazie danych

1. Pamięć RAM dla danych używanych w bieżącej chwili.
2. Dysk dla głównej bazy danych.
3. Taśma dla archiwalnych wersji danych i backupu.

Rys. 8.2 Hierarchia nośników przechowywania danych w bazie danych

Te same dane np. rekord pracownika mogą się w tej samej chwili znajdować w trzech różnych miejscach! Co więcej mogą się zmieniać w jednym z nich (w pamięci RAM) powodując brak (chwilowy) synchronizacji. Na ogół zmiany są propagowane z pamięci RAM do dysku i następnie na taśmę. Ale są sytuacje, np. operacja ROLLBACK albo awaria gdy rekord zapisany na dysku lub na taśmie zastępuje rekord zapisany w pamięci RAM.

Przejdziemy teraz do omówienia podstawowych modułów SZBD realizujących operacje na *stronach* (nie odwołując się do rekordów), z których korzystają inne moduły SZBD. Są to: moduł *zarządzania miejscem na dysku* oraz moduł *zarządzania buforami w pamięci RAM*.

**8.3** **Zarządzanie miejscem na dysku**

W pliku mamy dwa rodzaje stron:

* strony z wolnym miejscem do zapisu nowych rekordów, oraz
* strony całkiem albo prawie całkiem zapełnione.

Zakładamy także, że system ma możliwość rozszerzania pliku rekordów i alokacji nowych, pustych stron oraz że system ma możliwość zwrotu (dealokacji) nieużywanych stron.

Moduł *zarządzania miejscem na dysku* (ang. *disk space manager*) realizuje następujące funkcje:

1. Alokacja ciągu stron (położonych spójnie na dysku).
2. Zwrot (dealokacja strony).
3. Wyznaczenie strony do zapisu nowego rekordu.
4. Aktualizacja struktur danych na dysku związanych z przechowywanymi stronami.

Rozmiarem strony jest zwykle rozmiar bloku dyskowego i strony są przechowywane jako bloki dyskowe więc odczyt/zapis strony wykonuje się jako pojedyncza operacja We/Wy.

Moduł zarządzający miejscem na dysku ukrywa szczegóły sprzętu i systemu operacyjnego i umożliwia pozostałym modułom traktowanie danych na dysku jako zbioru stron.

**8.4** **Zarządzanie buforami danych (w RAM)**

*Moduł zarządzania buforami danych* (ang. *buffer manager*) jest odpowiedzialny za sprowadzanie stron z dysku do puli buforów danych w pamięci RAM. Dane muszą być umieszczone w buforach danych aby procesy SZBD mogły na nich operować! Zapis strony odbywa się w *ramce* (ang. *frame*).

Rys. 8.3 Pula ramek

Oprócz puli ramek w pamięci RAM są przechowywane:

* struktura danych zbioru par: <*nr\_ramki, id\_strony*> umożliwiająca znalezienie *id\_strony* w oparciu o *nr\_ramki* jak i *id\_ramki* w oparciu o *id\_strony* (np. dwie tablice haszowane);
* dla każdej ramki: ***licznik odwołań*** - ile różnych procesów używa ramki w danej chwili. Na początku po umieszczeniu strony w ramce: **licznik odwołań** = 1
* dla każdej ramki: ***bit modyfikacji*** – czy po sprowadzeniu do pamięci RAM zawartość ramki została zmodyfikowana (stan "*dirty*"), co oznacza, że strona na dysku będąca źródłem zawartości ramki może już być inna niż zawartość ramki w pamięci RAM. Na początku po umieszczeniu strony w ramce: **bit modyfikacji** = false

Ponadto wszystkie ramki, których **licznik odwołań** = 0, tworzą *listę wolnych ramek* (czasami lista ta jest tworzona z opóźnieniem).

Moduł zarządzania buforami danych jest wykorzystywany przez procesy SZBD - przede wszystkim przez procesy realizujące zlecenia użytkowników. Oto szkic jego działania.

Gdy procesowi jest potrzebna strona wykonywany jest algorytm:

1. Gdy potrzebnej strony nie ma w puli ramek:
   * Wybierz ramkę z listy wolnych ramek (**licznik odwołań** = 0).
   * Jeśli strona w wybranej ramce została zmieniona ale nie zaktualizowana na dysku (**bit modyfikacji** = true), zapisz ją na dysk.
   * Wczytaj potrzebną stronę w wybraną ramkę.
   * Ustaw jej **licznik odwołań** = 1.
2. Gdy potrzebna strona jest w puli ramek, zwiększ jej **licznik odwołań** o jeden.
   * Ewentualnie, jeśli ramka znajduje się na liście wolnych ramek, przenieś ją do części z używanymi ramkami.
3. Przekaż procesowi wskaźnik do ramki ze stroną.
4. Jeśli można z góry przewidzieć, że mają być sprowadzone sąsiadujące na dysku strony np. przy przeglądaniu sekwencyjnym pliku, sprowadź od razu kilka stron!

Gdy zmienia się zawartość ramki:

* Ustawiamy dla ramki **bit modyfikacji** = *true*.

Strona w buforze danych może być potrzebna wielu procesom:

* Nowe zapotrzebowanie na stronę w ramce zwiększa jej **licznik odwołań** o jeden.
* Gdy proces zwalnia stronę w ramce (np. w wyniku realizacji instrukcji COMMIT lub ROLLBACK), jej **licznik odwołań** zmniejsza się o jeden. Proces musi zapisywać numer ramki, z której korzysta, aby móc ją później zwolnić.
* Ramka staje się kandydatem do zastąpienia gdy jej **licznik odwołań** = 0. Zostaje wtedy wstawiona na listę wolnych ramek - niezależnie od wartości bitu modyfikacji.

Z jednej ramki w pamięci RAM może korzystać wiele procesów. Sposób korzystania jest określony przez specjalne procedury, o których będzie mowa w wykładzie o transakcjach.

**Strategie zastępowania stron w ramkach z listy wolnych ramek**

Strona jest na liście wolnych ramek gdy jej **licznik odwołań** jest równy 0. Stosowane są następujące strategie wyboru ramki do zastąpienia:

1. LRU – zastępuje się stronę, która najdłużej była nie używana,
2. MRU – zastępuje się stronę, która ostatnio była używana,
3. Clock - ustala się stałą, cykliczną kolejność pobierania wolnych ramek.

Wydawałoby się, że strategia LRU powinna być zawsze najlepsza. Okazuje się, jak pokazuje poniższy przykład, że nie jest tak zawsze.

Zjawisko sekwencyjnego zalewania puli ramek:

Stosując strategię LRU powtarzamy wielokrotnie sekwencyjne przeglądanie pliku. Gdy #ramek < #stron w pliku, wtedy każde żądanie strony powoduje operację We/Wy.

Strategia polegająca na wczytaniu najpierw #ramek stron do pamięci RAM, a następnie stosowaniu strategii MRU byłaby lepsza w tym przypadku. Część stron równa #ramek–1 pozostawałaby cały czas w puli buforów, natomiast pozostałe strony wczytywane byłyby kolejno do jednej z ramek.

Podobnie jeśli podejrzewamy, że wczytywane strony nie będą używane w najbliższym czasie, a tak możemy wnioskować w przypadku przeglądania wierszy dużej tabeli, nie opłaca się zastępować stron z rekordami, które są często używane przez strony pochodzące z pełnego przejrzenia wierszy dużej tabeli.

W praktyce proces wykonujący zlecenie użytkownika może posiadać informacje, że pewne strony są często używane i w związku należy je na stałe przechowywać w buforach w pamięci RAM. Komercyjne SZBD w większości realizują taką możliwość.

W stosowanych serwerach wzrastają rozmiary pamięci RAM i w związku z tym także rośnie liczba buforów danych. Wielokrotnie używane strony powinny więc być stale przechowywane w buforach. Gdy baza danych jest używana przez aplikację biznesową, przyjmuje się że średnio co najmniej 90% potrzebnych do wykonania zapytania danych powinno znajdować się w buforach. Ponadto, jest potrzebny niezależny proces serwera bazy danych, który będzie regularnie synchronizował zawartości stron w buforach z tymi na dysku.

**Obsługa wskaźników**

W modelu obiektowo-relacyjnym jak i przy implementacji modelu relacyjnego (np. przy indeksach) występują wartości, które są wskaźnikami do rekordów (wierszy,  obiektów). Wskaźniki są reprezentowane przez adresy obiektów składowanych na dysku. Gdy obiekty zostają zapisane w buforach, jest możliwość *przemiany* adresów dyskowych na adresy pamięci RAM (ang. *pointer* *swizzling*). Przetwarzanie takich obiektów staje się szybsze. Przy zapisie obiektu na dysk jest konieczna transformacja odwrotna adresów pamięci RAM na adresy dyskowe. Potrzebna jest więc tabela odwzorowująca wzajemnie adresy obiektów na dysku i w pamięci.

**8.5** **Formaty rekordów i stron**

Gdy użytkownik potrzebuje konkretnego rekordu z bazy danych, proces obsługujący zlecenie użytkownika:

1. najpierw oblicza adres strony, na której znajduje się dany rekord,
2. następnie sprowadza stronę z dysku i umieszcza ją w buforze pamięci RAM (przy tych operacjach są wywoływane moduły zarządzania miejscem na dysku i zarządzania buforami w pamięci RAM),
3. po czym wydobywa z niej szukany rekord i przekazuje go użytkownikowi.

W punkcie 3 jest potrzebna znajomość struktur zapisu: pól w ramach rekordu i rekordów na stronie. Omówimy je w tej chwili.

Liczba i typy pól są takie same dla wszystkich rekordów w pliku; są zapisane w słowniku danych (katalogu systemowym). Jest kilka alternatywnych formatów zapisu pól w ramach rekordu.

**Format rekordu: stała długość pól**

Zapis rekordu składa się ze spójnego obszaru zawierającego ciąg pól P*1*,...,P*n* o stałych rozmiarach *D1*,...,*Dn*. Mając adres bazowy (początek rekordu) *B* łatwo można obliczyć początek *i*-tego pola jako *B+D1+...+Di-1*.

Rys. 8.4 Zapis rekordu stałej długości

**Format rekordu: zmienna długość pól**

Istnieją dwa alternatywne formaty (przy założeniu, że #pól jest stała). W pierwszym przypadku, rozdzielamy pola za pomocą specjalnego separatora np. symbolu '$'. W drugim przypadku, na początku zapisu rekordu umieszczamy *tablicę wskaźników* (*offsetów*) do początków kolejnych pól.

Rys. 8.5 Zapis rekordu zmiennej długości

W drugim przypadku uzyskujemy:

* bezpośredni dostęp do wartości *i*-tego pola;
* efektywne przechowywanie wartości *Null*.

Powyższą strukturę danych można rozszerzyć do przypadku, gdy liczba pól jest zmienna – ale ich typ jest określony statycznie w słowniku danych (katalogu systemowym). Mianowicie przed katalogiem wskaźników do pól umieszczamy liczbę pól w bieżącym rekordzie (czyli rozmiar tablicy *offsetów*). Nie trzeba wtedy reprezentować pól na końcu rekordu nie mających ustalonych wartości, czyli będących *Null*.

Z kolei rozpatrzymy formaty zapisu rekordów w ramach strony.

**Format strony dla rekordów stałej długości**

Istnieją dwa alternatywne formaty. W obu przypadkach zapis strony składa się z ciągu miejsc, w każdym z nich albo jest zapisany rekord albo miejsce jest wolne. W pierwszym przypadku, najpierw są zgrupowane wszystkie miejsca zajęte, a następnie wolne. W drugim przypadku, miejsca wolne i zajęte są ze sobą przemieszane - to czy dane miejsce jest wolne czy zajęte wskazuje bit w dodatkowej tablicy *Occupied*:

*Occupied(i)*=1 wtedy i tylko wtedy, gdy *i*-te miejsce na stronie jest zajęte.

Rys. 8.6 Zapis rekordów stałej długości

Przy operowaniu na danych pojawia się potrzeba sięgania do konkretnych rekordów, a nie do wszystkich za każdym razem np. sięgamy do rekordu w oparciu o jego wcześniej wyliczony adres bądź poprzez jego adres znaleziony w indeksie.

Adres rekordu czyli identyfikator rekordu *rid*, identyfikujący jego położenie na dysku, jest określany następująco:

*rid* = <id\_strony, numer pozycji na stronie>

W przypadku pierwszej struktury przesuwanie rekordów na stronie  powoduje zmianę identyfikatora rekordu, co komplikuje odwołania do rekordu przez jego identyfikator *rid*. W przypadku drugiej struktury rekord nie jest przesuwany na stronie, nie zmienia się więc jego identyfikator.

**Format strony dla rekordów zmiennej długości**

W przypadku rekordów zmiennej długości adres zapisu rekordu na stronie jest określony przez wartość w *tablicy pozycji* Poz. Wartość Poz(*i*) wskazuje na początek zapisu *i*-tego rekordu, 1<= *i* <= *N*. Dodatkowo, wartość Poz(0) wskazuje na początek obszaru wolnych miejsc.

Rys. 8.7 Zapis rekordów zmiennej długości

Adres rekordu czyli identyfikator rekordu *rid*, identyfikujący jego położenie na dysku, jest określany następująco:

*rid* = <id\_strony, numer pozycji w tablicy Poz>

Gdy przesuwamy *i*-ty rekord na stronie, jego nowy adres na stronie aktualizujemy w Poz(*i*). Nie zmienia to indeksu *i* a zatem nie zmienia także identyfikatora rekordu *rid.*

Natomiast przy przenoszeniu rekordu na inną stronę, identyfikator rekordu zmienia się. Gdy tego rodzaju operacje są przewidywane, trzeba to uwzględnić w strukturze danych:

* Jedna metoda polega na przypisywaniu rekordom logicznych, niezmiennych adresów oraz dodaniu pomocniczej tablicy odwzorowującej logiczne adresy rekordów (np. wartości kluczy głównych) na fizyczne (zmienne) adresy rekordów.
* Druga metoda polega na pozostawieniu w oryginalnym miejscu nowego adresu przeniesionego rekordu. Wyszukując rekord znajdujemy go więc albo w miejscu podanym przez jego referencję albo w miejscu tym odczytujemy adres, pod który należy zajrzeć aby wyznaczyć szukany rekord. Postępowanie to może się powtórzyć wielokrotnie.

Na stronie dyskowej są zapisywane także dodatkowe informacje jak informacje o transakcjach, które aktualnie wykonują operacje na rekordach przechowywanych na danej stronie czy powiązania między stronami w ramach struktury danych pliku rekordów (o tym za chwilę).

**8.6** **Pliki rekordów**

Plik stanowi kolekcję stron, z których każda zawiera zero, jeden lub więcej rekordów. Jak wiemy, na pliku są wykonywane następujące rodzaje operacji:

1. *wstawianie/usuwanie/modyfikowanie* rekordu (o podanym *rid*),
2. *odczytywanie* konkretnego rekordu (o podanym *rid*),
3. *wyszukiwanie wszystkich rekordów* spełniających podane warunki.

Jest możliwe kilka organizacji pliku rekordów. Podstawowa różnica między nimi polega na tym, czy porządkują one rekordy według wartości pewnego klucza czy nie. Najpierw rozważymy organizacje zapisu nieuporządkowanego (nazywanego potocznie *stertą* ang. heap).

**Plik nieuporządkowany (ang. heap)**

1. Rekordy są przechowywane na stronach pliku w dowolnym porządku.
2. Nowy rekord jest wstawiany do pierwszej strony, na której jest wolne miejsce.
3. Przy wyszukiwaniu przechodzimy po wszystkich stronach do chwili napotkania szukanego rekordu albo do końca pliku, gdy rekordu nie ma w pliku.

Organizacja nieuporządkowana jest wygodna przy wykonywaniu zapytań dotyczących wszystkich rekordów lub większości rekordów np.

    SELECT \* FROM Emp;

lub

    SELECT \* FROM Emp e WHERE e.Sal > 1000;

gdy wiemy, że większość pracowników zarabia powyżej 1000.

Plik nieuporządkowany w wersji dwie listy

Rys. 8.8 Zapis pliku w postaci dwóch list

Na jednej liście trzymane są strony z wolnymi miejscami do wstawienia nowych rekordów; na drugiej liście są trzymane strony bez wolnego miejsca do wstawienia nowych rekordów.

Plik nieuporządkowany w wersji katalogu stron

Rys. 8.9 Zapis pliku w postaci katalogu stron

Stan zajętości stron jest zapisywany w osobnym katalogu.

Przy wpisywaniu nowego rekordu, najpierw wybieramy w katalogu stronę z wolnym miejscem, ściągamy ją i wpisujemy rekord.

Alternatywne w stosunku do pliku nieuporządkowanego ogranizacje zapisu polegają na zachowaniu pewnego ustalonego porządku względem klucza rekordu. Przedstawimy dwa takie rozwiązania: *plik posortowany* i *plik haszowany*.

**Plik posortowany**

Rekordy są zapisywane na kolejnych stronach zgodnie z porządkiem względem klucza wyszukiwania rekordu. Taka reprezentacja jest wygodna gdy rekordy przetwarza się zawsze w pewnym, ustalonym porządku lub tylko pewien ich zakres względem tego porządku np.

    SELECT \* FROM Emp e ORDER BY e.Sal;

lub

    SELECT \* FROM Emp e WHERE e.Sal BETWEEN 1000 and 2000;

W pliku posortowanym wyszukanie rekordu mając dany jego klucz wyszukiwania jest nieco szybsze niż dla pliku nieuporządkowanego, ale ze względu na to, że rekordy znajdują się na dysku, zastosowanie jednej z szybkich metod wyszukiwania jak wyszukiwanie binarne nie jest w pełni możliwe.

Są możliwe dwie implementacje:

1. Pełny ekstent – bloków sąsiadujących ze sobą na dysku – rekordy są uporządkowane według wartości klucza wyszukiwania. Jest problem ze wstawieniem nowego rekordu lub usunięciem rekordu z pliku. Jest możliwość zastosowania wyszukiwania binarnego.

|  |
| --- |
|  |

Rys. 8.10 Plik posortowany w postaci pełnego ekstentu

2. Lista bloków (lub ekstentów) – rekordy są uporządkowane według wartości klucza wyszukiwania ale kolejne bloki na liście nie muszą być sąsiednie. Nie ma problemu ze wstawieniem nowego rekordu i usunięciem rekordu z pliku. Bezpośrednio nie ma możliwości zastosowania wyszukiwania binarnego - potrzebna jest więc dodatkowa struktura danych wykorzystująca posortowanie danych (struktura indeksowa - temat ten zostanie omówiony na kolejnym wykładzie).

|  |
| --- |
|  |

Rys. 8.11 Plik posortowany w postaci listy

W obu powyższych metodach, rekordy o tej samej wartości klucza lub zbliżonej znajdują się na tej samej stronie lub tylko na kilku stronach dyskowych (w pierwszej metodzie nawet na sąsiednich). Jest to istotne przy wyszukiwaniu, gdyż wyznaczenie wszystkich rekordów o danej wartości klucza wyszukiwania (lub o wartości zbliżonej) wymaga sprowadzenia tylko kilku stron, a nie dla każdego rekordu - osobnej strony.

**Plik haszowany**

Plik jest kolekcją *segmentów* (ang. *bucket*) rekordów. *Segment* jest to strona *główna* plus zero lub więcej stron *nadmiarowych* alokowanych do segmentu w razie potrzeby. Przydział rekordu do segmentu odbywa się w oparciu o wartość funkcji nazywanej *funkcją* *haszującą* (*mieszającą*) zastosowanej do *klucza wyszukiwania rekordu*, którym jest jedno lub więcej pól rekordu. Mianowicie:

Funkcja haszująca h: h(*klucz wyszukiwania rekordu r*) = adres segmentu do którego wpada rekord *r*.

Rys. 8.12 Plik haszowany

Przy tej metodzie, rekordy o tej samej wartości klucza wyszukiwania znajdują się na tej samej stronie lub tylko na kilku stronach dyskowych. Jest to istotne przy wyszukiwaniu, gdyż wyznaczenie wszystkich rekordów o danej wartości klucza wyszukiwania wymaga sprowadzenia tylko kilku stron, a nie dla każdego rekordu osobnej strony.

Organizacja pliku haszowanego jest użyteczna przy wyborze rekordu z pliku w oparciu o wartość lub wartości pewnych pól rekordu np. przy wykonywaniu zapytania

SELECT \* FROM Emp e WHERE e.Ename=:Nazwisko;

Natomiast organizacja rekordów w pliku haszowanym nie zachowuje kolejności względem wartości klucza wyszukiwania.

Zastosowanie jednej organizacji rekordów w pliku zazwyczaj nie wystarcza w aplikacjach baz danych, w których wyszukiwanie odbywa się względem wartości różnych kluczy wyszukiwania. Rozwiązanie tego problemu polega na skorzystaniu z osobnych struktur indeksowych. Ich studiowanie rozpocznie się na następnym wykładzie.

**8.7 Podsumowanie**

W wykładzie tym czytelnik poznał podstawy fizycznego modelu bazy danych opartego na pojęciach pliku, strony, rekordu i buforu danych. Treścią kolejnych wykładów będą zasady funkcjonowania SZBD na gruncie przedstawionego modelu.

Podstawowe dwa moduły niskiego poziomu zasłaniające moduły wysokiego poziomu od szczegółów implementacyjnych modelu fizycznego to moduł *zarządzania miejscem na dysku* (Disk Space Manager) oraz moduł *zarządzania buforami w pamięci RAM* (Buffer Manager).

**8.8 Słownik pojęć**

[model fizyczny bazy danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Model) - reprezentacja tabel w terminach struktur przechowywania danych w komputerze. Podstawowy dyskowy model fizyczny jest następujący: *tabela* jest reprezentowana przez *plik* (dyskowy);  *plik* składa się ze *stron*, *strona* składa się z *rekordów*, *rekord* składa się z *pól*.

[plik](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Model) - reprezentacja tabeli w modelu fizycznym bazy danych.

[rekord](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Model) - fizyczna reprezentacja wiersza.

[pole](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Model) - reprezentacja wartości w kolumnie (elementu wiersza) w modelu fizycznym bazy danych.

[klucz wyszukiwania (rekordu)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Model) - wybrane pola rekordu, względem których ma odbywać się wyszukiwanie. Może być wiele kluczy wyszukiwania rekordu.

[strona](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Strona) (blok) - fizyczna jednostka przechowywania danych na dysku. Dane są przesyłane między pamięcią wewnętrzną i dyskiem stronami. Na jednej stronie mieści się pewna liczba rekordów.

[bufor danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Zarzadzanie buforami) - miejsce w pamięci RAM, do którego sprowadza się stronę z dysku.

[LRU](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#LRU) - podstawowa strategia zastępowania stron w puli buforów danych; mianowicie zastępuje się stronę, która najdłużej pozostawała nie używana.

[plik nieuporządkowany (heap)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Plik nieup) - plik, w którym rekordy są przechowywane na stronach w dowolnym porządku.

[plik posortowany](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Psort) - plik, w którym rekordy są zapisywane zgodnie z porządkiem względem pewnego klucza wyszukiwania rekordu.

[plik haszowany](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Plik h) - plik, który jest kolekcją segmentów. Podział rekordów na segmenty odbywa się w oparciu o wartość funkcji, nazywanej f*unkcją haszującą*, zastosowaną do klucza rekordu.

[funkcja haszująca](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w8.htm#Funkcja h) - funkcja, która wartościom klucza wyszukiwania,  przyporządkowuje adres segmentu w pliku haszowanym.

**Wykład 9**

***Indeksy i sortowanie zewnętrzne***

**Streszczenie**

W poprzednim wykładzie zostały omówione trzy podstawowe organizacje pliku rekordów: nieuporządkowana, posortowana i haszowana. Ze względu na różne sposoby dostępu do rekordów w danym pliku (np. ze względu na wartości różnych pól) potrzebne są osobne struktury danych wspomagające te sposoby dostępu. Obiekty bazy danych, które odpowiadają tym strukturom danym, nazywają się *indeksami*.

Wykład składa się z trzech części. W pierwszej części są omawiane podstawy struktur indeksowych w tym ich klasyfikacja.

W drugiej części są omawiane podstawowe struktury danych dla indeksów.

W trzeciej części jest rozważany centralny dla operacji na bazie danych problem sortowania zewnętrznego.

W przykładach są używane tabele *Emp* i *Dept* znajome z nauki SQL i PL/SQL.

**9.1** **Indeksy**

Na początku tego wykładu przedstawimy ogólne definicje związane z indeksami oraz ogólne ich klasyfikacje.

*Indeks* jest to struktura danych na dysku umożliwiająca szybkie wyszukiwanie danych w bazie danych na podstawie wartości klucza wyszukiwania takiego jak np. *nazwisko osoby*. Indeks w bazie danych ma takie samo znaczenie jak skorowidz (indeks) w książce!

W najprostszej postaci wyszukiwanie polega na tym, że mając *wartość* poszukujemy *rekordów*, w których ta *wartość* występuje w danym *polu*.

Rys. 9.1 Wyszukiwanie rekordów według wartości w polu

Na przykład:

1. Wyznacz dane studenta o nazwisku 'Kowalski'.
2. Wyznacz wszystkich studentów zapisanych na kurs 'Bazy danych'.

Operacja wyszukiwania może być nie trywialna do przeprowadzenia gdy liczba rekordów w pliku jest bardzo duża np. liczy kilkanaście milionów. Indeksy są strukturami danych, które wspomagają szybkie znajdowanie odpowiedzi na tego rodzaju zapytania.

*Klucz wyszukiwania* dla indeksu jest to wybrane pole lub pola rekordu względem których ma się odbywać wyszukiwanie. *Indeks* jest to struktura danych składająca się z *węzłów*, w których są zapisywane rekordy indeksu następujących postaci:

1. pozycje danych *k*\* określane względem wartości klucza wyszukiwania *k*. Są trzy rodzaje pozycji danych (przy czym w każdym indeksie jest stosowany tylko jeden rodzaj):
   1. sam rekord o kluczu *k*,
   2. para: (*k*,*w*) gdzie *k* jest kluczem rekordu a *w* jest wskaźnikiem do rekordu o tym kluczu,
   3. para: (*k*,*w1*,...,*wn*) gdzie *k* jest kluczem rekordu a *w1*,...,*wn* jest listą wskaźników do rekordów o kluczu *k*; rekord tej postaci jest bardziej zwarty niż (ii) ale za to wymaga zmiennej liczby pól;
2. pozycje indeksu kierujące wyznaczeniem właściwej pozycji danych *k\** w oparciu o wartość klucza wyszukiwania *k*; pozycją indeksu może być, na przykład, para postaci (*wartość klucza*, *wskaźnik do węzła w indeksie*).

Zawartość indeksu jest przechowywana w pliku nazywanym *plikiem indeksowym*.

W skład pliku danych wchodzą rekordy danych. W skład pliku indeksowego wchodzą rekordy indeksu będące pozycjami danych lub pozycjami indeksu.  Węzeł odpowiada na ogół stronie dyskowej i zawiera albo pozycje danych albo pozycje indeksu.

Indeks nazywamy *pogrupowanym* (*wewnętrznym*) (ang. clustered) gdy zachodzi przypadek 1i) w definicji pozycji danych oraz plik danych jest posortowany według wartości klucza wyszukiwania tego indeksu. W rezultacie, rekordy o tej samej wartości klucza lub zbliżonej znajdują się na tej samej stronie lub tylko na kilku stronach dyskowych. Może być tylko jeden indeks pogrupowany, bo plik danych można posortować według wartości tylko jednego klucza wyszukiwania. Indeks, który jest nie pogrupowany jest nazywany indeksem *zewnętrznym*.

Rys. 9.2 Dwa typy indeksów

Gdy klucz wyszukiwania zawiera klucz główny, indeks nazywa się indeksem *głównym*. Gdy klucz wyszukiwania zawiera klucz jednoznaczny, indeks nazywa się indeksem *jednoznacznym*.

**Złożone klucze wyszukiwania**

Złożone klucze wyszukiwania stanowią kombinację pól np. <*sal,age*>. W indeksie uporządkowanym kolejność pozycji danych w indeksie z kluczem złożonym jest oparta na porządku leksykograficznym.

Rys. 9.3 Pojedyncze i złożone klucze wyszukiwania

Na rysunku 9.3 oba indeksy ze złożonym kluczem wyszukiwania wspomagają wykonanie zapytania z klauzulą WHERE: age=20 AND sal=75

Natomiast tylko indeks z kluczem <age,sal> wspomaga wykonywanie zapytań zakresowych z klazulami WHERE: age<20 oraz age=20 AND sal>10

**Pseudo-wartość NULL**

Pseudo-wartość NULL stanowi problem przy wyszukiwaniu przez indeks, ponieważ nie można o żadnej wartości powiedzieć czy jest równa NULL czy nie. Obsługa NULL jest potrzebna w indeksie ponieważ za pomocą indeksu sprawdzane są więzy spójności klucza jednoznacznego a w składowych wartości klucza jednoznacznego są możliwe wartości NULL. Przyjmiemy założenie, że indeksowane są wszystkie wiersze, w których co najmniej jedna składowa klucza wyszukiwania indeksu nie jest NULL.

Na przykład, z powodu NULL za pomocą indeksu na kolumnie *Comm* nie można zrealizować wyszukiwania wierszy z nieokreśloną wartością w polu *Comm*:

SELECT Ename  
FROM Emp  
WHERE Comm IS NULL;

Rozważymy kilka rodzajów indeksów umożliwiających szybkie wyszukiwanie rekordów w dużym pliku rekordów.

**9.2** **Drzewo ISAM**

Popularną metodą wyszukiwania klucza w pliku (w ciągu) posortowanym jest  *wyszukiwanie binarne*. Porównujemy poszukiwany klucz z kluczem rekordu znajdującym się w środku ciągu i w zależności od wyniku porównania poszukujemy klucza albo w lewej części albo w prawej – stosując rekurencyjnie opisywaną metodę.

Jak wskazaliśmy to na poprzednim wykładzie, trudno jest stosować wyszukiwanie binarne do dynamicznego ciągu uporządkowanego. Ponadto musimy brać pod uwagę fakt, że elementy ciągu czyli rekordy nie znajdują się w indeksowanej tablicy, gdzie łatwo wyliczyć środkowy indeks, a tylko na dysku.

W przypadku przechowywania rekordów na dysku,  wyszukiwanie binarne jest łatwiej zastosować - nie bezpośrednio do pliku rekordów - ale do pliku zawierającego same klucze rekordów *k*1< *k*2< … <*kN*: gdzie *ki* jest najmniejszym kluczem na stronie o numerze *i*. Plik z kluczami stanowi *plik indeksowy*.

Rys. 9.4 Jednopoziomowy plik indeksowy

Wyszukiwanie binarne, zamiast na pliku rekordów, zostaje wykonane na znacznie mniejszym pliku indeksowym! Po wyznaczeniu najbliższego pasującego klucza wystarczy przejść do strony, na której znajduje się rekord o danym kluczu, o ile jest taki w pliku.

Nie ma przeszkód aby, tak jak dla posortowanego ciągu rekordów, tym razem do pliku indeksowego dobudować kolejny poziom sekwencyjnego indeksu z kluczami rekordów. Możemy tak postępować aż otrzymamy pojedynczy węzeł, a wszystkie poziomy sekwencyjnych indeksów dadzą nam strukturę drzewa. Dobieramy tak liczbę elementów w węźle aby jego zawartość była zapisywana na jednej stronie (ewentualnie kilku sąsiednich).

Nie ma przeszkód aby, tak jak dla posortowanego ciągu rekordów, tym razem do pliku indeksowego dobudować kolejny poziom sekwencyjnego indeksu z kluczami rekordów. Możemy tak postępować aż otrzymamy pojedynczy węzeł, a wszystkie poziomy sekwencyjnych indeksów dadzą nam strukturę drzewa. Dobieramy tak liczbę elementów w węźle aby jego zawartość była zapisywana na jednej stronie (ewentualnie w jednym ekstencie czyli na kilku sąsiednich stronach). Powstające w ten sposób drzewo nazywamy *drzewem ISAM* (skrót od ang. Indexed Sequential Access Method).

Zauważmy, że przy reprezantacji ciągu uporządkowanego za pomocą drzewa, element "środkowy" znajdujemy w korzeniu drzewa a ograniczenie wyszukiwania do podciągu realizujemy przez przejście krawędzią na kolejny, niższy poziom w drzewie.

Strona indeksu (węzeł drzewa)

Rys. 9.5 Strona indeksu

*Strona indeksu* składa się z *pozycji indeksu* <*Ki,Pi*>, gdzie *Ki* jest kluczem a *Pi* jest wskaźnikiem do węzła niższego poziomu. Pierwszą pozycję na stronie stanowi wskaźnik *P0* do węzła niższego poziomu.

|  |
| --- |
| * W poddrzewie wskazywanym przez *P0* wszystkie klucze są < od klucza *K1*. * W poddrzewie wskazywanym przez *Pi* wszystkie klucze są >= od klucza *Ki* dla i>=1. * W poddrzewie wskazywanym przez *Pi* wszystkie klucze są < od klucza *Ki+1* dla i<m. |

Na rysunku 9.6 są pokazane zasady wyboru gałęzi w poszukiwaniu danego klucza.

Rys. 9.6 Zasady wyboru gałęzi przy wyszukiwaniu

Na rysunku 9.7 jest przedstawiona struktura drzewa ISAM.

Rys. 9.7 Struktura drzewa ISAM

Przykład drzewa ISAM

Rys. 9.8 Przykład drzewa ISAM

Wstawiamy 23\*, 48\*, 41\*, 42\*. Otrzymujemy drzewo ISAM:

Rys. 9.9 Drzewo ISAM po wstawieniach

Ponieważ na stronach głównych nie było wolnego miejsca, utworzyły się doczepione do nich strony nadmiarowe - tworzące listy nieuporządkowane.

Struktura stron indeksu i stron głównych jest niezmienna. Gdy nie wystarczy miejsca na stronach głównych, doczepiane są strony nadmiarowe. Może się zdarzyć, że do jednej strony głównej trzeba doczepić długą listę stron nadmiarowych, co spowoduje, że wyszukiwanie rekordów na takiej liście będzie wymagać sprowadzenia do pamięci RAM wielu stron i może trwać długo.

Reasumując, drzewa ISAM dobrze nadają się do wyszukiwania rekordu na podstawie wartości jego klucza a także rekordów z zakresu wartości klucza:  *A*<= *klucz* <=*B* lub *A*<= *klucz* lub *klucz* <=*B*. Gdy liczba rekordów w pliku jest mniej więcej stała, struktura ISAM jest dobra. Z powodu niebezpieczeństwa grupowania się kluczy w ciągi na stronach nadmiarowych nie jest strukturą pewną w przypadku ogólnym. Na szczęście, istnieje jej niewielka modyfikacja do tak zwanych *drzew B+*, która nie ma już tej wady.

**9.3** **Drzewo B+**

*Drzewo B+* (nazywane też *B+ drzewem*) jest to drzewo o strukturze drzewa ISAM, którego kształt dynamicznie zmienia się w zależności od liczby pozycji danych i w którym nie używa się stron nadmiarowych. Drzewo B+ jest wyważone względem wysokości (ang. height-balanced), to znaczy każdy liść znajduje się w nim na tej samej głębokości. Ponadto, wymagana zajętość każdej strony indeksu wynosi minimum 50% (z wyjątkiem korzenia).

Maksymalna długość ścieżki w drzewie B+ od korzenia do liścia jest co najwyżej *c*log *N gdzie N =* #liści gdzie *c* jest stałą uzależnioną od średniej liczby pozycji indeksu na jednej stronie. Stąd wynika, że *wyszukiwanie/wstawianie/usuwanie* pozycji danych odbywa się w czasie rzędu logarytmicznego *c*log *N*.

Wyszukiwanie zaczyna się w korzeniu a porównania klucza wyszukiwania prowadzą do liścia tak jak dla drzewa ISAM.

Rys. 9.10 Drzewo B+

Do pozycji z kluczem 5 dochodzimy porównując w korzeniu 5 z 13 i wybierając skrajnie lewą gałąź.

Wyszukując 15, w korzeniu znajdujemy gałąź po której należy zejść, mianowicie leżącą między kluczami 13 i 17. W liściu znajdujemy tylko  elementy 14 i 16. To znaczy elementu 15 nie ma w tym drzewie.

Aby przekonać się czy element znajduje się w drzewie czy nie, wystarczy przejść ścieżką zaczynającą się w korzeniu drzewa a kończącą się w jednym z jego liści. Każdą stronę indeksu na tej ścieżce należy sprowadzić do buforu w pamięci RAM.

Aby wyznaczyć wszystkie elementy większe lub równe 24, znajdujemy najpierw element 24 (ogólnie najmniejszy element większy lub równy 24), następnie idąc w prawo po kolejnych liściach znajdujemy wszystkie elementy większe lub równe 24.

Przyjmujemy, że liście drzewa znajdują się na *dwukierunkowej strukturze listowej* - ułatwia ona wykonywanie zapytań zakresowych takich jak powyższe.

**B+ drzewo jako indeks pogrupowany i nie pogrupowany**

Tak jak opisaliśmy to na początku wykładu indeks może być *pogrupowany* i wtedy pozycje danych pokrywają się z rekordami danych – to znaczy rekordy danych są zapisywane w strukturze B+ drzewa zgodnie z porządkiem klucza wyszukiwania.

Albo indeks jest *nie pogrupowany* i wtedy rekordy z danymi są przechowywane poza indeksem w dowolnym porządku.

Dla jednej tabeli może być zbudowany tylko jeden indeks pogrupowany i wiele indeksów nie pogrupowanych względem różnych kluczy wyszukiwania.

W indeksie pogrupowanym zbudowanym na B+ drzewie rekordy z danymi są zapisywane na stronach reprezentujących liście drzewa. Z definicji algorytmów wstawiania i usuwania wynika, że rekordy mogą być przesuwane między stronami. To znaczy rekordy mogą zmieniać stronę a zatem nie może być do nich z zewnątrz bezpośrednich wskaźników używających identyfikatora strony. Za identyfikator rekordu używa się wtedy wartość jego klucza głównego a dostęp do rekordu jest realizowany poprzez indeks główny.

Są możliwe dwa przypadki. Na ogół, SZBD wymaga aby indeks pogrupowany był jednocześnie indeksem głównym. Wtedy wyszukiwanie rekordu według wartości klucza wyszukiwania indeksu nie pogrupowanego wymaga przejścia dwóch indeksów: najpierw nie pogrupowanego, w którym znajdujemy wartość klucza głównego szukanego rekordu, a następnie indeksu pogrupowanego głównego, w którym w oparciu o wartość klucza głównego znajdujemy szukany rekord.

Jest też możliwe, że indeks główny jest indeksem nie pogrupowanym, w którego pozycjach danych są zapisywane (zmienne) wskaźniki do rekordów. W tym przypadku, przy zmianie położenia rekordu w indeksie pogrupowanym jego adres w pozycji danych indeksu głównego też musi być uaktualniony. Reasumując, przy wyszukiwaniu rekordu  przechodzimy tak jak poprzednio dwa indeksy: nie pogrupowany i główny. Natomiast przy zmianach położenia rekordu w indeksie pogrupowanym wymagane jest znalezienie pozycji danych tego rekordu w indeksie głównym i zapisanie w niej nowego adresu tego rekordu.

Zatem, z jednej strony zbudowanie i użycie indeksu pogrupowanego przyśpiesza wyszukiwanie przez ten indeks - szczególnie w przypadku mało selektywnego wyszukiwania np. zakresowego dla indeksu pogrupowanego - jednak z drugiej strony spowalnia on wyszukiwania przez pozostałe indeksy.

|  |
| --- |
| Drzewa B+ w praktyce - oszacowania   * Średni stopień *d* wynosi około 100. Średnie zapełnienie stron: 67% * Typowa pojemność:   + Wysokość 4: 312,900,700 rekordów   + Wysokość 3:    2,352,637 rekordów * W praktyce wysokość jest <=3. * Dwa lub nawet trzy kolejne górne poziomy drzewa mogą być przechowywane w buforze danych w  RAM. Oto ich przybliżone rozmiary:   + Poziom 1 = 1 strona = 8 KB   + Poziom 2 = 133 strony = 1 MB   + Poziom 3 = 17,689 stron = 133 MB |

**Strategia zastępowania ramek dla stron B+ drzewa**

Indeks jest zapisywany w pliku i jego strony są sprowadzane do buforów w pamięci RAM tak jak strony rekordów z danymi. Dla pliku indeksowego:

LRU nie jest dobrą metodą!

Dla drzew B+ lepszą strategią jest następująca:

* korzeń drzewa i najwyższe poziomy drzewa dobrze jest trzymać cały czas w pamięci RAM.

**Dodatkowe operacje na B+ drzewach**

Wszystkie komercyjne systemy zarządzania bazą danych implementują B+ drzewa, z tym że na ogół pozostawiają w drzewie usunięte pozycje indeksu i danych. Na życzenie użytkownika lub w przypadku gdy zajętość w drzewie spada poniżej pewnego progu – następuje wywołanie operacji **REBUILD**, która tworzy indeks od nowa lub **COALESCE**, która łączy ze sobą sąsiednie strony o zajętości poniżej 50%.

Gdy na samym początku mamy dany duży zbiór rekordów, to wtedy zamiast powtarzania kolejnych operacji INSERT opłaca się zastosować algorytm **Bulk Loading**, którego działanie polega na posortowaniu rekordów, a następnie dobudowaniu nad posortowanym ciągiem kolejnych poziomów indeksowych drzewa B+.

**Dlaczego nie używamy zwykłych B drzew?**

Dla pełności tematu pokazujemy na rys. 9.11 schemat węzła zwykłego B drzewa (bez plusa).

Rys. 9.11 Węzeł zwykłego B drzewa

Wyszukiwanie w zwykłych B drzewach jest szybsze. Natomiast z powodu niejednolitości węzłów dla B drzew operacje INSERT i DELETE są bardziej skomplikowane. Dlatego w bazach danych preferuje się B+ drzewa.

|  |
| --- |
| Podsumowanie drzew   1. Zastosowania indeksów o strukturze drzewa:    * wyszukiwanie zakresowe,    * wyszukiwanie równościowe,    * sortowanie (np. ORDER BY),    * wyszukiwanie przy uzgadnianiu wierszy w trakcie złączania tabel. 2. Drzewo ISAM – struktura statyczna:    * struktura prostsza niż drzewa B+,    * modyfikowane są tylko liście,    * wymagane są strony nadmiarowe – mogące w przypadku pesymistycznym istotnie pogorszyć czas wykonywania instrukcji SELECT, chyba że akceptujemy wykonywanie operacji REBUILD, powiedzmy raz na dzień w nocy. 3. Drzewo B+ – struktura dynamiczna:    * w praktyce wysokość <=3,    * zakładając, że korzeń B+ drzewa zawsze jest trzymany w buforze RAM, koszt wyszukania pozycji danych rekordu wynosi co najwyżej 3 operacje We/Wy. Tę liczbę będziemy używać w dalszych oszacowaniach. |

**9.4** **Indeks  haszowany**

Indeks haszowany umożliwia szybkie wyszukiwanie rekordów w oparciu o wartość klucza wyszukiwania. Nie umożliwia wyszukiwania zakresowego. Indeks haszowany jest oparty na tej samej strukturze danych tablicy haszowanej co plik haszowany.

*Indeks haszowany* to struktura danych oparta na rozłożeniu pozycji danych do *segmentów* (ang. *bucket*) na podstawie wartości funkcji haszującej od klucza wyszukiwania. Segment składa się ze strony głównej i ewentualnie listy stron nadmiarowych. Wartość funkcji haszującej określa adres strony głównej segmentu, gdzie należy szukać rekordu. W przypadku większej liczby rekordów tworzone są strony nadmiarowe doczepiane do strony głównej.

|  |
| --- |
|  |

Rys. 9.12 Indeks haszowany

Przy prawidłowo dobranych: funkcji haszującej oraz wartości *M* prawie zawsze wszystkie pozycje danych znajdują się na stronie głównej i nie ma stron nadmiarowych (średnia długość listy stron dla jednej wartości funkcji haszującej wynosi 1.2). Stąd dla indeksu haszowanego średnia liczba operacji We/Wy przy wyszukiwaniu rekordu wynosi 1.2. (Używamy tu założenia, że adres segmentu dla danej wartości funkcji haszującej jest przechowywany bądź może być wyliczony w oparciu o informacje zapisane w buforze w RAM.)

Przykładowa funkcja haszująca:

**h**(*k*) = *k* mod *M* =  numer segmentu, do którego należy pozycja danych o kluczu *k* (*M* = #segmentów).

Przykład indeksu haszowanego po wykonaniu ciągu operacji INSERT (przy założeniu 2 rekordów na stronie) jest pokazany na rys. 9.13.

Rys. 9.13 Przykład indeksu haszowanego

Mówimy o *statycznym haszowaniu*, gdy liczba segmentów jest ustalana na samym początku i nie zmienia się. Wraz ze wzrostem liczby pozycji danych (rekordów danych) zwiększa się rozmiar segmentów i co za tym idzie czas wyszukiwania, który jest proporcjonalny do liczby stron w segmencie. W przypadku *dynamicznego haszowania* liczba segmentów dopasowuje się dynamicznie do liczby pozycji danych. Oto dwie metody dynamicznego haszowania:

1. Okresowa (np. raz na dzień) przebudowa REBUILD pliku haszowanego przez zmianę liczby segmentów. Jest pożądane aby funkcja haszująca miała własność umożliwiającą łatwe podwojenie rozmiaru indeksu przez podział każdego segmentu na dwa. Wspomniana powyżej funkcja mod ma tę własność.

2. Metoda rozszerzalnego haszowania

Unika się stron nadmiarowych przez dynamiczną rozbudowę struktury segmentów. Wprowadza się pośredni katalog kierujący do stron. Gdy strona segmentu się przepełni, rozdziela się ją na dwie.

|  |
| --- |
| Podsumowanie klasyfikacji indeksów  Klasyfikacja indeksów obejmuje cztery ortogonalne wymiary:   1. czy jest główny, jednoznaczny czy zwykły; 2. czy jest oparty na strukturze drzewowej czy haszowanej; 3. czy jest pogrupowany,  czy nie.   Własności indeksu wynikają z zaliczenia jego do odpowiedniej grupy uzyskanej przez wybór konkretnych czterech atrybutów z tych czterech wymiarów; na przykład: pogrupowany zwykły indeks drzewowy czy nie pogrupowany jednoznaczny indeks haszowany. |

**9.5** **Sortowanie zewnętrzne - wielofazowe sortowanie przez scalanie**

Rozpatrzmy konkretny problem: posortować 2GB danych w buforze pamięci RAM rozmiaru 100MB. Klasyczne metody sortowania wewnętrznego w pamięci RAM nie dadzą się bezpośrednio zastosować.

W bazach danych tego typu problem sortowania pojawia się często. Oto przykłady:

1. Klauzula ORDER BY - dane są wymagane w pewnym porządku.
2. Budowa indeksu - początkowego B+ drzewa dla zbioru rekordów (w ramach metody BULK LOADING lub REBUILD).
3. Złączanie tabel metodą sortowania przez scalanie.
4. Realizacja operatorów DISTINCT, GROUP BY, UNION DISTINCT, EXCEPT.

*Sortowanie zewnętrzne* to problem sortowania pliku rekordów, który nie mieści się w pamięci wewnętrznej RAM.

Metodą stosowaną w bazach danych jest *wielofazowe sortowanie zewnętrzne przez scalanie*. Algorytm składa się z dwóch etapów:

Etap 1: Tworzenie początkowych uporządkowanych bloków rekordów (ang. *run*) – za pomocą jednej z metod [sortowania wewnętrznego](javascript:popUp('fig9/ok1.htm',500,130)) i dystrybucja ich do dwóch lub więcej obszarów dyskowych. Oznaczmy ich liczbę przez *B*.

Etap 2: Wielofazowe odczytywanie obszarów dyskowych z danymi, scalanie kolejnych bloków z zapisywaniem ich do obszarów dyskowych – powtarzane dopóki nie otrzyma się pojedynczego, uporządkowanego bloku rekordów.

Rys. 9.15 Scalanie danych w puli buforów

W każdej fazie odczytujemy i zapisujemy każdą stronę w pliku.

Rys.9.16 Przykład wielofazowego sortowania zewnętrznego przez scalanie

Liczba operacji We/Wy przy wykonywaniu sortowania zewnętrznego jest <= 2N\*#*faz* gdzie *N* jest liczbą stron w pliku rekordówa współczynnik 2 bierze się stąd, że każdą stronę trzeba odczytać i zapisać.

W praktyce średnio liczba faz scalania nie przekracza 3. Łącznie z fazą tworzenia początkowych bloków liczba wszystkich faz jest średnio co najwyżej 4. Zatem liczba operacji We/Wy potrzebnych do wykonania sortowania zewnętrznego jest co najwyżej *8N* (*N* jest liczbą stron w pliku rekordów).

**Sortowanie zewnętrzne - zastosowanie B+ drzew**

Załóżmy, że na pliku rekordów jest założony indeks na B+ drzewie z uporządkowaniem względem wartości klucza sortowania. Czy przejście po liściach tego drzewa od lewej strony do prawej z wypisywaniem kolejnych rekordów jest dobrą metodą sortowania względem wartości klucza sortowania (klucza indeksu)? Tak, ale tylko wtedy, gdy indeks jest pogrupowany!

Jeśli indeks nie jest pogrupowany trzeba ściągnąć do pamięci wewnętrznej tyle stron dyskowych ile jest rekordów (a nie tyle, na ilu stronach są zapisane rekordy w pliku!) – co przy dużej liczbie rekordów może dać czas większy niż czas wielofazowego sortowania przez scalanie ~ 8*N* operacji We-Wy, gdzie *N* jest liczbą stron w pliku rekordów.

|  |
| --- |
|  |

Rys. 9.17 Ilustracja indeksu nie pogrupowanego.

Zaletą sortowania za pomocą B+ drzew jest *interaktywny* charakter wyznaczania wyników. W przeciwieństwie do sortowania zewnętrznego wypisywanie wyników na ekran użytkownika może się rozpocząć prawie natychmiast bez konieczności obliczania całości. Gdy użytkownik ogląda pierwszą partię wyników, w tym czasie system może wyliczać następną ich część.

**9.7 Podsumowanie**

W wykładzie tym zaznajomiliśmy się z budową indeksów oraz z sortowaniem zewnętrznym.

**9.8 Słownik pojęć**

[klucz wyszukiwania](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/Klucz%20wyszukiwania) (rekordu) - wybrane pola rekordu względem których ma odbywać się wyszukiwanie. Może być wiele kluczy wyszukiwania rekordu.

[indeks](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Indeksy) - struktura danych na dysku umożliwiająca szybkie wyszukiwanie danych w bazie danych na podstawie klucza wyszukiwania rekordu np. *nazwiska osoby*. Indeks jest zapisywany w pliku nazywanym *indeksowym*.

[plik indeksowy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Indeksy) - plik, w którym jest przechowywany indeks.

[indeks pogrupowany](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Ipogrup) - indeks, który organizuje pozycje danych i rekordy danych w kolejności uporządkowanej względem wartości klucza wyszukiwania. Oznacza to, że rekordy o tej samej wartości klucza lub zbliżonej znajdują się na tej samej stronie lub tylko kilku stronach dyskowych. W przeciwnym razie indeks nazywamy niepogrupowanym. Może być tylko jeden indeks pogrupowany ale wiele niepogrupowanych.

[indeks główny](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Gdy klucz) - indeks, którego klucz wyszukiwania zawiera klucz główny. Może być tylko jeden indeks główny.

[indeks jednoznaczny](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Gdy jedn) - indeks, którego klucz wyszukiwania zawiera klucz jednoznaczny. Może być wiele indeksów jednoznacznych.

[drzewo ISAM](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#ISAM) - implementacja indeksu w postaci statycznego drzewa opartego na wyszukiwaniu binarnym.

[drzewo B+](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Bdrzewo) - implementacja indeksu w postaci dynamicznego drzewa opartego na wyszukiwaniu binarnym.

[indeks haszowany](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Tablica haszowana) - implementacja indeksu w postaci tablicy haszowanej.

[sortowanie zewnętrzne](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Sortowanie) - sortowanie pliku danych, który nie mieści się w pamięci wewnętrznej RAM.

[wielofazowe sortowanie zewnętrzne przez scalanie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Sortowanie) - podstawowa metoda sortowania zewnętrznego. Składa się z dwóch etapów: tworzenia początkowych uporządkowanych bloków rekordów – za pomocą jednej z metod sortowania wewnętrznego oraz wielofazowego odczytywania uporządkowanych bloków rekordów i sukcesywnego ich scalania, dopóki nie otrzyma się pojedynczego uporządkowanego bloku rekordów.

**Wykład 10**

***Wykonywanie zapytań***

**Streszczenie**

Wykład składa się z dwóch części. Pierwsza część zaznajamia z metodami realizacji operatorów relacyjnych, z których składają się zapytania SQL.

W drugiej części są omówione zasady wykonywania zapytań. Przed wykonaniem zapytania przez SZBD włącza się moduł *optymalizatora zapytań*, którego zadaniem jest znaleźć możliwie najlepszy *plan wykonania* tego zapytania.

**10.1** **Operatory relacyjne**

Każdą instrukcję SQL można rozłożyć na części przy czym każda z tych części jest związana z użyciem jednego operatora relacyjnego działającego na jednej lub więcej tabeli. Oto podstawowe operatory relacyjne:

1. *Selekcja -* selekcja podzbioru wierszy (określona przez warunek w klauzuli WHERE).
2. *Projekcja* - pominięcie z wyniku pewnych kolumn (klauzule SELECT i SELECT DISTINCT).
3. *Złączenie* - złączenie tabel (relacji).
4. *Operatory algebraiczne* UNION (DISTINCT) i EXCEPT na tabelach (relacjach).
5. *Agregacja* - zastosowanie funkcji agregujących SUM, MIN, itd. i klauzuli GROUP BY.

Przypominamy, że na koszt realizacji operacji bazodanowych największy wpływ ma liczba stron dyskowych przesyłanych między dyskiem a pamięcią wewnętrzną.

**Implementacja selekcji**

Rozważmy przykład.

SELECT \*  
FROM Emp e  
WHERE e.Ename < 'C' AND e.Sal > 1000;

Załóżmy, że warunek w klauzuli WHERE ma postać koniunkcji. (Wiadomo z wykładu z matematyki dyskretnej, że każdą formułę logiczną można sprowadzić do koniunkcji alternatyw prostych warunków – czyli do tzw. *postaci normalnej*.) Są dwie metody realizacji takiej selekcji:

1. przejście sekwencyjne całego pliku rekordów (operacja *scan*),
2. wybór najbardziej "selektywnego" pola, które występuje w prostym warunku koniunkcji i na którym jest założony indeks, po czym wyszukanie przez ten indeks (tzn. sprowadzamy warunek WHERE do postaci koniunkcji: *p* AND *Reszta* gdzie *p* jest prostym predykatem dotyczącym pojedynczego pola w tabeli). Najlepiej, gdy indeks jest haszowany dla selekcji równościowej oraz gdy indeks jest na B+ drzewie dla selekcji zakresowej.

Wybór drugiej metody jest wskazany w trzech przypadkach:

1. gdy indeks jest główny lub jednoznaczny,
2. gdy indeks jest pogrupowany oraz wyszukiwanie jest zakresowe,
3. gdy oszacowanie rozmiaru zbioru wyszukiwanych przez indeks wierszy wskazuje na niewielką ich liczbę (to znaczy gdy wyszukiwanie przez indeks jest selektywne) - powiedzmy do 5-10%.

W przypadku zastosowania indeksu nie pogrupowanego wskazane jest, jeśli możliwe, posortowanie identyfikatorów zwracanych rekordów według adresów stron dyskowych i ściągnięcie każdej potrzebnej strony tylko jeden raz.

**Implementacja projekcji**

Rozważmy przykład:

      SELECT DISTINCT e.Job  
    FROM Emp e;

Gdy nie ma operatora DISTINCT – wystarczy przejść cały plik (*scan*) i przepisać wartości wyrażeń na liście SELECT.

Problem stanowi tylko klauzula SELECT DISTINCT, która wymaga eliminacji powtórzeń co można uzyskać przez posortowanie zbioru wynikowego.

Do eliminacji powtórzeń można też użyć metody polegającej na rozrzuceniu wynikowych wartości do segmentów tablicy haszowanej  poprzez zastosowanie funkcji haszującej. Eliminacja powtórzeń odbywa się wtedy w ramach jednego segmentu dla znacznie mniejszej liczby rekordów – można wtedy użyć sortowania wewnętrznego albo kolejnego rozrzucenia do segmentów - jeśli rozmiar segmentu jest zbyt duży aby zmieścił się w pamięci wewnętrznej.

**Implementacja operatorów zbiorowych**

Operatory UNION (DISTINCT) i EXCEPT są realizowane podobnie jak SELECT DISTINCT – wymagają usunięcia powtórzeń albo przez sortowanie zewnętrzne albo przez haszowanie.

Operator INTERSECT jest szczególnym przypadkiem złączenia. Na przykład, instrukcja:

SELECT Deptno FROM Dept  
INTERSECT  
SELECT Deptno FROM Emp;

jest równoważna:

SELECT d.Deptno  
FROM Dept d, Emp e  
WHERE d.Deptno=e.Deptno;

Metody realizacji operatora złączenia są omówione dalej na tym wykładzie.

**Implementacja agregacji**

Bez GROUP BY

Rozważmy przykład.

SELECT AVG(e.Sal)  
FROM Emp e;

Wymagane jest albo przejście całego pliku rekordów.

Z GROUP BY

Rozważmy przykład.

SELECT e.Job, AVG(e.Sal)  
FROM Emp e  
GROUP BY e.Job;

Można posortować według wartości pól grupujących, przechodząc cały plik rekordów. Inną metodą realizacji grupowania może być użycie haszowania.

**Implementacja złączenia tabel**

W celu zaprezentowania metod złączania tabel rozważymy dla uproszczenia złączenie równościowe z jedną kolumną złączenia. Założymy mianowicie, że interesuje nas złączenie dowolnych tabel *E* i *D* względem *i*-tej kolumny w *E* oraz *j*-tej kolumny w *D*. To znaczy przyjmiemy, że warunkiem złączenia jest *Ei*=*Dj*.

Rozważmy przykład.

SELECT E.Ename, D.Loc  
FROM Emp E, Dept D  
WHERE E.Deptno = D.Deptno;

Przegląd metod złączania tabel zaczynamy od najprostszego algorytmu opartego na rozważeniu wszystkich możliwych kombinacji wierszy tabel *E* i *D*.

**Algorytm Simple Nested Loops Join**

foreach row e in E do  
foreach row d in D do  
if ei == dj then dodaj <e, d> do obliczanego wyniku;

Dla każdego wiersza tabeli *E*, przeglądamy wszystkie wiersze tabeli *D*.

Tabela *E* nazywa się *tabelą zewnętrzną* (złączenia), tabela *D* nazywa się *tabelą wewnętrzną* (złączenia).

Oczywiste ulepszenie polega na zastosowaniu postępowania:  dla każdej strony w *E*, sprowadź każdą stronę w *D*. Będziemy dalej używać metody Simple Nested Loops Join razem z tym ulepszeniem.

**Algorytm Index Nested Loops Join**

{*E* – tabela zewnętrzna złączenia, *D* – tabela wewnętrzna złączenia, na kolumnie złączenia *Dj*  jest założony indeks}

foreach row *e* in *E* do  
{weź wartość  *ei*kolumny złączenia *Ei* i poprzez indeks na *Dj* wyznacz wszystkie wiersze *d* mające tę samą wartość w kolumnie złączenia *Dj* (ei == dj) – połącz oba wiersze <*e*,*d*> i dodaj <*e*, *d*> do obliczanego wyniku}

Rys. 10.1 Ilustracja algorytmu Index Nested Loops Join

Dla każdego wiersza w *E* najpierw wyszukujemy pozycję danych w indeksie na kolumnie *Dj* . Mając pozycję danych w indeksie na kolumnie *Dj*, przechodzimy do wierszy w *D.*

Wydajność tej metody zależy od rodzaju indeksu. W przypadku indeksu głównego, jednoznacznego, pogrupowanego jej koszt jest względnie niewielki - liniowy względem liczby wierszy w tabeli zewnętrznej i nie zależy od liczby wierszy w tabeli wewnętrznej.

Natomiast w pozostałych przypadkach koszt istotnie zależy od selektywności wyszukiwania przez indeks i może być bardzo duży (nawet kwadratowy względem liczby wierszy w tabeli zewnętrznej i wewnętrznej).

**Użycie klastra tabel**

Można się lepiej przygotować do często występujących złączeń tabel przez umieszczenie ich w jednym klastrze z kluczem będącym kolumną złączenia obu tabel. Połączenie tabel w klaster powoduje, że złączenie odbywa się tak jakby to była pojedyncza operacja przejścia jednej tabeli. Realizacja naszego przykładowego zapytania zostanie przyśpieszona jeśli obie tabele *Emp* i *Dept* umieścimy w jednym klastrze tak jak to zrobliśmy na [wykładzie 3](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Klaster).

Więcej informacji o implementacji i zastosowaniu klastra będzie podane na następnym wykładzie przy okazji omawiania struktur indeksowych w systemie Oracle.

W przypadku złączania gdy brak klastra albo odpowiedniego indeksu, SZBD stosuje jedną z dwóch poniższych metod.

**Algorytm Sort Merge Join**

|  |
| --- |
| Posortuj wiersze obu tabel względem wartości kolumn złączenia. Dokonaj scalenia obu ciągów produkując wynikowy zbiór wierszy. |

**Algorytm Hash Join**

|  |
| --- |
| Rozrzuć wiersze tabel *E* i *D* do segmentów używając funkcji haszującej *h1* określonej na wartościach kolumn złączenia. Powstające segmenty zapisz na dysku. |

Rys. 10.2 Ilustracja algorytmu Hash Join

|  |
| --- |
| Rozpatruj kolejno odpowiadające sobie segmenty tabel *E* i *D* szukając par wierszy *e* i *d* dla których ei=dj. Nie trzeba już uzgadniać wierszy pochodzących z różnych segmentów – bo na takich wierszach wartość funkcji haszującej jest różna a więc i wartości w kolumnach złączenia też są różne. Dla każdej uzgodnionej pary wierszy, dodaj <*e*, *d*> do obliczanego wyniku.  Jeśli rozmiary jednej z par odpowiadających sobie segmentów są tak duże, że żaden z nich nie mieści się w pamięci wewnętrznej, należy do nich zastosować drugą funkcję haszującą *h2* (istotnie inną niż *h1*) i powtórzyć postępowanie opisane powyżej. |

**Złączanie tabel obiektowo-relacyjnych**

Przy złączaniu tabel obiektowo-relacyjnych możemy skorzystać z referencji i kolekcji referencji. Obie operacje zarówno przejście przez referencję jak i przejście przez kolekcję referencji są szybsze niż odpowiednie operacje przejścia przez indeksy zewnętrzne dla tabel relacyjnych. Wadą referencji i kolekcji referencji (oprócz utraty niezależności od wartości modelu fizycznego) jest dodatkowy narzut czasowy i miejsca na dysku związany z reprezentacją i przetwarzaniem kolekcji (rekord zawierający obszerną kolekcję referencji może wymagać więcej niż jednej strony do zapisu; przejście do rekordów wskazywanych przez referencje wymaga sprowadzenia tylu stron ile jest referencji w kolekcji.)

|  |
| --- |
| Porównanie metod złączania  Gdy SZBD chce wykonać złączenie tabel, rozważa możliwe metody w następującej kolejności:   * Gdy jest zbudowany klaster, złączenie tabel sprowadza się do przejścia klastra tak jakby to była jedna tabela. Kluczem klastra powinna być kolumna złączania tabel. * W przypadku złączania tabel, których powiązanie jest określone nie przez związek klucz obcy-klucz główny ale bezpośrednio przez referencje, możemy zastosować przejścia przez te referencje. Przeciwskazaniem może być tylko przewidywana duża liczba referencji do przejścia (duża liczba stron do sprowadzenia do pamięci wewnętrznej). * Metoda Simple Nested Loops Join jest prosta i to jest jej podstawowa zaleta. Może być używana w sytuacji, gdy jedna ze złączanych tabel ma niewielki rozmiar. * Na koszt metody Index Nested Loops Join istotny wpływ mają własności indeksu np. czy jest pogrupowany, czy jest selektywny jak np. indeks główny, jednoznaczny. W połączeniu z selekcją na tabeli zewnętrznej złączenia bywa najszybszą metodą. * Metoda Hash Join  wypada lepiej w oszacowaniach średniej liczby operacji We/We niż metoda Sort-Merge ale w przypadku pesymistycznym może się okazać bardzo zła. * Metoda Hash Join wypada lepiej od Sort-Merge gdy rozmiary sortowanych plików zasadniczo się różnią. Jest łatwiejsza do zrównoleglenia niż Sort-Merge. * Metoda Sort Merge jest lepsza gdy rozmiary sortowanych plików są zbliżone. Jest mniej wrażliwa na mało losowe dane oraz rezultat złączenia jest posortowany. |

**Strategia tylko-indeks**

Przy realizacji operatorów selekcji, projekcji, agregowania i grupowania w przypadku gdy wszystkie elementy klauzul instrukcji SELECT należą do klucza wyszukiwania jednego indeksu – można ograniczyć się do przejścia tylko pliku indeksowego zamiast całego pliku rekordów. Metoda ta nosi nazwę *strategii tylko-indeks*. Jej zastosowanie wymaga aby wszystkie potrzebne do wyznaczenia wyniku zapytania wiersze tabeli były indeksowane.

Na przykład, jeśli mamy indeks założony na kolumnach *Ename* i *Comm* tabeli *Emp* oraz w kolumnie *Ename* nie występuje pseudo-wartość NULL (np. z powodu użycia więzów spójności NOT NULL na kolumnie *Ename* w tabeli *Emp*), to możemy strategię tylko-indeks zastosować do obliczenia wyniku instrukcji:

SELECT e.Ename, e.Comm  
FROM Emp e;

jak również w przypadku indeksu założonego tylko na kolumnie Comm w przypadku instrukcji:

SELECT Avg(e.Comm)  
FROM Emp e;

ponieważ to czy w kolumnie *Comm* występuje NULL nie ma tu znaczenia, bowiem operator Avg nie bierze w ogóle pod uwagę pseudo-wartości NULL.

**10.2** **Optymalizacja zapytań**

Zapytanie SQL ma charakter deklaratywny: określa **co** ma być wyznaczone w bazie danych, a nie **jak** to ma być znalezione. Dla każdego zapytania istnieje wiele sposobów jego realizacji. Który sposób jest najlepszy, zależy od dodatkowych okoliczności. SZBD rozważa różne alternatywy, szacuje ich koszt oraz wybiera możliwie najlepszy, "optymalny" plan. Proces ten nazywa się *optymalizacją zapytania* a moduł go realizujący *optymalizatorem zapytań*.

1. W SZBD zapytaniem najpierw zajmuje się ***parser***,dokonujący analizy składniowej zapytania.
2. Po analizie składniowej włącza się ***optymalizator zapytań*** w skład którego wchodzą dwa główne moduły:
   * ***generator planów*** - moduł generujący możliwe plany wykonania zapytania, i
   * ***estymator kosztu*** - moduł obliczający przybliżony koszt wykonania zapytania według danego planu.
3. Przy szacowaniu kosztu planu estymator korzysta z informacji statystycznych zapisanych w słowniku danych (katalogu systemowym) takich jak: liczba rekordów w pliku, liczba stron na których są zapisane rekordy w pliku, liczba różnych wartości w kolumnie, rozkład wartości w kolumnie (histogram).
4. Optymalizator wybiera plan o najniższym koszcie i przekazuje go do ***ewaluatora planu*** - modułu wykonującego zapytanie.
5. Gdy zapytanie zostało wcześniej przeanalizowane i kontekst jego użycia się nie zmienił, system może użyć wyliczony wcześniej plan.

Rys. 10.3 Schemat optymalizatora zapytań

*Plan wykonania zapytania* obejmuje:

* drzewo operatorów SQL tego zapytania,
* metody dostępu do każdego wystąpienia tabeli w tym zapytaniu,
* metody realizacji dla każdego wystąpienia operatora relacyjnego w zapytaniu.

Niektóre plany wykonania zapytania nie korzystają z tymczasowych tabel - *działając* *w miejscu*. Ich działanie polega na tym, że przy określonym sposobie dostępu do rekordów każdej tabeli utrzymuje się tylko *kursory* przebiegające rekordy w plikach (ewentualnie pozycje danych w pliku indeksowym) bez zapisywania pomocniczych tabel. Unikamy w ten sposób zapisywania tymczasowych wyników na dysk aby je potem sprowadzać powtórnie do pamięci RAM.

Plany mające postać drzewa skierowanego w lewo (omawiane dalej) w powiązaniu z metodami Simple Nested Loops Join i Index Nested Loops Join umożliwiają działanie w miejscu. Natomiast metody Sort-Merge Join i Hash Join wymagają użycia pomocniczych plików na dysku, więc nie działają w miejscu. Zastosowanie klastra lub kolekcji referencji zamiast operatora złączenia też umożliwia działanie w miejscu.

Jeśli chodzi o połączenie ze sobą operatorów w planie wykonania zapytania, to jest używana zasada *przetwarzania potokowego*. Wynik jednego operatora jest przekazywany na wejście drugiego operatora. Oznacza to, że nie jest potrzebna tymczasowa tabela, więc też mamy do czynienia z działaniem w miejscu.

W przykładach będziemy używać oznaczeń na operatory SQL zebranych w Tabeli 10.1.

|  |  |
| --- | --- |
| **Operator** | **Symbol** |
| Selekcja |  |
| Projekcja |  |
| Złączenie |  |

Tab. 10.1 Oznaczenia operatorów relacyjnych

Zapytanie jest przedstawiane w postaci drzewa operatorów SQL. Na przykład, zapytanie

SELECT E.Ename  
FROM Emp E, Dept D  
WHERE E.Deptno=D.Deptno AND  
    E.Mgr=100 AND D.Loc='Oz';

jest reprezentowane przez drzewo:

Rys. 10.4 Drzewo operatorów instrukcji SQL

Plan 1

Najprostszy plan wykonania tego zapytania to:

* wykonywać złączenie tabel *Emp E* i *Dept D* metodą Simple Nested Loops Join i dla każdego wiersza złączenia sprawdzać warunek E.Mgr=100 AND D.Loc='Oz';
* jeśli warunek zachodzi, wydobywać wartość z kolumny *E.Ename* i przekazywać ją do zbioru wyników.

Rys. 10.5 Najprostszy plan wykonania zapytania

Plan ten działa w miejscu (bez tymczasowych tabel) i nie wykorzystuje indeksów. Nie jest zbyt dobry.

Plan 2

Rozważmy następujący alternatywny plan:

* osobno przechodzimy *Emp E* i *Dept D* z jednoczesną selekcją E.Mgr=100 oraz odpowiednio  D.Loc='Oz'; wyniki selekcji zapisujemy w dwóch tymczasowych tabelach;
* stosujemy Sort-Merge Join do złączenia (alternatywnie, zamiast Sort-Merge Join możemy użyć Hash Join.);
* gdy wyniki selekcji mieszczą się w pamięci wewnętrznej stosujemy od razu uzgodnienie wierszy z obu zestawów.

Rys. 10.6 Plan stosujący ograniczanie przed złączeniem

Główna różnica z poprzednim planem polega na tym, że zanim rozpocznie się złączanie rekordów - najpierw są wykonywane selekcje. Jest nadzieja, że istotnie ograniczą one liczbę złączanych rekordów w porównaniu z poprzednim planem. Przed złączaniem, oprócz selekcji, można byłoby jeszcze dokonywać eliminacji nie używanych dalej kolumn czyli, inaczej mówiąc, moglibyśmy zastosować projekcje *Ename*,*Deptno* dla *Emp* oraz *Deptno* dla *Dept*. W ten sposób zmniejszylibyśmy rozmiar tabel tymczasowych *T1* i *T2* a co za tym idzie również liczbę operacji We/Wy.

Drobna zmiana w powyższym planie polegałaby na zastąpieniu operacji *scan* wyszukiwaniem przez indeksy odpowiednio na kolumnach *E.Mgr* i *D.Loc*. W przypadku indeksów: pogrupowanego na B+ drzewie lub o dobrej selektywności zmiana istotnie przyśpieszyłaby cały proces obliczeniowy.

Plan 3

Rozważmy jeszcze jeden alternatywny plan tym razem taki, w którym złączenie jest oparte na indeksie (czyli jest stosowana metoda Index Nested Loops Join). Dodatkowo na tabeli zewnętrznej złączenia stosujemy selekcję przez indeks, która może istotnie ograniczyć liczbę wierszy rozpatrywanych jako kandydaci do złączenia.

* Korzystając z indeksu haszowanego na *E.Mgr* wybieramy wiersze spełniające warunek E.Mgr=100.
* Dla każdego otrzymanego wiersza z *E,* korzystając z indeksu na *D.Deptno*, znajdujemy pasujące (D.Deptno=E.Deptno) do niego wiersze z tabeli *D*.
* Złączamy ze sobą oba wiersze, sprawdzamy czy zachodzi warunek D.Loc='Oz' a na koniec dokonujemy projekcji na kolumnę *E.Ename*.

Rys. 10.7 Plan zorientowany na stosowanie indeksów i przetwarzanie potokowe

Zauważmy, że:

* *Emp E* jest tabelą zewnętrzną złączenia. *Dept D* jest tabelą wewnętrzną złączenia.
* Stosujemy metodę *Index Nested Loops Join* razem z przetwarzaniem potokowym (pipelining). Nie ma potrzeby zapisywania wyniku selekcji jako tymczasowej relacji - przetwarzanie jest potokowe i działa w miejscu.
* Istotna jest selektywność wyszukiwania względem warunku E.Mgr=100 oznaczająca, że liczba pracowników mających kierownika o numerze 100 jest niewielka.
  + W przypadku złej selektywności lepszy będzie *scan* całej tabeli *E*.
* Zauważmy, że przy przetwarzaniu potokowym każdy węzeł może działać równolegle z innymi, przekazując wynik swojego działania kolejnemu węzłowi a sam podejmując przetwarzanie kolejnego elementu: albo wygenerowanego przez siebie albo otrzymanego na wejściu od innego węzła.

Generowanie przez optymalizator planów wykonania zapytania

Jest dużo możliwych planów wykonania jednego zapytania, ponieważ kolejność wykonywania złączeń jest dowolna na podstawie praw przemienności i łączności operatora złączenia.

Ze względu na dużą liczbę możliwości optymalizator ogranicza się do pewnej klasy wszystkich drzew tzw. *drzew skierowanych w lewo*.

Drzewo skierowane w lewo zawiera "rdzeń" w postaci gałęzi węzłów, na której każdy kolejny węzeł jest lewym następnikiem poprzedniego i tylko węzły leżące na tej gałęzi mogą mieć stopień dwa odpowiadający operatorowi złączenia (pozostałe węzły w drzewie mają stopień 0 lub 1).

Z trzech drzew na rysunku 10.8 tylko środkowe jest skierowane w lewo. Drzewa skierowane w lewo dają plany umożliwiające "potokowe" wykonywanie zapytania "w miejscu" tj. bez tymczasowych plików.

Oczywiście złączanie metodą Sort Merge Join czy Hash Join wymaga zapisywania pomocniczych plików, więc nawet jeśli zastosujemy plan oparty o drzewo skierowane w lewo, to wykonanie zapytania nie będzie działać potokowo „w miejscu”.

Rys. 10.8 Tylko środkowe drzewo jest skierowane w lewo

Zauważmy, że dla zapytania zawierającego *n-1* operatorów złączenia jest co najmniej *n*! drzew skierowanych w lewo odpowiadających *n*! permutacjom operatorów złączenia.

Przykład

Faza 1: Generujemy drzewo planu wykonania zapytania: *Emp* - tabela zewnętrzna, *Dept* - tabela wewnętrzna. (Drugie możliwe drzewo to wariant pierwszego drzewa, w którym z lewej strony znajduje się tabela *Dept* a z prawej *Emp*.)

Rys. 10.8 Drzewo operatorów instrukcji SQL

Faza 2: Analiza planów dostępu do wierszy tabel *Emp* i *Dept*:

**Emp**:

* Indeks haszowany na *Emp*.*Mgr*
* Indeks haszowany na *Emp.Deptno*
* Scan

**Dept**:

* B+ drzewo na *Dept.Loc*
* Indeks główny haszowany na *Dept.Deptno*
* Scan

Faza 3: Rozpatrujemy każdy plan dostępu, bierzemy pod uwagę możliwe dla tego planu dostępu metody złączenia (SNLJ, INLJ, SMJ, HJ) i liczymy orientacyjny koszt korzystając ze statystyk zebranych przez system takich jak liczba wierszy w tabeli, liczba stron w pliku z danymi i w pliku indeksu.

**Podzapytania**

Podzapytania są optymalizowane niezależnie od głównego zapytania. Główne zapytanie jest optymalizowane z branym pod uwagę kosztem „wywoływanych” podzapytań. Alternatywnie, podzapytanie jest sprowadzane do złączeń i optymalizowane łącznie z całym zapytaniem.

Ogólne strategie optymalizacyjne

* Dokonuj jak najwcześniej selekcji zmniejszającej liczbę rozważanych rekordów – istotne szczególnie wtedy gdy wynik selekcji przekazujemy do złączenia – które jest najbardziej kosztowną operacją. W szczególnym przypadku wynik selekcji może się cały dać zapisać w buforach pamięci RAM co przyśpieszyłoby istotnie wykonywanie zapytania.
* Do wykonania selekcji stosuj indeks - najlepiej indeks główny, jednoznaczny, pogrupowany lub względem selektywnego warunku - powiedzmy wybierającego mniej niż 5-10% wszystkich rekordów w pliku. Jeśli takiego indeksu nie da się zastosować, zamiast wyszukiwać przez indeks, bardziej opłaca się sekwencyjnie przejrzeć cały plik (scan) z wyborem rekordów spełniających zadany warunek.
* Staraj się wiązać selekcje z iloczynem kartezjańskim, w celu zidentyfikowania rodzaju złączenia tabel.
* Do wykonania złączenia stosuj indeks na tabeli wewnętrznej (preferowany indeks główny, jednoznaczny, pogrupowany lub względem selektywnego warunku).
* Wybierz plan działający "w miejscu" bez tymczasowych tabel np. w postaci drzewa skierowanego w lewo. Stosuj przetwarzanie potokowe (pipelining) do wykonywania ciągu operatorów jednoargumentowych jak selekcje i projekcje.
* Zamiast operatora złączenia zastosuj klaster, który też umożliwia działanie w miejscu.
* Jeśli to możliwe - ograniczaj się do przechodzenia indeksów a nie tabel (strategia *tylko-indeks*).
* Wyszukuj wspólne podwyrażenia i obliczaj je tylko raz.
* Przetwórz wstępnie plik we właściwy sposób (indeksy, sortowanie, haszowanie).
* Gromadź statystyki ilościowe dotyczące tabel, kolumn i indeksów – w tym histogramy to znaczy dystrybucje wartości w kolumnach tabel. Korzystaj ze statystyk gromadzonych w katalogu systemowym.
* Szacuj koszt każdego planu i wybieraj plan o najmniejszym koszcie. Przy obliczaniu kosztu planu szacuj koszt realizacji każdego operatora relacyjnego i rozmiar jego wyników.
* Zapamiętuj plan wykonania zapytania, aby móc ten plan zastosować w tych samych warunkach.

**10.4 Podsumowanie**

W wykładzie 10 zostały omówione metody implementacji operatorów relacyjnych stanowiących cegiełki, z których składa się całe zapytanie SQL. Implementację tych operatorów można dokładnie dostroić. Zostały omówione podstawowe zasady wykonywania zapytań w tym podstawowy problem optymalizacji zapytania. 

**10.5 Słownik pojęć**

[operator relacyjny](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Operatory relacyjne) - selekcja, projekcja, złączenie, suma i agregacja. Wykonywanie zapytania SQL sprowadza się do złożenia implementacji tych podstawowych operatorów.

[selekcja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Implementacja selekcji) - operator relacyjny polegający na ograniczeniu pliku rekordów do podzbioru.

[projekcja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Implementacja projekcji) - operator relacyjny polegający na ograniczeniu pliku rekordów do wybranych pól.

[suma (union)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Implementacja operatorów zbiorowych) - operator relacyjny polegający na zsumowaniu dwóch plików rekordów.

[agregacja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Implementacja agregacji) - operator relacyjny polegający na wyliczeniu statystyk na danym pliku rekordów według podziału na grupy rekordów.

[złączenie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#Implementa) - operator relacyjny polegający na połączeniu dwóch plików rekordów według wartości wspólnych pól.

[Simple Nested Loops Join](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#NLJ) - metoda złączenia polegająca na rozpatrzeniu po kolei każdego rekordu z pierwszego pliku rekordów a z kolei dla niego przejrzenia wszystkich rekordów z drugiego pliku w poszukiwaniu wszystkich par rekordów, które dadzą się złączyć ze sobą.

[Index Nested Loops Join](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#INLJ) - metoda złączenia polegająca na rozpatrzeniu po kolei każdego rekordu z pierwszego pliku rekordów a z kolei dla niego zastosowania wyszukiwania przez indeks w celu wyznaczenia wszystkich rekordów z drugiego pliku, które dadzą się z nim złączyć.

[Sort Merge Join](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#SMJ) - metoda złączenia polegająca na posortowaniu plików rekordów według wartości w kolumnach złączenia a następnie dokonaniu ich scalenia.

[Hash Join](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w9.htm#HJ) - metoda złączenia polegająca na rozrzuceniu rekordów w złączanych plikach rekordów według wartości funkcji haszującej na wartościach w kolumnach złączenia a następnie dokonaniu ich scalenia.

[optymalizacja zapytań](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w10.htm#Optymalizacja zapyt) -  zadanie wykonywane przez SZBD polegające na analizie różnych planów wykonania zapytania SQL i wyboru "najoptymalniejszego".

[optymalizator zapytań](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w10.htm#Optymalizacja zapyt) - moduł SZBD, którego zadaniem jest znaleźć możliwie najlepszy ("optymalny") plan wykonania zapytania SQL.

**Wykład 11**

***Planowanie indeksów***

**Streszczenie**

Wykład składa się z dwóch części. W pierwszej części są omówione zasady projektowania fizycznego schematu bazy danych i jego dostrajanie: jakie założyć indeksy, czy pogrupować tabele w klastry, jak poprawić schemat logiczny bazy danych z punktu widzenia szybkości działania zapytań.

W drugiej części, jako uzupełnienie do tematu budowy i wyboru indeksów w bazie danych, są przedstawione rodzaje indeksów realizowane w SZBD Oracle.

**11.1** **Projektowanie fizycznej bazy danych**

**Ulepszanie schematu tabel i postacie normalne**

Redundancja na poziomie logicznym (schematu tabel) pociąga za sobą redundancję zapisu na nośniku danych, bo wymagane jest więcej miejsca na dysku, oraz anomalie przy wstawianiu, usuwaniu i aktualizacji danych. Redundacje w schemacie tabel są eliminowane przy użyciu analizy zależności funkcyjnych i dekompozycji tabel na mniejsze. Jednak wówczas może się okazać, że do wykonania zapytania będzie potrzebne złączenie dwóch lub więcej tabel, co może istotnie zwiększyć czas realizacji zapytania.

1. Jeśli wszystkie tabele są w postaci normalnej BCNF (Boyce'a-Codda), to są one wolne od redundancji związanych z zależnościami funkcyjnymi.
2. Jeśli tabela nie jest w postaci BCNF, staramy się dokonać jej dekompozycji na zbiór tabel w postaci BCNF:
   * Jeśli przy dekompozycji nie daje się zachować zależności funkcyjnych – poprzestajemy na 3-ciej postaci normalnej;
   * Jednocześnie z dekompozycjami bierzemy pod uwagę wymagania dotyczące szybkości działania zapytań na bazie danych; jeśli wymaganie szybkości działania ma charakter priorytetowy a dekompozycja istotnie spowalnia wykonanie zapytania – nie doprowadzamy procesu dekompozycji do końca, albo dokonujemy logicznej denormalizacji - połączenia dwóch rozdzielonych tabel w jedną. Aby przyśpieszyć złączanie tabel - alternatywą może być:
     + zbudowanie klastra tabel (fizyczna denormalizacja),
     + albo, jeśli zbudowanie klastra nie jest możliwe,  skorzystanie z kolumn typu referencji i kolekcji referencji (w miejscu odpowiednio kolumn klucza obcego i głównego).
3. Gdy aplikacja przetwarza osobno dwa zbiory wierszy może opłacać się rozdzielić tabelę na dwie np. tabelę *Osoby* na tabele: *Studenci* i *Pracownicy*. Tutaj dekompozycja jest *pozioma*, przy normalizacji natomiast, dekompozycja jest *pionowa*.
4. Sprawdzamy czy tabela jest wolna od zależności wielowartościowych. Jeśli nie, dokonujemy odpowiedniej dekompozycji.

**Projektowanie indeksów**

W pierwszej kolejności określamy *pole działania* (ang. *workload*) tworzonej aplikacji bazodanowej.

Pole działania aplikacji bazodanowej tworzą:

* Najważniejsze zapytania razem z informacją jak często będą używane.
* Najważniejsze aktualizacje razem z informacją jak często będą używane.
* Pożądana szybkość działania tych zapytań i aktualizacji.

Dla każdego zidentyfikowanego zapytania:

* Do których tabel jest wymagany dostęp?
* Które kolumny występują w warunkach selekcji/złączenia? Jak bardzo te warunki są selektywne?

Dla każdej zidentyfikowanej aktualizacji:

* Które kolumny występują w warunkach selekcji? Jak bardzo te warunki są selektywne?
* Typ aktualizacji (INSERT/DELETE/UPDATE) i których kolumn dotyczy?

W oparciu o zebrane informacje podejmujemy decyzje:

* Na których tabelach i kolumnach należy utworzyć indeksy? Jakiego typu: główny/jednoznaczny/zwykły? pogrupowany/zwykły? haszowany/drzewowy? dynamiczny/statyczny?  
  (Bierzemy oczywiście pod uwagę jakie rodzaje indeksów realizuje stosowany przez nas SZBD - na ogół tylko część z nich. Indeksy realizowane w Oracle będą omówione w dalszej części tego wykładu.)
* Czy warto dokonać *poziomego podziału* tabeli np. tabeli *Osoby* na osobne tabele *Pracownicy* i *Studenci* - co byłoby wskazane w przypadku gdyby kierowane do bazy danych zapytania dotyczyły osobno albo pracowników albo studentów oraz odpowiednie warunki wyszukiwania sugerowałyby indeksy na innych kolumnach?
* Czy połączyć zapis kilku tabel w klaster? Ich złączenie staje się  szybsze; operacje na pojedynczych tabelach stają się nieco wolniejsze niż bez klastra. Na przykład, czy zapisywać pozycje zamówień razem z zamówieniami w jednym pliku, w taki sposób, aby na dysku obok rekordu zamówienia były zapisane wszystkie jego pozycje?
* Czy w celu wykonania konkretnego, ważnego zapytania utworzyć indeks z kluczem wyszukiwania obejmującym kolumny występujące w zapytaniu pamiętając, że indeksy mogą przyśpieszyć wykonanie zapytania ale spowalniają aktualizacje i zwiększają zajętość miejsca na dysku?
* Czy w celu wykonywania konkretnego, ważnego zapytania zawierającego złożone konstrukcje jak złączenia, agregacje lub podzapytania, nie utworzyć dla niego *perspektyw*y *zmaterializowanej* (przypominamy, że  *perspektywa zmaterializowana* to perspektywa, której zawartość jest obliczana i zapisywana w tabeli, w regularnych odstępach czasu) z założonym odpowiednim indeksem pogrupowanym pamiętając, że zapytanie będzie wykonywane szybko ale kosztem aktualizacji perspektywy zmaterializowanej i zwiększenia zajętości miejsca na dysku? Na przykład, czy zamiast liczyć przy każdym zapytaniu statystykę stanu kont bankowych nie warto, mieć ją obliczoną raz na dzień lub raz na godzinę?

Jeszcze krótkie podsumowanie projektowania indeksów:

* Kolumny występujące w klauzuli WHERE są kandydatami na składniki kluczy wyszukiwania w indeksach.
* Indeks jest automatycznie tworzony przez system dla każdego klucza głównego i jednoznacznego.
* Indeks bywa zakładany na kolumnach:
  + których wartości ograniczają wyszukiwanie wierszy w tabeli np. Emp.Job='MANAGER' przy czym istotne jest aby wyszukiwanie było selektywne tj. aby procent wyszukiwanych wierszy nie był zbyt duży, powiedzmy co najwyżej 5 do 10%;
  + które są kluczami obcymi, co przyśpiesza wykonywanie złączeń dwóch tabel względem warunku *klucz obcy=klucz główny* – zakładając, że wykonywanie zapytania ze złączeniem zaczyna się od wyboru wiersza z tabeli nadrzędnej, tj. z kluczem głównym, po czym szukamy wierszy w tabeli podrzędnej, tj. z odpowiadającym kluczem obcym;
  + które występują w ważnych zapytaniach systemu informacyjnego umożliwiając realizację strategii *tylko-indeks*.
* Wyszukiwanie równościowe sugeruje indeks haszowany.
* Wyszukiwanie zakresowe i wypisywanie w kolejności uporządkowanej sugerują indeks drzewowy.
* Indeks pogrupowany jest szczególnie istotny dla zapytań zakresowych, z klauzulą ORDER BY oraz przy duplikatach - nie jest natomiast istotny gdy stosuje się strategię *tylko-indeks*.
* Przy stosowaniu metody Index Nested Loops Join indeks dla wewnętrznej tabeli złączenia powinien być: albo *główny* albo *jednoznaczny* albo *pogrupowany* albo *selektywny* - o ile oczywiście trzeba sięgać do wartości w niektórych innych kolumnach tabeli wewnętrznej oprócz samych wartości klucza wyszukiwania. Jeśli jest to niemożliwe, lepiej nie zakładać indeksu a do złączania tych tabel używać metody Sort-Merge Join lub Hash Join.

Rozważymy teraz na przykładach problem projektowania indeksów. Przypominamy, że potrzeba założenia indeksu ma swoje źródło w wymaganiu szybkiego wykonania jednego lub więcej zapytań na budowanej bazie danych.

Przykład 1

SELECT E.Ename, D.Loc  
FROM Emp E, Dept D  
WHERE E.Deptno=D.Deptno AND  
           D.Dname='SALES';

Indeks na *D.Dname* wspomaga selekcję D.Dname='SALES' gdy *D* jest  tabelą zewnętrzną. Potrzebny jest indeks albo jednoznaczny albo selektywny albo pogrupowany.

Indeks na *E.Deptno* wspomaga złączenie (*E* – tabela wewnętrzna). Powinien być pogrupowany, ponieważ spodziewamy się wybrania wielu wierszy z *E*.

Jeszcze lepszym rozwiązaniem byłby klaster obu tabel. Wtedy złączenie sprowadzilibyśmy do selekcji na klastrze zbudowanym w oparciu o indeks na kolumnie *D.Dname*. Przy odpowiednim zapisie klastra (jak w implementacji Oracle opisanej w dalszej części wykładu) dla danego departamentu rekordy wszystkich pracowników pracujących w tym departamencie byłyby zapisane na tej samej stronie co rekord departamentu lub tylko na kilku. Zastosowanie klastra jest więc szczególnie istotne gdy selekcja dotyczy tabeli po stronie *jeden* związku między tabelami (to jest po stronie klucza głównego) a dołączane przy złączaniu rekordy pochodzą z tabeli po stronie klucza obcego.

W krytycznych zastosowaniach można byłoby pomyśleć o zaprojektowaniu perspektywy zmaterializowanej liczącej tę konkretną instrukcję SELECT. Wynik byłby dostępny natychmiast.

Można byłoby też pomyśleć o perspektywie zmaterializowanej liczącej całe złączenie tabel *Dept* i *Emp*:

SELECT D.Dname, E.Ename, D.Loc  
FROM Emp E, Dept D  
WHERE E.Deptno=D.Deptno;

i założenie odpowiedniego indeksu na kolumnie *D.Dname* w celu przyśpieszenia wyszukiwania po kolumnie *D.Dname*.

Przykład 2

SELECT E.Ename, D.Loc  
FROM Emp E, Dept D  
WHERE E.Deptno=D.Deptno AND  
    E.Sal BETWEEN 10000 AND 20000 AND E.Job='SALESMAN';

Ze względu na warunki ograniczające dla tabeli *E*, wybieramy *Emp E* jako tabelę zewnętrzną złączenia. Wtedy *Dept D* będzie tabelą wewnętrzną złączenia.

Kolumna *D.Deptno* jest kluczemgłównym, posiada zawsze indeks, który można użyć przy złączaniu.

Jaki indeks jest potrzebny na tabeli *Emp E*? Albo indeks na *E.Sal* albo na *E.Job* – wybór zależy od selektywności warunków wyszukiwania - przy złej selektywności potrzebny byłby indeks pogrupowany na *E.Job*.

Alternatywnie system może zastosować przejście przez cały plik rekordów (*scan*) tabeli *E -* wtedy ewentualne indeksy w *E* nie byłyby zastosowane.

Klaster też mógłby być zastosowany do przyśpieszenia wykonania zapytania. Tym razem selekcja dotyczy tabeli po stronie *wiele* (to jest po stronie klucza obcego): dla rekordu pracownika obok na tej samej stronie moglibyśmy odczytać rekord departamentu.

Przykład 3

SELECT E.Deptno, COUNT(\*)  
FROM Emp E  
WHERE E.Sal>2000  
GROUP BY E.Deptno;

Potrzebny jest indeks pogrupowany na kolumnie *E.Deptno*. Jeśli nie jest możliwe założenie takiego indeksu, system posortuje względem *E.Deptno* plik rekordów tabeli *Emp E -* biorąc pod uwagę tylko rekordy dla których E.Sal>2000. Następnie wykona zliczanie COUNT(\*).

Byłoby jeszcze lepiej, gdybyśmy dysponowali indeksem drzewowym na kolumnach <E.Deptno, E.Sal>. Wtedy moglibyśmy wykonać zapytanie przechodząc tylko indeks bez potrzeby przechodzenia do pliku rekordów – tzn. stosując strategię *tylko-indeks*.

Przykład 4

Pewne zapytania, jak ostatnie, można wykonać bezpośrednio przez indeks - bez przechodzenia do pliku rekordów. Oto kilka dodatkowych przykładów:

|  |  |
| --- | --- |
| (a) | SELECT D.Dname FROM Dept D, Emp E WHERE D.Deptno=E.Deptno; |

W tym przypadku wystarczy dowolny indeks na *E.Deptno*, ponieważ w ogóle nie trzeba przechodzić do pliku rekordów tabeli *E*! Zapytanie jest realizowane przez przejście (*scan*) całego pliku rekordów tabeli *D* i za każdym razem sięgnięcie do indeksu na *E.Deptno* w celu sprawdzenia czy istnieje pozycja danych z wartością klucza wyszukiwania równą *D.Deptno*.

|  |  |
| --- | --- |
| (b) | SELECT E.Deptno, COUNT(\*) FROM Emp E GROUP BY E.Deptno; |

Wystarczy dowolny indeks na E.Deptno, ponieważ nie trzeba przechodzić do pliku rekordów tabeli *E*.

|  |  |
| --- | --- |
| (c) | SELECT E.Deptno, MIN(E.sal) FROM Emp E GROUP BY E.Deptno; |

Przydałby się indeks drzewowy na <E.Deptno, E.sal>. Wtedy nie trzeba byłoby przechodzić do pliku rekordów tabeli *E*.

|  |  |
| --- | --- |
| (d) | SELECT AVG(E.Sal) FROM Emp E WHERE E.Deptno=25 AND E.Sal BETWEEN 3000 AND 5000; |

Przydałby się indeks drzewowy na <E.Deptno,E.Sal> lub na <E.Sal, E.Deptno>. Wtedy nie trzeba by było przechodzić do pliku rekordów tabeli *E*.

Heurystyki optymalizacyjne dotyczące zapytań i indeksów

* Wybieraj do zaprojektowania indeksy, które wspomagają wykonywanie wielu zapytań.
* Buduj indeksy umożliwiające zastosowanie strategii *tylko-indeks*.
* Podejmij decyzję czy założyć indeks pogrupowany dla tabeli (o ile twój system umożliwia zakładanie takich indeksów). Tylko jeden indeks pogrupowany może być założony dla jednej tabeli. Pamiętaj, że wyszukiwanie przez inne indeksy będzie spowolnione.
* Unikaj, na ile to możliwe, zagnieżdżonych zapytań (podzapytań),  złożonych warunków, DISTINCT, GROUP BY, OR, wyrażeń arytmetycznych/napisowych.
* Przebudowuj okresowo wszystkie indeksy, w szczególności statyczne.
* Zbieraj statystyki używane przez optymalizator zapytań. Okresowo odświeżaj statystyki.
* Sprawdzaj plan wybrany przez optymalizator - ewentualnie zmień indeks lub  zapytanie. Ewentualnie, dołącz do zapytania wskazówki optymalizacyjne, jeśli system je umożliwia.

**11.2** **Indeksy w Oracle**

Pokażemy teraz jak przedstawione idee budowy i użycia indeksów stosują się w przypadku konkretnego SZBD, mianowicie firmy Oracle.

W Oracle rezultatem wyszukiwania w indeksie są *identyfikatory wierszy* będące wartościami specjalnego typu danych ROWID. Są następujące rodzaje indeksów.

**1. Indeks oparty na B+ drzewie**

Pozycja danych składa się z wartości indeksowanych kolumn oraz z identyfikatora ROWID wiersza w tabeli - określającego fizyczne położenie danego wiersza na dysku. Umożliwia szybkie znalezienie wiersza w oparciu o daną wartość klucza wyszukiwania oraz realizację zapytań zakresowych. Odpowiada koncepcji indeksu nie pogrupowanego.

Przy wykonywaniu zapytania system używa indeksu opartego na B+ drzewie tylko wtedy gdy jest zapewniona wystarczająca selektywność wyszukiwania, powiedzmy 5 do 10% wszystkich rekordów w pliku.

Zwykły indeks oparty na B+ drzewie jest automatycznie tworzony dla każdego klucza głównego i jednoznacznego.

**2. Tabela połączona z indeksem opartym na B+ drzewie**

(tworzona za pomocą klauzuli ORGANIZATION INDEX w instrukcji CREATE TABLE)

Pozycjami danych indeksu głównego są rekordy pliku tzn. wiersze tabeli są trzymane w indeksie. Jest zapewniony bardzo szybki dostęp do wierszy przez wartości klucza głównego. Wiersze nie posiadają swoich identyfikatorów (ROWID) – identyfikacja wierszy przebiega wyłącznie przez wartości klucza głównego;  w pozostałych indeksach wynikiem wyszukania jest wartość klucza głównego – a nie ROWID. Odpowiada to koncepcji indeksu pogrupowanego - ale budowanego tylko dla klucza głównego.

Przydatność tabeli połączonej z indeksem prześledzimy na następującym przykładzie. Załóżmy, że chcemy dokonywać analizy klientów wyszukując klientów mieszkających w określonym mieście. Załóżmy więc, że mamy dwie tabele:

* Miasta(Id\_miasta, Nazwa\_miasta)
* Klienci(Id\_miasta, Id\_klien\_w\_miescie, Nazwisko, Hobby, Wiek)

z podkreślonymi kluczami głównymi. Jeśli tabela *Klienci* jest połączona z indeksem na swoim kluczu głównym, istotnie można przyśpieszyć wykonywanie zapytań w rodzaju:

SELECT Klienci.Nazwisko, Klienci.Hobby  
FROM Klienci, Miasta  
WHERE Klienci.Id\_miasta = Miasta.Id\_miasta  
     AND Miasta.Nazwa\_miasta = 'WARSZAWA'  
     AND Klienci.Wiek BETWEEN 18 and 25;

ponieważ po znalezieniu *Id\_miasta* Warszawy wszyscy klienci z Warszawy będą zapisani razem tylko na kilku stronach a nie rozrzuceni po całym dysku, jak mogłoby się zdarzyć bez połączenia tabeli z indeksem.

Przykład ten jest interesujący i z tego powodu, że pokazuje zastosowanie złożonego klucza głównego.

Dla pełności prezentacji podajemy składnię instrukcji tworzącej tabelę *Klienci*:

CREATE TABLE Klienci(

Id\_miasta INTEGER,

Id\_klien\_w\_miescie INTEGER,

Nazwisko VARCHAR2(80),

Hobby VARCHAR(20),

Wiek INTEGER,

CONSTRAINT Klienci\_pk PRIMARY KEY(Id\_miasta,Id\_klien\_w\_miescie),

CONSTRAINT Klienci\_fk FOREIGN KEY(Id\_miasta) REFERENCES Miasta

)

ORGANIZATION INDEX;

**3. Indeks oparty na B+ drzewie z odwróconymi wartościami kluczy**

(klauzula REVERSE w instrukcji CREATE TABLE)

Jest stosowany tylko z predykatem równości – nie można w szczególności wypisywać wierszy wynikowych w kolejności uporządkowanej względem wartości klucza wyszukiwania. Bywa stosowany przy podziale tabeli i indeksu na partycje, gdzie równomierne rozłożenie wartości kluczy w poddrzewach B+ drzewa ma istotne znaczenie.

**4. Indeks oparty na klastrze jednej lub więcej tabel, indeks haszowany**

W klastrze wiersze kilku tabel są przechowywane razem uporządkowane względem wartości wybranych kolumn - nazywanych *kluczem klastra*, np. zamówienie razem ze swoimi pozycjami (kluczem klastra jest numer zamówienia). Jest to dobra struktura w przypadku gdy aplikacje często używają złączenia dwóch lub więcej tabel wchodzących w skład klastra.

Na każdym klastrze jest zakładany indeks zewnętrzny jednego z dwóch rodzajów:

* zwykły indeks oparty na B+ drzewie,
* indeks haszowany - do wartości klucza stosuje się funkcję haszującą otrzymując wartość, w oparciu o którą można łatwo wyznaczyć adres na dysku, gdzie znajduje się odpowiednia grupa wierszy. Indeks haszowany jest stosowany tylko wtedy gdy wyszukiwanie jest oparte na równości klucza. Indeks haszowany w SZBD Oracle występuje tylko w odniesieniu do klastra tabel, który jednak w szczególnym przypadku może zawierać pojedynczą tabelę.

Rozważmy przykładowe zapytanie liczące wartości zamówień złożonych przez klienta Kowalskiego.

SELECT z.Id\_zam, SUM(p.Wartość\*p.Ilość)  
FROM Zamówienia z, Pozycje\_zam p, Klienci k  
WHERE p.Id\_zam = z.Id\_zam  
    AND z.Id\_klienta = k.Id\_klienta  
    AND k.Nazwisko = 'Kowalski'  
GROUP BY z.Id\_zam;

Wykonanie tego zapytania można istotnie przyśpieszyć korzystając z klastra zawierającego tabele *Zamówienia* i *Pozycje\_zam*, ponieważ wszystkie pozycje zamówień znajdują się albo na tej samej stronie co samo zamówienie, albo tylko na kilku stronach na dysku.

CREATE CLUSTER Klast\_zam(Id\_zam INTEGER)  
  SIZE 2000;

Parametr SIZE określa ilość miejsca w bajtach przeznaczoną do zapisania jednego zamówienia i jego pozycji. Domyślną wartością jest rozmiar strony dyskowej. Jeśli wszystkie wiersze dla danej wartości klucza klastra nie mieszczą się w jednym bloku, są zapisywane na liście bloków.

Zatem chociaż indeks klastra nie jest pogrupowany, to jednak poprzez dobór odpowiedniej wartości SIZE można spowodować, że wszystkie rekordy z daną wartością klucza znajdą się na tej samej lub sąsiednich stronach dyskowych. Oznacza to, że wyszukiwanie przez klaster może być szybkie nawet przy nie najlepszej selektywności wyszukiwania przez indeks klastra. Sztywne ustawienie wartości SIZE może jednak  spowodować nie efektywne wykorzystanie miejsca na dysku – gdy zarezerwujemy więcej miejsca na rekordy z daną wartością klucza niż będzie to na ogół potrzebne.

Po utworzeniu tabel a przed wstawieniem do nich danych należy utworzyć indeks klastra.

CREATE INDEX Idx\_zam ON CLUSTER Klast\_zam;

Pełny przykład tworzenia klastra został podany na [wykładzie 3](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Klaster).

W przykładzie z zamówieniami należałoby mieć założone także zwykłe indeksy na kolumnach *Klienci.Nazwisko* i *Zamówienia.Id\_klienta*.

Podamy przykład zastosowania indeksu haszowanego dla klucza głównego *Id\_konta* tabeli:

* Konta(Id\_konta, Saldo, Imie, Nazwisko, Adres)

Zakładamy, że tabela *Konta* zawiera bardzo dużo wierszy oraz że często wielu kasjerów w banku równocześnie wykonuje zapytanie:

SELECT \*  
FROM Konta k  
WHERE k.Id\_konta = :numer;

Najpierw definiujemy klaster korzystając z opcji SINGLE TABLE:

CREATE CLUSTER Klast\_konta(Id\_konta INTEGER)  
  SIZE 512  
  SINGLE TABLE  
  HASHKEYS 100000 HASH IS mod(Id\_konta, 100003);

gdzie:

* parametr SIZE określa ilość miejsca w bajtach przeznaczoną do zapisania wierszy z tą samą wartością funkcji haszującej dla ich kluczy,
* klauzula HASHKEYS specyfikuje liczbę wartości funkcji haszującej – Oracle zaokrągla ją do najbliższej, większej od niej liczby pierwszej – w tym przypadku 100003,
* klauzula HASH IS określa funkcję haszującą stosowaną do klucza.

Liczbę wartości funkcji haszującej można odczytać z perspektywy słownika danych:

SELECT u.Hashkeys  
FROM User\_clusters u  
WHERE u.Cluster\_name = 'KLAST\_KONTA';

Następnie definiujemy tabelę *Konta*:

CREATE TABLE Konta(Id\_konta INTEGER PRIMARY KEY,  
     Saldo NUMBER,  
     Imie VARCHAR2(20),  
     Nazwisko VARCHAR2(50),  
     Adres VARCHAR2(70))  
CLUSTER Klast\_konta(Id\_konta);

W przypadku klastra haszowanego nie trzeba explicite tworzyć indeksu na kolumnie klucza klastra.

Dostęp do informacji o koncie klienta jest teraz bardzo szybki – na ogół wystarczy sprowadzić dwie strony z dysku (jedną indeksu i jedną z pliku rekordów), aby odczytać wartości w wierszu. W przypadku pesymistycznym może się jednak zdarzyć, że wiele kluczy da tę samą wartość funkcji haszującej i wtedy po przekroczeniu rozmiaru SIZE kolejne rekordy będą umieszczane na stronach nadmiarowych.

**5. Indeks z pozycjami określonymi za pomocą wyrażeń**

Kolumny indeksu mogą być wyrażeniami zawierającymi funkcje (z wyłączeniem funkcji agregujących). Na przykład, założenie indeksu

CREATE INDEX Uppercase\_idx ON Emp(UPPER(Ename));

pomaga w realizacji wyszukiwania pracowników w oparciu o ich nazwiska - nie biorąc pod uwagę wielkości liter w nazwisku:

SELECT \* FROM Emp e  
WHERE UPPER(e.Ename)='KOWALSKI';

**6. Indeks bitmapowy**

zostanie omówiony w wykładzie na temat hurtowni danych.

**11.3 Podsumowanie**

W wykładzie 11 zostały omówione zasady projektowania fizycznego schematu bazy danych i jego dostrajanie: jakie założyć indeksy, czy pogrupować tabele w klastry, jak poprawić schemat logiczny bazy danych z punktu widzenia szybkości działania zapytań.

Dodatkowo zostały przedstawione rodzaje indeksów, które występują w systemie Oracle. Pokazuje to, jak ogólna teoria dotycząca indeksów jest realizowana w praktyce.

**11.4 Słownik pojęć**

[pole działania](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w10.htm#Pole) - składa się z informacji o aplikacji bazy danych:

* jakie są najważniejsze zapytania i jak często będą używane,
* jakie są najważniejsze aktualizacje i jak często będą używane,
* jaka jest pożądana szybkość działania tych zapytań i aktualizacji.

**Wykład 12**

***Zarządzanie transakcjami***

**Wprowadzenie**

Do tej pory czytelnik zapoznał się z podstawowymi strukturami danych na dysku i w pamięci wewnętrznej RAM, które umożliwiają przechowywanie danych na dysku i realizację instrukcji języka SQL. Nie braliśmy przy tym pod uwagę czy z bazy danych korzysta jeden czy wielu użytkowników. Teraz uwzględnimy możliwość równoczesnej pracy z bazą danych przez wielu użytkowników. System bazy danych musi zapewnić aby użytkownicy nie przeszkadzali sobie nawzajem w pracy z bazą danych np. gdy kilku z nich chce zarezerwować sobie miejsce w samolocie na ten sam lot.

Zlecenie użytkownika kierowane do SZBD ma postać *transakcji* – czyli ciągu instrukcji SQL, które są wykonywane jako całość: albo wszystkie albo żadna z nich. Przypominamy, że transakcje w języku SQL zostały już  omówione na [wykładzie 3](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w3.htm#Transakcje).

W tym wykładzie przedstawimy problemy jakie napotyka się, gdy akcje współbieżnie wykonywanych transakcji są przeplatane oraz w jaki sposób SZBD radzi sobie z pojawiającymi się problemami. Podstawą realizacji transakcji przez SZBD jest protokół ścisłego blokowania dwufazowego w skrócie Strict-2PL (ang. *Strict Two-Phase Locking*).

**12.1 Transakcje**

**Współbieżność**

Współbieżne wykonywanie programów użytkowników jest istotne dla szybkości działania aplikacji baz danych. Dostęp do danych na dysku jest częsty i względnie wolny, więc procesor może jednocześnie wykonywać kilka programów.

Instrukcje SQL tworzące transakcję mogą być podawane przez użytkownika w pewnych odstępach czasu i zależeć od wyniku poprzednich. Oczywiste jest więc, że również z tego powodu system nie może rezerwować czasu całego procesora do wykonania jednej transakcji.

Z punktu widzenia systemu SZBD transakcja stanowi reprezentację programu użytkownika i jest *ciągiem odczytów i zapisów do bazy danych*. Współbieżność uzyskuje się przez przeplecenie ze sobą odczytów i zapisów różnych transakcji. Efekt powinien być taki jakby transakcje były wykonywane niezależnie od siebie w czasie. Moduł *zarządzania transakcjami* zajmuje się organizacją współbieżnego wykonywania transakcji użytkowników.

**Poprawność i sp****ójność**

Podstawowe znaczenie ma poprawność wykonywanych transakcji. Tę poprawność wspomagają więzy spójności *definiowane* przez administratora danych i projektanta bazy danych a *sprawdzane* przez system. Stan bazy danych jest *spójny* gdy są spełnione wszystkie więzy spójności. Nie wszystkie warunki poprawności mogą być sprawdzone przez system. System może sprawdzić w ramach więzów spójności, na przykład, czy format numeru telefonu jest prawidłowy; ale nie może sprawdzić czy jest to rzeczywiście numer telefonu osoby, o której informacje trzymamy w bazie danych. Należy więc do obowiązków użytkownika *zadbanie*, aby dane wprowadzane i aktualizowane przez transakcję *wiernie odpowiadały* rzeczywistym danym – zdefiniowane przez projektanta a sprawdzane przez system więzy spójności mogą tu tylko pomóc.

Oprócz sprawdzania więzów spójności, do zadań SZBD należy zadbanie aby poprawne transakcje zostały *poprawnie zrealizowane* w sytuacji współbieżnego wykonywania transakcji, czyli w sytuacji przeplatania ich akcji. Za *poprawny stan bazy danych* *po współbieżnym wykonaniu zbioru transakcji* przyjmuje się każdy stan, który można otrzymać przez szeregowe (sekwencyjne) wykonanie tych transakcji - przy czym różna kolejność wykonywania transakcji może prowadzić do innego poprawnego stanu. Dochodzi do tego jeszcze możliwość wycofania każdej transakcji ze zbioru (anulowania wszystkich wprowadzonych przez nią zmian). Ewentualne wznowienie tej samej transakcji użytkownika, oznacza już nową transakcję z punktu widzenia systemu.

**Aksjomaty ACID**

Są cztery ogólne wymagania, jakie stawia się przed SZBD co do współbieżnego wykonywania transakcji. Noszą one nazwę aksjomatów wykonywania transakcji ACID od czterech słów angielskich: A - Atomicity (*atomowość*), C - Consistency (*spójność*), I - Isolation (*izolacja*) oraz D - Durability (*trwałość*).

1. *Atomowość* (*niepodzielność*) - każda transakcja jest niepodzielną operacją z punktu widzenia użytkownika: albo wszystkie akcje wchodzące w skład transakcji są wykonywane albo żadna z nich.
2. *Spójność* - po wykonaniu zbioru transakcji stan bazy danych powinien być spójny (pod warunkiem, że przy rozpoczynaniu transakcji stan bazy danych był spójny oraz że każda z wykonywanych transakcji jest z osobna poprawna).
3. *Izolacja* - transakcje powinny sobie wzajemnie nie  przeszkadzać w działaniu. Każdy użytkownik powinien mieć iluzję, że sam korzysta z bazy danych. Przy najwyższym (zalecanym) stopniu izolacji wymaga się aby transakcja działała na spójnym, nie zmienianym przez innych użytkowników fragmencie bazy danych.
4. *Trwałość* - dane zatwierdzone przez transakcję powinny być dostępne nawet w sytuacji awarii programu, komputera lub nośnika danych.

**Mechanizmy współdzielenia zasobów bazy danych**

SZBD stosuje następujące mechanizmy służące do zapewnienia aksjomatów ACID:

* *blokady* (*zamki*) (ang. *lock*) zakładane na obiekty – ograniczające albo wręcz uniemożliwiające działanie innych transakcji na zablokowanym obiekcie,
* *dziennik* (ang. *log*) - zapisywanie wszystkich zmian w bazie danych do specjalnego dziennika (logu), aby w razie potrzeby móc:
  + dla nie zatwierdzonej transakcji wycofać wprowadzone przez nią zmiany,
  + w przypadku awarii odtworzyć spójny stan bazy danych.
* *kopia zabezpieczająca* bazy danych (backup) - wykonywana w regularnych odstępach czasu, na przykład raz na dzień; w przypadku awarii danych na dysku pozwala przywrócić spójny stan bazy danych.
* *wielowersyjność* - możliwość odczytywania danych zmienianych równocześnie przez inne transakcje w takiej postaci w jakiej istniały w pewnych chwilach w przeszłości (np. w chwili rozpoczynania się danego zapytania lub danej transakcji).

**Atomowość transakcji**

Transakcja może zakończyć swoje działanie na cztery sposoby:

* *zatwierdzeniem* (COMMIT) po zakończeniu wszystkich swoich akcji,
* *samo-wycofaniem* (ROLLBACK) - anulowaniem wszystkich wykonanych akcji,
* może zostać *przerwana przez SZBD* ("abortowana", ang. *abort*) a następnie *wycofana* i  *restartowana* (np. z powodu *zakleszczenia* - ang. *deadlock*),
* zanim dojdzie do końca może zostać *przerwana z powodu awarii serwera lub dysku* - po podniesieniu systemu po awarii dotychczasowe zmiany wprowadzone przez transakcję zostają na chwilę przywrócone a następnie wycofane przez system (ponieważ transakcja nie zakończyła się zatwierdzeniem).

W każdej więc z tych sytuacji albo cała transakcja zostaje wykonana albo żadna jej część nie zostaje wykonana. Transakcję można traktować jak pojedynczą, atomową operację na bazie danych.

SZBD zapisuje wszystkie wykonywane akcje w dzienniku (logu) tak aby w razie potrzeby, gdy nie jest możliwe doprowadzenie transakcji do końca, móc ją wycofać czyli anulować wszystkie jej akcje.

Prześledźmy na prostym przykładzie problem przeplotu akcji przy współbieżnej realizacji dwóch transakcji.

Przykład

|  |
| --- |
| T1: BEGIN A=A+100, B=B-100 END T2: BEGIN A=1.06\*A, B=1.06\*B END |

Intuicyjnie transakcja *T1* dokonuje transferu $100 z konta B na konto A. Transakcja *T2* dopisuje do obu kont 6% odsetki.

Poprawna, współbieżna realizacja obu transakcji powinna być równoważna albo szeregowemu wykonaniu *T1* potem *T2* albo szeregowemu wykonaniu *T2* potem *T1*. W przykładzie, w obu przypadkach efekt jest taki sam.

Rezultat współbieżnego wykonania kilku transakcji może być niejednoznacznie określony.

Oto możliwy, poprawny (z punktu widzenia otrzymywanego wyniku) *przeplot akcji* obu transakcji (czyli *plan* ich wykonania):

|  |
| --- |
| T1: A=A+100,                  B=B-100 T2:                 A=1.06\*A,                B=1.06\*B |

A to niepoprawny plan (z powodu niepoprawnego wyniku):

|  |
| --- |
| T1: A=A+100,                                    B=B-100 T2:                 A=1.06\*A, B=1.06\*B |

SZBD abstrahuje od znaczenia poszczególnych operacji na danych. Z punktu widzenia SZBD drugi plan jest postaci:

|  |
| --- |
| T1: R(A), W(A),                                     R(B), W(B) T2:                  R(A), W(A), R(B), W(B) |

gdzie R(*A*) oznacza operację odczytu obiektu *A*, a W(*A*) operację zapisu obiektu *A*. Z punktu widzenia zarządzania transakcjami jeszcze dwie operacje są istotne: Commit - zatwierdzenie transakcji oraz Rollback - anulowanie transakcji.

Na początku naszych rozważań przyjmiemy upraszczające założenia, że baza danych jest zbiorem niezależnych (niepodzielnych) obiektów oraz że zbiór ten nie zmienia się w trakcie realizacji transakcji. Potem pokażemy jak rozszerzyć otrzymane wyniki na ogólny przypadek.

**Plan wykonania transakcji**

*Plan* jest to ustalenie kolejności wykonywania akcji odczytu i zapisu na obiektach bazy danych przez współbieżnie działające transakcje. SZBD stosując pewne reguły, o których będzie mowa w dalszej części, na bieżąco podejmuje decyzje, akcja której transakcji ma być wykonana jako następna - dynamicznie tworząc pewien plan wykonywania akcji zbioru współbieżnych transakcji.

*Plan szeregowy* jest to plan, ustawiający wykonywanie transakcji w ciąg: najpierw akcje jednej transakcji, następnie akcje drugiej transakcji itd.

Dwa plany są *równoważne* jeśli efekt realizacji obu planów jest taki sam dla każdego stanu bazy danych tzn. po realizacji każdego z planów otrzymujemy ten sam stan bazy danych.

*Plan szeregowalny* jest to plan, który jest równoważny pewnemu planowi szeregowemu. Plan szeregowy jest poprawny a więc również plan szeregowalny jako jemu równoważny jest też poprawny. Zauważmy, że:

* Dla planu szeregowalnego realizacja transakcji przez SZBD ma własność izolacji - użytkownik może sobie wyobrazić, że w czasie wykonywania jego transakcji nie działa żadna inna transakcja, bo ze względu na równoważność realizowanego planu z planem szeregowym, może przyjąć, że transakcje są wykonywane szeregowo jedna po drugiej.
* Jeśli każda transakcja zachowuje spójność (poprawność) bazy danych, to realizacja planu szeregowalnego, zapewnia spójność (poprawność) bazy danych, zatem realizacja transakcji przez SZBD ma własność spójności.
* Plan szeregowalny nie daje natomiast gwarancji, że przy wycofywaniu transakcji nie trzeba wycofać akcji innych splecionych z nią transakcji, co czasem może się okazać nie wykonalne, gdy jedna z transakcji zostanie w międzyczasie zatwierdzona. Zatem sam plan szeregowalny nie gwarantuje jeszcze ani własności *atomowości* ani *trwałości*.

Przykład

Oto plan (rozważany już wcześniej), który nie jest szeregowalny:

|  |
| --- |
| T1: R(A), W(A),                                                  R(B), W(B) T2:                   R(A), W(A), R(B), W(B) |

Jego nieszeregowalność wnioskuje się rozważając tzw. *graf zależności* akcji transakcji. Gdy zapis zmiennej *A* przez transakcję *T1* poprzedza odczyt zmiennej *A* przez transakcję *T2*, rysujemy krawędź od *T1* do *T2* i opatrujemy ją etykietą *A* i podobnie dla *B*. Zbudujmy graf zależności dla rozpatrywanego planu:

|  |
| --- |
|  |

Rys. 12.1 Cykl w grafie zależności

Przyczyna nie szeregowalności tego planu leży w cyklu grafu zależności. Wynik *T2* zależy od *T1* i vice-versa. Nie istnieje równoważny plan, który by uszeregował kolejność wykonywania obu transakcji - przestawiając akcje jednej z nich przed drugą.

**Zjawiska niepożądane przy przeplataniu akcji**

Przy analizie współbieżnego wykonywania transakcji istotne są jeszcze inne, niezależne od pojęcia szeregowalności własności, które teraz rozważymy.

Analizując współbieżne wykonywanie transakcji bierzemy pod uwagę tylko operacje odczytu i zapisu. Konflikty między transakcjami i w szczególności niemożliwość uszeregowania ich akcji, pojawiają się w związku z dokonywaniem przez nie zapisów. Oto możliwe sytuacje konfliktowe dla dwóch transakcji.

**Odczyt niezatwierdzonych danych "dirty read"**

Transakcja *T2* odczytuje i ewentualnie zmienia niezatwierdzone jeszcze przez transakcję *T1* (a więc potencjalnie niepoprawne) dane. Transakcja *T1* może nawet je chcieć następnie wycofać, po tym jak *T2* je już zatwierdzi!

Chociaż nie są używane powszechnie, to czasami stosuje się transakcje "na niby", których celem jest przetestowanie pewnych hipotez ("co by było gdyby Kowalski został dyrektorem? czy nasza firma by upadła?") a następnie wycofanie całej transakcji. W trakcie takich transakcji stan obiektów bazy danych może być zupełnie niezgodny z rzeczywistością (użytkownik bazy danych mógłby na przykład z niej nagle odczytać, że firma znajduje się w stanie bankructwa).

**Niepowtarzalny odczyt - odczyt zatwierdzonych danych**

Transakcja odczytuje dwukrotnie ten sam obiekt i za każdym razem widzi inne dane - bo w międzyczasie zmieniła je i zatwierdziła inna transakcja.

Jednak, niepowtarzalny odczyt może prowadzić do nieprawidłowego działania aplikacji bazy danych, gdy jedna transakcja podejmuje decyzję w oparciu o odczyt danych. Na przykład, odczytuje, że jest miejsce w samolocie na lot, informuje o tym użytkownika, po czym chce zarezerwować to miejsce, a w tym momencie już zostało ono zarezerwowane przez inną transakcję.

**Nadpisanie niezatwierdzonych danych**

Transakcja *T1* zmieniła dane i nie zatwierdziła zmian. Za chwilę transakcja *T2* zmieniła te same dane i zatwierdziła swoje zmiany.

Nawet wtedy gdy transakcja tylko zapisuje wartość bez wcześniejszego jej odczytu, nadpisanie niezatwierdzonych danych może spowodować niespójny stan bazy danych. Załóżmy, że spójny stan bazy danych to taki, w którym zmienne *A* i *B* mają tę samą wartość. Transakcja *T1* zapisuje na obu zmiennych wartość 0, a transakcja *T2* zapisuje na obu zmiennych wartość 1. Przemieszanie akcji tych transakcji doprowadzi po ich zatwierdzeniu do niespójnego stanu bazy danych, w którym *A* nie jest równe *B*.

**Plan odtwarzalny**

Plan, który umożliwia wycofanie każdej transakcji nazywa się planem *odtwarzalnym*. Plan odtwarzalny jest istotny do zapewnienia własności atomowości i trwałości.

Zastanówmy się kiedy jesteśmy w stanie wycofać transakcję *T*. Jeśli w danym planie nie ma zjawisk nie zatwierdzonego odczytu ani nadpisania niezatwierdzonych danych, to żadna inna transakcja nie korzysta z niezatwierdzonych zmian transakcji *T* i dlatego transakcję *T* można wycofać. Jeśli jakaś transakcja skorzystała z wyników transakcji *T*, to tę transakcję też trzeba wycofać, co może być już niemożliwe, jeśli ta transakcja została wcześniej zatwierdzona.

Podamy teraz rozwiązanie problemu jak zagwarantować powstawanie i realizację planów wyłącznie szeregowalnych i odtwarzalnych.

**12.2 Zarządzanie współbieżnością oparte na blokadach**

**Podstawowe rodzaje blokad**

Podstawowym mechanizmem zapobiegającym konfliktom przy współbieżnie wykonywanych transakcjach są *blokady* (nazywane też *zamkami*) zakładane na obiekty. Są dwa rodzaje blokad:

*Współdzielona, typu S* (ang. *shared lock*) - daje transakcji współdzielony dostęp do zasobu, na przykład, kilka transakcji może jednocześnie odczytywać wiersze tej samej tabeli. Jeśli transakcja zakłada współdzieloną blokadę, inne transakcje też mogą założyć współdzieloną blokadę, ale nie mogą założyć blokady drugiego rodzaju, to jest wyłącznej blokady. Operację założenia blokady współdzielonej na obiekcie *A* oznaczamy przez S(*A*).

*Wyłączna, typu X* (ang. *exclusive lock*) - daje transakcji wyłączne prawo do wprowadzania zmian obiektu. Tylko jedna transakcja może mieć założoną wyłączną blokadę na obiekcie i w tym czasie nie może być na nim założonej żadnej innej blokady nawet współdzielonej. Operację założenia blokady wyłącznej na obiekcie *A* oznaczamy przez X(*A*).

Istnieje metoda zakładania blokad gwarantująca powstawanie i realizację planów wyłącznie szeregowalnych i odtwarzalnych. Jest ona powszechnie używana przez SZBD. Oto ona sformułowana w postaci ogólnego protokołu zakładania blokad na obiektach:

**Protokół ścisłego blokowania dwufazowego (Strict 2PL):**

1. Każda transakcja musi uzyskać blokadę S (współdzieloną) na obiekcie zanim odczyta ten obiekt oraz blokadę X (wyłączną) na obiekcie przed zapisaniem go.
2. Jeśli transakcja trzyma blokadę X na obiekcie, żadna inna transakcja nie ma prawa założyć żadnej blokady (ani typu S ani X) na tym obiekcie.
3. Jeśli transakcja trzyma blokadę S na obiekcie, żadna inna transakcja nie ma prawa założyć blokady X na tym obiekcie.
4. Gdy transakcja nie może założyć blokady na obiekcie, może ustawić się w kolejce oczekujących transakcji stowarzyszonej z tym obiektem albo może zostać wycofana.
5. Wszystkie blokady trzymane przez transakcję są zwalniane jednocześnie, w chwili gdy transakcja kończy się (wycofaniem lub zatwierdzeniem).

Punkty 2 i 3 są powtórzeniami definicji blokad S i X - przytaczamy je dla pełności. Zwracamy uwagę na istotność ostatniego punktu protokołu.

Twierdzenie  
Protokół Strict 2PL gwarantuje realizację wyłącznie planów szeregowalnych i odtwarzalnych.

Protokół Strict 2PL nazywa się *dwufazowym*, ponieważ determinuje dwie fazy działania każdej transakcji związane z blokadami:

1. transakcja zakłada blokady i dokonuje wymaganych odczytów i zapisów na obiektach, na których założyła blokadę;
2. transakcja wykonuje COMMIT/ROLLBACK jednocześnie zwalniając wszystkie blokady.

Zakładanie blokad i ich zwalnianie są oddzielonymi w czasie fazami.

Zastanówmy się czy protokół Strict-2PL generuje wyłącznie plany wolne od anomalii.

* Niezatwierdzony odczyt nie może zajść, ponieważ aby zmienić wartość obiektu transakcja musi założyć blokadę X i od tej chwili do momentu zwolnienia blokady na koniec wykonywania transakcji żadna inna transakcja nie ma dostępu do zablokowanego obiektu (nawet w celu jego odczytu).
* Niepowtarzalny odczyt też nie może zajść, ponieważ przy pierwszym odczycie transakcja zakłada blokadę S (lub X) na odczytywany obiekt i zwalnia tę blokadę dopiero na koniec. W międzyczasie żadna inna transakcja nie uzyska blokady X na tym obiekcie aby go zmienić.
* Nadpisywanie niezatwierdzonych zmian też nie może mieć miejsca, ponieważ zapisanie obiektu wymaga wcześniejszego założenia blokady X i do czasu zakończenia transakcji żadna inna transakcja nie może ani odczytać ani zapisać tego obiektu.

Podsumujmy wynik naszego rozumowania w postaci twierdzenia.

Twierdzenie  
Protokół Strict-2PL nie dopuszcza do powstania anomalii niezatwierdzonego odczytu, niepowtarzalnego odczytu i nadpisywania nie zatwierdzonych danych.

Uwaga

Warunek 5 protokołu Strict 2PL można złagodzić zachowując jego dwufazowość i szeregowalność. Zamiast:

* Blokady trzymane przez transakcję są zwalniane, gdy transakcja kończy się

można przyjąć:

* Transakcja nie może założyć żadnej nowej blokady po zwolnieniu jakiejkolwiek blokady

Otrzymujemy protokół blokowania dwufazowego oznaczany przez 2PL, który także prowadzi do planów szeregowalnych. Przy stosowaniu jego może się jednak pojawić problem z wycofaniem transakcji. Gdy transakcja *T1* zwolni blokadę obiektu *A*, wtedy inna transakcja *T2* może dokonać zmiany tego obiektu i zatwierdzić zmiany. Po czym transakcja *T1* może chcieć wykonać ROLLBACK, co wymagałoby wycofania również transakcji *T2*, która jednak już została zatwierdzona! Zatem protokół 2PL nie gwarantuje planów odtwarzalnych!

**Zarządzanie transakcjami**

Moduł zarządzania transakcjami operuje na transakcjach, obiektach bazy danych i na rekordach dzienników. Buduje i wykorzystuje struktury danych przechowujące dane o powiązaniu między sobą obiektów bazy danych, transakcji i rekordów dziennika:

1. *transakcje* (tablica transakcji w RAM) i *obiekty* (bazy danych) są powiązane związkiem: "*transakcja* trzyma określoną blokadę na *obiekcie* bazy danych" (na ogół blokowana jest *strona*, na której leży obiekt - nie tylko sam obiekt);
2. *transakcje* (tablica transakcji w RAM) i *rekordy dziennika* są powiązane związkiem: "ostatnio wykonana przez *transakcję* operacja jest opisana w *rekordzie* dziennika" (informacja ta jest potrzebna do rozpoczęcia wycofywania transakcji i odtwarzania po awarii serwera).

Szczegóły algorytmów stosowanych w module zarządzania transakcjami pomijamy jako wykraczające poza zakres tego wykładu.

**Zakleszczenia (deadlocks)**

Protokół Strict-2PL nie zapewnia, że działanie transakcji dojdzie do końca (nie zostanie wstrzymane przez SZBD). Sytuacja taka może się pojawić w związku ze zjawiskiem *zakleszczenia*, kiedy dwie lub więcej transakcji wzajemnie blokują sobie - potrzebne do kontynuowania swojego działania - obiekty.

Na przykład, transakcja *T1* dokonała blokady obiektu *A*, a transakcja *T2* obiektu *B*. W kolejnym kroku *T1* chce założyć blokadę na obiekt *B*, a *T2* na obiekt *A*. Obie transakcje przestają działać oczekując, potencjalnie w nieskończoność, na zwolnienie blokady przez drugą transakcję.

*Zakleszczenie* (ang. deadlock) jest to cykl transakcji oczekujących wzajemnie na zwolnienie blokady przez inną transakcję w cyklu. Są trzy sposoby radzenia sobie z zakleszczeniami:

1. *zapobieganie*,
2. *wykrywanie*,
3. *ustalenie limitu oczekiwania na blokadę* (timeout).

**Zapobieganie** polega na ustaleniu priorytetu między transakcjami np. transakcja, która rozpoczęła się wcześniej ma z definicji wyższy priorytet. Nie dopuszcza się do tego aby transakcja z wyższym priorytetem czekała na transakcję z niższym priorytetem. Aby do tego nie dopuścić, transakcja z niższym priorytetem zostaje wycofana przez system.

**Wykrywanie zakleszczeń** polega na analizie, która transakcja oczekuje na zwolnienie blokady przez którą transakcję i sprawdzaniu czy występuje cykl.

1. Utwórz *graf oczekiwań na zwolnienie blokady*:
   * węzłami są transakcje;
   * istnieje krawędź od *Ti* do *Tj* jeśli transakcja *Ti* oczekuje na zwolnienie blokady przez transakcję *Tj*.
2. Co jakiś czas sprawdzaj, czy w grafie jest cykl. Jeśli jest, wycofaj jedną z transakcji w cyklu.

Przykład rozpoznania zakleszczenia

Dla czterech transakcji:

|  |
| --- |
| T1: S(A), R(A),                       S(B) T2:                  X(B),W(B)                               X(C) T3:                                                 S(C), R(C)                   X(A) T4:                                                                          X(B) |

budujemy graf oczekiwań na zwolnienie blokady (rys. 12.2).

Rys. 12.2 Graf oczekiwań na zwolnienie blokady

Próba wykonania operacji X(*A*) przez transakcję T3 wprowadza cykl do rozważanego grafu (czerwona krawędź zamyka cykl). Aby nie dopuścić do zakleszczenia wycofujemy transakcję T3.

**Zapobieganie zakleszczeniom** - metoda *timeout*

Gdy transakcja czeka bezczynnie na zwolnienie blokady dłużej niż ustalony odcinek czasu *timeout*, transakcja zostaje wycofana przez system.

**Zjawisko przeładowania**

Im więcej transakcji jest równocześnie aktywne, tym większe jest  prawdopodobieństwo, że dana transakcja będzie czekać na zwolnienie blokady przez inną działającą transakcję. Od pewnej liczby aktywnych transakcji działanie systemu zaczyna się pogarszać – coraz mniejsza liczba akcji jest wykonywana. Dla określonej bazy danych i jej aplikacji administrator bazy danych powinien znaleźć ten punkt "przeładowania systemu" i powinien ograniczać liczbę równocześnie działających transakcji. Drugą metodą dawania sobie rady z przeciążeniem systemu jest identyfikacja najczęściej używanych (blokowanych) obiektów (ang. *hot spots*) i próba rozładowania zapotrzebowania na nie np. przez zastosowanie replikacji lub zmianę aplikacji.

**Problem fantomów**

Protokół Strict-2PL (w dotychczasowej postaci) jest poprawny pod warunkiem, że baza danych jest ustaloną, nie zmieniającą się kolekcją obiektów. Oto przykład dwóch transakcji, dla których realizacja za pomocą protokołu Strict-2PL daje plan nie szeregowalny.

1. *T1* blokuje wszystkie strony zawierające rekordy pracowników z E.Job='SALESMAN' i wyznacza zarabiającego najwięcej (E.Sal = 3000).
2. Następnie *T2* wstawia nowego pracownika: E.Job='SALESMAN', E.Sal = 3500.
3. *T2* usuwa najlepiej zarabiającego pracownika z E.Job='MANAGER' (zarabiającego powiedzmy E.Sal = 5000) i zatwierdza.
4. *T1* blokuje wszystkie strony zawierające rekordy pracowników z E.Job='MANAGER' i wyznacza najlepiej zarabiającego (powiedzmy E.Sal = 4500).

Wykonywania tych dwóch transakcji nie da się uszeregować!

Rozwiązanie

Nie wystarcza to, że transakcja *T1* zakłada blokadę wszystkich istniejących rekordów pracowników z E.Job='SALESMAN'. Potrzebne są blokady na całe zbiory rekordów określone przez predykaty np. E.Job='SALESMAN'. Można to uzyskać przez zablokowanie węzła indeksu z E.Job='SALESMAN', jeśli taki indeks istnieje albo trzeba zablokować całą tabelę, gdy indeksu nie ma. Dodajemy ten postulat do definicji protokołu Strict-2PL.

Rys. 12.4 Blokada węzła indeksu

Zatem indeksy oprócz zastosowania do wyszukiwania i zapewnienia zachodzenia więzów klucza głównego i jednoznacznego, pełnią także istotną rolę przy zapewnieniu szeregowalnego planu wykonywania transakcji!

**Blokady zakładane na węzłach B+ drzewa**

Przy wykonywaniu operacji na wierszach tabeli należy również dokonać odpowiednich modyfikacji w każdym indeksie dla tej tabeli np. dodać nowe pozycje danych z kluczem wstawianego rekordu przy INSERT lub usunąć pozycje danych dla usuwanego rekordu przy DELETE. Rozważmy przypadek indeksu zbudowanego na B+ drzewie. Zmiany dotyczą liści drzewa, ale mogą również dotyczyć innych węzłów na ścieżce od korzenia do liścia. Jednocześnie na indeksie może być wykonywane kilka operacji pochodzących z różnych transakcji. Potrzebne są więc blokady zakładane na węzły drzewa, blokujące innym operacjom dostęp do węzła na którym pracuje dana operacja.

Przy samym wyszukiwaniu, węzły na ścieżce od korzenia do liścia nie muszą być blokowane (a dokładniej, węzeł jest blokowany tylko w czasie jego odczytu).

Przy wykonywaniu instrukcji INSERT, węzeł na ścieżce od korzenia do modyfikowanego liścia musi zostać zablokowany w trybie X tylko jeśli proces podziału węzłów może zostać propagowany do niego od modyfikowanego liścia (podobnie dla DELETE). Schodząc  w dół drzewa dokonujemy blokady aktualnego węzła i sprawdzamy, czy możemy zdjąć blokadę z węzłów, które leżą wyżej na ścieżce do korzenia. Zatem zawsze co najmniej jeden węzeł w drzewie pozostaje zablokowany w trybie wyłącznym.

**12.3 Poziomy izolacji transakcji**

Przypominamy z wykładu 3, że standard ANSI/ISO definiuje cztery poziomy izolacji określające, jakie zmiany dokonywane przez inne współbieżnie działające transakcje, widzi dana transakcja.

|  |  |  |  |
| --- | --- | --- | --- |
| **Poziom izolacji** | **Niezatwierdzony odczyt** | **Niepowtarzalny odczyt** | **Fantomy** |
| READ UNCOMMITED | TAK | TAK | TAK |
| READ COMMITED | NIE | TAK | TAK |
| REPEATABLE READS | NIE | NIE | TAK |
| SERIALIZABLE | NIE | NIE | NIE |

Tab. 12.1 Poziomy izolacji transakcji w SQL

Wyjaśnimy, jak są one implementowane przez SZBD.

1. SERIALIZABLE - transakcja *T* odczytuje tylko obiekty, których zmiany zostały zatwierdzone; żadna wartość odczytana lub zmieniona przez *T* nie może być zmieniona przez inną transakcję, dopóki T nie skończy się; wyniki instrukcji SELECT wyliczone przez transakcję *T* nie zmieniają się, dopóki *T* nie skończy działać.

Transakcja *T* uzyskuje blokady do odczytu i zapisu obiektów zgodnie z protokołem Strict-2PL plus blokady typu S na zbiory wierszy wynikowych instrukcji SELECT (realizowane albo poprzez zablokowanie węzła indeksu albo poprzez zablokowanie całej tabeli).

2. REPEATABLE READS - transakcja *T* może odczytywać tylko obiekty, których zmiany zostały zatwierdzone; żadna wartość odczytana lub zmieniona przez *T* nie może być zmieniona przez inną transakcję dopóki *T* nie skończy się.

Transakcja *T* uzyskuje blokady do odczytu i zapisu obiektów zgodnie z protokołem Strict-2PL. Nie mamy do czynienia z blokadami zakładanymi na zbiory wierszy wynikowych instrukcji SELECT.

3. READ COMMITED - transakcja *T* może odczytywać tylko obiekty, których zmiany zostały zatwierdzone; żadna wartość zmieniona przez *T* nie może być zmieniona przez inną transakcję, dopóki *T* nie skończy się.

Transakcja *T* uzyskuje blokady X aby wykonać zmiany i utrzymuje te blokady do końca swojego działania. Do wykonania odczytu uzyskuje blokadę S. Po zakończeniu odczytu blokada jest natychmiast zwalniana (nie czekając na koniec transakcji).

W implementacji Oracle uzyskanie blokady S do odczytu obiektu nie jest konieczne. Zamiast zakładać blokadę S do odczytu, Oracle korzysta z cechy wielowersyjności danych (co jest omówione dalej). Poziom READ COMMITTED w Oracle umożliwia większy poziom współbieżności niż w Standardzie, gdyż umożliwia odczytywanie obiektów nawet wtedy gdy inna transakcja założyła na tym obiekcie blokadę X (co jest niedozwolone w wersji opisanej przez Standard).

4. READ UNCOMMITED - *T* odczytuje obiekty w dowolnej chwili; *T* nie dokonuje żadnych zapisów.

Transakcja nie zakłada żadnych blokad ale też nie ma prawa wprowadzać żadnych zmian do bazy danych.

**Optymistyczne blokowanie**

Alternatywą do omawianej do tej pory metody nazywanej *blokowaniem pesymistycznym* jest metoda *blokowania optymistycznego* polegająca na nie blokowaniu obiektów i wykonywaniu operacji przez transakcję najpierw w jej lokalnych buforach.

Faza 1: Transakcja wczytuje potrzebne dane do swoich lokalnych buforów i na nich dokonuje zmian bez zakładania żadnych blokad.

Faza 2: Transakcja sprawdza czy dokonane przez nią odczyty i zapisy nie pozostają w konflikcie z odczytami i zapisami zatwierdzonych już transakcji. Jeśli nie, następuje przepisanie zmian z lokalnych buforów do globalnych i zatwierdzenie transakcji. Jeśli tak, następuje restartowanie jeszcze raz tej samej transakcji - nie jest potrzebne wycofywanie zmian w bazie danych, bo żadne nie zostały wprowadzone.

Tylko w czasie realizacji Fazy 2 jest konieczność założenia blokad X na zmieniane obiekty.

**Wielowersyjność danych**

Realizację wielowersyjności danych można sobie wyobrazić w ten sposób, że procesy zapisujące na obiekcie tworzą nową wersję obiektu, podczas gdy procesy odczytujące korzystają ciągle ze starej wersji.

Rys. 12.5 Koegzystencja różnych wersji tego samego obiektu

Zapytanie odwołuje się do stanu bazy danych z pewnej chwili w przeszłości - na przykład, z chwili kiedy nastąpiło rozpoczęcie wykonywania zapytania lub, z chwili kiedy nastąpiło rozpoczęcie wykonywania transakcji.

|  |
| --- |
| Przyjmując model wielowersyjności danych, transakcje tylko odczytujące mogą działać bez zakładania blokad, w tym odczytywać obiekty z założoną blokadą X. |

W trybie READ COMMITED system Oracle pozwala zawsze odczytywać dane niezależnie od założonych blokad X (dane te mogą być w trakcie zmieniania przez inne transakcje). System używa wielowersyjności – podobnie jak przy wykonywaniu transakcji typu *tylko-odczyt*:

            SET TRANSACTION READ ONLY;

**Blokady w Oracle**

Przy wykonywaniu instrukcji SQL Oracle sam zakłada blokady na wiersze i tabele. Użycie instrukcji LOCK TABLE prowadzi do spowolnienia działania wykonywania transakcji, więc powinno być używane tylko w specjalnych przypadkach gdy chcemy zawczasu ograniczyć w odpowiedni sposób innym transakcjom dostęp do tabeli.

**12.4 Podsumowanie**

Wykład 12 przedstawił metody stosowane przez SZBD do współbieżnego wykonywania transakcji.

**12.5 Słownik pojęć**

[transakcja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Transakcje) - ciąg instrukcji SQL, które są wykonywane jako całość: albo wszystkie albo żadna z nich.

[aksjomaty ACID](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#ACID) - podstawowe właściwości jakie powinna spełniać implementacja transakcji przez SZBD: atomowość, spójność, izolacja, trwałość.

[atomowość (niepodzielność)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Atom) - aksjomat realizacji transakcji polegający na tym, że albo wszystkie akcje wchodzące w skład transakcji są wykonywane albo żadna.

[spójność](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Spoj) - aksjomat realizacji transakcji polegający na tym, że po wykonaniu transakcji stan bazy danych powinien być spójny (pod warunkiem, że przy rozpoczynaniu transakcji stan bazy danych był spójny).

[izolacja](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Izol) - aksjomat realizacji transakcji polegający na tym, że wynik działania transakcji powinien być taki sam, jakby od chwili rozpoczęcia transakcji nie działała na wspólnych danych żadna inna transakcja. Każdy użytkownik powinien mieć iluzję, że sam korzysta z bazy danych.

[trwałość](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Trwa) - aksjomat realizacji transakcji polegający na tym, że  dane zatwierdzone przez transakcję powinny być dostępne (ewentualnie do odtworzenia) nawet w sytuacji awarii oprogramowania lub sprzętu.

[plan](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Plan) - zaplanowanie w czasie wykonywania akcji odczytu i zapisu na obiektach bazy danych przez współbieżnie działające transakcje.

[plan szeregowy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Pls) - plan wykonania akcji transakcji, ustawiający wykonywanie transakcji w ciąg: najpierw akcje jednej transakcji, następnie akcje drugiej transakcji itd.

[plan szeregowalny](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Plss) - plan wykonania akcji transakcji, który jest równoważny pewnemu planowi szeregowemu.

[blokada obiektu](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Blokady) - ograniczenie używania obiektu przez innych użytkowników.

[blokada współdzielona, typu *S*](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Wsp) - daje transakcji współdzielony dostęp do zasobu. Np. kilka transakcji może jednocześnie pracować na tej samej tabeli. Jeśli transakcja zakłada współdzieloną blokadę, inne transakcje też mogą założyć współdzieloną blokadę, ale nie mogą założyć wyłącznej blokady.

[blokada wyłączna, typu X](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Wlcz) - daje transakcji wyłączny dostęp do obiektu. Tylko jedna transakcja może mieć założoną wyłączną blokadę na obiekcie i w tym czasie nie może być założonej żadnej innej blokady nawet współdzielonej.

[protokół ścisłego blokowania dwufazowego (Strict 2PL)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Strict 2PL) - zasady wykonywania transakcji przez system oparte na zakładaniu i zwalnianiu blokad gwarantujące realizację tylko planów szeregowalnych (co zapewnia zachodzenie aksjomatów izolacji i spójności).

[zakleszczenie (deadlock)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Zakls) - cykl transakcji oczekujących wzajemnie na zwolnienie blokady.

[fantom](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Fantomy) - wiersz, który zostaje wstawiony do tabeli po tym jak transakcja wykonała operację na tej tabeli a przed jej zatwierdzeniem (co potencjalnie oznacza, że gdyby ten wiersz był obecny przy wykonywaniu operacji, jej wynik byłby inny).

[optymistyczne blokowanie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Optymistyczne blokowanie) - wykonywanie transakcji w jej lokalnych buforach i dopiero na koniec sprawdzenie, czy można dokonane zmiany wprowadzić do bazy danych. Zmniejsza się w ten sposób blokowanie zasobów ale nie daje gwarancji, że po wykonaniu wszystkich akcji będzie można przepisać rezultat transakcji do bazy danych (jeśli nie będzie można, trzeba będzie uzyskany rezultat porzucić i zacząć wykonywanie transakcji od nowa).

[wielowersyjność](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w12.htm#Wielowersyjno) - tworzenie kopii obiektów tak aby transakcja zapisująca na obiekcie mogła być wykonywana współbieżnie z transakcjami tylko odczytującymi stan obiektu. Co więcej, każda transakcja odczytująca powinna mieć swoją własną wersję, aktualną w chwili rozpoczynania się transakcji

**Wykład 13**

**Dzienniki bazy danych i odtwarzanie**

**Wprowadzenie**

Ten wykład jest kontynuacją poprzedniego wykładu o transakcjach i jest poświęcony zabezpieczeniom realizowanym przez SZBD na wypadek różnego rodzaju awarii: serwera, dysku, komputera. Są one realizowane na poziomie transakcji użytkowników, to znaczy w chwili awarii wszystkie nie zatwierdzone transakcje są anulowane a efekt wszystkich zatwierdzonych transakcji jest trwały niezależnie od rodzaju awarii.

**13.1** **Dzienniki bazy danych**

W dzienniku bazy danych są zapisywane informacje o operacjach wykonywanych na bazie danych przez transakcje oraz inne dodatkowe informacje o zachodzących w bazie danych zdarzeniach.

Każdy rekord dziennika ma przypisany sobie identyfikujący go numer LSN (ang. *log sequence number*).  Przykładowo, rekord dziennika opisujący zmianę w bazie danych ma postać:

|  |
| --- |
| <typ operacji, IDtransakcji, IDstrony,  offset, długość, before-image, after-image, poprzedniLSN> |

gdzie poprzedniLSN to identyfikator LSN rekordu dziennika dla poprzedniej akcji w ramach tej samej transakcji. Rekord ten opisuje zmianę zawartości strony o identyfikatorze *IDstrony* od jej miejsca *offset* (czyli pozycji na stronie)i na fragmencie długości *długość*: wartość *before-image* (czyli poprzednia wartość) przechodzi na *after-image* (czyli nową wartość).

Rys. 13.1 Rejestracja zmian na stronie

Oprócz zmiany danych, rekordy dziennika są tworzone także dla innych akcji takich jak: *zatwierdzenie transakcj*i, *wycofanie transakcji*.

Są dwa rodzaje dzienników: *dzienniki wycofań* i *dzienniki powtórzeń*. Dziennik wycofań jest związany z wycofywaniem transakcji oraz z wielowersyjnością. Dziennik powtórzeń jest związany z odtwarzaniem bazy danych po awarii serwera lub nośnika danych. Ze względu na inne spełniane funkcje oba dzienniki zwykle są zapisywane osobno: dziennik wycofań razem z danymi w bazie danych, a dziennik powórzeń na innym nośniku danych niż pliki z danymi. W niektórych systemach używa się tylko dziennika powtórzeń, który, tak czy owak, zawiera w sobie informacje dziennika wycofań.

**13. 2** **Dziennik wycofań**

W celu umożliwienia wycofania transakcji SZBD zapisuje wszystkie zachodzące na stronach dyskowych zmiany w specjalnym *dzienniku wycofań* (ang. *undo log*) nazywanym w Oracle *segmentami wycofań* (ang. *undo segments*). Dziennik wycofań jest przechowywany razem z danymi w bazie danych. Gdy trzeba wycofać transakcję, system odczytuje w tył zapisy o zmianach wprowadzonych przez transakcję i przywraca poprzednie wartości danych w bazie danych - jednocześnie kasując rekordy dziennika wycofań.

**Dziennik wycofań i wielowersyjność**

Dziennik wycofań może być podstawą realizacji wielowersyjności. Zamiast explicite utrzymywać stare wersje obiektów, można je rekonstruować z aktualnego stanu bazy i informacji wpisanych do dziennika wycofań.

W przypadku wykonywania zapytania jego wyniki powinny być spójne i odpowiadać chwili rozpoczęcia jego wykonywania. W trakcie realizacji zapytania, inne transakcje mogą zmieniać zawartość stron, które są potrzebne do wykonania zapytania (jeśli nie stosujemy blokad współdzielonych S przy wykonywaniu zapytania). W takiej sytuacji, w oparciu o dziennik wycofań należy obliczyć zawartość potrzebnej strony w chwili rozpoczynania realizacji zapytania i użyć tę zawartość do wyznaczenia wyniku zapytania.

Mianowicie, system utrzymuje licznik SCN nazywany *systemowym numerem zmiany*. Każda zatwierdzona transakcja zwiększa ten licznik o jeden. Wartość SCN może być uważana za identyfikator zatwierdzanej transakcji. Na każdej stronie z danymi jest zapisany numer SCN ostatniej transakcji, która ją zmieniła.

Algorytm wykonywania zapytania

Niech q\_SCN będzie aktualnym numerem SCN w chwili rozpoczęcia wykonywania zapytania. W trakcie wykonywania zapytania są odczytywane strony z danymi. Dla każdej takiej strony z jej nagłówka jest odczytywana zapisana w niej wartość s\_SCN będąca numerem transakcji, która ją ostatnio zmieniła.

* Jeśli s\_SCN <= q\_SCN, wtedy aktualną zawartość strony można użyć w obliczeniach.
* Jeśli s\_SCN > q\_SCN, wtedy w oparciu o segmenty wycofań należy obliczyć zawartość strony w chwili q\_SCN i użyć tej zawartości do wykonania zapytania.

W podobny sposób są wykonywane transakcje raportujące typu READ ONLY.

**13.3** **Dziennik powtórzeń i odtwarzanie**

Dziennik rejestrujący wszystkie zmiany zachodzące w bazie danych jest nazywany *dziennikiem powtórzeń* (ang. *redo log*). Jest on z założenia trzymany na innym nośniku danych niż pliki z danymi w bazie danych. Na ogół, dokonuje się stale jego archiwizacji przepisując go na taśmę.

**Zasada WAL**

Dane zmienione przez transakcję nie muszą zostać zapisane na dysk w chwili kończenia transakcji (COMMIT/ROLLBACK) ale mogą zostać zapisane:

* przed jej zakończeniem (mówi się wtedy o zjawisku *kradzieży ramek* - ang. *frame stealing*);
* albo później, po jej zakończeniu (mówi się wtedy o strategii *nie używania siły* - ang*. no force*)

– nie ma to znaczenia dla operacji COMMIT, ROLLBACK i możliwości odtworzenia danych w przypadku awarii. Po zapisie do dziennika powtórzeń nawet awaria serwera lub dysku z danymi nie spowoduje utraty danych bo można je odtworzyć.

Informacja o zatwierdzeniu transakcji razem z informacjami o dokonanych przez nią zmianach są najpierw zapisywane do dziennika powtórzeń przechowywanego na innym nośniku danych (dysku) niż pliki z danymi. Jest to częścią zasady nazywanej *najpierw-zapis-do-dziennika* w skrócie  WAL – ang. *write-ahead logging*. Zasada WAL składa się z dwóch punktów:

(1) Najpierw do dziennika powtórzeń na dysku jest zapisywany rekord opisujący zmianę zawartości strony dopiero potem strona może być zapisana na dysku (zapewnia możliwość przywrócenia poprzedniej zawartości strony w przypadku awarii czyli zapewnia atomowość transakcji).

(2) Transakcja kończy się wtedy, gdy wszystkie rekordy dziennika powtórzeń dotyczące jej akcji zostaną przepisane z pamięci wewnętrznej na dysk a nie wtedy gdy dane zmienione przez transakcję zostaną zapisane na dysk (po zatwierdzeniu nawet w przypadku awarii jest możliwość przywrócenia danych czyli zapewnia trwałość transakcji).

**Odtwarzanie (ang. recovery)**

Gdy nastąpi awaria dysku, rekordy dziennika powtórzeń (z części on-line na osobnym dysku lub części archiwizacyjnej na taśmie) zastosowane do kopii zabezpieczającej bazy danych pozwalają odtworzyć stan bazy danych i struktur danych w pamięci RAM w chwili awarii. Jest to proces nazywany *odtwarzaniem do przodu*.

Podobnie, w przypadku awarii serwera bazy danych analiza stron bazy danych i dziennika powtórzeń pozwala na odtworzenie stanu bazy danych w chwili awarii.

Ponieważ w dzienniku powtórzeń zapisane są również rekordy  dziennika wycofań, jest możliwość wycofania nie zatwierdzonych transakcji, których działanie zostało przerwane w chwili awarii:

* na przykład, przy transferze pieniędzy - z jednego konta pieniądze zostają zdjęte, w tej chwili następuje awaria, na drugie konto pieniądze już nie zostają przelane.

Jest to proces nazywany *odtwarzaniem do tyłu*. W rezultacie tych dwóch odtwarzań jest możliwość doprowadzenia stanu bazy danych do spójnego stanu.

**Punkt kontrolny**

Odtwarzanie po awarii serwera musi mieć określony punkt wyjściowy.

W trakcie normalnej pracy systemu w celu przyśpieszenia ewentualnego odtwarzania bazy danych po awarii serwera, jest realizowana operacja nazywana *punktem kontrolnym* ang*. checkpoint*, polegająca na przepisaniu wszystkich zmienionych stron w buforze RAM na dysk, również tych, których zmiany nie zostały jeszcze zatwierdzone. Oznacza to synchronizację aktualnych zawartości buforów pamięci RAM i dysku.

Ponadto, zapisywana jest do dziennika powtórzeń informacja o wszystkich transakcjach i stronach, aktualnie przetwarzanych w buforze pamięci RAM. Dla każdej transakcji jest podawany identyfikator rekordu dziennika powtórzeń rejestrującego ostatnią zmianę, jaka zaszła dla tej transakcji. W osobnym, bezpiecznym miejscu, zapisujemy identyfikator rekordu dziennika powtórzeń opisującego ostatni punkt kontrolny. Od niego rozpoczynamy odtwarzanie po awarii serwera. Dla każdej strony, która w tym momencie była w buforze w RAM, powtarzamy operacje zapisane w dzienniku powtórzeń. Operacja punktu kontrolnego powinna być wykonywana w regularnych odstępach czasu na bazie danych.

W punkcie kontrolnym istotne jest zapisanie w dzienniku powtórzeń, jakie transakcje i jakie strony znajdują się aktualnie w pamięci RAM. Można przyśpieszyć wykonywanie punktu kontrolnego, co za tym idzie i działanie całego serwera bazy danych, przez nie przepisywanie w tym momencie na dysk zmienionych zawartości buforów bazy danych. W przypadku konieczności odtworzenia po awarii serwera, system może odtworzyć stan każdej strony w oparciu o rekordy dziennika powtórzeń wychodząc od ostatnio zapisanej zawartości strony w bazie danych. Oznacza to, że wtedy odtwarzanie trwa dłużej, ale za to punkty kontrolne, trwają krócej. Punkt kontrolny tego rodzaju nosi nazwę *niepełnego punktu kontrolnego* (ang. *fuzzy checkpoint*).

**13. 4** **Rezerwowa baza danych (*stand-by*)**

Rezerwowa baza danych jest to dodatkowa instalacja bazy danych na osobnym komputerze utrzymywana stale w specjalnym trybie *standby* – z ciągle dokonywanym odtwarzaniem w oparciu o kopie zarchiwizowanego dziennika powtórzeń generowane przez główną, operacyjną bazę danych.

W przypadku awarii dysku lub katastrofy w rodzaju ataku terrorystycznego, trzęsienia ziemi, pożaru czy kradzieży, rezerwowa bazy danych przechodzi z trybu *standby* w tryb *read write* i przejmuje obowiązki głównej bazy danych.

Rezerwowa baza danych znajduje się zwykle w fizycznie oddalonym węźle, do którego stale są przesyłane kolejne części zarchiwizowanego dziennika powtórzeń.

Rezerwowej bazy danych można używać do raportowania – przechodząc w tryb *read only*  – napływające pozycje zarchiwizowanego dziennika powtórzeń utrzymywane są w kolejce, dopóki nie wrócimy do trybu *standby*. Po przejściu bazy danych w tryb *read write* nie jest już możliwy jej powrót do trybu *standby*.

Zamiast rezerwowej bazy danych alternatywę stanowią:

1. użycie repliki bazy danych (będzie o nich mowa w wykładzie 14);
2. użycie zwierciadlanego odbicia bazy danych - dokładnej jej kopii, która w każdej chwili może przejąć jej funkcje.

Dla pełności obrazu trzeba jeszcze wspomnieć, że aksjomat ACID trwałości może być zrealizowany przez użycie *macierzy dysków RAID* (skrót od *Redundant Array of Independent Disks*) umożliwiającej redundantny zapis danych; gdy jeden dysk ulegnie awarii - dane są pobierane z drugiego.

**13.5 Podsumowanie**

Wykład 13 przedstawił metody stosowane przez SZBD do zabezpieczeń przed utratą danych w sytuacji zaistnienia awarii.

**13.6 Słownik pojęć**

[dziennik wycofań](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w13.htm#Dziennik wycofa) - dziennik rejestrujący wszystkie zmiany zachodzące na stronach dyskowych umożliwiający wycofanie transakcji. Dziennik ten jest zapisywany razem z danymi. W Oracle dziennik ten nazywa się *segmentami wycofań*.

[dziennik powtórzeń](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w13.htm#Dziennik) - dziennik rejestrujący wszystkie zmiany zachodzące na stronach dyskowych umożliwiający odtworzenie bazy danych po wystąpieniu awarii. Dziennik ten jest zapisywany w innym miejscu niż dane np. na innym dysku. Na ogół jest stale archiwizowany tworząc zarchiwizowany dziennik powtórzeń – przechowywany na nośniku pamięci masowej np. na taśmie magnetycznej. Umożliwia to cykliczne wykorzystywanie miejsca na zapis dziennika powtórzeń na dysku.

[odtwarzanie](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w13.htm#Recover) - proces odtwarzania, odzyskiwania danych straconych w wyniku awarii serwera lub dysku z danymi.

[punkt kontrolny](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w13.htm#ckpt) - synchronizacja pamięci RAM i struktur bazy danych zapisanych na dyskach. Stanowi punkt startowy dla procesu odtwarzania po awarii serwera.

[rezerwowa baza danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w13.htm#Rezerwowa) - dodatkowa instalacja bazy danych na osobnym komputerze utrzymywana stale w specjalnym trybie oczekiwania (*standby*)– z ciągle dokonywanym odtwarzaniem w oparciu o kopie zarchiwizowanego dziennika powtórzeń generowane przez główną, operacyjną bazę danych.

## Wykład 14

***Rozproszona baza danych***

**Streszczenie**

Wykład składa się z trzech części. W pierwszej przedstawione jest pojęcie *rozproszonej bazy danych*. Omówione zostają jej zalety i wady w porównaniu z konwencjonalną, scentralizowaną bazą danych. Przedstawione jest rozszerzenie algorytmów optymalizacji zapytań i zatwierdzania transakcji na przypadek rozproszonej bazy danych.

W drugiej części wykładu przedstawiona jest implementacja pojęć rozproszonej bazy danych w konkretnym systemie baz danych – Oracle.

W trzeciej części wykładu jest rozważony problem integracji informacji pochodzących z różnych baz danych.

**14.1** **Baza scentralizowana**

W sytuacji rozproszenia użytkowników w różnych węzłach sieci internetowej są możliwe różne organizacje bazy danych. Najprostszym rozwiązaniem jest *baza scentralizowana*, w której dane są przechowywane w jednym węźle sieci. W takim przypadku można utrzymywać jeden centralny system kontroli danych. Przetwarzanie transakcji i odtwarzanie po awarii są objęte sprawdzonymi algorytmami i protokołami. Jeśli kierowane z węzłów sieci do bazy danych transakcje dotyczą wszystkich danych (nie zależą istotnie od samego węzła), wówczas rozwiązanie scentralizowane jest dobre. Przykładem takich systemów mogą być ogólnokrajowe rejestry: podmiotów gospodarczych, podatników, samochodów.

Wady bazy scentralizowanej:

1. Potencjalnie długi czas oczekiwania na rezultaty z odległego węzła sieci – bardzo długi przy awarii sieci lub centralnego węzła.
2. Brak kontroli nad danymi specyficznymi dla danego miejsca.
3. Generowanie dużego ruchu w sieci.

**14.2 Baza rozproszona**

W przypadku, gdy transakcje w lokalnych węzłach dotyczą na ogół danych lokalnych i stosunkowo rzadko zachodzi potrzeba sięgnięcia do danych dotyczących węzłów odległych, lepszym rozwiązaniem może być zbiór połączonych ze sobą baz lokalnych – z danymi przechowywanymi w węzłach lokalnych.

|  |
| --- |
| Z *rozproszoną bazą danych* mamy do czynienia wtedy, gdy zbiór lokalnych baz stanowi całość w sensie jednego modelu danych i koordynacji wykonywanych transakcji. |

W rozproszonej bazie danych występuje rozłożenie danych do węzłów sieci poprzez ich *fragmentację* (*podział*) lub *replikację* - do różnych konfiguracji sprzętowych i programistycznych na ogół rozmieszczonych w różnych (geograficznie) miejscach organizacji.

* *Fragment danych* stanowi pewien podzbiór wszystkich danych całej bazy danych przechowywany w jednym węźle sieci.
* *Replika danych* stanowi kopię całości lub jakiejś części danych przechowywanych w innym miejscu niż oryginał.

Rozproszona baza danych powinna reprezentować pojedynczy model danych firmy. Przypuśćmy, że tworzymy model danych odzwierciedlający zarządzanie kadrami w pewnej organizacji i projektujemy rozproszoną bazę danych:

* w biurze w Gdańsku znajdują się dane dotyczące Polski Północnej,
* w biurze w Krakowie znajdują się dane dotyczące Polski Południowej,
* a w biurze w Warszawie znajdują się dane dotyczące Polski Środkowej.

Informacje  o strukturze firmy są replikowane i przechowywane w każdym węźle. Zakładamy, że okresowo dane dotyczące wszystkich trzech regionów są zbierane razem, dając obraz całego kraju.

Są dwa typy fragmentacji: *pionowa* i *pozioma*.

*Fragment poziomy* stanowi podzbiór wierszy w tabeli, np. dane z jednego rejonu kraju.

*Fragment pionowy* stanowi podzbiór kolumn w tabeli, np. identyfikator, data urodzenia i zarobki pracowników.

Fragmentację pionową wykonuje się przez wykonanie rzutu na podzbiór kolumn zawierający w sobie klucz główny. Odtworzenie oryginalnej tabeli polega na zastosowaniu złączenia w oparciu o wartości klucza głównego.

Fragmentację poziomą uzyskuje się przez zastosowanie operacji selekcji. Odtworzenie oryginalnej tabeli polega na zastosowaniu operacji sumy UNION.

|  |  |  |
| --- | --- | --- |
| **Operacja** | ***Fragmentacja pozioma*** | ***Fragmentacja pionowa*** |
| ***rozkład*** | selekcja | rzut |
| ***złożenie*** | suma | złączenie |

Dla użytkownika rozproszona baza danych powinna wyglądać dokładnie tak samo jak jedna, scentralizowana baza danych. W związku z tym rozproszona baza danych powinna mieć trzy właściwości *przezroczystości*:

1. *Przezroczystość geograficzna*. Użytkownicy nie muszą wiedzieć, w którym dokładnie węźle są przechowywane dane, które oglądają.
2. *Przezroczystość fragmentacji*. Użytkownicy nie muszą wiedzieć, w jaki sposób dane są podzielone. W przykładzie zarządzania kadrami mamy do czynienia logicznie z jedną bazą danych, a fizycznie z trzema jej fragmentami.
3. *Przezroczystość replikacji*. Użytkownicy nie muszą wiedzieć, w jaki sposób dane są replikowane i z której repliki pochodzą dane, które oglądają. W przykładowej bazie danych, w każdym z trzech biur jest przechowywana kopia informacji o strukturze firmy. Kiedy zachodzi potrzeba okresowej aktualizacji tych danych, użytkownicy nie muszą być świadomi tego, że aktualizacja dotyczy każdego z trzech miejsc.

Zalety rozproszenia (w porównaniu z systemem scentralizowanym)

1. Odwzorowanie w bazie danych geograficznego podziału organizacji. Jeśli organizacja ma charakter narodowy lub ponadnarodowy, system musi uwzględniać podział organizacji na departamenty, oddziały itd.
2. Większą kontrolę nad danymi można uzyskać przechowując je w miejscu, gdzie są one potrzebne. Jeśli biurowi w Gdańsku przekażemy odpowiedzialność za dane dotyczące Polski Północnej, wówczas tylko personel w Gdańsku będzie miał prawo aktualizować te dane.
3. Utrzymywanie replikacji danych zwiększa niezawodność systemu. Przechowywanie danych w miejscach, gdzie są one potrzebne, poprawia ich dostępność. W dużych bankach bazy danych obsługujące transakcje klientów muszą działać 24 godziny przez wszystkie dni w tygodniu. Powszechnie używa się dwóch lub więcej bliźniaczych baz danych, jednej do bezpośredniego wykonywania transakcji, drugiej jako jej kopii (bazy rezerwowej lub repliki).
4. Działanie systemu może się istotnie poprawić, jeśli dokona się prawidłowego rozproszenia danych. Jeśli większość zapytań generowanych w lokalnej bazie danych będzie wykonywana w tej lokalnej bazie danych zamiast w dużej scentralizowanej, wówczas jest duża szansa przyśpieszenia dostępu do danych przy operacjach wyszukiwania i aktualizacji.

System zarządzania rozproszoną bazą danych jest bardziej skomplikowany w porównaniu z bazą scentralizowaną.

1. Słownik danych (katalog systemowy) rozproszonej bazy danych jest znacznie bardziej złożony. Obejmuje on na przykład informacje o położeniu fragmentów i replik tabel bazowych.
2. Problemy związane ze współbieżnością są zwielokrotnione w systemach rozproszonych. Propagowanie aktualizacji do szeregu różnych węzłów jest skomplikowane. Gdy jest wiele kopii tego samego obiektu, trudno jest utrzymać spójność danych. Trudno jest zrealizować naturalny postulat, że każdy użytkownik, niezależnie skąd loguje się, widzi to samo. Obok dotychczasowych rodzajów awarii system rozproszony musi sobie dać radę również z awariami połączeń sieciowych.
3. Optymalizator zapytań w prawdziwym systemie rozproszonym powinien być w stanie użyć właściwości topologiczne sieci (np. o koszcie przesłania danych między dwoma węzłami) przy decydowaniu, jak najlepiej wykonać dane zapytanie.
4. Aby zapewnić odporność na awarie, system zarządzania rozproszoną bazą danych nie powinien być ulokowany w jednym miejscu. Zarówno oprogramowanie, jak i dane powinny zostać rozproszone po różnych miejscach w sieci.

**Transakcje rozproszone**

W kontekście rozproszonej bazy danych odróżnia się zapytania i transakcje *odległe* od *rozproszonych*.

*Zapytanie odległe, transakcja odległa* – dane do realizacji znajdują się w jednym, odległym węźle sieci.

*Zapytanie rozproszone, transakcja rozproszona* – dane do realizacji znajdują się w różnych węzłach sieci.

Do kosztu wykonywania zapytania dochodzi koszt przesłań danych między węzłami w sieci - i ten koszt ma największy wpływ na czas wykonywania zapytania.

**Zapytania rozproszone**

Załóżmy, że dane do poniższego zapytania znajdują się w różnych węzłach sieci.

SELECT AVG(E.Sal)  
 FROM Emp E  
 WHERE E.Deptno > 30 AND E.Deptno < 70

* Fragmentacja pozioma: Załóżmy, że wiersze z *E.Deptno* < 5O są w Krakowie, natomiast z *E.Deptno* >= 50 w Warszawie. Aby wykonać zapytanie, obliczamy osobno w obu węzłach SUM(*E.Sal*) i COUNT(*E.Sal*). Gdyby był warunek, np. *E.Empno*>70, wtedy wykonywanie zapytania odbywałoby się tylko w jednym węźle.
* Fragmentacja pionowa: Załóżmy, że kolumna E.Sal jest w Warszawie, E.Deptno w Krakowie, natomiast E.Empno w obu bazach. Wystraczy sprowadzić dane potrzebne do wykonania zapytania do jednego węzła (można wykorzystać warunki ograniczające) i zrekonstruować potrzebny fragment tabeli, po czym wykonać obliczenia. W naszym przypadku dane z Krakowa (odpowiednia projekcja selekcji) zostałyby przesłane do Warszawy.
* Replikacja: Załóżmy, że kopie tabeli *Emp* są dostępne w Warszawie i Krakowie. Wybór węzła wykonującego zapytanie jest uzależniony od lokalnego kosztu wykonania zapytania i od kosztu przesłania wyników.

**Rozproszone złączenia**

Jest kilka metod wykonywania rozproszonego złączenia. Rozważymy je na przykładzie. Załóżmy, że tabela *Dept* jest dostępna w Warszawie a  *Emp* jest dostępna w Krakowie.

 SELECT E.Ename, d.Dname  
 FROM Emp E INNER JOIN Dept D ON E.Deptno=D.Deptno  
 WHERE E.Job='MANAGER' AND d.Loc='Warszawa'

Oto te metody.

1. Zastosuj metodę Nested Loops Join.

2. Sprowadź jedną tabelę (ewentualnie jej odpowiednią projekcję selekcji) do węzła gdzie jest druga tabela i tam wykonaj zapytanie.

3. Gdy rozmiar wyniku jest duży w porównaniu z rozmiarem tabel, sprowadź tabele do końcowego węzła i tam je złącz.

4. Metoda *półzłączeń*

1. W Warszawie, dokonaj projekcji tabeli *Dept* na kolumny złączenia i prześlij wynik do Krakowa. Można skorzystać z selekcji d.Loc='Warszawa' ograniczającej zbiór wierszy.
2. W Krakowie, dokonaj złączenia projekcji tabeli *Dept* z tabelą *Emp*. Wynik nazywa się *redukcją* tabeli *Emp* względem *Dept*. Prześlij redukcję tabeli *Emp* do Warszawy. Można skorzystać z selekcji E.Job='MANAGER' ograniczającej zbiór wierszy.
3. W Warszawie, dokonaj ostatecznego złączenia tabeli *Dept* z redukcją tabeli *Emp*.

* Idea metody półzłączeń: koszt przesłania całej tabeli zastępujemy kosztem obliczenia i przesłania kolejno projekcji i redukcji.

Rys. 14.1 Przesyłanie danych w sieci w metodzie półzłączeń

**Optymalizacja rozproszonego zapytania**

Rozważ wszystkie plany, wybierz najtańszy; podobnie jak w scentralizowanym przypadku. Oto różnice w porównaniu z przypadkiem scentralizowanym.

Różnica 1: Trzeba uwzględnić koszty komunikacji między węzłami; trzeba podejmować decyzję, którą replikę wybrać.

Różnica 2: Trzeba wziąć pod uwagę specyfikę każdego węzła lokalnego (np. inny rodzaj SZBD).

Różnica 3: Trzeba wziąć pod uwagę specyficzne metody rozproszonego złączenia.

**Odświeżanie** **replik**

Są dwie metody synchronizacji (odświeżania) replik:

* *Synchroniczna replikacja*. Zanim modyfikująca transakcja zostanie zatwierdzona należy dokonać aktualizacji wszystkich replik (obejmuje zakładanie blokad, wymianę komunikatów w sieci). Możliwe jest astosowanie zarówno blokowania pesymistycznego jak i optymistycznego. Wynik odczytywania danych tabeli jest zawsze taki sam dla każdej jej repliki.
* *Asynchroniczna replikacja*. Kopie zmodyfikowanej tabeli są tylko okresowo aktualizowane – metoda znacznie tańsza; ale chwilowo repliki mogą nie być zsynchronizowane ze zmodyfikowaną tabelą.

**Modyfikacje przez repliki**

Na ogół repliki stanowią kopie tabeli nadrzędnej, na której wyłącznie są wykonywane wszystkie operacje INSERT, DELETE i UPDATE.

*Replika modyfikowalna* to replika przez którą można dokonywać zmian w taleli oryginale wykonując instrukcje INSERT/DELETE/UPDATE. Możliwe jest zastosowanie zarówno blokowania pesymistycznego jak i optymistycznego.

W przypadku zastosowania blokowania optymistycznego, problemem jest, co robić w przypadku konfliktu, np. gdy w dwóch różnych węzłach dokonano różnych modyfikacji tego samego wiersza - szczególnie wtedy kiedy obie modyfikujące transakcje zakończyły się już zatwierdzeniem. Możliwe metody rozwiązywania kolizji są następujące:

1. nie pozwolić jednej z transakcji na zatwierdzenie i po stwierdzeniu zaistnienia konfliktu, wycofać ją - zgodnie ze standardowymi regułami,
2. ustalić ogólne reguły np. oparte na priorytetach,
3. pozostawić konflikty do rozstrzygnięcia administratorowi bazy danych.

Ustalenie zasad ograniczających możliwości modyfikacji przez repliki ma prowadzić do zachowania aksjomatu trwałości wykonywania transakcji (czwarty aksjomat ACID) - albo przy najmniej tylko do kontrolowanego jego naruszania.

**14.3** **Rozproszone zatwierdzanie i odtwarzanie**

W porównaniu z przypadkiem scentralizowanym pojawiają się nowe problemy:

1. Dodatkowe rodzaje awarii, np. związane z połączeniami sieciowymi i odległymi węzłami.
2. Gdy "pod-transakcje" całej transakcji są wykonywane w różnych węzłach, wszystkie razem  wykonują COMMIT albo żadna z nich. Potrzebny jest specjalny protokół zatwierdzania.
3. W każdym węźle jest utrzymywany odrębny dziennik (log) wykonywanych akcji, tak jak w scentralizowanej bazie danych. W tym dzienniku są odnotowywane akcje protokołu zatwierdzania.

Transakcja rozproszona jest realizowana przez zbiór węzłów w sieci. Jest jeden wyróżniony węzeł, który inicjuje i koordynuje transakcję – jest nazywany *koordynatorem transakcji*.

Rys. 14.2 Model wykonywania transakcji rozproszonej

Strzałki oznaczają przepływ sterowania, nie koniecznie przepływ danych. Baza danych użytkownika może być jedną z baz danych biorących udział w transkcji, ale nie koniecznie.

Podstawą realizacji transakcji rozproszonej jest następujący protokół.

Protokół dwufazowego zatwierdzania (2PC ang. Two-Phase Commit)

1. Koordynator wysyła do wszystkich węzłów komunikat *prepare*.
2. Węzły podejmują lokalnie decyzję o gotowości do zatwierdzenia *prepare* lub konieczności wycofania transakcji *abort* i zapisują w swoim dzienniku odpowiednio albo rekord *prepare* albo rekord *abort* i następnie wysyłają do koordynatora odpowiednio komunikat albo *yes* albo *no*.
3. Gdy koordynator uzyska jednomyślną odpowiedź *yes*, zapisuje do swojego dziennika rekord *commit* i wysyła do wszystkich węzłów komunikat *commit*. W przeciwnym przypadku zapisuje do swojego dziennika rekord *abort* i wysyła do wszystkich węzłów komunikat *abort*.
4. Węzły zapisują w swoim dzienniku odpowiednio rekord *abort/commit* albo *end,* kończą transakcję usuwając informację o niej z bufora bazy danych w pamięci RAM*.* a następnie wysyłają do koordynatora komunikat *ack*.
5. Po otrzymaniu wszystkich potwierdzeń *ack* koordynator zapisuje do swojego dziennika rekord *end* i kończy transakcję usuwając informację o niej z buforu bazy danych w pamięci RAM*.*

Komentarz na temat protokołu 2PC

1. Dwufazowość - dwie rundy komunikacji: pierwsza - głosowanie; druga - kończenie. Obie są inicjowane przez koordynatora.
2. Koordynator w fazie głosowania, gdy nie ma odpowiedzi z jednego z węzłów, może zadecydować o wycofaniu (*abort*) całej transakcji - jest zwykle określone ograniczenie czasowe oczekiwania na odpowiedź.
3. Każdy węzeł przed lub w czasie fazy głosowania może zadecydować o wycofaniu (*abort*) transakcji - również wtedy gdy nie może się połączyć z koordynatorem.
4. Każdy komunikat odzwierciedla decyzję nadawcy; aby mieć pewność odporności na awarie, decyzja jest najpierw zapisywana do dziennika transakcji (logu).
5. Dwu-fazowe zatwierdzanie (2PC) z domyślnym wycofaniem: w przypadku podjęcia decyzji o wycofaniu zarówno koordynator jak i węzeł lokalny od razu dokonują wycofania transakcji. Brak transakcji w pamięci RAM oznacza, że została ona wycofana.

**14.4** **Implementacja rozproszonych baz danych w Oracle**

Przede wszystkim Oracle dostarcza oprogramowanie sieciowe *Oracle Net* umożliwiające komunikację między bazami danych Oracle oraz obsługę transakcji rozproszonych działających na więcej niż jednej bazie danych – w tym zatwierdzanie takich transakcji i ich wycofywanie.

Z punktu widzenia projektanta rozproszonej bazy danych najważniejsze są dwa nowe rodzaje obiektów tworzonych w lokalnej bazie danych, za pomocą których uzyskuje się dostęp do obiektów w odległych bazach danych. W ten sposób ze zbioru lokalnych baz danych można utworzyć jedną rozproszoną bazę danych. Obiekty te to:

1. *Powiązanie z bazą danych* (ang. *database link*) – jest to zapisana w bazie danych ścieżka sieciowa do odległej bazy danych.

2. *Migawka, perspektywa zmaterializowana* (ang. *snapshot*, *materialized view*) – lokalna kopia (replika) danych znajdujących się w jednej lub więcej odległych bazach danych. Migawka ma charakter perspektywy odświeżanej co określony przedział czasu (może więc nie zawierać danych w pełni aktualnych!).

**Powiązanie z bazą danych**

Składnia tworzenia powiązania z bazą danych jest następująca:

|  |
| --- |
| CREATE DATABASE LINK *nazwa\_powiązania*  CONNECT TO *użytkownik* IDENTIFIED BY *hasło*  USING ’*nazwa\_usługi*’; |

gdzie:

* *użytkownik/hasło* – dotyczą konta, na które ma zostać dokonane logowanie w odległej bazie danych, jeśli ich brak – używana jest nazwa użytkownika i hasło z lokalnej bazy danych;
* *nazwa\_usługi* – nazwa usługi (aliasu bazy danych) Oracle Net np. zdefiniowanej w lokalnym pliku konfiguracyjnym TNSNAMES.ORA.

Po utworzeniu powiązania z bazą danych można korzystać z tabel i perspektyw w tej odległej bazie danych, tak jakby znajdowały się one w lokalnej bazie danych – dołączając do nazwy tabeli lub perspektywy napis @*nazwa\_powiązania*. Na przykład, instrukcja:

CREATE DATABASE LINK baza  
CONNECT TO scott IDENTIFIED BY tiger  
USING 'elektron';

tworzy powiązanie bazodanowe o nazwie *baza* z odległą bazą danych określoną przez sieciową usługę Oracle o nazwie *elektron*. Przy wykonywaniu instrukcji

SELECT \*  
FROM Emp@baza;

lokalny serwer Oracle łączy się, używając oprogramowania Oracle Net, z odległą bazą danych *elektron*. Oprogramowanie Oracle Net znajdujące się na docelowym komputerze przechwytuje zgłoszenie i dokonuje logowania w bazie danych *elektron* na konto *scott/tiger*. Serwer wykonuje przesłaną instrukcję SELECT i przesyła przez sieć z powrotem wyniki zapytania, jednocześnie wylogowując użytkownika *scott/tiger*.

System Oracle potrafi wykonywać dowolne instrukcje SQL, tak jakby tabele i perspektywy z odległych baz danych znajdowały się w lokalnej bazie danych np.

SELECT \* FROM Emp@warszawa  
UNION  
SELECT \* FROM Emp@gdansk  
UNION  
SELECT \* FROM Emp@katowice;

czy

UPDATE Emp@baza  
SET Salary = Salary \* 1,1;

**Synonimy**

Definiuje się zwykle synonimy dla obiektów w odległej bazie danych, np.

CREATE SYNONYM Klienci  
FOR Klienci@Baza\_szkola;

Następnie używamy nazwy Klienci, tak jakby to była to zwykła tabela.

**Migawki (****repliki)**

Drugi rodzaj obiektów specyficzny dla rozproszonej bazy danych *replika* (*migawka*, *perspektywa zmaterializowana*) jest tworzona za pomocą instrukcji:

|  |
| --- |
| CREATE SNAPSHOT *nazwa\_migawki* REFRESH NEXT *przedział\_czasu* AS *zapytanie*; |

gdzie:

1. *przedział\_czasu* – określa, co jaki czas należy odświeżać migawkę,
2. *zapytanie* – określa zapytanie, które tworzy zawartość migawki i które jest ponawiane przy każdym odświeżaniu migawki.

Zamiast słowa kluczowego SNAPSHOT można też używać słowo kluczowe MATERIALIZED VIEW.

Instrukcja:

CREATE SNAPSHOT Wszyscy\_prac  
REFRESH NEXT Sysdate+1  
AS SELECT \* FROM Emp@warszawa  
   UNION  
   SELECT \* FROM Emp@krakow  
   UNION  
   SELECT \* FROM Emp@gdansk;

tworzy migawkę złożoną z danych o pracownikach pochodzących z trzech oddziałów firmy w Warszawie, Krakowie i Gdańsku. Zawartość migawki ma być odświeżana raz na dzień. Po zdefiniowaniu migawki *Wszyscy\_prac* można jej używać w zapytaniach, tak jakby to była zwykła tabela lub perspektywa. Na przykład:

SELECT \*  
FROM Wszyscy\_prac  
WHERE Job = 'MANAGER';

wypisuje informacje o wszystkich osobach pracujących w firmie na stanowisku MANAGER.

**Implementacja migawki**

W lokalnej bazie danych dla każdej migawki są tworzone trzy obiekty:

1. *tabela*, do której jest zapisywany wynik zapytania (stąd nazwa *perspektywa zmaterializowana*),
2. *indeks* dla klucza głównego,
3. *perspektywa* służąca do wyświetlania i używania zawartości migawki.

**Odświeżanie migawki**

Rozróżniamy dwa typy migawek względem instrukcji SELECT występującej w jej definicji: *proste* i *złożone*. W przypadku migawki *złożonej* nie ma żadnych ograniczeń. Jeśli w instrukcji SELECT nie występują ani klauzule GROUP BY, CONNECT BY, DISTINCT ani funkcje sumaryczne, ani operatory zbiorowe, ani złączenia tabel, to taka migawka jest *prosta*.

Dla migawki prostej jest możliwe *szybkie odświeżanie* (opcja REFRESH FAST) pod warunkiem utworzenia w węźle, w którym znajduje się tabela-oryginał tej migawki, specjalnego dziennika. *Dziennik migawki* jest tabelą, do której są wpisywane zmiany dokonywane na tabeli-oryginale. Następnie, przy odświeżaniu migawki zamiast przesyłać całą zawartość tabeli-oryginału, jest przesyłana tylko zawartość dziennika migawki. Oto przykład instrukcji zakładającej dziennik migawki na tabeli *Emp*:

CREATE SNAPSHOT LOG ON Emp  
TABLESPACE users;

Ze względu na spójność referencyjną - łączy się migawki w grupy i dokonuje się ich jednoczesnego odświeżania.

**Modyfikowanie danych przez migawkę**

Jeśli w definicji migawki prostej użyjemy klauzuli FOR UPDATE, utworzymy *migawkę modyfikowalną*:

CREATE SNAPSHOT Rep\_emp  
REFRESH FAST NEXT Sysdate+1  
FOR UPDATE  
AS SELECT \* FROM Emp@warszawa;

Można wówczas dokonywać zmian w taleli oryginale wykonując operacje INSERT/DELETE/UPDATE na migawce, np.

UPDATE Rep\_emp  
SET Sal = Sal\*1.05  
WHERE Ename = 'SCOTT';

**14.5** **Problem integracji informacji**

W zastosowaniach baz danych coraz częściej pojawia się problem integracji danych pochodzących z różnych źródeł danych - w szczególności z baz danych.

1. Powiązane dane istnieją w różnych miejscach i może zaistnieć potrzeba jednoczesnego ich użycia przez jedną aplikację.
2. Bazy danych mogą się różnić:
   * *modelem* (np. relacyjny, obiektowo-relacyjny, hierarchiczny, XML, pliki MS Excel),
   * *schematem* (np. znormalizowany, nieznormalizowany),
   * *terminologią* (np. czy konsultanci firmy są pracownikami, czy emerytowani pracownicy są pracownikami),
   * *konwencjami* (np. stopnie Celsjusza lub Fahrenheita; mile lub kilometry).

Przykład

Dwie szkoły wyższe chcą się połączyć. Każda z nich ma swoją osobną bazę danych. Władze nowej szkoły wyższej chcą mieć zintegrowany widok na całość.

**Dwa podejścia do integracji**

1. *Hurtownia danych*. Skopiuj dane źródłowe do centralnej bazy danych dokonując ich transformacji do wspólnego schematu. Hurtownie danych są tematem osobnego wykładu.
   * Sprowadzanie danych i ich transformacja jest dokonywane raz na pewien czas – np. raz na dzień.

Rys. 14.3 Rozwiązanie problemu integracji za pomocą hurtowni danych

1. *Mediator*. Utwórz perspektywę wszystkich źródeł danych, tak jakby były zintegrowane.
   * Wyznaczaj wynik zapytania do perspektywy przez jego tłumaczenie do źródeł danych a następnie tam wykonuj zapytanie.

Rys.14.4 Rozwiązanie problemu integracji za pomocą modułu mediatora

**Moduł osłony (wrapper) i rola języka XML**

Moduł stojący między źródłową bazą danych a hurtownią danych bądź mediatorem nazywa się *osłoną* (ang. wrapper).

Filtruje on dane ze źródłowej bazy danych na zewnątrz (zgodnie ze specyfikacją). Najwygodniejszym uniwersalnym językiem wynikowym jest XML (do komunikacji, co najmniej, między mediatorem a osłoną).

Zapytanie do źródłowej bazy danych (przygotowywane przez mediatora) też może być sformułowane przy użyciu XML. Tłumaczenia na język źródłowej bazy danych dokonuje konkretny moduł osłony związany z konkretną bazą danych.

**14.6 Podsumowanie**

Zostało przedstawione pojęcie rozproszonej bazy danych i omówione jej zalety i wady w porównaniu z konwencjonalną, scentralizowaną bazą danych.

Przedstawiona została implementacja rozproszonej bazy danych w konkretnym systemie baz danych – Oracle.

Rozważony też został problem integracji informacji pochodzących z różnych heterogenicznych baz danych.

**14.7 Słownik pojęć**

[fragment danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#Fragment) - podzbiór wszystkich danych całej bazy danych przechowywany w jednym węźle sieci.

[replika danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#Replika) - kopia całości lub jakiejś części danych przechowywanych w innym węźle sieci niż oryginał.

[replikacja synchroniczna](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#Modyfikacja replik) - zanim modyfikująca transakcja zostanie zatwierdzona dokonuje się aktualizacji (odświeżenia) wszystkich replik (obejmuje zakładanie blokad, wymianę komunikatów w sieci).

[replikacja asynchroniczna](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#Modyfikacja replik) - kopie zmodyfikowanej tabeli są tylko okresowo aktualizowane (odświeżane) – metoda znacznie tańsza; ale chwilowo różne kopie mogą nie być ze sobą zsynchronizowane.

[zapytanie odległe](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#odleg) – zapytanie, w którym dane do realizacji znajdują się w jednym odległym węźle sieci.

[transakcja odległa](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#odleg) – transakcja, której dane do realizacji znajdują się w jednym odległym węźle sieci.

[zapytanie rozproszone](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#rozprosz) – zapytanie, w którym dane do realizacji znajdują się w różnych, odległych węzłach sieci.

[transakcja rozproszona](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#rozprosz) – transakcja, której dane do realizacji znajdują się w różnych odległych węzłach sieci.

[protokół dwufazowego zatwierdzania (2PC)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#dwufaz) - protokół zatwierdzania transakcji rozproszonej obowiązujący węzły w sieci biorące udział w realizacji transakcji rozproszonej. Składa się z dwóch faz komunikacji między węzłami: pierwsza - *głosowanie* czy zatwierdzić transakcję; druga - *kończenie* transakcji po podjęciu decyzji. Obie fazy są inicjowane przez jeden wyznaczony do tego węzeł nazywany *koordynatorem transakcji*.

[powiązanie bazodanowe (z bazą danych](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#Powi)) - zapisana, jako obiekt bazy danych Oracle,  ścieżka sieciowa do odległej bazy danych.

[integracja informacji](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w14.htm#Problem integracji informacji) - problem zbudowania jednolitego intefejsu do zbioru heterogenicznych baz danych. Dwa stosowane podejścia to albo kopiowanie danych do jednej bazy danych nazywanej *hurtownią danych* albo zbudowanie modułu *mediatora* między bazami danych. Jako język do komunikacji jest używany język XML.

**Wykład 15**

***Hurtownie danych a bazy danych***

**Streszczenie**

Tematem wykładu są hurtownie danych i ich związek z bazami danych.

Wykład ten jest podzielony na dwie części. W pierwszej jest omówiona specyfika hurtowni danych, w szczególności czym hurtownie danych różnią się od baz danych rozpatrywanych do tej pory.

W drugiej części są przedstawione narzędzia do tworzenia i używania hurtowni danych dostępne w serwerze Oracle.

**15.1 Wprowadzenie do hurtowni danych**

Przykłady zastosowań baz danych, którymi zajmowaliśmy się do tej pory, miały specjalne cechy. Przede wszystkim dotyczyły wspomagania bieżącego funkcjonowania firmy. Umożliwiały rejestrację wszystkich zmian, jakie zachodzą w odniesieniu do obiektów działalności firmy. Umożliwiały szybkie znajdowanie zapisanych na dysku informacji o obiektach, także w powiązaniu z innymi obiektami. Podstawowymi problemami, z jakimi spotykaliśmy się, były: utrzymanie spójności zmienianych danych (więzy spójności, transakcje) oraz szybkość znajdowania odpowiednich informacji (np. poprzez indeksy). Zastosowania tego rodzaju nazywają się *operacyjnymi* lub *OLTP* (ang. *On-Line Transaction Processing*), a używane bazy danych bazami *operacyjnymi*.

Oprócz tego istnieją zastosowania *analityczne*, nazywane też *OLAP* (ang. *On-Line Analytical Processing*), których celem jest wyciąganie z danych przechowywanych w bazie danych informacji biznesowych takich jak informacje o wzorcach i trendach biznesu, z myślą o wspomaganiu procesu podejmowania decyzji w firmie. Dziedziną, na której one operują, są dane historyczne z działalności firmy zwykle z ostatnich kilku lub kilkunastu lat. Używane w tym celu bazy danych nazywają się *hurtowniami danych* (używa się też nazw *magazyny danych* lub *bazy analityczne*). Oto ich podstawowe cechy:

* przechowują zintegrowane dane pochodzące z różnych, często  heterogenicznych baz operacyjnych używanych w firmie;
* przechowują dane obejmujące długie okresy czasu, często rozszerzone o informacje sumaryczne;
* mają rozmiary liczone w gigabajtach i terabajtach;
* oczekuje się szybkiej odpowiedzi na złożone zapytania – dlatego staje się konieczne wcześniejsze przetworzenie danych;
* rzadko ma miejsce modyfikowanie danych a tylko zachodzi regularne dopisywanie nowych danych napływających ze źródłowych baz operacyjnych.

Typową hurtownią danych jest baza danych o sprzedaży. Dane do takiej hurtowni spływają z wielu operacyjnych baz używanych w oddziałach firmy. W oparciu o wielomilionowy zbiór rekordów osoby, które podejmują decyzje w firmie, chcą uzyskiwać odpowiedzi na takie pytania jak:

* Które produkty przynoszą firmie najwięcej zysku?
* Którzy klienci przynoszą najwięcej zysku? Co mają ze sobą wspólnego?
* W których regionach kraju firma osiąga największe zyski, z jakiego rodzaju działalności te zyski pochodzą?
* Jakie usługi są najbardziej pożądane dla klientów firmy?
* Kiedy powinno się wprowadzić na rynek nowe produkty?
* Które produkty powinno się wycofać z rynku?
* Którzy dostawcy są najmniej pewni?

Typowym pytaniem zadawanym przy wykonywaniu analizy danych jest:

|  |
| --- |
| Jakie są dane o ... w rozbiciu na ... ? |

np. dane o sprzedaży tegorocznej i ubiegłorocznej w rozbiciu na produkty i miesiące roku; dane o zyskach w rozbiciu na regiony kraju i rodzaj działalności; dane o obrotach w rozbiciu na poszczególnych klientów i sektory rynku.

**Własności hurtowni danych**

Hurtownia danych powinna być:

* *zorientowana tematycznie* – wokół tematów takich, jak np. klienci, produkty, partnerzy handlowi, dostawcy, dane o sprzedaży czy pracownicy. Dane dotyczące jednego tematu mogą się znajdować w kilku aplikacjach operacyjnych: na przykład dane o klientach firmy w aplikacji obsługi zamówień i w aplikacji księgowej.
* *nieulotna* - po załadowaniu bazy analitycznej jej dane są tylko odczytywane. Spostrzeżenie to pociąga za sobą wiele konsekwencji: jeżeli dane mają być tylko odczytywane, nie jest tak ważne jak w przypadku baz operacyjnych zapewnienie im trzeciej postaci normalnej w celu uniknięcia anomalii przy uaktualnianiu; nie jest też konieczne zapewnienie mechanizmów transakcyjnych - jeżeli dane zostały do niej raz zapisane, to powinny pozostawać tam bez zmian.
* *wielowersyjna-czasow*o - dane pochodzą z okresu kilku, a czasem nawet, kilkunastu lat. Istotnym elementem wszystkich danych jest czas. Jeśli w danych przychodzących do hurtowni danych nie ma wymiaru czasowego, trzeba go dodać.

Z baz operacyjnych dane wpływają do hurtowni danych regularnie (powiedzmy raz na dzień). Najpierw następuje ich transformacja do jednolitego formatu, następnie ich przetworzenie przez obliczenie pewnych ich agregatów i w końcu załadowanie wszystkiego do hurtowni danych. Dane są pobierane ze źródłowych baz operacyjnych, przetwarzane i wpisywane zgodnie z pewną ich organizacją określoną przez *repozytorium metadanych* (*słownik danych*).

Rys. 15.1 Analityczne przetwarzanie danych za pomocą hurtowni danych

Z danych przechowywanych w hurtowni danych korzystają konkretne narzędzia analityczne OLAP przedstawiające dane w postaci informacji. Stosuje się zapytania podobne do operacji na arkuszach kalkulacyjnych i na wielowymiarowych kostkach danych. Specjalne znaczenie ma specjalistyczne oprogramowanie, które w oparciu o zgromadzone dane prowadzi eksploracyjne wyszukiwanie (ang. *data mining*) interesujących trendów i anomalii.

W hurtowni danych problem nie polega na szybkim odszukaniu pojedynczego rekordu (jak w aplikacji operacyjnej); obliczenia należy wykonywać dla dużej grupy rekordów, często dla wszystkich rekordów ze zbioru.

Projektując hurtownię danych stosujemy zupełnie odwrotne podejście niż przy projektowaniu baz operacyjnych. Zamiast pytania: "Po co przechowywać miesięczne salda kont bankowych, skoro można je wyliczyć mając zapis ciągu wykonanych na nim operacji?" powinniśmy postawić pytanie: "Czemu nie wyliczyć raz i nie przechowywać miesięcznych stanów kont, skoro 90% wykonywanych analiz wymaga właśnie tak przygotowanych danych?". Dobrze zaprojektowane wstępne przetworzenie danych może istotnie przyśpieszyć czas wykonywania 90% analiz!

W hurtowni danych stosuje się dwie metody pomagające rozwiązać powyższe problemy:

* *agregacje* - wstępne wyliczenie pewnych miar przydatnych w późniejszych analizach;
* *podziały na partycje* - podział tabel na części tak, by zmniejszyć rozmiar danych, które trzeba przeczytać w trakcie wykonywania analiz lub tak, aby umożliwić ich równoległe przetwarzanie.

**15.2 Schemat hurtowni danych**

**Fakty i wymiary**

Mając ustalony temat, wyróżniamy dla niego dwa rodzaje danych:

1. *wielkości analizowane* (*fakty*) - dane ilościowe opisujące pewne fakty jak np. sprzedaż, zyski, obroty;

2. *wielkości klasyfikujące* (*wymiary*) - dane klasyfikujące opisywane fakty według okoliczności ich zaistnienia, takich jak czas, miejsce, osoba itp.

Stąd wynika, że każdy fakt istnieje w wielowymiarowej przestrzeni, np. fakt pojedynczej sprzedaży istnieje w wielowymiarowej przestrzeni, w której poszczególne wymiary to: czas, struktura sprzedaży, struktura klientów, struktura produktów itp.

Każdemu wymiarowi jest przypisany zbiór *atrybutów opisowych* (*klasyfikujących*).

Atrybuty opisowe są układane w hierarchie, w naturalny sposób obrazujące drogę przejścia od danych szczegółowych do ogólnych.

Na przykład, dla wymiaru czasu:

|  |
| --- |
| *dzień -> miesiąc -> kwartał -> rok*  *dzień -> tydzień -> rok* |

dla wymiaru geograficznego:

|  |
| --- |
| *sklep -> miasto -> województwo*  *sklep -> oddział* |

dla wymiaru produktów:

|  |
| --- |
| *opakowanie -> produkt -> grupa produktów* |

Przy czym hierarchie nie muszą być jednoznaczne, np. jeden oddział firmy może obsługiwać dwa województwa oraz jednocześnie może istnieć województwo z dwoma oddziałami.

Dla użytkownika końcowego dane do analizy to zestaw osobnych tabel opisujących fakty w rozbiciu na wymiary, plus ewentualnie informacja o tym, jak poszczególne atrybuty klasyfikujące wymiary łączą się w hierarchie.

**Schemat gwiazda**

Naturalną strukturą danych do przeprowadzania analiz danych jest *schemat gwiazda*. W schemacie tym mamy jedną tabelę faktów w środku układu i zbiór tabel po bokach, każda z nich opisująca jeden wymiar. Identyfikatory wartości wymiarów stanowią klucze obce w tabeli faktów.

Oto przykład schematu gwiazda w odniesieniu do tabeli faktów sprzedaży w podziale względem wymiaru struktury sprzedaży, czasu, wymiaru geograficznego i wymiaru produktów.

Rys. 15.2 Schemat gwiazda

Tabele wymiarów nie są na ogół w trzeciej postaci normalnej np. w rozważanym przykładzie istnieją zależności takie jak: *miesiąc* determinuje *kwartał*, *miasto* determinuje *województwo*. Czasami tabele wymiarów dodatkowo normalizuje się uzyskując schemat nazywany *płatkiem śniegu*.

Same tabele z wymiarami mogą posłużyć użytkownikom do znajdowania odpowiedzi na pytania dotyczące klasyfikacji danych, np.

* Na jakie regiony geograficzne zostały podzielone dane?
* Jakie województwa zawiera dany makroregion?
* Ile pozycji będzie zawierał raport w rozbiciu na oddziały firmy?

Na pytania takie można odpowiedzieć nie sięgając w ogóle do tabeli faktów.

Osobnym problemem jest, gdzie zapisywać wyniki operacji agregujących (podsumowujących). Do tego celu można albo użyć tego samego zestawu tabel - dodając w każdej tabeli wymiaru atrybut poziomu agregacji danych, względem którego są agregowane wartości analizowane, albo zapisywać wyniki agregacji do osobnych tabel – najlepiej kontrolowanych przez sam system.

**Schemat kostki wielowymiarowej**

Jak stwiedziliśmy uprzednio, każdy fakt, np. dotyczący sprzedaży, można zinterpretować jako funkcję z wielowymiarowej przestrzeni wymiarów w zbiór wartości analizowanych np. dla schematu gwiazdy powyżej z pominięciem czwartego wymiaru - sprzedawcy:

Rys. 15.3 Kostka 3-wymiarowa

Operacje na danych uzyskują interpretację geometryczną.

Każdą taką kostkę czyli funkcję można bezpośrednio reprezentować za pomocą wielowymiarowej tablicy. Przy dużej liczbie pozycji trzeba przechowywać zawartość tej tablicy na dysku i tylko częściami sprowadzać do pamięci wewnętrznej w celu wykonania obliczeń.

Podobnie każdą tabelę wymiaru można zapisać w tablicy. Powstające w ten sposób tablice wymiarów można zapisać w pamięci wewnętrznej, ponieważ nie są zbyt duże. W ten sposób otrzymujemy hurtownię danych, która nie jest oparta na relacyjnej bazie danych, tylko na zbiorze tablic. Rozwiązanie takie nosi nazwę MOLAP – ang. *multidimensional on-line analytical processing*, w odróżnieniu od poprzednio rozważanego rozwiązania, nazywanego ROLAP – ang. *relational on-line analytical processing*.

**Operacje w hurtowni danych**

Niezależnie od sposobu implementacji (ROLAP czy MOLAP) są rozważane następujące operacje na prezentowanych użytkownikowi danych odwołujące się do modelu wielowymiarowego:

* *pivoting* – wskazanie do analizy: miary np. *sprzedaży* oraz dwóch wymiarów np. *miasta* i *roku* oraz przypisaniu wartościom tych wymiarów pewnej agregacji wartości wybranej miary np. sumarycznej sprzedaży w danym mieście w danym roku. Wygodną reprezentacją wyniku operacji pivoting jest tablica; w przykładzie, wiersze tablicy są etykietowane latami, a kolumny miastami.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **WA** | **KA** | **Razem** |
| **1995** | 63 | 81 | 144 |
| **1996** | 38 | 107 | 145 |
| **1997** | 75 | 35 | 110 |
| **Razem** | 176 | 223 | **339** |

Tab. 15.1 Wynik operacji pivoting

* *drill-down* – *rozwijanie* – rozwinięcie danego wymiaru do kolejnego elementu tego wymiaru – np. mając wyniki sprzedaży w rozbiciu na lata, chcemy poznać wyniki sprzedaży w konkretnym roku w rozbiciu na miesiące. Operacją odwrotną jest *roll-up* - *zwijanie* od rozbicia na bardziej szczegółowe elementy wymiaru do bardziej ogólnych.
* *slice-and-dice* – *wycinanie* – operacja projekcji na wybrany podzbiór wymiarów dla wybranych wartości innych wymiarów np. dane o sprzedaży poszczególnych produktów (projekcja na wymiar produktów) w ubiegłym roku (selekcja na wymiarze czasu).

**15.3** **Implementacja hurtowni danych w Oracle**

Przedstawimy teraz konstrukcje w serwerze Oracle służące do definiowania i używania hurtowni danych:

**1.** **Histogramy**

Histogramy określają dystrybucję wartości w kolumnie tabeli. Zakres wartości w kolumnie dzieli się na przedziały o tych samych rozmiarach i oblicza się ile wartości wpada do poszczególnych przedziałów. Histogramy oblicza się za pomocą instrukcji ANYLYZE TABLE:

ANALYZE TABLE Sprzedaz  
COMPUTE STATISTICS FOR COLUMNS Wartosc\_sprzedazy  
SIZE 10; -- liczba przedziałów, na które dzieli się wartości sprzedaży

Histogramy są używane przez optymalizator zapytań i można je odczytać z perspektyw słownika danych USER\_HISTOGRAMS i USER\_TAB\_COLUMNS.

**2.** **Wersja równoległa serwera Oracle**

Tworząc tabelę możemy określić jej stopień zrównoleglenia:

CREATE TABLE XXX(....)  
PARALLEL (DEGREE 8); -- stopień zrównoleglania zapytań dla tej tabeli

Proces przyjmujący w serwerze Oracle zgłoszenie użytkownika będzie się starał przyporządkować do obsługi zapytania z tą tabelą liczbę procesów serwera równą temu stopniowi. Urównolegleniu może ulec ładowanie danych do tabeli z różnych plików, również tworzenie tabel i indeksów.

**3.** **Obiekty z partycjami (partycjonowane)**

Partycje umożliwiają rozłożenie obiektów bazy danych do różnych dysków w celu zrównoleglenia operacji wejścia/wyjścia. Dotyczy to zarówno tabel, indeksów jak i perspektyw zmaterializowanych, o których będzie dalej mowa.

CREATE Klienci(  
Id NUMBER(5) PRIMARY KEY,  
Kraj CHAR(2), ......)  
PARTITION BY RANGE (kraj)  
(PARTITION p1 VALUES LESS THAN ('C')  
TABLESPACE Data01;  
PARTITION p2 VALUES LESS THAN ('I')  
TABLESPACE Data02;  
...  
PARTITION p19 VALUES LESS THAN MAXVALUE  
TABLESPACE Data19);

Następnie możemy utworzyć indeks podzielony na partycje

CREATE INDEX Indx\_Klienci\_Kraj ON Klienci(Kraj) LOCAL;

Słowo kluczowe LOCAL oznacza, że podział na partycje jest zgodny z podziałem wierszy tabeli. Gdybyśmy chcieli podzielić pozycje indeksu w inny sposób, musielibyśmy użyć klauzuli GLOBAL PARTITION BY RANGE specyfikując inne zasady podziału.

**4.** **Indeksy bitmapowe**

Indeksy bitmapowe są zakładane na kolumnach, których zbiór wartości jest niewielki i które występują często w warunku WHERE zapytań z równością np.

WHERE (Rok\_produkcji=1992 OR Rok\_produkcji=1993) AND (Producent = 'Fiat' OR Producent='Ford') AND (Kraj\_klienta='Polska' OR Kraj\_klienta='USA')

Są na przykład odpowiednie dla tabeli zawierającej milion wierszy oraz na kolumnie mającej tylko 1000 różnych wartości.

CREATE BITMAP INDEX Ind\_Sprzedaż\_Producent ON Sprzedaż(Producent);

Rozważmy prostszą przykładową tabelę:

|  |  |  |
| --- | --- | --- |
| **Nazwisko** | **Płeć** | **Stanowisko** |
| Kowalski | M | Dyrektor |
| Jankowski | M | Sprzedawca |
| Malinowski | M | Sprzedawca |
| Gazda | K | Kasjer |
| Wiśniewski | M | Kasjer |
| Bojanowska | K | Sprzedawca |

Tab. 15.2 Przykładowa tabela

Oto indeksy bitmapowe na kolumnach *Płeć* i *Stanowisko*:

|  |
| --- |
| Indeks na *Płeć*    Indeks na *Stanowisko*  Nr wiersza    M K               D S K ------------ ----------------- --------------------- 1             1 0               1 0 0 2             1 0               0 1 0 3             1 0               0 1 0 4             0 1               0 0 1 5             1 0               0 0 1 6             0 1               0 1 0 |

Tab. 15.3 Dwa indeksy bitmapowe

Warunek:

WHERE: (Płeć = 'K' AND Stanowisko = 'K') OR (Płeć = 'M' AND Stanowisko = 'D')

powoduje wykonanie trzech operacji na wektorach bitowych będących odpowiednimi kolumnami w indeksach przedstawionych powyżej:

([000101] AND [000110]) OR ([111010] AND [100000]) = [100100]

dając jako wynik zapytania pierwszą i czwartą osobę – wskazane przez miejsca jedynek w wektorze.

Zauważmy, że aby móc użyć indeksu bitmapowego potrzebujemy dwóch dodatkowych struktur danych:

* jedna, która w oparciu o stałą wartość atrybutu pozwala szybko wyznaczyć przypisany jej wektor bitów (np. B+ drzewo);
* druga, która w oparciu o numer wiersza pozwala szybko wyznaczyć ten wiersz (np. tablica lub B+ drzewo).

W hurtowniach danych indeksy bitmapowe są zakładane w tabeli faktów na kolumnach kluczy obcych odwołujących się do tabel wymiarów.

**5.** **Transformacja STAR**

W przypadku wykrycia prawidłowości:

jedna duża tabela faktów i wiele drobnych tabel wymiarów

optymalizator stosuje specjalne postępowanie - wykorzystujące w szczególności indeksy bitmapowe.

[Jest dodatkowe wymaganie w systemie Oracle.](javascript:popUp('fig15/ok3.htm',500,230))

Przykład

Transformacja STAR przekształca instrukcję:

SELECT \* FROM Sprzedaż, Miejsce, Towar, Czas  
   WHERE Sprzedaż.Id\_miejsca = Miejsce.Id  
      AND Sprzedaż.Id\_towaru = Towar.Id  
      AND Sprzedaż.Id\_czasu = Czas.Id  
      AND Miejsce.Miasto IN ('WAW','KRA','RAD','LUB')  
      AND Towar.Kategoria = 'OPROGRAMOWANIE'  
      AND Czas.Rok > 1996;

na:

SELECT \* FROM Sprzedaż  
WHERE Sprzedaż.Id\_miejsca IN  
               (SELECT Miejsce.Id FROM Miejsce  
                        WHERE Miejsce.Miasto IN ('WAW','KRA','RAD','LUB'))  
   AND Sprzedaż.Id\_towaru IN  
               (SELECT Towar.Id FROM Towar  
                         WHERE Towar.Kategoria = 'OPROGRAMOWANIE')  
   AND Sprzedaż.Id\_czasu IN  
               (SELECT Czas.Id FROM Czas  
                         WHERE Czas.Rok > 1996);

W rezultacie:

* najpierw oblicza się podzapytania,
* następnie stosuje się indeksy bitmapowe i operacje na wektorach bitów, aby znaleźć fakty spełniające jednocześnie wszystkie trzy warunki IN.

Uwaga: W rzeczywistości obie instrukcje SELECT nie są równoważne i wynik drugiej trzeba jeszcze złączyć z tabelami wymiarów – ale operacje te są już wykonywane na znacznie mniejszych tabelach, niż oryginalne złączenie. Poza tym, na ogół w wyjściowej instrukcji SELECT chodzi o wyliczenie pewnych agregacji, które jeszcze bardziej ograniczają rozmiar wyniku drugiej instrukcji SELECT.

**6. Perspektywy zmaterializowane (migawki)**

Po zaprojektowaniu tabel faktów i wymiarów projektuje się następnie perspektywy zmaterializowane określające wymagane złączenia i agregacje (podsumowania) danych z bazowych tabel. Na perspektywie zmaterializowanej można zakładać indeksy w szczególności bitmapowe, więc przy wykonywaniu zapytań, perspektywa zmaterializowana ma te same własności co tabela.

Ponadto można zażyczyć sobie, aby zapytania pisane w terminach tabel faktów i wymiarów były automatycznie przekształcane przez optymalizator do zapytań korzystających z perspektyw zmaterializowanych.

Aby optymalizacja była możliwa, w definicji perspektywy zmaterializowanej należy użyć klauzuli ENABLE QUERY REWRITE.

Na przykład:

CREATE MATERIALIZED VIEW Sprzedaz\_mv  
ENABLE QUERY REWRITE  
AS  
   SELECT s.Nazwa\_sklepu, SUM(f.Wielkosc) AS Suma  
   FROM Sklep s, Fakt f  
   WHERE f.IdSklepu = s.IdSklepu  
   GROUP BY s.Nazwa\_sklepu;

[Jest dodatkowe wymaganie w systemie Oracle.](javascript:popUp('fig15/ok4.htm',500,230))

Użycie opcji QUERY REWRITE umożliwia zasłonięcie szczegółów technicznych związanych z agregacjami danych w hurtowni danych przed użytkownikiem, który pisze i używa aplikacji w terminach tabel faktów i wymiarów. W szczególności, gdy zmienią się definicje perspektyw zmaterializowanych, aplikacje będą dalej działać poprawnie. W ten sposób użycie perspektyw zmaterializowanych jest podobne do użycia indeksów – w obu przypadkach stosuje je optymalizator zapytań. Oba rodzaje obiektów muszą być aktualizowane po wykonaniu operacji INSERT, DELETE i UPDATE.

Obok wcześniejszego wyliczania agregacji perspektywy zmaterializowane używane są też do wcześniejszego wyliczania złączeń tabeli faktów z pewnymi wybranymi tabelami wymiarów. Przy takim zastosowaniu perspektywy zmaterializowane pełnią rolę tzw. *indeksów złaczeniowych* (przy bitmapowej implementacji - *bitmapowych indeksów złączeniowych*).

**7.** **Grupowanie ROLLUP i CUBE**

Używając samej klauzuli GROUP BY nie uzyskamy częściowych podsumowań po poszczególnych poziomach wymiarów. Do wprowadzenia wyników częściowych podsumowań służą operatory ROLLUP i CUBE. Zapytanie:

SELECT Czas.rok, Miejsce.miasto, Towar.kategoria, SUM(Sprzedaż.wartość)AS "W sumie"  
FROM Sprzedaż, Miejsce, Towar, Czas  
GROUP BY ROLLUP(Czas.Rok, Miejsce.Miasto, Towar.Kategoria);

da wynik w rodzaju (oczywiście używane narzędzie OLAP przedstawi wynik tego zapytania w bardziej czytelny sposób np. za pomocą wykresu):

ROK MIASTO  KATEGORIA W sumie

------ ---------- ----------- ----------

1996 Warszawa Komputery     100000

1996 Warszawa Telewizory     50000

....

1996 Warszawa       \*          1000000

1996 Łódź       Komputery     50000

1996 Łódź       Telewizory     25000

.....

1996 Łódź            \*          250000

.....

1996   \*             \*         2000000

1997 Warszawa Komputery   1500000

....

 \*       \*            \* 10000000

Tab. 15.4 Wynik operacji ROLLUP

Operator ROLLUP daje wszystkie częściowe podsumowania po poszczególnych wymiarach – w danej kolejności. W wyniku istotna jest kolejność grupowanych kolumn. W szczególności zapytanie:

SELECT e.Deptno, e.EmpNo, SUM(e.Sal) AS Zarobki  
FROM Emp e  
GROUP BY ROLLUP(e.Deptno, e.EmpNo);

daje wynik:

DEPTNO EMPNO ZAROBKI

---------- ---------- ----------

10 7782 2450

10 7839 5000

10 7934 1300

10 8750

20 7369 800

20 7566 2975

20 7788 3000

20 7876 1100

20 7902 3000

20 10875

30 7900 950

30 7499 1600

30 7521 1250

30 7654 1250

30 7698 2850

30 7844 1500

30 9400

29025

Tab. 15.5 Wynik operacji ROLLUP

Operator ROLLUP daje więc tę możliwość, której brakowało dotychczas w SQL – zwracania w wyniku zarówno szczegółowych wierszy jak ich podsumowań.

Operator CUBE daje nawet więcej niż ROLLUP, bo również wszystkie częściowe podsumowania po poszczególnych wymiarach – we wszystkich możliwych kolejnościach.

Na przykład, gdybyśmy w powyższej instrukcji SELECT zamienili ROLLUP na CUBE, dostalibyśmy oprócz powyższych wierszy również takie wiersze jak:

\*   Warszawa    Komputery    1000000

...

\*   Warszawa         \*       9000000

...

\*       \*      Komputery    8000000

....

1996   \*       Komputery   500000

Tab. 15.5 Wynik operacji CUBE

Jest też możliwość łączenia zwykłego grupowania z grupowaniem typu ROLLUP i CUBE. Na przykład:

GROUP BY Miejsce.Miasto, ROLLUP(Czas.Rok, Towar.Kategoria)

**15.4 Podsumowanie**

W pierwszej części wykładu została omówiona specyfika hurtowni danych, w szczególności czym hurtownie danych różnią się od baz danych rozpatrywanych do tej pory.

W drugiej części wykładu zostały  przedstawione narzędzia do tworzenia i używania hurtowni danych dostępne w serwerze Oracle.

**15.5 Słownik pojęć**

[miara (fakt)](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Fakty i wymiary) - dane ilościowe opisujące pewne fakty biznesowe jak sprzedaż, zyski, obroty.

[wymiar](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Fakty i wymiary) - dane klasyfikujące opisywane fakty wg okoliczności ich zaistnienia jak czas, miejsce, osoba itp.

[schemat gwiazda](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Schemat gwiazda) - schemat  złożony z tabeli faktów w środku układu i zbioru tabel, po bokach, każda z nich opisująca jeden wymiar. Identyfikatory wartości wymiarów stanowią klucze obce w tabeli faktów.

[schemat kostki wielowymiarowej](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Kostka) - nierelacyjna reprezentacja hurtowni danych w postaci wielowymiarowej tablicy.

[indeks bitmapowy](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Indeksy bitmapowe) - specjalny indeks zakładany na kolumnach, których zbiór wartości jest niewielki i które często występują w warunku WHERE razem z operatorem równości lub operatorem IN.

[transformacja STAR](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Opt) - przekształcenie zapytania, w którym występuje złączenie tabeli faktów z tabelami wymiarów, na zapytanie dotyczące tylko tabeli faktów tak aby można było zastosować indeksy bitmapowe na kolumnach kluczy obcych (odpowiadającym wymiarom).

[perspektywa zmaterializowana](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Pers) - perspektywa, której wynik jest obliczany i zapisywany do bazy danych. W hurtowni danych perspektywa zmaterializowana określa wymagane agregacje (podsumowania) danych z bazowych tabel. Na perspektywie zmaterializowanej można zakładać indeksy, więc ma ona podobne  własności co tabela.

[ROLLUP](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Grup) - operator rozszerzający możliwości GROUP BY. Operator ROLLUP daje wszystkie częściowe podsumowania po poszczególnych wymiarach – w danej kolejności. W wyniku istotna jest kolejność grupowanych kolumn.

[CUBE](https://edu.pjwstk.edu.pl/wyklady/sbd/scb/w15.htm#Grup) - operator rozszerzający możliwości zarówno GROUP BY jak i ROLLUP. Operator CUBE daje wszystkie częściowe podsumowania po poszczególnych wymiarach – we wszystkich kolejnościach grupowanych kolumn.