1. Vnitřní programovací konstrukce (Embedded SQL) – procedurální prostředky v rámci jazyka SQL, jazyk PL/SQL

Embedded SQL

Je metoda vkládání řádkových **SQL příkazů do kódu programovacího jazyka, protože tento programovací jazyk neumí parsovat SQL. Vkládané SQL je parsováno v SQL preprocesoru.**

Přímý přístup programovacího jazyka do databázových struktur – překladač jazyka je obohacený o konstrukce pro SQL

Embedded SQL ("vložené SQL") je jednou z metod, jak přistupovat k databázi z prostředí nějakého procedurálního programovacího jazyka. Jak název napovídá, jde o **vkládání SQL příkazů mezi příkazy programovacího jazyka**.

- SQL dotazy jsou psány přímo ve zdrojovém kódu. Syntaxe SQL je přizpůsobena syntaxi daného programovacího jazyka.
- Před kompilací programu se musí zdrojový kód zpracovat preprocesorem Embedded SQL, kdy SQL dotazy jsou nahrazeny odpovídajícím kódem programovacího jazyka.
- Nejčastěji v kombinaci s jazyky C/C++.

Podpora v databázích

Podporují klasické velké databázové systémy

- IBM DB2 (C/C++, Java, Cobol)
- Oracle (Cobol, *Pro*C* - embedded SQL Oraclu pro C/C++)
- PostgreSQL (C/C++)

Nepodporují

- MySQL
- Microsoft SQL Server (starší verze podporovali C)

Příklad syntaxe

Oracle Embedded SQL v jazyce C:

```
    {
        int a;
        /* ... */
        EXEC SQL SELECT plat INTO :a
                 FROM Zamestnanci
                 WHERE id=876543210;
        /* ... */
        printf("Plat je %d\n", a);
        /* ... */
    }
```

Embedded SQL

The first technique for sending SQL statements to the DBMS is embedded SQL. Because SQL does not use variables and control-of-flow statements, it is often used as a database sublanguage that can be added to a program written in a conventional programming language, such as C or COBOL. This is a central idea of embedded SQL: placing SQL statements in a program written in a host programming language. Briefly, the following techniques are used to embed SQL statements in a host language:

- Embedded SQL statements are processed by a special SQL precompiler. All SQL statements begin with an introducer and end with a terminator, both of which flag the SQL statement for the precompiler. The introducer and terminator vary with the host language. For example, the introducer is "EXEC SQL" in C and "&SQL(" in MUMPS, and the terminator is a semicolon (;) in C and a right parenthesis in MUMPS.
- Variables from the application program, called host variables, can be used in embedded SQL statements wherever constants are allowed. These can be used on input to tailor an SQL statement to a particular situation and on output to receive the results of a query.
- Queries that return a single row of data are handled with a singleton SELECT statement; this statement specifies both the query and the host variables in which to return data.
- Queries that return multiple rows of data are handled with cursors. A cursor keeps track of the current row within a result set. The DECLARE CURSOR statement defines the query, the OPEN statement begins the query processing, the FETCH statement retrieves successive rows of data, and the CLOSE statement ends query processing.
- While a cursor is open, positioned update and positioned delete statements can be used to update or delete the row currently selected by the cursor.

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms713573(v=vs.85).aspx>

The following code is a simple embedded SQL program, written in C. The program illustrates many, but not all, of the embedded SQL techniques. The program prompts the user for an order number, retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen.

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms714570(v=vs.85).aspx>

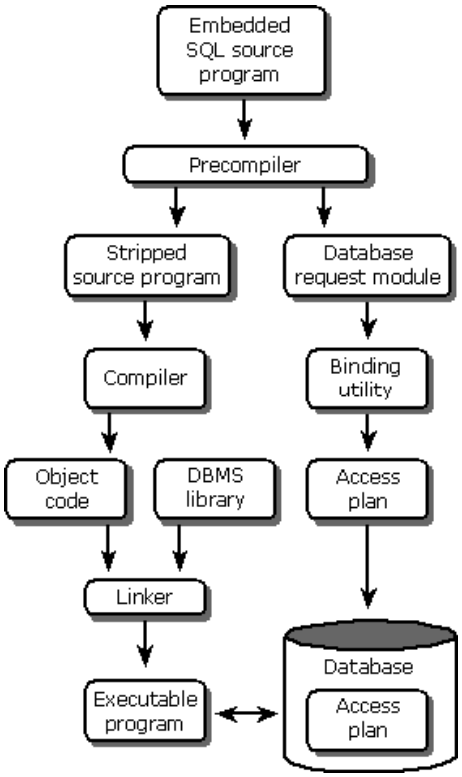
```
int main() {
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int OrderID;          /* Employee ID (from user)         */
      int CustID;             /* Retrieved customer ID         */
      char SalesPerson[10]    /* Retrieved salesperson name      */
      char Status[6]          /* Retrieved order status        */
   EXEC SQL END DECLARE SECTION;
/* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;
/* Prompt the user for order number */
   printf ("Enter order number: ");
   scanf_s("%d", &OrderID);
```

```
/* Execute the SQL query */
   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;
/* Display the results */
   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson: %s\n", SalesPerson);
   printf ("Status: %s\n", Status);
   exit();
query_error:
   printf ("SQL error: %ld\n", sqlca->sqlcode);
   exit();
bad_number:
   printf ("Invalid order number.\n");
   exit();
}
```

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms714570(v=vs.85).aspx>

Because an embedded SQL program contains a mix of SQL and host language statements, it cannot be submitted directly to a compiler for the host language. Instead, it is compiled through a multistep process. Although this process differs from product to product, the steps are roughly the same for all products.

This illustration shows the steps necessary to compile an embedded SQL program.



Five steps are involved in compiling an embedded SQL program:

1. The embedded SQL program is submitted to the SQL precompiler, a programming tool. The precompiler scans the program, finds the embedded SQL statements, and processes them. A different precompiler is required for each programming language supported by the DBMS. DBMS products typically offer precompilers for one or more languages, including C, Pascal, COBOL, Fortran, Ada, PL/I, and various assembly languages.
2. The precompiler produces two output files. The first file is the source file, stripped of its embedded SQL statements. In their place, the precompiler substitutes calls to proprietary DBMS routines that provide the run-time link between the program and the DBMS. Typically, the names and the calling sequences of these routines are known only to the precompiler and the DBMS; they are not a public interface to the DBMS. The second file is a copy of all the embedded SQL statements used in the program. This file is sometimes called a database request module, or DBRM.
3. The source file output from the precompiler is submitted to the standard compiler for the host programming language (such as a C or COBOL compiler). The compiler processes the source code and produces object code as its output. Note that this step has nothing to do with the DBMS or with SQL.
4. The linker accepts the object modules generated by the compiler, links them with various library routines, and produces an executable program. The library routines linked into the executable program include the proprietary DBMS routines described in step 2.
5. The database request module generated by the precompiler is submitted to a special binding utility. This utility examines the SQL statements, parses, validates, and optimizes them, and then produces an access plan for each statement. The result is a combined access plan for the entire program, representing an executable version of the embedded SQL statements. The binding utility stores the plan in the database, usually assigning it the name of the application program that will use it. Whether this step takes place at compile time or run time depends on the DBMS.

Notice that the steps used to compile an embedded SQL program correlate very closely with the steps described earlier in Processing an SQL Statement. In particular, notice that the precompiler separates the SQL statements from the host language code, and the binding utility parses and validates the SQL statements and creates the access plans. In DBMSs where step 5 takes place at compile time, the first four steps of processing an SQL statement take place at compile time, while the last step (execution) takes place at run time. This has the effect of making query execution in such DBMSs very fast.

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms713968(v=vs.85).aspx>

Procedurální prostředky SQL (procedurální rozšíření SQL)

- SQL bylo původně navrženo pro získávání dat z relačních databází. SQL je deklarativní jazyk a ne imperativní, jako například C nebo Java.
- Dodavatelům DB systémů možnosti SQL nedostačovaly a tak si začali implementovat vlastní procedurální rozšíření SQL.
- příklad: kursory, bloky, proměnné

Motivace

- Šetření komunikačního kanálu
- Menší množství odesílaných povelů
- v jednom povelu je větší množství příkazů
- Podstatně menší objem přenesených dat - Data se zpracují na serveru bez přenosu na klienta
- Odlehčení klienta - Možnost ukládat a vykonávat kód na serveru

Na co si dát při návrhu pozor

- Hlídat na úrovni databáze všechny manipulace s daty, které ohlídat jdou
  - Cokoli jde zadat uživatelem špatně, bude zadáno špatně
- Integritní omezení, triggery
  - Později je čištění nekonzistentních dat namáhavé a opravy často nemožné
  - Lépe ohlídat vše

Procedurální rozšíření v SQL:1999

SQL:1999 standardizuje procedurální rozšíření. Rozšíření se jmenuje *SQL/PSM - SQL/Persisted Stored Modules*. **Nikdo to moc nedodržuje a všichni si dělají vlastní standardy/implementace**

- funkce a procedury - lze zapsat v SQL i v hostitelském programovacím jazyce
- řídící konstrukce - cykly, větvení, přiřazení

Podpora v databázích

SQL:1999 (SQL/PSM)

- IBM DB2
- MySQL
- PostgreSQL

Vlastní (proprietární) rozšíření

- Oracle (PL/SQL)
- PostgreSQL (PL/pgSQL)
- Microsoft (T-SQL)
- Sybase (T-SQL)

Jazyk PL/SQL

SQL - *Structured Query Language* (Strukturovaný dotazovací jazyk) je standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích

- Zpracování dat na straně serveru – klientovi výsledek

Rozdělení standardního SQL

SQL příkazy se dají rozdělit do několika kategorií podle toho, co provádíte

Data definition language (DDL) neboli příkazy určené pro práci se strukturou databázových objektů. Nejčastěji tabulek.
CREATE - vytvoření
ALTER - změně
DROP - odstranění
RENAME - přejmenování
TRUNCATE - smazání, aniž by se data ukládala do koše
COMMENT - přidání komentáře

Data manipulation language (DML) neboli příkazy určené pro manipulaci s daty
SELECT - vybrání dat z databáze
INSERT - vložení
UPDATE - úprava nebo také editace či změna
DELETE - smazaní
MERGE - sloučení

Data control language (DCL) neboli příkazy sloužící k přídání či odebrání opravnění k databázi a objektů v ní.
GRANT - přiřazení
REVOKE = odebrání

Řízení transakcí. Jednotlivé příkazy DML můžete slučovat do transakcí, ale nemusíte.
COMMIT - slouží k potvrzení veškerých změn
ROLLBACK - provede rollback veškerých změn
SAVEPOINT - vytvoří časovou značku ke které se můžete vracet.

PL/SQL (Processing Language/Structured Query Langugae)

PL/SQL přidává k jazyku SQL konstrukce procedurálního programování.

- Jazyk SQL je jazykem deklarativním, který neobsahuje procedurální rozšíření jako jsou cykly, podmínky, procedury, funkce atd
- PL/SQL je rozšířením jazyka SQL o proceduralitu od společnosti ORACLE
- Jazyk PL/SQL umožňuje deklarovat konstany, proměnné a kurzory, podporuje transakční zpracovaní řeší chybové stavy pomocí výjimek. PL/SQL podporuje modularitu.

PL/SQL (Procedural Language/Structured Query Language) je procedurální nadstavba jazyka SQL od firmy Oracle založená na programovacím jazyku Ada.

PL/SQL je rozšíření jazyka SQL o procedurální rysy. Je specifické pro produkty firmy Oracle, procedurální rozšíření SQL produktů jiných firem se zpravidla navzájem liší. Výjimkou je ŠRBD DB2 společnosti IBM, který podporuje jak vlastní procedurální jazyk SQL PL, tak je plně kompatibilní s jazykem PL/SQL včetně datových typů. Základním stavebním kamenem PL/SQL je tzv. **PL/SQL blok, který může být buď tělem triggeru, procedry a funkce, nebo samostatný.** Struktura PL/SQL bloku je viz "základní konstrukce"

Základním stavebním kamenem v PL/SQL je **blok**. Program v PL/SQL se skládá z bloků, které mohou být vnořeny jeden do druhého. Obyčejně každý blok spouští jednu logickou akci v programu. Blok má následující strukturu:

Základní konstrukce BLOKU

DECLARE
Deklarace proměnných konstant a kurzorů
BEGIN
výkonná část – příkazy, jediná tahle část je povinná
EXCEPTION
ošetření výjimek
END;

Komentáře:

-- komentář do konce řádky
/* komentář od do */

Pouze výkonná sekce je povinná, ostatní jsou doporučené. Jediné příkazy jazyka SQL, které jsou ve výkonné sekci **povolené, jsou SELECT, INSERT, UPDATE, DELETE** a několik dalších pro manipulaci s daty a pro kontrolu transakcí. **Definiční příkazy jazyka SQL jako CREATE, DROP nebo ALTER nejsou povoleny**. Avšak použití těchto příkazů lze pomocí direktivy EXECUTE IMMEDIATE "PŘÍKAZ". PL/SQL není citlivé na velikost písmen a mohou být použity komentáře ve stylu jazyka C.

Kurzor

- Kurzor je pracovní oblast obsahující data (výsledná množina, tzv. result set), které lze dále využívat prostřednictvím operací nad kurzory
- Existují **implicitní** a **explicitní** kurzory
  - Implicitní jsou jednořádkové SQL (INTO)
  - Explicitní můžeme deklarovat v části DECLARE pomocí klíčového slova CURSOR
- Práce s kurzory se podobá souborům

Procedury + funkce

Bloky příkazů jazyka PL/SQL lze pojmenovat a uložit ve spustitelné formě do databáze = procedury / funkce
• Jsou uloženy ve zkompilovaném tvaru v databázi.
• Mohou volat další procedury či funkce, či samy sebe.
• Lze je volat ze všech prostředí klienta.
Funkce, na rozdíl od procedury, vrací jedinou hodnotu (procedura může vracet hodnot více, resp. žádnou).

Triggery

- uživatelsky definovaný blok PL/SQL sdružený s určitou tabulkou. Je implicitně spuštěn (proveden), jestliže je nad tabulkou prováděn aktualizační příkaz

Trigger má 4 části:

- typ Triggeru = BEFORE/AFTER
- spouštěcí událost = INSERT/UPDATE/DELETE
- omezení triggeru = nepovinná klauzule WHEN
- akce triggeru = PL/SQL BLOK

Trigger 2 druhy:

- příkazový (statement) se spustí jedenkrát pro příkaz bez ohledu na počet aktualizovaných řádků
- řádkový trigger se spustí pro každý aktualizovaný řádek tabulky

Anonymous Blocks (jenom BEGIN až END bez názvu, takže zapomenout EXEC)

Packages

- Programový balík je sdružením řady funkcí a procedur s vlastním jmenným prostorem a vlastním persistentním prostorem pro proměnné v rámci jedné session
- Umožňuje uchovávat hodnoty v rámci session pro řadu procedur a funkcí
- Ne náhodná analogie s objekty

Pokročílé dative struktury PL/SQL – Kolekce a Záznamy

Kolekce = homogenní skupiny prvků (odpovídá polím), prvky identifikovány pořadím

1. Asociativní pole – identifikace prvků pomocí indexu
2. Nested table (hnízděná tabulka) – odpovídá virtuální databázové tabulce, práce s ní je sekvenční
3. Dynamické pole (variable-size array) – pomalejší přístup pomocí SQL než k nested tables

Záznamy= heterogenní data, jsou členěny na jednotilvá pole

Nested Tables

- Představují pole - množina (set, bag) v některých programovacích jazycích
- Lze ukládat tyto tabulky v databázových tabulkách (oboustranná kompatibilita)
- Deklarace:
  DECLARE TYPE ntable IS TABLE
  OF element_type;
- Typ prvku může být lib. PL/SQL typ mimo odkazu (REF) a kurzoru (CURSOR)

Varrays

- variable-size array (dynamické pole) – tzv. varrays odpovídají klasickým dynamickým polím, uchovávají definovaný počet hodnot,
- pomalejší přístup SQL nástroji než k nestedtables

Loops (FOR/WHILE)

```
LOOP
pocet:= pocet +1
IF pocet =100 THEN EXIT;
END IF;
END LOOP;

Nebo

LOOP
EXIT WHEN podmínka;
END LOOP;
```

IF

```
IF podmínka THEN příkazy_1;
ELSIF podmínka THEN příkazy_2;
.
ELSE příkazy_n;
END IF;
```

CASE

```
CASE proměnná
WHEN výraz_1 THEN příkazy_1;
WHEN výraz_2 THEN příkazy_2;
WHEN výraz_n THEN příkazy_n;
ELSE příkazy_n+1
END CASE
```

Deklarace typu v PL/SQL

- Explicitně jménem:
  X NUMBER(7,2);
- Kopírováním typu jiné proměnné
  Y X%**Type;**
- Kopírování typu sloupce a nebo řádku

Proměnné v PL/SQL

Proměnné je možné užívat jak v PL/SQL tak v SQL kódu (příkazy SQL mohou být volně užívány v PL/SQL bloku). Užití např: parametry v SQL, výpočty výrazů, podmínky

Možnosti přirazení hodnot do proměnné: příkaz přiřazení (:=), výběrem hodnoty z dotazu který vrací jeden řádek, výstupní hodnota procedury

Konstanty – klíčové slovo **CONSTANT,** inicializovány při deklaraci (pi CONSTANT REAL DEFAULT 3.1415;)

2. Kurzory – definice, klasifikace, použití kurzorů

Kurzor = pracovní oblast obsahující data (výsledná množina, tzv. result set), které lze dále využívat prostřednictvím operací nad kurzory

Kurzor = (abstraktní datový typ/ konstrukce / objekt) umožňující procházet záznamy vybrané dotazem, který je s kurzorem spojen

Kurzor = prostředek pro získání informace z databáze a předání do programu v jazyce PL/SQL

Když SELECT vrací více řádků, přes kurzor je možné výsledky procházet

Rozdělení kurzorů

IMPLICITNÍ kurzor = jednořádkové SQL (INTO)

- je deklarován a prováděn přímo v těle programu
- v tomto typu kurzoru jsou povoleny pouze příkazy SQL, které vrací jednotlivé řádky nebo nevrací žádné řádky,
- příkazy SELECT, UPDATE, INSERT a DELETE obsahují implicitní kurzory
- musí se shodovat datové typy sloupců a proměnných
- implicitní kurzor SELECT musí vracet pouze **jeden řádek (SELECT INTO** !)...

```
DECLARE
suma NUMBER;
BEGIN
 SELECT SUM(plat) INTO suma FROM Zamestnanci
END;
```

EXPLICITNÍ kurzor – můžeme deklarovat v části DECLARE pomocí klíčového slova CURSOR

- nutno deklarovat, otevřít, načíst data a uzavřít DECLARE CURSOR <jméno kurzoru>IS <dotaz>; OPEN <jméno kurzoru>; FETCH <jméno kurzoru>INTO <jméno proměnné1>, <jméno proměnné2>, …; CLOSE <jméno kurzoru>;

Deklarační část a příklad explicitního

Deklarace kurzoru mu přiřazuje název a spojuje s ním příslušný příkaz SELECT. Deklarace kurzorů je součástí části DECLARE společně s deklaracemi proměnných.

CURSOR jméno IS SELECT sloupce FROM tabulka;

```
DECLARE
  p_jmeno char(15);
  CURSOR k1 IS SELECT jmeno FROM zamestnanec;
BEGIN
  OPEN k1;
  LOOP
    FETCH k1 INTO p_jmeno;
    dbms_output.put_line(p_jmeno);
    EXIT WHEN k1% NOTFOUND;
  END LOOP;
  CLOSE k1;
END
```

Operace a Postup zpracování kurzoru

- funguje stejně jako přístup k souborům
- operace:
    - **OPEN** cursor
    - **FETCH** cursor INTO record
    - **CLOSE** cursor
- Postup: otevření a postupné čtení jednotlivých záznamů (čtení s posunem) a zavření

Atributy (explicitních) kurzoru (pro zjištění stavu kurzoru)

- **cursor%FOUND** obsahuje záznamy?
- **cursor%NOTFOUND** neobsahuje záznamy?
- **cursor%ISOPEN** je otevřený?
- **cursor%ROWCOUNT** dosud načtených řádků příkazem FETCH

Atributy pro definice typů:

- **tab%ROWTYPE** záznam typu řádku tabulky - nemusíme deklarovat proměnnou pro každý sloupec
  (DECLARE zamestnanec zamestnanci%ROWTYPE; BEGIN SELECT * INTO zamestnanec FROM zamestnanci; END;)

Použití kurzorů

Když je potřeba iterovat přes hromadu položek a pro každou položku něco provést (tedy ne pro všechny najednou, ale pro každou zvlášť)

- v triggerech
- v uložených procedurách

1. **Otevření** – tento krok zakládá pracovní množinu n-tic (řádků), která bude naplněna příkazem FETCH
2. **Načtení záznamu do kurzoru** – načtení n-tic (řádků) příslušného SELECTu do bloku PL/SQL se provádí příkazem FETCH. **FETCH** načte vždy jeden řádek příslušného SELECTu (prochází ho po řádcích). Proto FETCH dávat do cyklu
3. **Zavření** – příkaz **CLOSE** zavírá kurzor a znepřístupňuje množinu dat vybranou příkazem SELECT, zároveň je uzavřen příkazem EXIT

Aktualizační operace s kurzorem

n-tice přesunutá do kurzoru může být z databázové tabulky **vymazána**, resp. **aktualizována.** Pokud chceme využít této možnosti, je nutné, aby byl kurzor deklarován

FOR UPDATE OF (položka pro aktualizace)

a v příkazu FETCH uvedena klauzule

WHERE CURRENT OF

Např.

CURSOR k1 IS SELECT jmeno FROM zamestnanec WHERE plat <20000 **FOR UPDATE OF** plat;

FETCH k1 INTO p_jmeno
IF podmínka THEN DELETE zamestnanec **WHERE CURRENT OF** k1;
ELSE UPDATE zamestnanec SET … **WHERE CURRENT OF** k1;

Parametrické kurzory

- Některé kurzory mohou být závislé na parametru
- Deklarují se s uvedením parametrů a jejich typů – tyto **parametry lze potom užit v SQL** příkazu
- Konkrétní hodnotu parametru definujeme při otevírání kurzoru

```
DECLARE
CURSOR k1 (min IN Number) IS SELECT jmeno FROM zamestnanci
WHERE vek > min ORDER BY vek DESC;
BEGIN
OPEN k1(18);
```

(Omezení kurzorů)

- Jsou určeny pro statické SQL dotazy (nemohou zajistit variantní tvorbu SQL dotazu)
- Existuje PL/SQL podpora také pro dynamické SQL dotazy, které jsou tvořeny „za běhu“

3. Uložené procedury, funkce a balíky procedur a funkcí, kompilace, spouštění

Bloky příkazů jazyka PL/SQL, které jsou pojmenované a uložené ve spustitelné formě v databázi označujeme jako procedury a funkce.

Podprogram je pojmenovaný blok, který může být opakovaně volán a může přebírat aktuální parametry. Typy podprogramů jsou **funkce a procedury**. Procedury a funkce lze sdružovat do logických celků – balíků (package).

(Všechny anonymní bloky PL/SQL lze natrvalo uložit do databáze, a to buď ve tvaru procedury, nebo funkce.)

Vlastnosti procedur a funkcí

- Jsou uloženy ve zkompilovaném tvaru v databázi
- Mohou volat další procedury či funkce, či samy sebe
- Lze je volat ze všech prostředí klienta
- Funkce, na rozdíl od procedury, vrací jedinou hodnotu (procedura může vracet žádnou nebo více)

Uložené podprogramy – procedury a funkce

Procedury a funkce mohou být trvale uloženy do DB a mohou být použity jakoukoli aplikací, která s databázovým strojem komunikuje. Jednou přeložený a uložený program patří mezi databázové objekty a může být referován libovolným počtem aplikací.

Uložené podprogramy mohou být samostatné, nebo součástí balíku.

Podprogramy lze volat z:

- DB triggerů
- Jiných uložených podprogramů
- Aplikačních programů zapsaných ve vyšším programovacím jazyce, pro nějž existuje předkompilátor (ORACLE Pro)
- Interaktivně (SQL*Plus) : EXECUTE jmeno_procedury(parametry)

PROCEDURY

CREATE PROCEDURE jmeno_procedury [(parametry)] **AS** [lokální deklarace = proměnné]
BEGIN
[výkonné příkazy]
[EXCEPTION ošetření výjimek]
END;

Parametry procedury

Parametry mohou být vstupní, výstupní a vstupně-výstupní, parametry odděleny čárkou
Pouze vstupní parametry mohou být inicializovány (buď klauzulí DEFAULT nebo přirazením hodnoty :=)
jmeno_parametru [IN OUT IN OUT] typ_parametru [:= hodnota]

např:
CREATE PROCEDURE deleni (delenec IN number, delitel IN number)
AS
BEGIN
dbms_output.put_line(delenec/delitel);
END;
.
RUN
SET SERVEROUT ON //přesměrování výstupu na konzoli
EXECUTE deleni (10,2);

Kompilace a spuštění (SQL* PLUS)

Ukončení zápisu procedury na nový řádek **. (tečka)**

RUN – příkaz pro přeložení

EXECUTE [(parametry procedury)] – vykonání procedury

FUNKCE

CREATE FUNCTION jmeno_funkce [(parametry)] **RETURN** typ_navratove_promenne **AS** [lokální deklarace = proměnné]
BEGIN
[výkonné příkazy]
[EXCEPTION ošetření výjimek]
END;

CREATE FUNCTION f_deleni (delenec IN number, delitel IN number) RETURN Number
AS
vysledek Number;
BEGIN
vysledek :=delenec/delitel;
RETURN vysledek;
EXCEPTION when_zero_divide then
dbms_output.put_line("chyba deleni nulou");
END;

Vyvolání funkce

SET SERVEROUT ON
BEGIN
Dbms_output.put_line(f_deleni(12,4));
END;

Zrušení procedury či funkce

DROP PROCEDURE jmeno;
DROP FUNCTION jmeno**;**

Uložené balíky

Programový balík je sdružením řady funkcí a procedur s vlastním jmenným prostorem a vlastním persistentním prostorem **pro proměnné v rámci jedné session**

To **umožňuje uchovávat hodnoty v rámci session pro řadu procedur a funkcí**

Uložené balíky procedur a funkcí slouží ke sdružení logicky spolu souvisejících procedur a funkcí, ale i typů a objektů. Mohou obsahovat i globální proměnné, jejichž platnost je omezena délkou aktuálního spojení s databází.

Dvě části - definice balíku představuje definici

> **VEŘEJNÉ DEKLARACE (rozhraní balíku) –** plné hlavičky funkcí a procedru určených pro volání mimo balík
> (deklarace typů, proměnných, konstant, podmínek definujících nestandardní stavy, kurzorů, podprogramů dostupných zvenčí)
> (= co tam je za funkce parametry, ale bez těla) a
>
> **TĚLA balíku** – ostatní deklarace proměnných, definice příslušných veřejných objetků a další privátní podprogramy**, zpřístupnění pomocí tečkové notace**
> (implementuje specifikaci)
> (=normálně napsané procedury včetně implementace).

Př. Veřejné deklarace (Rozhraní):

```
CREATE OR REPLACE PACKAGE arithmetic AS usage INTEGER;

FUNCTION sub(a IN INTEGER, b IN INTEGER) RETURN INTEGER;
PROCEDURE inc(a IN OUT INTEGER);
END;
```

Př. Těla:

```
CREATE OR REPLACE PACKAGE BODY arithmetic AS

FUNCTION sub(a IN INTEGER, b IN INTEGER) RETURN INTEGER IS
BEGIN
usage := usage + 1;
RETURN (a - b);
END;

PROCEDURE inc(a IN OUT INTEGER) IS
BEGIN
usage := usage + 1;
a := a + 1;
END;
BEGIN
usage := 0;
END;
```

Kompilace

Pokud je volána procedura/funkce ve stavu INVALID, kompiluje se automaticky.

- Pokud se kompilace nezdaří, dojde k výjimce.

Ruční kompilace

- `ALTER PROCEDURE[FUNCTION] jmproc COMPILE;`
  - Pokud se kompilace nezdaří, dojde k výjimce

SQL*Plus a chyby kompilace – pomocí něj můžeme zjistit, jaké chyby se vyskytly při kompilaci. Pokud uložení procedury/funkce neproběhlo bez chyb, nelze ji používat a je nutné ji opravit. Pomocí SQL*Plus příkazů:

- SHOW ERROR – vypíše popis poslední chyby, na kterou při ukládání (kompilaci) narazil
- SHOW ERR typ jméno – např. SHOW ERR FUNCTION F – pro uvedený objekt
- Pomocí dotazu na tabulku USER_ERRORS

Spouštění

Proceduru můžeme zavolat (spustit různými způsoby:

- Pomocí direktivy EXEC – EXEC nastav_plat(123, 10000);
- V těle jiného PL/SQL bloku

```
BEGIN
...
Nastav_plat(123, 10000);
...
END;
```

Funkci můžeme volat také dvěma způsoby:

- V příkazu SELECT – SELECT secti(2, 3) FROM dual;
- V těle jiného PL/SQL bloku

```
DECLARE
a INTEGER;
b INTEGER;
BEGIN
a := 5;
b := secti(a, 2);
dbms_output.put_line('Vysledek : '||b);
END;
```

Při volání funkcí/procedur z balíků používáme tečkovou notaci (package.funkce()) pro kvalifikaci jejich jména.

Zabalený podprogram lze volat z db triggeru, jiného uloženého programu, aplikace napsané pro některý z předkompilátorů, standardních klientských nástrojů (SQL*Plus)

Standardní balík ***STANDARD*** = definuje prostředí PL/SQL.

4. Aktivní databáze – Oracle triggery, klasifikace a spouštění triggerů

- Aktivní pravidla = Trigger
- Databázový trigger je uživatelsky definovaný blok PL/SQL sdružený s určitou tabulkou. (Je implicitně spuštěn (proveden), jestliže je nad tabulkou prováděn aktualizační příkaz)
- Triggery (=“spoušťě) = jsou PL/SQL objekty spouštěné vyvoláním příslušné události v DB, vyvolání mohou způsobit 3 případy:
  - DML událost
  - DDL operace
  - Speciální DB událost
- Triggery jsou od SQL1999
- Triggery – proprietární řešení výrobců DB systémů

V řadě IS, které běží nad nějakou databází, potřebujeme v případě vzniku nějaké události (např. modifikujeme řádek v nějaké tabulce) automaticky spustit příkaz, který provede nějaké operace. K tomuto účelu slouží triggery (spouštěče). Trigger je speciální typ uložené procedury, která se aktivuje při splnění nějaké podmínky na serveru (aktualizace dat, události spojené s DB nebo session).

Triggery jsou vlastně procedury, které automaticky volá (spouští) SŘBD při definované události. Touto událostí může být buď vložení (INSERT), rušení (DELETE), nebo aktualizace (UPDATE OF) záznamu v tabulce (lze vázat i na update konkrétního sloupce).Pravidla lze kombinovat pomocí OR. Triggery bývají obvykle volány buď:

- Před specifikovanou událostí – **BEFORE**
- Po specifikované události – **AFTER**
- Místo specifikované události – **INSTEAD OF**
- Volání je také možno omezit podmínkou

Aktivní pravidla - TRIGGERY

ECA-paradigma

- Event = Událost – SQL příkaz pro manipulaci s daty (INSERT, DELETE, UPDATE)
- Condition = Podmínka - SQL podmínka, pouze řádkové Triggery
- Action = Akce – provádí příkazy (PL/SQL kód) (SELECT, INSERT, DELETE, UPDATE), nesmí být DDL příkazy

Sémantika aktivních pravidel: „Když nastane Událost, pokud je splněna Podmínka, proveď Akci“

Trigger = Pravidlo je:

- **SPUŠTĚNO - pokud nastane příslušná Událost – pokud událos sleduje více pravidel, tvoří se *KONFLIKTNÍ MNOŽINA* (**confict set)
- **VYHODNOCENO – po vyhodnocení dané Podmínky**
- **VYKONÁNÉ - po provedení jeho Akce**

Sémantika – cykly – mohou nastat pokud Trigger1 způsobí akci co znovu zpustí T1

„Konečný stav“ je určen prázdnou konfliktní množinou, zajistit konečnost vyvolávání triggerů je na autorovi pravidel

Postup při spouštění databázového triggeru

1. do ORACLE je předán příkaz INSERT, UPDATE nebo DELETE
2. provede se **příkazový** Trigger **BEFORE**
3. pro každý řádek, kterého se příkaz SQL týká:
    a. se provede **řádkový** trigger **BEFORE**
    b. změna řádku a kontrola integritního omezení
    c. se provede **řádkový** tirgger **AFTER**
4. proveď kontrolu integrity na úrovni příkazu
5. provede se **příkazový** trigger **AFTER**
6. návrat do aplikace

- Od uložených procedur a funkcí se odlišují tím, že jsou spuštěny *při modifikaci tabulky*, definují se pouze pro db tabulky a nepřijímají argumenty a lze je spustit jen při zmiňovaných DML příkazech.
- U triggeru lze rovněž specifikovat podmínku, kdy má být vykonáno jeho tělo (PL/SQL blok) a to použitím klauzule WHEN.
- Triggery jsou plně zkompilované po spuštění příkazem CREATE TRIGGER a po uložení procedurálního kódu v systémovém katalogu.

Trigger lze také

- deaktivovat (zakázat) - ALTER TRIGGER jmeno DISABLE;
- aktivovat – ALTER TRIGGER jmeno ENABLE;
- zrušit – DROP TRIGGER jmeno;

Výhody triggerů

nepovolí neplatné datové transakce, zajišťují komplexní bezpečnost, zajišťují referenční integritu přes všechny uzly db, zajišťují audit (sledování), spravují synchronizaci tabulek, zaznamenávají statistiku často modifikovaných tabulek.

Klasifikace triggerů

Příkazové triggery (statement level)
Příkazový trigger se spustí jedenkrát pro příkaz, bez ohledu na počet aktualizovaných řádků.

Triggery se spustí nad tabulkou bez ohledu na to, kolik tabulka obsahuje řádek. Např. pokud chci logovat změny provedené v DB, která obsahuje tabulky PRACOVISTE, ZAMESTNANCI do tabulky LOGY. Po každé operaci I, U, D nad tabulkami PRAC a ZAM se do tabulky LOGY vloží záznam o modifikaci tabulky a typu modifikace.

```
CREATE TRIGGER tai_pracoviste
AFTER INSERT ON pracoviste
BEGIN
INSERT INTO logy VALUES ( 'PRACOVISTE', 'I');
END;
```

Řádkové triggery – FOR EACH ROW
Řádkový trigger se spustí pro každý aktualizovaný řádek tabulky.

Trigger se spouští jednou pro každý řádek tabulky. Např. z předchozího příkladu chci mít u každého záznamu informaci, kdy byl zadán a kdo jej zadal.

```
CREATE TRIGGER trbi_pracoviste
BEFORE INSERT ON pracoviste
FOR EACH ROW
BEGIN
:new.zadal := user;
:new.datum := sysdate;
END;
```

Vlastní obsluhu události lze nadefinovat v PL/SQL bloku. Uvnitř PL/SQL bloku (a také klauzuli WHEN) řádkového triggeru se lze odkazovat na původní a nový záznam pomocí pseudoproměnných

:new (obsahuje vkládaný záznam) a

:old (původní záznam).

Je zřejmé, že při vkládání nového záznamu není definována :old a při mazání :new. Jejich jména lze předefinovat v klauzuli **REFERENCINGOLD AS.**

Business rules triggery
Triggery jsou také často používány při realizaci tzv. business rules, tj. **integritních omezení** specifických pro danou oblast použití. Např. použijeme – studenti si mohou zapsat max 20 kreditů za semestr. Pokud při vkládání dat do tabulky ZAPIS překročíme maximální povolenou hodnotu, dostaneme chybové hlášení – a to zařídí business trigger.

Zápis

CREATE OR REPLACE TRIGGER jméno
BEFORE | **AFTER** | INSTEAD OF
DELETE | **INSERT** | **UPDATE** OF cols
ON tabulka
[způsob odkazování]
[**FOR EACH ROW**]
[WHEN (podmínka)]
AS pl/sql kód

Další způsob rozdělení

DML Triggery:

- při rušení řádků DELETE
- při vkládání řádků INSERT
- při modifikaci sloupců UPDATE OF

způsoby vyvolání triggerů

- pro BEFORE a AFTER trigger chápán jako statement = příkazový – je vyvolán jedenkrát, ale lze stanovit jinka pomocí klauzule FOR EACH ROW
- pro INSTEAD OF je trigger chápán jako řádkový, statement nemá moc praktický význam

DDL TRIGGERY

- vyvolány po provedení DDL přikazu
- mohou být BEFORE nebo AFTER
- mohou být omezeny podmínkou WHEN
- definují se:
  jméno události ON DATABASE
  události = CREATE, ALTER, DROP, RENAME

TRIGGERY DATABÁZOVÝCH UDÁLOSTÍ

- pracují jako DDL trigger, pouze jiná množina povolených událostí
- typicky události zásadní pro celou db instance – STARTUP, SHUTDOWN, LOGON.
- Uvnitř DDL a databázovýc htrigerů nelze provide jiné DDL operac

Logické proměnné trigger

Pokud chceme zjistit jakou operací je Trigger volán, v případě že může být volán vice opracemi (INSERT OR DELETE) – lze to v průběhu trigger rozlišit

- INSERTING
- DELETING
- UPDATING

Omezení triggerů

- BEFORE a AFTER triggery nelze specifikovat nad pohledy
- V BEFORE triggerech není možné zapisovat do :old záznamů
- V AFTER triggerech nelze zapisovat do :old ani do :new záznamů
- INSTEAD OF triggery pracují jen s pohledy, mohou číst :old i :new, ale nemohou zapisovat ani do jednoho
- Nelze kombinovat INSTEAD OF a UPDATE
- Nelze definovat trigger nad LOB atributem

Dvě zásadní omezení

- Nelze použít transakce, pokud je zpracovávána jiná transakce, tedy prakticky nelze použít transakce vůbec
- Není možné sledovat ani modifikovat data v tabulce, která způsobila vyvolání DML triggeru
- -> jediné známé řešení je zrcadlení tabulek

Sémantika Oracle Triggerů

- **Spouštění probíhá okamžitě při události**, nelze je spustit explicitně (např. uživ. Příkazem)
- **Vnořené spouštění triggerů** - činnost triggeru může vyvolat jiný trigger - kontext aktuálního triggeru se uloží, začne se vykonávat nový trigger, pak se zas obnoví a pokračuje ten původní
- Maximální hloubka zanoření je 32, pak to hodí výjimku
- Při vyjímce /chybě jsou změny odrolovány „rollback“ – podpra částečného rollback

Doplňující pojmy

události (events)

- změna stavu databáze
- časové události
- externí, definované aplikací

podmínky (conditions)

- databázový predikát
- databázový dotaz

akce (actions)

- libovolná manipulace s daty
    - transakční příkazy
    - pravidla zpracování
    - externí procedury

Spouštění triggerů

- Vykonání triggeru
    - Okamžité (immediate)
        - Před událostí
        - Po události
        - Namísto události
    - Odložené (deferred)
        - Na konci transakce – odstartované příkazem COMMIT WORK
        - Po uživatelském příkazu
        - Následkem uživatelského příkazu
    - Oddělené (detached)
        - V kontextu samostatné transakce vypuštěné z počáteční transakce poté, co nastala událost
        - Možné kauzální závislosti počáteční a oddělené transakce
- Vykonání akce
    - Okamžité (immediate)
        - Následuje ihned po vyhodnocení podmínky
    - Odložené (deferred)
        - Akce je odsunuta na konec transakce
        - Akci vyvolá uživatelský příkaz
    - Oddělené (detached)
        - Probíhá v kontextu samostatné transakce vypuštěné z počáteční transakce ihned po vyhodnocení podmínky
        - Možné kauzální závislosti počáteční a oddělené transakce

Konfliktní množina

Obecně triggery jeden po druhém. ALE

Skupina aktivních pravidel, která mohou být aktivována současné. **Je zapotřebí metoda, která určí pořadí v konfliktní množině:**

- **Úplné uspořádání – trigger svázán s číselnou prioritou**
- **Částečné uspořádání – systémové uspořádíní + uživatelská priorita -> nedeterministcký**
- **Bez priorit – systémově definované - nedeterminismus**

5. Transakce, dvoufázový uzamykací protokol, detekce uváznutí

Transakce je logická jednotka zpracování dat, která se skládá z jednoho nebo více SQL příkazů provedených jedním uživatelem.
Transakce je uspořádaná skupina databázových operací (dotazů, procedur), která se vnímá a provádí jako jediná jednotka a to celá, nebo vůbec. Převádí databázi z jednoho konzistentního stavů do druhého.

Transakční zpracování – existuje posloupnost DML operací u kterých vyžadujeme **atomické** provedení „zdánlivě jako jedna jediná operace". Z důvodů výkonu se ovšem transakce jako atomická operace neprovádí (jedná se o několik dílčích operací). Před i po provedení transakce je databáze v konzistentním tvaru, v průběhu transakce tomu tak být nemusí

Databázové transakce musí splňovat tzv. vlastnosti **ACID**

- **A – Atomicity** - *atomičnost*
  Databázová transakce je jako operace dále nedělitelná (atomická). Provede se buď jako celek, nebo se neprovede vůbec (a daný databázový systém to dá uživateli na vědomí, např. chybovou hláškou).
- **C - Consistency** - *konzistentnost*
  Při a po provedení transakce není porušeno žádné integritní omezení.
- **I - Isolation** - *izolovanost*
  Operace uvnitř transakce jsou skryty před vnějšími operacemi. Vrácením transakce (ROLLBACK) není zasažena jiná transakce, jinak i tato musí být vrácena. V důsledku tohoto chování může dojít k tzv. řetězovému vrácení (cascading rollback).
- **D - Durability** - *trvalost*
  Změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi a již nemohou být ztraceny.

Způsoby ukončení transakce

- **COMMIT –** veškeré změny transakce jsou přijaty, jsou viditelné pro ostatní transakce v databázovém systému
- **ROLLBACK –** situace se vrací do okamžiku před zahájením transakce, všechny změny jsou označeny za neplatné

Globální vs. lokální

- **Lokální** transakce probíhá pouze na jediném uzlu.
- **Globální** (**distribuovaná**) transakce přesahuje rozsah jednoho uzlu.

Stavy transakce

- **Aktivní** – počáteční stav, transakce v něm setrvá po dobu provádění
- **Částečně potvrzený** - stav po provedení posledního příkazu transakce
- **Chybový stav** – po zjištění, že normální provádění není dál
- **Zrušený** - nastane po skončení operace ROLLBACK, databáze bude ve stavu před transakcí
- **Potvrzený** - po úspěšném vykonání COMMIT

Prováděné operace

Pro práci s transakcemi je nutné zavést následující operace:

- **BEGIN** - začátek transakce
- **COMMIT** - ukončení transakce a uložení dosažených výsledků do databáze
- **ROLLBACK** - odvolání změn - není-li definován savepoint, (místo, po které lze provedené změny vrátit zpět) tak návrat do stavu před započetím vykonávání transakce

Řešení souběhů transakcí

Rozvrh = plán posloupnosti všech operací v rámci všech uvažovaných transakcí

Protokol = soubor předem stanovených pravidel pro konstrukci Rozvrhu

Řešení souběhu pak spočívá v **uspořádatelnosti rozvrhu –** hledá se uspořádání operací v jednotlivých transakcích, aby byly vlastnosti ACID zachovány, zejména izolace.

Možnosti uspořádání rozvrhu

- **Serializace rozvrhu –** serializace potlačuje paralelní zpracování transakcí – snižuje tedy významně výkon databázového serveru
- Hledání jiných způsobů uspořádatelnosti rozvrhů by bylo časově velmi neefektivní (iterativní úloha) => konstrukce rozvrhů podle předem **stanovených pravidel –** soubor těchto pravidel označujeme jako **Protokol**
- Nejčastěji se **protokoly** konstruují na základě **zamykání a odemykání** objektů – k zamčenému objektu (LOCK) nemá jiná transakce přístup

Protokoly (soubory pravidel) pro konstrukci Rozvrhů se nejčastěji vyvářejí na základě:

A) Dvoufázový uzamykací protokol

- V první fázi uzamyká vše, co bude potřebovat při svém průběhu
- Druhá fáze začíná okamžikem prvního odemknutí
- Ve druhé fázi se zamčené objekty pouze odemykají – tj. nelze již použít operaci LOCK

Tedy transakce musí mít všechny objekty uzamčeny předtím, než nějaký objekt odemkne. Dá se dokázat, že pokud jsou všechny transakce v dané množině transakcí **dobře formované a dvoufázové**, pak **každý jejich legální rozvrh je uspořádatelný**. Dvoufázový protokol **zajišťuje uspořádatelnost**, ale **ne zotavitelnost** ani bezpečnost proti kaskádovému rušení transakcí nebo uváznutí. (deadlock = vzájemné uváznutí – vzájemné čekání na dokončení operace druhé transakce)

B) Metoda časových razítek

Metoda zajišťuje uspořádatelnost rozvrhu
Ke každému objektu jsou přiřazena dvě tzv. časová razítka:

- **TSR** – čas posledního čtení
- **TSW** – čas posledního zápisu

Na počátku každá transakce T dostane vlastní hodnotu **TS**(T), která je unikátní a stále roste.
Metoda časových razítek lze rozšířit na metodu **nezpůsobující uváznutí = deadlock**

Dobře formovaná transakce (souvisí se (ode)zamykáním)

- Před přístupem k objektu provádí jeho uzamknutí LOCK
- Neprovádí LOCK na objekty, které už zamkla
- Neprovádí UNLOCK na objekty, které nezamkla
- Před koncem provede UNLOCK na všechny objekty, které ve svém průběhu zamkla a jsou dosud zamčené

Definice uváznutí

Situace kdy jednotlivé transakce nekonečně dlouho čekají na uvolnění zamknutého objektu.

deadlock = vzájemné uváznutí – vzájemné čekání na dokončení operace druhé transakce

Uváznutí (deadlock) je situace, kdy pro každou transakci ze skupiny transakcí jsou splněny následující podmínky:

- Každá transakce ze skupiny je blokována čekáním na objekt databáze (na jeho odemknutí)
- Ukončení všech transakcí mimo uvedenou skupinu transakci neumožní odblokování žádné transakce ve skupině.

Je nutné **zabránit vzniku** uváznutí nebo toto **uváznutí detekovat a odstranit ho:**

- **Detekci** uváznutí a následné **odstranění** užíváme při **uzamykacím protokolu**.
- **Předcházení** uváznutí řešíme metodou **časových razítek**

Oracle: základní systém trasakcí podporuje ACID, využívá dvoufázového zamykání a provádí detekci uváznutí (deadlocků) s jejich přerušením formou výjimky

Optimistické vs. pesimistické zamykání

- U ***pesimistického zpracování*** se v jeho průběhu změny zaznamenávají do dočasných objektů (například a nejčastěji: do řádků tabulek s příznakem dočasných dat, platných jen po dobu transakce) a teprve po přesunu/změně dat se odznačí příznak dočasnosti a data se stanou platnými. (Tento způsob se dá přibližně připodobnit přepisu souboru, při kterém se nejdříve nová verze souboru nakopíruje pod dočasným jménem a teprve poté se tento soubor přejmenuje za starý a tím ho nahradí.)
- U ***optimistického zpracování*** se (optimisticky) předpokládá, že při transakci nenastane chyba a nebude třeba ji vrátit zpět (přestože tato možnost je zachována). Měněné záznamy v tabulkách jsou při optimistickém zpracování transakce zapisovány „natvrdo“, současně s tím se však vytváří tzv. rollback log coby seznam SQL příkazů, které dokáží prováděné změny vrátit zpět. V případě, že při transkaci dojde k nějaké nezotavitelné chybě, tento log se provede a transakce (aby dodržela pravidlo atomicity) skončí ve výchozím stavu s chybou. Naopak, na konci transakce, při které k žádné takové chybě nedošlo, se rollback log maže.

Žurnály

Jsou záznamy, které uchovávají informace o průběhu transakcí a slouží k zotavení po vzniklé chybě. Žurnály musí být v každém uzlu a obsahují záznamy o historii každé transakce.

2-fázové protokoly podrobněji

Dvoufázový protokol (2PL)

Dvoufázová transakce v první fázi zamyká vše co je potřeba a od prvního odemknutí (druhá fáze) již jen odemyká co měla zamčeno (již žádná operace LOCK). Tedy transakce musí mít všechny objekty uzamčeny předtím, než nějaký objekt odemkne. Dá se dokázat, že pokud jsou všechny transakce v dané množině transakcí dobře formované a dvoufázové, pak každý jejich legální rozvrh je uspořádatelný. Dvoufázový protokol zajišťuje uspořádatelnost, ale ne zotavitelnost ani bezpečnost proti kaskádovému rušení transakcí nebo uváznutí.

Striktní dvoufázový protokol (S2PL)

Problémy 2PL jsou nezotavitelnost a kaskádové rušení transakcí. Tyto nedostatky lze odstranit pomocí striktních dvoufázových protokolů, které uvolňují zámky až po skončení transakce (COMMIT). Zřejmá nevýhoda je omezení paralelismu. 2PL navíc stále nevylučuje možnost deadlocku. Read-lock je tu možné uvolnit kdykoli během 2. fáze, write-lock jen na konci.

SS2PL

Prakticky se dnes používá SS2PL, což je Strong Strict twophase locking, které sice nevylučuje deadlock, ale pokud k němu dojde, tak ho umí automaticky vyřešit. Liší se od SS2PL jen tím, že i read-lock uvolňuje až na konci.

Konzervativní dvoufázový protokol (C2PL)

Rozdíl oproti 2PL je ten, že transakce žádá o všechny své zámky, ještě než se začne vykonávat. To sice vede občas k zbytečnému zamykání (nevíme co přesně budeme potřebovat, tak radši zamkneme víc), ale stačí to již k prevenci uváznutí (deadlocku). Pomalý, je třeba vědět předem, co se bude číst/zapisovat, nepoužívá se.

Detekce uváznutí a zotavení

- Uváznutí se detekuje pomocí čekacího grafu
  - Vrcholy jsou transakce Ti
  - Orientovaná hrana Ti → Tj značí, že Ti čeká, až Tj odemkne datovou položku
  - Je-li v čekacím grafu cyklus, došlo k uváznutí
- Hledá se takový plán transakcí, aby se co nejmíň kryly a tak, aby byl dodržen princip ACID (hlavně Isolation), když se takový plán povede najít, nazývá se uspořadatelný
- **Well formed transakce** (správně zamykat a odemykat)
- **Když se zjistí uváznutí**
  - Je nutno nalézt obětní transakci a vnutit jí abort (a tím i obnovu dat). Obětuje se obvykle nejmladší transakce, tj. ta, která ještě neudělala mnoho změn
  - Transakce mohou stárnout, bude-li za oběť vybírána vždy nejmladší transakce. Proto je vhodné do kritéria výběru obětí zahrnout i počet transakcí provedených návratů.
  - Která data se ale mají obnovovat?
    - **Totální obnova** transakci úplně zruší, data se vrátí do počátečního stavu, a transakce se restartuje. To může být velmi nákladné
    - Efektivnější je, když se transakce "vrací postupně" do stavu, kdy uváznutí zmizí. Tento postup je ale náročný na evidenci kroků a změn transakcí provedených: metoda kontrolních bodů (***checkpointing***) – konzistentní mezistavy

Zajištění uspořadatelnosti pomocí pořadových čísel transakcí

Transakce vyvolá write W(x):

1. TSR(x) > TS(t): zápis do „později přečtené“ paměti > ROLLBACK
2. TSW(x) > TS(t): zápis do „později přepsané“ paměti > ROLLBACK
3. jinak proveď zápis

Transakce vyvolá read R(x):

4. TSW(x) > TS(t): čtení z „později zapsané“ paměti > ROLLBACK
5. jinak read

6. Optimalizace dotazu, jednotlivé přístupy (např. Cost Based optimalizace (CBO)), podstata optimalizátoru, přínos optimalizace.

SQL je velmi flexibilní jazyk – Dvěma či více různými dotazy je možno obdržet stejná data.

Rychlost různých dotazů ovšem nemusí být stejná i přesto, že vracejí stejná data.

Důvodem optimalizace je minimalizace nákladů na:

- Zdrojový čas
- Kapacitu paměti (prostor)
- Programátorskou práci
  tzn. snaha dosáhnout maximálního výkonu se stávajícími prostředky

Na optimalizaci se podílí: návrhář databáze, vývojář, správce databáze, uživatel

Zpracování SQL dotazů v Oracle

se skládá z následujících komponent

- parser
- optimalizátor – jádro celého zpracování, analyzuje sémantiku dotazu, hledá optimální způsob provedení, liší se v přístupu, jakým heldájí optimální plán vykonávání (RBO/CBO)
- generátor řádkových zdrojů
- vlastní provádění SQL

Obecná pravidla pro psaní SQL dotazů

Vyplývají z technik optimalizace:

- V selectu nepoužívat v seznamu sloupců *, protože ve většině případů nepracujeme se všemi, ale vyjmenovat sloupce
- Používat co nejméně klauzuli LIKE, IN, NOT IN (vhodnější je WHERE a WHERE NOT EXISTS)
- Používat klauzule typu LIMIT
- Ve WHERE na začátek dávat podmínky, po kterých vypadne co nejvíc záznamů (DS nejprve vyhledá záznamy, vyhovující první podmínce, z nich pak vybírá ty co vyhovují druhé podmínce – na začátku vyhodit co nejvíce)
- Výběr vhodného pořadí spojení
- Používat hinty (podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu)
- Nastavit indexy
- Jeden dotaz psát všude přesně stejně

Přínos:

- zdrojový čas
- kapacitu paměti či prostoru
- programátorskou práci
- přenesená data

Optimalizace

SŘBD Oracle již sám používá optimalizačních technik pro vyhodnocení jakéhokoli dotazu. Těmito technikami jsou:

- **Rule based optimaliaztion (RBO)**
- **Cost based optimalization (CBO)** – od verze Oracle 9i je preferovaný

Jak zjistit způsob provedení příkazu? Abychom mohli zjistit, jak ve skutečnosti daná optimalizace vyhodnocení dotazu funguje, potřebujeme vytvořit tabulku **PLAN_TABLE** (podle skriptu *utlxplan.sql),* kam optimalizátor ukládá své vítězné plány právě vyhodnoceného dotazu. Pro vysvětlení (tj. zjištění optimálního plánu) vyhodnocení dotazu použijeme příkaz **EXPLAIN_PLAN.**

```
explain plan for select p.NAZEV,m.ZKRATKA, ...
```

Vykonáním tohoto příkazu se vítězný plán uloží do tabulky PLAN_TABLE v podobě několika záznamů. Informace vyčteme s použitím dotazu:

```
select plan_table_output from table(dbms_xplan.display());
```

CBO/RBO

Oracle doporučuje používat pouze CBO, který je stále vylepšován a RBO je implementován hlavně kvůli zpětné kompatibilitě.

RBO (Rule Based Optimization)

Vyhodnocuje jednotlivé přístupové cesty pomocí **předem daného system pravidel**

- Starší přístup, dnes často deprecated (Oracle),
- Odvozuje plán ze syntaxe příkazu a existence indexů (a systému pravidel)
  řídí se předem sestavenou sadou pravidel, která nezohledňují např:
  - velikost tabulky -- ***Malá tabulka*** (obsahuje 5 řádků a vejde se do jednoho datového bloku), je rychlejší jí přečíst celou než hledat podle indexu (1 IO operace vs. čtení bloku indexů a pak dat)
  - Pokud existuje více neunikátních indexů na jedné tabulce, nemusí optimalizátor vybrat ten nejlepší. Použití určitého indexu je možné optimalizátoru znemožnit použitím výrazu v dotazu.

Ceny přístupu k podmnožině řádek v tabulce v klesajícím pořadí:

- **Plný přístup (Full scan)** – prochází se celá tabulka – všechny záznamy, u každé řádky se ověří podmínka. Vhodné, pokud procento vyhovujících řádek bude velké.
- **Index-Range-Scan** – vyhledání intervalu v indexu. Ověření ostatních podmínek v odkazovaných řádcích
- **Unique-Index-Scan** – vyhledání jediné možné vyhovující řádky podle unikátního indexu
- **ROWID-Scan** – vyhledání řádky na základě známé hodnoty jejího fyzického identifikátoru v DB.

Z důvodu kompatibility je možné jej však aktivovat odpovídajícím hintem RULE.

CBO (Cost Based Optimization)

Hledá plán s nejmenšími náklady – používá statistiky

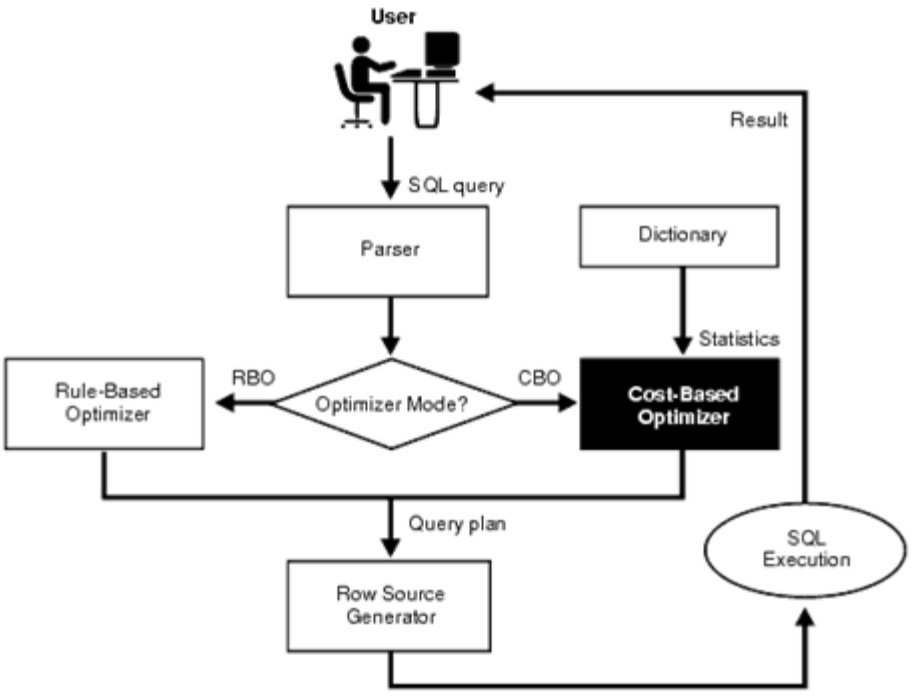
Pro jednotlivé plány se počítá cena provedení v řadě hledisek. Hledá plán s nejmenšími náklady pomocí statistik. Optimalizace je založena na statistikách, počítá cenu zdrojů provedení operace (čas, CPU, paměť, množství I/O,…). Využívá **statistiky** o tabulkách a datech, které jsou uloženy v Data Dictionary:

STATISTIKY

- Počet různých hodnot ve sloupci, histogramy rozložení hodnot ve sloupci, počet řádek v tabulce, průměrná délka jedné řádky.
- Některé statistiky jsou přístupné i pro uživatele db pomocí pohledů, či tabulek
- Aktualizují se výpočtem nebo odhadem
- **Typy statistik**:
  - **Údaje o tabulkách** – počet řádků v tabulce, bloků, délka jedné řádky
  - **Údaje o sloupcích** – počet unikátních hodnot ve sloupci a NULL, histogram distribuce dat
  - **Údaje o indexech** – počet listových bloků, clustering
- Dokáže rozlišit plány i pro různé typy konstant v dotazu.
- Použití CBO je doporučeno firmou Oracle. Vybírá se plán s nejnižší váženou cenou.

Podstata optimalizátoru a přínos

Podstata: stejná data lze z databáze získat různými dotazy (SELECT * vs. SELECT col1, col2…), výsledek bude stejný, ovšem zpracování se může lišit potřebným časem a systémovými nároky.



Výstupem optimalizátoru je plán vykonávání (**execution plan**), který určije:

- Přístupové cesty k jednotlivým tabulkám používaných dotazem a pořadí jejich spojování (JOIN order)

Hints

Hint = podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu. Hinty se aplikují na blok dotazu, ve kterém se vyskytují.

V CBO lze využít pro optimalizaci i nápovědu – Hints. Prostřednictvím ní mohu optimalizátoru vnutit některou operaci, protože si myslím, že její užití přispěje k lepší optimalizaci. Tato nápověda se zapisuje jako komentář specifického tvaru.

```
explain plan for select /*+ INDEX (predmety predmety_index1) */
p.NAZEV,…
```

Indexy

- Tvorba indexů není v SQL-92 standardizována
- Jednotlivé databázové systémy řeší tvorbu indexů svými prostředky, které jsou navzájem více či méně podobné
    - Může se lišit syntaxe, podpora různých typů indexů, jejich použití/nepoužití pro daný dotaz

Procházení tabulky pomocí indexu trvá mnohem kratší dobu než procházení tabulky bez jeho použití.

A) B-tree indexy

- Obvykle redundantní B+ stromy
    - Hodnoty v listech
    - Listy oboustranně linkované pro snadný sekvenční průchod
    - **Vhodné pro sloupce s vysokou selektivitou (počtem různých hodnot ve sloupci)**
    - Vícesloupcové (složené) indexy mohou zvýšit selektivitu
    - Nad jednou tabulkou v jednom dotazu nelze obvykle kombinovat více B-tree indexů. Dotaz se vyhodnocuje s použitím jednoho z indexů a ostatní podmínky se dopočítávají.

B) Bitmapové indexy

- **Pro každou hodnotu sloupce/výrazu vytvořen binární řetězec obsahující 1 právě pro řádky s danou hodnotou**
    - **Vhodné pro sloupce s nízkou selektivitou**
    - Lze kombinovat více bitmapových indexů nad jednou tabulkou pro zvýšení selektivity
    - Kombinací více bitmap se zvyšuje selektivita indexu

Indexy nepomohou

- Pokud je procento vyhovujících záznamů velké (zvýšená režie s přístupem k řádkům v nesekvenčním pořadí daném indexem)
- Při dotazech na hodnotu null – v indexech se běžně neukládá

Indexy pomohou

- V dotazech na rovnost sloupce s konstantou
- V dotazech na to, zda je hodnota v intervalu

Indexy jsou automaticky vytvářeny

- Pro primární klíče
- Pro sloupce s UNIQUE (kandidátní klíče)

Vždy vytvářet indexy pro cizí klíče!!!

- Zrychlení odezvy při manipulaci s nadřízenou tabulkou
- Průchod přes index najde efektivně všechny existující závislé řádky bez nutnosti čtení celé tabulky

Výběr typu optimalizace

Je dalším krokem, kterým můžeme ovlivnit optimalizaci. Jedná se o změnu optimalizačního typu optimalizátoru SŘBD Oracle. Typy optimalizace jsou:

Typ optimalizace se vybírá příkazem:

```
ALTER SESSION SET OPTIMIZER_GOAL = ALL_ROWS;
```

- CHOOSE – výběr podle (ne)přítomnosti statistik nejsou-li k dispozici, potom RBO, jinak CBO.
- ALL_ROWS – vždy CBO, minimalizuje se cena za získání všech řádek odpovědi. Vhodné pro dávkové zpracování.
- FIRST_ROWS – vždy CBO, minimalizuje se cena za získání prvních řádek odpovědi. Vhodné pro interaktivní zpracování.
- RULE – vždy RBO.

Další možnost ladění je změna módu optimalizátoru. Dotazy mohou být optimalizovány na:

- Nejlepší průchodnost – ALL_ROWS
- Nejrychlejší odezvu – FIRST_ROWS_1 – zkrátí čas vyhodnocení dotazu.

Př. nastavení módu:

```
ALTER SESSION SET optimizer_mode = all_rows;
```

7. Postrelační databáze – výhody a nevýhody, mapování, RDB, ORDB, OODB

Postrelační databázový systém je relační databázový systém rozšířený o nějakou specializaci na databázové úrovni, jelikož aplikační řešení by bylo nedostačující

Proč více databázových technologií? – požadavky nových aplikací:

- Nové typy objektů a funkcí
- OO analýza a návrh vs. Relační db

Cíl: integrace a správa dat v jednom systému (fotky, dokumenty, mapy, maily)

Příklady postrelačních DB systémů

- *Prostorové databáze* — rozšířeny o práci s prostorovými objekty a vztahy mezi nimi
- *Objektově orientované databáze* — rozšířeny o objektový model dat a vazby
- *Deduktivní databáze* — rozšířeny o funkce pro analýzu dat
- *Temporální databáze* — rozšířeny o temporální logiku
- *Multimediální databáze* — rozšířeny o funkce pro práci s multimediálním obsahem
- *Aktivní databáze* — rozšířeny o aktivní pravidla

V současnosti všechny používané databázové systémy jsou postrelační, jelikož obsahují nějaká rozšíření oproti původnímu relačnímu schématu (např. triggery).

- **RDB** (RSŘBD) - relational database
- **ORDB** (ORSŘBD) - object-relational database
- **OODB** (OOSŘBD) - object oriented database

Jedná se o všechny současné databáze - jde o relační databáze doplněné o nějakou "funkci" navíc, např. aktivní databáze (triggery) už jsou postrelační databáze.

Relační databáze

Technologie relačních databází byla původně navržena E.F.Coddem a později ji implementovala IBM a jiní. Standard je popsán ANSI a ISO normou, častěji se na ni ovšem odvoláváme jako na SQL + číslo verze. Poslední je tedy SQL2. Novější verze SQL3 obsahuje navíc některá objektová rozšíření.

Datový model

RDB uchovává data v databázi skládající se z řádků a sloupců. Řádek odpovídá záznamu (record, tuple); sloupce odpovídají atributům (polím v záznamu). Každý sloupec má určen datový typ. Datových typů je omezené množství, typicky 6 nebo víc (např. znak, řetězec, datum, číslo...). Každý atribut (pole) záznamu může uchovávat jedinou hodnotu. Vztahy nejsou explicitní, ale spíše plynou z hodnot ve speciálních polích, tzv. cizí klíče (foreign keys) v jedné tabulce, který se rovná hodnotám v jiné tabulce.

Dotazovací jazyk

Pohled (view) je podmnožina databáze, která je výsledkem vyhodnocení dotazu. V RDB je pohled tabulka. RDB využívá SQL pro definici dat, řízení dat a přístupu a získávání dat. Data jsou získávána na základě hodnoty v určitém poli záznamu (buňka v řádce).

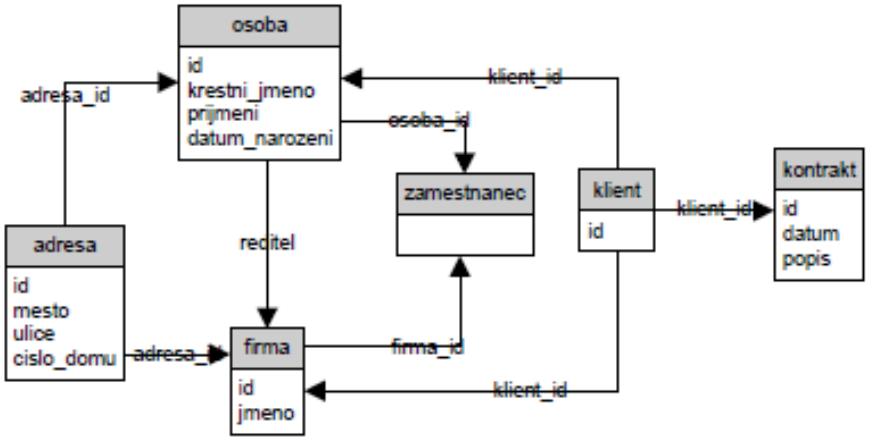
Výpočetní model

Veškeré zpracováváni je založeno na hodnotách polí záznamů. Záznamy nemají jednotné identifikátory, které jsou neměnné po dobu existence záznamu. Neexistují žádné odkazy z jednoho záznamu na jiný. Vytvoření výsledku je prováděno pod kontrolou kurzoru, který umožňuje uživateli sekvenčně procházet výsledek po jednotlivých záznamech. Totéž platí pro update.

Výhody:

- Výkonné OLTP
- Dostupnost dat
- Utajení
- Prostředky pro správu dat
- Standardní jazykové rozhraní
- Řízení paměti
- Souběžné zpracování dat
- Integrita

Příklad:



obr. č. 2. - relační implementace databáze

Objektově orientované databáze

- V roce 1993, návrh standardu ODMG-93
- Dotazovací Obejct Query Language (OQL)

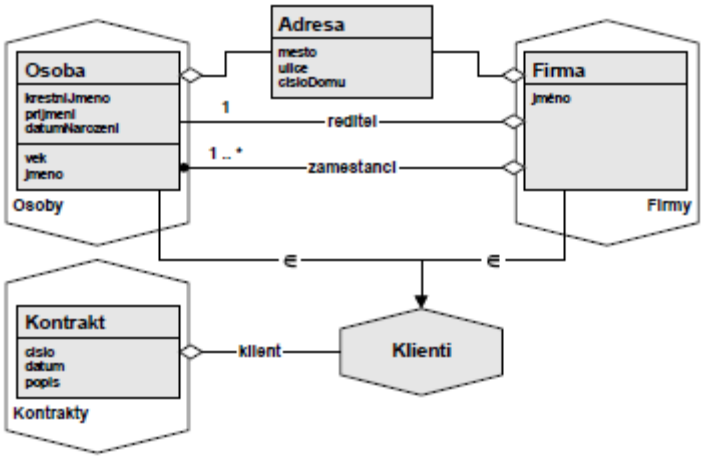
Základní koncepty ODMG-93

- Třída – šablona pro instance
- Instance – objekty, s atributy a metodami
- Atribut – primitivní typy nebo ADT
- Metoda – práce s atributy
- Identifikátor objektu OID
- Zapouzdření
- Hierarchie tříd, dědění

Objektově orientované databáze

Vedle relačních databází lidé se začal vyvíjet nový typ databázových systémů, založených na principech objektového programování. Co nového objektové databáze přináší? Tak jako jsme mohli vnímat přechod od strukturálního programování k objektovému programování (např. klasický Turbo Pascal a Delphi), tak můžeme vnímat i přechod z relačních databází na objektové databáze. Základem OO databáze není tentokrát tabulka, ale objekt. Každý objekt má atributy/vlastnosti (zde je vidět analogie se sloupci v tabulce) a metody, které nějakým způsobem manipulují s hodnotami vlastností. Jednotlivé "záznamy" jsou instance objektu s konkrétními hodnotami (v relačních databázích - 1 řádek). Lze zde využít všech výhod dědičnosti (a to i mnohonásobné), zapouzdřenosti a polymorfismu. Díky tomu OO databáze výrazně rozšiřují možnosti tvorby databázových aplikací.

Příklad oproti relační:



obr. č. 3. - objektová implementace databáze

Pro objektové databáze neexistuje žádný oficiální standard. Standardem je de facto kniha Morgana Kaufmana The Object Database Standard: ODMG-V2.0. Důraz ODB je na přímou korespondenci mezi následujícími:

* Objekty a objektové vztahy v aplikaci napsané v OO jazycích

* jejich uchovávání v databázi.

- Objektově orientované databáze (OODB) využívají objektových principů, jako jsou abstraktní datové typy, zapouzdření, inheritance, polymorphismus apod. Struktura objektu je (i, c, v) = (unique id, constructor, stav objektu). Unique Object identifiers, …

OQL = Object Query Language = deklarativní jazyk, přidaná flexibilita

http://goldberg.berkeley.edu/courses/F04/215/215-OODB.ppt

Datový model

Objektové databáze využívají datového modelu, který má objektově orientované aspekty jako třídy s atributy a metodami a integritními omezeními; poskytují objektové identifikátory (OID) pro každou trvalou instanci třídy; podporují zapouzdření (encapsulation); násobnou dědičnost (multiple inheritance) a podporují abstraktní datové typy.

Objektové databáze kombinují prvky objektově orientovaného programování s databázovými schopnostmi. Rozšiřují funkčnost objektových programovacích jazyků (C++, Smalltalk, Java) a poskytují plnou schopnost programování databáze. Datový model aplikace a datový model databáze se ve výsledku hodně shodují a výsledný kód se dá mnohem efektivněji udržovat.

Dotazovací jazyk

Objektově orientovaný jazyk (C++, Java, Smalltalk) je jazykem jak pro aplikaci, tak i pro databázi. Poskytuje těsný vztah mezi objektem aplikace a uloženým objektem. Názorně je to vidět v definici a manipulaci s daty a v dotazech.

Výpočetní model

V RDB rozumíme dotazovacím jazykem vytváření, přístup a aktualizaci objektů, ale v ODB, ačkoliv je to stále možné, je toto prováděno přímo pomocí objektového jazyka (C++, Java, Smalltalk) využitím jeho vlastní syntaxe. Navíc každý objekt v systému automaticky obdrží identifikátor (OID), který je jednoznačný a neměnný během existence objektu. Objekt může mít buď vlastni OID, nebo může ukazovat na jiný objekt.

Výhody:

- Operace na složitých objektech
- Rekurzivní struktury
- Abstraktní datové typy
- Rozhraní k OO jazyku
- Složité transakce

Objektově-relační databáze

Důvod nástupu OR db: obdržet maximu z investic do relačních technologií, využít přínosů OO modelování

"Rozšířená relační" a "objektově-relační" jsou synonyma pro databázové systémy, které se snaží sjednotit rysy jak relačních, tak objektových databází. ORDB je specifikována v rozšíření SQL standardu — SQL3. Do této kategorie patří např. Informix, **IBM, Oracle** a Unisys.

Datový model

ORDB využívají datový model tak, že "přidávají objektovost do tabulek". Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu, nazývanou abstraktní datové typy (ADT). ADT je datový typ, který vznikne zkombinováním základních datových typů. Podpora ADT je atraktivní, protože operace a funkce asociované s novými datovými typy mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu. ORDB jsou nadmnožinou RDB a pokud nevyužijeme žádné objektové rozšíření jsou ekvivalentní SQL2. Proto má omezenou podporu dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem.

Dotazovací jazyk

ORDB podporuje rozšířenou verzi SQL — SQL3. Důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod. ORDB je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření, pracuje právě s nimi.

Výpočetní model

Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s databází. Přímá podpora objektových jazyků stále chybí, což nutí programátory k překladu mezi objekty a tabulkami.

Dva přístupy:

- **univerzální paměť**, kdy všechny druhy dat jsou řízeny SŘBD), jde o integraci (různými způsoby!) ⇒ univerzální servery
- **univerzální přístup**, kdy všechna data jsou ve svých původních (autonomních) zdrojích

Technika: middleware

- brány (min. dva nezávislé servery)
- zobrazení schémat, transformace dotazů
- objektové obálky: Persistence Software, Ontologic, HP,
- Next, ... (problémy: výkon)
- DB založené na Web

Rozšiřitelnost, uživatelsky definované typy a funkce

Možnost přidávání nových datových typů + programů (funkcí) „zabalených do speciálního modulu“

- **UDT – uživatelsky definované typy**
- **UDF – uživatelsky definované funkce**

Oracle používá k rozšíření **Cartridges**

Objektově relační modelování

Rozšíření relačního modelu o objekty a konstrukty pro manipulaci nových datových typů.

Atributy n-tic jsou složité typy, včetně hnízděných relací

Zachovány jsou relační základy včetně deklarativního přístupu k datům

Shrnutí

Relační model je jednoduchý a elegantní, ale je naprosto rozdílný od objektového modelu. Relační databáze nejsou navrhovány pro ukládání objektů a naprogramování rozhraní pro ukládání objektů v databázi je velmi složité. Relační databázové systémy jsou dobré pro řízení velkého množství dat, vyhledávání dat, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. Jazyky jako SQL umožňují tabulky propojit za běhu, aby vyjádřily vztah mezi daty.

Naproti tomu **objektově orientovaný model** je založen na objektech, což jsou struktury, které kombinují daný kód a data. Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty "v databázi"; tj. místo věčného přeskakování mezi jazykem aplikace (např. C) a dotazovacím jazykem (např. SQL) může programátor jednoduše používat objektový jazyk k vytváření a přístupu k metodám. Krátce řečeno, ODB jsou výborné pro manipulaci s daty.

Hlavní rozdíl je v přístupu ke vztahům

- v OO databázích jsou vztahy reprezentovány pomocí OIDs, což zlepšuje přístup k datům
- v relačních databázích jsou vztahy mezi n-ticemi specifikovány atributy se stejnou doménou.

Nevýhody OO

- Chabý výkon (ORM 15-20% slabší než samotný JDBC driver). Ve srovnání s relačními jsou optimalizátory pro OO DB velmi složité.
- Problémy se škálovatelností, neschopnost podporovat rozsáhlé systémy.

Doplnit z přednášky…, mapování

Standardizace SQL/MM - např: full-text, prostorová data, obrázky (i videa) – jde o řadu UDT a UDF dle SQL:1999

Relační model:

- Identifikace klíčem
- Klíč je modifikovatelný
- Klíč je jedinečný v rámci relace

OO model:

- Identifikace OID
- OID nelze měnit
- OID je jedinečný v databázi

8. ANSI/ISO normy SQL – objektové vlastnosti jazyka SQL99

ANSI/ISO normy SQL

Vývoj standardů SQL:
SQL86
SQL89
SQL92
SQL/Call Level Interface 95
SQL/Persistent Stored Module Language Interface 96
SQL/Java
SQL99
SQL/Object Language Bindings 2000
SQL/Management External Data 2000
SQL/OLAP
SQL/temporal
SQL/Schemata
SQL/XML
SQL/MM – Framework, Full text, Spatial, Still Image

Pracovní názvy:
SQL1 (>SQL86)
SQL2 (>SQL92),
SQL3 (>SQL99),
SQL4

SQL-86 (SQL 87)

První standard formalizovaný ANSI

SQL-92 (SQL2)

Standard je rozdělen na tři úrovně: *entry*, *intermediate* a *full*. Někdy je také uváděn mezistupeň mezi entry a intermediate jako *transitional*. Úrovně slouží k tomu, aby mohlo být u implementací standardu (jednotlivých databází) uvedeno do jaké míry splňují daný standard.

Změny možno klasifikovat jako:

Entry

- Jen formální změny oproti SQL-86

Transitional

- Podpora různých druhů spojení jako NATURAL JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN
- Podpora nových datových typů DATE, TIME, TIMESTAMP and INTERVAL, including various datetime and interval features (excluding time zones)

Intermediate

- Podpora dlouhých identifikátorů (do 128 znaků)
- Podpora kurzorů a směru získávání dat příkazem FETCH
- Podpora definice a používání znakových sad

Full

- Podpora dočasných tabulek
- Možnost výběru přesnosti u datových typů TIME a TIMESTAMP
- Možnost testování pravdivostních hodnot pomocí TRUE, FALSE or UNKNOWN
- Vnořené tabulky ve FROM
- Podpora UNION JOIN a CROSS JOIN
- Podpora pro collations znakových sad
- Vylepšené udělování práv

SQL:1999 (SQL3)

Jedná se o standard pro relačně-objektový dotazovací jazyk (na rozdíl od předchozích verzí, které byly pouze relační)

- Podpora objektů
- Uložené procedury
- Triggery
- Rekurzivní dotazy
- Regulární výrazy
- Rozšíření pro OLAP
- Procedurální rozšíření (Příkazy řízení běhu - LOOP, IF...)
- Objektové rozšíření
- nové typy STRING, BOOLEAN, REF, ARRAY, typy pro full-text, obrázky, prostorová data

The SQL:1999 standard introduced a number of object-relational database features into SQL, chiefly among them **structured user-defined types**, usually called just **structured types**. These can be defined either in plain SQL with CREATE TYPE but also in Java via SQL/JRT. SQL structured types allow single inheritance.

Existují i novější standardy

- **SQL:2003** (Představeny XML-vázané funkce, window funkce, standardizované sekvence a sloupce s automaticky generovanými hodnotami)
- **SQL:2006** (SQL může být použito ve spojení s XML - možnost importu a skladování XML dat v SQL databázi, manipulaci s nimi a publikace dat v XML formě. Možnost využití XQuery)
- **SQL:2008** (ORDER BY mimo definici kurzoru, INSTEAD OF triggery, přidán TRUNCATE příkaz)
- Jednotlivé databázové servery ne vždy dodržují ANSI normu – obvykle pouze SQL-92 Entry
- Čím vice se při vývoji aplikace využijí rysy vyšší než SQL-92 Entry, tím je menší šance, že aplikace bude provozuschopná i na jiné databázi

většina z těchto rozšíření lze najít v jiných otázkách, proto zde nebudou více rozepsána.

Vlastnosti SQL1999

Pět částí SQL1999

- SQL/Framework
- SQL/Foundations
- SQL/CLI (Call Level Interface) – alternative k volání SQL z aplikačních program (implementace ODBC)
- SQL/PSM (Persisten Store Modules) – procedurální jazky pro psaní transakcí
- SQL/Bindings

Objekty

SQL3 pro podporu objektů používá:

- Uživatelem definované typy (ADT, pojmenované typy řádků a odlišující typy)
- Kolekce
- Uživatelem definované funkce UDF
- Uživatelem definované procedury UDP
- Velké objekty (Large Objects neboli LOB) – např uložení videa "video BLOB(3G)"

Standard SQL:1999 – podmnožina celkové koncepce

Nové typy SQL99

A) Konstruované **atomické** typy:

- **Reference**
- **Odlišující typy** "CREATE TYPE nejaky_typ AS CHAR(5) FINAL;" – odvození od předdefinovaných typů. Např. CHAR(5) si nějak pojmenuju a to je ono.

B) Konstruované kompozitní (**strukturované**) typy:

- **Array** – uspořádaný seznam dané maximální délky, nejsou povoleny žádná pole polí (vícedimenz), je to podtyp Collection
- **Row**
- **ADT**

U pole ARRAY poziční přístup ke složkám "autori VARCHAR (30) ARRAY[8]" -> "autori[2]"

Objektové vlastnosti SQL99

- Kompatibilní s existujícími jazyky
- **OID**
- **Hnízděné tabulky**
- Uživatelem definované typy
    - **Abstraktní datové typy** – jsou typem atributu relace
    - **Řádkové typy** – jsou typem relace
    - **Odlišující typy** – musí být FINALL

UDT mohou být organizovány do hierarchií s děděním

Chování UDT je realizováno pomocí procedur, funkcí a (metod u ADT)

Řádkové typy

CREATE ROW TYPE jmeno (deklarace komponent)
Např: CREATE ROW TYPE typadresa (ulice CHAR VARYING (50), město CHARVARYING (20));
Zpřístupnění komponent řádkového typu tečkovou notací – adresa.ulice

Abstraktní datové typy

Umožňují **zapouzdření atributů a operací** (na rozdíl od řádkových typů), hodnoty jejich typů mohou být umístěny do sloupců tabulek.

```
CREATE TYPE typzamestnanec AS (
id_zam INTEGER,
jmeno CHAR (20),
adresa typadresa,
vedouci typzamestnanec,
datum nastupu DATE,
zakladní plat DECIMAL(6,2) )

INSTANTIABLE NOT FINAL,

METHOD odpr_leta( ) RETURNS INTEGER; /*jen signatury*/
METHOD mzda ( ) RETURNS DECIMAL;

CREATE METHOD odpr_leta
FOR typzamestnanec
BEGIN … END ;
```

Instance ADT vznikají:

- Konstruktorem – jmenotypu()
- Operátorem – NEW jmeno(hodnota,…)
- Příkazem – INSERT

Podtypy – dědí atributy i metody svých nadtypů, strukturované typy mohou být podtypem

CREATE TYPE typkulisak UNDER typzamestnanec AS („další atributy a metody“);

Podtabulky

- dědí atributy, IO, triggery z nadtabulky
- Mohou mít další sloupce

CREATE TABLE osoba (
Jmeno CHAR(30),
Vek INTEGER);
CREATE TABLE zamestnanec UNDER osoba (
Mzda FLOAT);

Uživatelsky definované procedury a funkce

= programy vyvolatelné v SQL

- Procedury mají parametry typu IN, OUT, INOUT
- Funkce mají parametry jen typu IN, vracejí hodnotu

Konstrukce podprogramů:

- hlavička i tělo v SQL (1 SQL příkaz nebo BEGIN .. END)
- hlavička v SQL, tělo externě definované

Volání podprogramů

- **procedura: CALL** jmeno_procedury(param1,param2)
- **funkce:** funkce(x,y)

Reference a Dereference

Výhody používání REF: sdílení objektů – nejsou zbytečně kopírována data, změna na jednom místě

Objektová rozšíření SŘBD Oracle

V současné době směřuje trend ve vývoji databázových systémů směrem k objektovým SŘBD, neboť objektové SŘBD umožňují snadněji a přesněji modelovat většinu z různých tříd aplikací. Proto se výrobci relačních SŘBD snaží své produkty rozšířit o některé základní vlastnosti objektových SŘBD, čímž vznikají tzv. *objektově-relační* SŘBD. Zmíněný tred se promítá i do nového standardu **SQL 3**, jehož součástí jsou i abstraktní datové typy, persistentní programové moduly a vhnízděné tabulky. Jedním z objektově-relačních SŘBD je i systém Oracle (od verze 8i), na kterém si nyní ukážeme některá objektová rozšíření relačního SŘBD.

Abstraktní datové typy

Abstraktní datový typ lze nadefinovat příkazem **CREATE TYPE**. Tento uživatelem definovaný ADT lze pak použít všude, kde jsou používány standardní datové typy SQL. Následující příklad ukazuje definici typu *t_adresa* reprezentujícího adresu obyvatele. Ukázána je i definice tabulky, kde jedním z atributů je objekt typu *t_adresa* :

```
CREATE TYPE t_adresa AS OBJECT (
  ulice    VARCHAR2(30),
  cislo    NUMBER(3),
  obec     VARCHAR2(30),
  psc      NUMBER(5)
);
```

Metody objektů

Samozřejmě, že objekty v SQL 3 mohou mít kromě datových prvků i metody. Následující ukázka slouží pro ilustraci definice objektu s několika metodami. Implementace metod je definována v příkazu **CREATE TYPE BODY**.

Řádek v tabulce může být reprezentován i objektem. Tabulku pak nadefinujeme jednoduše takto :
CREATE TABLE osoby **OF** t_osoba;

Pole jako atributy

V praxi se často vyskytují případy, kdy je třeba uložit do jednoho atributu více hodnot. Typickým případem jsou alternativní čísla telefonu, na kterých je dosažitelná určitá osoba. Relační model dat nás nutí vytvořit novou entitu pro telefonní čísla, což se nám může oprávněně zdát poněkud nepřirozené. V takovýchto případech lze s výhodou použít pole proměnné délky - **VARRAY**. Pole **VARRAY** může obsahovat různý počet položek stejného datového typu (tzn. i objekty), který nesmí překročit definovanou velikost pole. Způsob použití polí je ukázán v následujícím příkladě :

```
CREATE TYPE t_tel_seznam AS VARRAY(5) OF VARCHAR2(30);
```

Vhnízděné (vnořené) tabulky

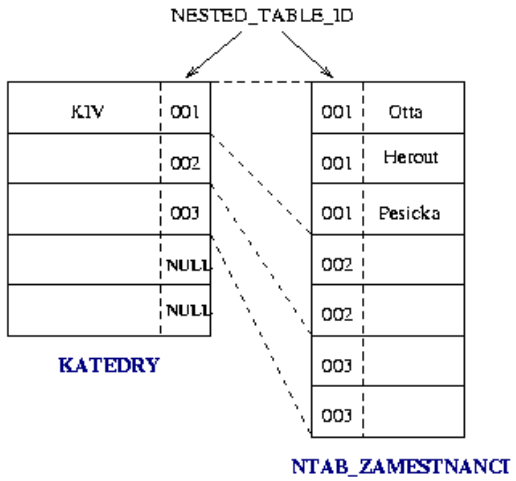
Vnořené tabulky mají oproti polím tu výhodu, že z hlediska aktualizace lze přistupovat k jednotlivým řádkům, jejichž počet není nijak omezen (teoreticky). Nevýhodou však je, že prvky (řádky) v tabulce nemají definované pořadí, jak tomu je u polí. Vnořenou tabulku vytvoříme tak, že nadefinujeme uživatelský datový typ podobně jako u polí :

```
CREATE TYPE t_zamestnanec AS OBJECT (
 jmeno    VARCHAR2(30),
 prijmeni VARCHAR2(30),
 plat     NUMBER(5)
);
CREATE TYPE tab_zamestnanci AS TABLE OF t_zamestnanec;
```

Definujeme-li tabulku, která obsahuje vnořenou tabulku, musíme definovat jméno tabulky, kde budou fyzicky uloženy záznamy z vnořených tabulek. V uvedeném příkladě to je tabulka NTAB_ZAMESTNANCI :

```
CREATE TABLE katedry (
 cislo_kat   NUMBER(5),
 nazev       VARCHAR2(50),
 zamestnanci tab_zamestnanci
) NESTED TABLE zamestnanci STORE AS ntab_zamestnanci;
```

Ve skutečnosti jsou vnořené tabulky řešeny "klasickým způsobem" - vazba mezi mateřskou a vnořenou tabulkou je realizována přes skrytý sloupec NESTED_TABLE_ID, jak ukazuje spodní obrázek :



Reference na objekty

V některých případech nechceme v atributu uchovávat celý objekt, ale pouze referenci (ukazatel, odkaz) na něj. Proto byl v SQL3 zaveden typ *reference*. Uvažme případ, že máme danou databázi pracovišť a předmětů (nábytek apod.) uložených na jednotlivých pracovištích. Vzhledem k centrální správě předmětů by bylo poněkud nevýhodné mít u každého pracoviště vnořenou tabulku předmětů.

9. Objektově relační databáze

Za posledních 25 let je mohutný trend přechodu od strukturovaného k objektově orientovanému programování, a to i v oblasti zpracování dat a databází. Objektově orientované DB se objevili v 90. Letech minulého století a kladou si za cíl urychlit a ulehčit práci s daty. Situace ovšem není jednoduchá, protože relační a objektový přístup je od základu rozdílný. Existuje tak mnoho výhod i mnoho nevýhod pro relační i objektové DB. Na současném trhu existují 3 základní typy:

- Relační DB (Relational Database Management System, RDBMS) – výkonné na tradičních datech. Př. Oracle 7.x, DB2.
- Objektově-relační (Object Relational Database Management t System, ORDBMS) – Poskytuje programátorské pohodlí, rychlý a udržitelný vývoj aplikací. Př. Oracle 8.x, 9.x, 10.x.
- Objektové DB (Object Database Management system, ODBMS) – Výkonné na netradičních datech, slabší v databázových rysech. Př. Jasmine, Gemstone, O2.

Objektově relační databáze

Objektově relační DB se snaží přinést objektové rysy do relačních databází. ORDBMS je specifikována v rozšíření SQL standardu – SQL3. Do této kategorie patří např. Informix, IBM, Oracle a Unisys.

Objektově relační mapování přináší vrstvy mezi OO aplikací a SQL databází. Charakteristiky jsou:

Myšlení v objektech, ev. Cache objektů, zodpovědné za persistenci.

Datový model ORDBMS

Rozšiřují datový model tak, že přidávají objektovost do tabulek. Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu (porušení 1. NF), nazývanou abstraktní datové typy, tzv. ADT:

- ADT je datový typ, který vznikne kombinací základních datových typů
- Podpora dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem je omezená
- Funkce a operace jsou asociované s novými datovými typy, mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu.

Dotazovací jazyk

ORDBMS podporují rozšířenou verzi SQL – SQL3 (SQL 99), důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). ORDBMS je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření. Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod.

Výpočetní model

Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s DB.

Objektové vlastnosti SQL99 (SQL3)

Objektové rozšíření standardizované v SQL99 zahrnuje:

- **Strukturované uživatelské typy** – Oracle od verze 9.i včetně jednoduché dědičnosti. Mohou být organizovány do hierarchie s děděním. Chování uživatelem definovaných typů je realizováno pomocí procedur a funkcí a (metod u ADT). Jedná se o řádkové typy, ADT, odlišující typy.
- **Pole s proměnnou délkou (VARRAY)** – CREATE TYPE typ AS VARRAY(5) OF VARCHAR(15)
- **Hnízděné tabulky** – typ TABLE

- **Typ REF** – odkaz ukazatel – jeho obsahem je OID nějakého záznamu, nelze s ním manipulovat jako s hodnotou, ale jako s odkazem. Zajišťuje objektovou identitu.
  - Výhoda - pro sdílení objektů (nejsou zbytečně kopírována data a změna se provádí na jednom místě)
- **CREATE TYPE TypHerec AS (jmeno CHAR(30), nejlepsiFilm REF (FilmTyp))**
- Dereference př. SELECT Film->titul FROM hrajev WHERE herec->jmeno='Chaplin';

SQL99 je kompatibilní s existujícími jazyky a další vlastnosti jsou:

- **Řádkové typy** (jsou typem relace)
  Vytvoření řádkových typů:

```
CREATE ROW TYPE typadresa (ulice CHAR VARYING(50), mesto CHAR
VARYING(20));
```

  Příklad tvorby tabulky s řádkovým typem:

```
CREATE TABLE FilmovyHerec OF typherec;
CREATE TABLE FilmovyHerec (
jmeno CHAR VARYING(30),
adresa ROW (
ulice CHAR VARYING (50),
mesto CHAR VARYING(20)
)
);
```

  Tvorba dotazů:

```
SELECT FilmovyHerec.jmeno, FilmovyHerec.adresa.ulice
FROM FilmovyHerec WHERE FilmovyHerec.adresa.mesto = 'Plzeň';
```

- Lze definovat **i podtypy** a **podtabulky** CREATE TYPE typSekretarka UNDER typZamestnanec AS(...
- **Abstraktní datové typy** (jsou typem atributu relace) – umožňují zapouzdření atributů a operací (na rozdíl od řádkových typů). Hodnoty jejich typů mohou být umístěny do sloupců tabulek.
  - Př. (Oracle)

```
CREATE TYPE typZamestnanec AS (
c_zam INTEGER,
METHOD mzda() RETURNS DECIMAL);
CREATE METHOD mzda ... FOR typZamestnanec
BEGIN ... END
```

  - Instance vznikají konstruktorem jmenoTypu(), operátorem NEW jmeno hodnota, příkazem INSERT INTO osoby VALUES(...)

  - Funkce a procedury vyjádřeny v SQL/PSM (Persistent stored module), nebo C/C++, Java, ADA.... Jsou svázány s ADT. Metody jsou uložené ve schématu typu definovaného uživatelem. Metody se dědí. Metody i funkce mohou být polymorfní (liší se způsobem výběru). CREATE PROCEDURE zjisti_cenu ...; CALL zjisti_cenu(...);
- **Odlišující typy** (musí být FINAL) – emulace domén – strong typing
- **OID** – záznamy mají/mohou mít OID (v relačních DB mohou být použity jako primární klíče), které zajišťují objektovou identitu. V jiných záznamech atribut typu REF – odkaz ukazatel. Zpřístupnění klauzulemi REF IS SYSTEM GENERATED a REF IS USER GENERATED.

Srovnání DBMS

Srovnání databázových systémů			
Kritérium	**RDBMS**	**ORDBMS**	**ODBMS**
Definovaný standard	SQL2 (ANSI X3H2)	SQL3/4 (in process)	ODMG-V2.0
Podpora pro objektově orientované programování	Špatná; programátoři stráví 25% času kódování mapováním objektového programu do databáze	Omezená hlavně na nové datové typy	Přímá a rozsáhlá
Jednoduchost používání	Strukturám tabulky je jednoduché porozumět; mnoho dostupných nástrojů pro koncové uživatele	Totéž co RDBMS, navíc s nějakými matoucími rozšířeními	OK pro programátory; nějaký SQL přístup pro koncové uživatele
Jednoduchost vývoje	Poskytuje nezávislost dat z aplikace, dobrá pro jednoduché vztahy	Poskytuje nezávislost dat z aplikace, dobrá pro jednoduché vztahy	Objekty jsou přirozenou cestou k modelu; může vyhovět širokým rozsahem typů a vztahů
Rozšiřitelnost a obsah	Žádná	Omezená hlavně na nové datové typy	Může pracovat s libovolnou složitostí; uživatelé mohou psát metody a jakékoliv struktury
Složité datové vztahy	Pro model obtížné	Pro model obtížné	Může pracovat s libovolnou složitostí; uživatelé mohou psát metody a jakékoliv struktury
Výkon versus spolupracovatelnost	Úroveň bezpečnosti se mění s dodavatelem, je třeba vzájemně porovnat; dosáhnutí obojího vyžaduje rozsáhlé testování	Úroveň bezpečnosti se mění s dodavatelem, je třeba vzájemně porovnat; dosáhnutí obojího vyžaduje rozsáhlé testování	Úroveň bezpečnosti se mění s dodavatelem; většina ODBMSs dovoluje programátorům rozšířit funkčnost DBMS definováním nových tříd
Distribuce, replikace, a spojené databáze	Rozsáhlá	Rozsáhlá	Podle dodavatele; pár jich poskytuje rozsáhlou podporu
Vyspělost produktu	Velmi vyspělé	Nezralé; rozšíření jsou nová, stále se definují a jsou relativně neprozkoušená	Relativně vyspělé
Podpora pro lidi a univerzálnost SQL	Široká podpora nástrojů a trénovaných vývojářů	Může využívat výhod nástrojů RDBMS a vývojářů	Vybaveno SQL, ale určeno pro objektově orientované programování.

Softwarové ekosystémy	Poskytováno hlavními RDBMS společnostmi	Poskytováno hlavními RDBMS společnostmi	ODBMS výrobci začínají emulovat RDBMS výrobce, ale žádný nenabízí velký obchod jiným ISV
Životaschopnost výrobce	Očekávaná pro hlavní zaběhnuté RDBMS výrobce	Očekávaná pro hlavní RDBMS výrobce; UniSQL bojuje	Menší než se čekalo; stále se očekává zmenšování
Zdroj: International Data Corporation, 1997			

Shrnutí

Relační model je jednoduchý a elegantní, ale je naprosto rozdílný od objektového modelu. Relační databáze nejsou navrhovány pro ukládání objektů a naprogramování rozhraní pro ukládání objektů v databázi je velmi složité. Relační databázové systémy jsou dobré pro řízení velkého množství dat, vyhledávání dat, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. Jazyky jako SQL umožňují tabulky propojit za běhu, aby vyjádřily vztah mezi daty.

Naproti tomu objektově orientovaný model je založen na objektech, což jsou struktury, které kombinují daný kód a data. Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty "v databázi"; tj. místo věčného přeskakování mezi jazykem aplikace (např. C) a dotazovacím jazykem (např. SQL) může programátor jednoduše používat objektový jazyk k vytváření a přístupu k metodám. Krátce řečeno, ODBMS jsou výborné pro manipulaci s daty. Pokud navíc opomeneme programátorskou stránku, dá se říct, že některé typy dotazů jsou efektivnější než v RDBMS díky dědičnosti a referencím.

10. Vlastnosti objektově orientovaného datového modelu

Objektový datový model je v souladu s viděním světa (entita – objekt), definice složitých objektů a jejich manipulace.

- Operace na složitých objektech
- Rekurzivní struktury
- Abstraktní datové typy
- Rozhraní k OO jazyku
- Složité transakce

Návrh standardu ODMG-93

- Dotazovací Object Query Language – OQL
- Rozhraní k OO programovacím jazykům – k Java – JDO = Java Data Objects

Koncepty ODMG-93

- Třída – šablona pro instance (objekty)
- Instance = objekt vytvořený dle vzoru – třídy, mohou sdílet atributy a metody
- Atributy – primitivní typ, abstraktní datový typ, odkaz
- Metody
- OID = jednoznačný identifikátor objektu
- V OO databázích jsou vztahy reprezentovány pomocí OIDs, což zlepšuje přístup k datům.
- Zapouzdření – data jsou zabalena spolu s metodami (jednotkou zapouzdření je objekt)
- Hierarchie tříd, dědění – dědění je proces znamenající pro podtřídu osvojení všech atributuů a meto z nadtřídy

Nevýhody OO

- Chabý výkon. Ve srovnání s relačními jsou optimalizátory pro OO DB velmi složité.
- Problémy se škálovatelností, neschopnost podporovat rozsáhlé systémy.

The Object-Oriented Data Model

1. A data model is a logic organization of the real world objects (entities), constraints on them, and the relationships among objects. A DB language is a concrete syntax for a data model. A DB system implements a data model.
2. A core object-oriented data model consists of the following basic object-oriented concepts:
   (1) **object and object identifier**: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).
   (2) **attributes and methods**: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.

   [An attribute is an instance variable, whose domain may be any class: user-defined or primitive. A class composition hierarchy (aggregation relationship) is orthogonal to the concept of a class hierarchy. The link in a class composition hierarchy may form cycles.]
   (3) **class**: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.
   (4) **Class hierarchy and inheritance**: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

Objektová databáze

Pro objektové databáze neexistuje žádný *oficiální* standard. Standardem je de facto kniha Morgana Kaufmana *The Object Database Standard: ODMG-V2.0*. Důraz ODBMS je na přímou korespondenci mezi následujícími:

- Objekty a objektové vztahy v aplikaci napsané v OO jazycích
- jejich uchovávání v databázi.

Datový model

Objektové databáze využívají datového modelu, který má objektově orientované aspekty jako třídy s atributy a metodami a integritními omezeními; poskytují objektové identifikátory (OID) pro každou trvalou instanci třídy; podporují zapouzdření (encapsulation); násobnou dědičnost (multiple inheritance) a podporují abstraktní datové typy.

Objektové databáze kombinují prvky objektově orientovaného programování s databázovými schopnostmi. Rozšiřují funkčnost objektových programovacích jazyků (C++, Smalltalk, Java) a poskytují plnou schopnost programování databáze. Datový model aplikace a datový model databáze se ve výsledku hodně shodují a výsledný kód se dá mnohem efektivněji udržovat.

Dotazovací jazyk

Objektově orientovaný jazyk (C++, Java, Smalltalk) je jazykem jak pro aplikaci, tak i pro databázi. Poskytuje těsný vztah mezi objektem aplikace a uloženým objektem. Názorně je to vidět v definici a manipulaci s daty a v dotazech.

Výpočetní model

V RDBMS rozumíme dotazovacím jazykem vytváření, přístup a aktualizaci objektů, ale v ODBMS, ačkoliv je to stále možné, je toto prováděno přímo pomocí objektového jazyka (C++, Java, Smalltalk) využitím jeho vlastní syntaxe. Navíc každý objekt v systému automaticky obdrží identifikátor (OID), který je jednoznačný a neměnný během existence objektu. Objekt může mít buď vlastni OID, nebo může ukazovat na jiný objekt.

Další Vlastnosti Objektově orientovaných databází

HDM, SDM a RDM = záznamově orientované modely. V 90. letech se začaly objevovat první objektově orientované SŘBD (OOSŘBD), které umožňují pracovat s datovou abstrakcí na úrovni objektů.

- Výhody
  - snadnější aktualizace dat
  - přímé vyjádření složitých objektů modelované reality v databázi (odpadají mezikroky převodu objektů do normalizovaných tabulek relační databáze. Stejně tak je zjednodušen i opačný krok načítání objektů z databáze do aplikace)
  - součástí uložených objektů je také jejich chování (metody atd.)
- Nevýhoda
  - Vývoj a návrh objektově orientovaných modelů v jejich univerzálnosti a komplexnosti je velmi složitý proces ⇒ menší uplatnění tohoto typu modelu v reálných aplikacích.

Objektově orientované programování (OOP)

Koncepce objektově orientovaných databází vychází z principů používaných v OOP - základem je objekt (prvek typu třída). Data se nazývají atributy, funkce metody (služby). Metoda je aktivována příchodem zprávy do jiného objektu.

Hlavní vlastnosti OOP: zapouzdření dat, dědičnost, polymorfismus

Objektově orientované modelování dat

Postup objektově orientovaného modelování dat lze stručně shrnout do následujících bodů:

- Vyhledají se objekty jako nositelé aktivit (např. metodou gramatické inspekce: podstatná jména představují objekty nebo třídy, přídavná jména představují hodnoty atributů a slovesa představují většinou aktivity).
- Identifikují se třídy zobecňování objektů se stejnými atributy (+ zkoumá se možnost uspořádat třídy hierarchicky podle dědičnosti) a vztahy.
- Stanoví se integritní omezení na hodnoty jednotlivých atributů a případně na typy atributů.
- Vyhotoví se seznam nabízených a požadovaných služeb pro všechny metody (přehled toku zpráv)
- Implementují se metody (teprve po jednoznačném vymezení všech funkcí systému)

Hlavní rozdíly mezi RDM a ODM:

	RDM	ODM
1	• relační tabulka • jeden záznam • manipulace s atributy záznamu	• množina objektů • jeden objekt • přenos a zpracování zpráv
2	normalizace relací (dekompozice) vede k rozptýlení popisu vlastností složitého objektu do mnoha tabulek	spojuje jednotlivé složky pomocí odkazů
3	záznamy relací jsou omezeny na jednoduché datové typy	složité strukturované datové entity - objekty, které lépe vystihují prvky reálného světa
4	manipulace s hodnotami atributů záznamů	operace posílání zpráv poskytuje větší možnosti
5	každá tabulka musí mít identifikační klíč (ten nemusí odrážet požadavky zadání)	zabezpečuje identifikaci objektů vlastními systémovými prostředky (OID)
6	při zpracování dotazů dochází často k získávání údajů z několika tabulek ⇒ narůstá čas potřebný k vyhodnocení dotazu	ke spojování množin dochází v daleko menší míře; dotazovací konstrukce lze díky polymorfismu aplikovat i na množiny obsahující různé typy objektů

Pozn.: RDM za určitých podmínek představuje zvláštní případ ODM.

Produkty: Objectivity, Versant, POET, CACHÉ, db4o, Ozone, GOODS, XL2, ZODB, Prevayler

Bariéry rozšíření objektových databázových systémů:

- neochota vývojářů a jejich klientů k přechodu od tradičního relačního přístupu k objektovému
- nedostatek kvalifikovaných vývojářů
- nízká podpora standardů
- nízká podpora dotazovacích jazyků
- neexistence mechanismu pro řízení přístupu k datům

Původní text (dle mého "mimo mísu")

Objektový (konceptuální model)

- představuje statický model reality (businessu)
- popisuje z čeho je realita složena a jaké jsou základní (podstatné/statické) složky (objekty) a vazby mezi nimi.
- Akce (metody) a algoritmy, vázané k objektům v jejich životních cyklech, zde mají význam též statický (jsou podřízeny statickému - pohledu).
- K popisu lze využít specifický diagram – Diagram tříd (Class Diagram - základní diagram jazyka UML).

- Model (entitních, business, analytických) objektů (podstata struktury reality) sleduje základní stavební kameny, z nichž se realita (problémová doména) skládá.

Diagram tříd

- Původní strukturální přístup k analýze IS spočíval v rozdělení systému na funkční a datovou část (např. DFD - Data Flow diagramy a ER - Entity Relationship model).
- Přínosem byla funkční hierarchická dekompozice – psaní programu shora dolů a datové konceptuální modelování.
- Rostoucí složitost systémů (magická hranice 1000 entit a 10000 funkcí) znemožňuje soudržnost datové a funkční vrstvy.
- Objektový přístup čelí složitosti systému tím, že třída (objekt) jako nositel (funkční) odpovědnosti (dovednosti), plně odpovídá za svá data.
- Objekt má svou identitu, vlastnosti, chování a odpovědnost. Síla odpovědnosti spočívá v tom, že je nedělitelná – žádný jiný objekt nemůže odpovědnost sdílet – dělit se o ni, plést se do ní.
- Modelování tříd a objektů je klíčová aktivita objektově orientovaného vývoje.

Třída, Objekt

- **Třída** - popis množiny objektů sdílejících stejné vlastnosti (atributy), chování (operace/metody) a vztahy.
- **Objekt** - instance třídy (chybně se pojem třída a objekt volně zaměňují).

Definice – J. Rumbauh: objekt je diskrétní entita s jasně definovaným rozhraním, které zapouzdřuje stav a chování.

- Třídu si můžeme představit jako razítko, objekty jsou pak otisky tohoto razítka, které vidíme na papíře.
- Při návrhu třídy neuvažujeme o konkrétním naplnění atributů, pouze určíme jejich název a typ. Teprve při vzniku instance objektu se atributům přiřadí skutečné hodnoty.
- Třída je jednoznačně určena svým názvem (v příslušném názvovém prostoru – balíčku). Pro třídu je možno definovat vlastnosti - atributy (Attribute) a chování - operace (Operation).
- Hledání tříd, jejich atributů a kompetencí - vyberme z reality objekty, kandidáty pro zobecnění na třídy a prověřme jejich vhodnosti:
  - Potenciální třída je smysluplná, pokud je nezbytná pro funkci systému.
  - Potenciální třída je dostatečně stabilní a invariantní vůči vnějším změnám např. technologie, legislativy apod.

Hledání tříd na základě analýzy podstatných jmen a sloves.

Analyzujeme jazyk problémové domény, např. text sebraných požadavků. Podstatná jména a jejich spojení mohou označovat třídy nebo atributy. Slovesa mohou označovat odpovědnosti, chování tříd.

Pozor na skryté, utajené třídy, které nejsou v textu uvedeny.

Klasifikace, zařazení do třídy přenáší význam na formální objekt, je nositelem sémantiky. Klasifikace je jedním z nejdůležitějších způsobů, jímž lidé uspořádávají, vnímají, chápou okolní svět. Existuje mnoho způsobů jak klasifikovat okolní svět, proto je analýza tak náročná.

Atributy

Definice atributů

Atribut určuje vlastnosti objektu, je nositelem informace o objektu.

Atributy popisují hodnoty (stavy) udržované v jednotlivých objektech.

Objekty jsou vymezeny (popsány) množinou atributů.

Atributy popisují vlastnosti objektů, které potřebujeme k dosažení daného cíle. Reálný objekt ve své nekonečné složitosti nelze vymezit omezenou množinou atributů. Základní problém analýzy IS - výběr rozumného množství relevantních atributů.

S atributy mohou manipulovat výhradně služby daného objektu.

Klíčovou otázkou je zodpovědnost určitého objektu za uchování informací. Hledání atributů je řízeno otázkami: Jak je objekt popsán v kontextu zodpovědností daného systému. V jakých stavech se může objekt v průběhu svého životního cyklu nacházet

Specifikace atributů

Atribut je definován: jménem, typem (formátem) viditelností (veřejný – public, soukromý – private a chráněný – protected).

Každý atribut pečlivě pojmenujte. Volte názvy, které jsou běžné v aplikační oblasti a jsou rozumné délky a pevné struktury. Ke každému atributu připojte vysvětlující text.

Hledejte omezující podmínky pro hodnoty atributů. Omezení se vztahují na: formáty, rozsah, výčet přípustných hodnot, přesnost implicitní hodnoty, požadavek na nastavení výchozích hodnoty atributu prevalidační a postvalidační podmínky – podmínky, které musí být splněny před a po změně hodnoty atributu, za jakých podmínek je povolen přístup k atributu (např. v závislosti na hodnotách ostatních atributů) závislost atributů, viz datové modelování, jak změna jednoho atributu ovlivňuje hodnoty jiných – závislých atributů, viz normalizace relačního modelu.

Identita objektu

Objekt je vedle svého stavu a chování jednoznačně určen, je jedinečný, má identitu, atribut, který jej jednoznačně identifikuje mezi všemi ostatními objekty dané třídy, má své ID. Atribut zajišťující identitu, se v datovém modelování nazývá primární klíč.

Čtenář (číslo čtenáře, jméno, adresa, kontakt) Kniha (isbn, autor, titul) Exemplář (číslo exemp, datum nákupu) Exempláře knihy se liší inventárním číslem a sledujeme u nich datum nákupu.

Problematika volby primárního klíče,

Umístění atributů

Ve třídách, které jsou vázány dědičností (generalizace) umístíme atribut do co nejvyšší třídy, ve které atribut platí pro všechny její generické podtřídy (specializace).

Hledání tříd, doporučení

Každá třída by měla mít 3-5 klíčových odpovědností První extrém - není dobré, když existuje velké množství malých tříd Druhý extrém – není dobré mít velké třídy Nezavádějte „funktiody“ – pro jednotlivé funkce systému nezavádějte třídy Vyhýbejte se stromům dědičnosti s mnoha úrovněmi

Vazby – relace mezi třídami

Vazba asociace (Association)

Vazba asociace mezi třídami je vyjádřením abstraktního vztahu mezi objekty (instancemi tříd). Asociace říká, že objekty mají mezi sebou přímý vztah, že o sobě ví.

Zaměstnanec pracuje v daném oddělení, mohu se ptát: V jakém oddělení pracuje zaměstnanec, mohu získat seznam všech zaměstnanců v oddělení. Vazba je nositelem významu – sémantiky, odpovídá – mapuje požadavky kladené na systém. Je trvalejšího charakteru.

Asociace je společný typ vazby pro:

- agregaci, vyjadřující vztah mezi celkem a částí
- prostou asociaci, vyjadřující prostou objektovou referenci.

Vazba asociace je specifikována řadou vlastností, z nichž některé jsou vázány přímo k vazbě asociace (například název), ostatní k zakončením vazby (například role). Podrobné určení vlastností až v okamžiku návrhu – specifikace návrhových tříd a jejich vazeb má „implementační“ důsledky.

Vazbu asociace lze zavézt jako orientovanou (Navigability), přičemž neorientovaná vazba je považována za obousměrnou (dva jednosměrné vztahy).

Třídy v asociaci mohou vůči sobě vystupovat v rolích (Role) (Objednávka – Zaměstnanec, Zaměstnanec vystupuje ve vztahu k objednávce v roli Prodejce). Každá strana asociace má své jméno – roli. Role popisuje vlastnost, funkci třídy „viděné“ z druhé strany.

V asociaci lze určit násobnost vazby, (kardinalitu) multiplicitu, která vyjadřuje počet možných vazeb objektů tříd v asociaci (0, 0..1, 0..*, 1, 1..*, *, M..N, …).

Násobnost vazby definuje, kolik může k jednomu objektu, tj. k jedné instanci třídy A na jedné straně vztahu existovat minimálně (parcialita) a maximálně (kardinalita) objektů ze třídy B na druhé straně vztahu a obráceně.

- Standardně je vazba asociace implementována zavedením atributu třídy - role pro zachycení objektové reference (množiny referencí pro parcialitu 0..*) na objekty druhé třídy. Implementace vazby – objekt si sebou nese reference na asociované objekty.
- S vazbami je třeba šetřit.
- Na rozdíl implementace v relačním datovém modelu se jedná o explicitní vyjádření vazby. V relačním modelu se pro vyjádření vazby používají tzv. cizí klíče – implicitní implementace vazby.

Další vlastnosti vazeb související s implementací vazby (mimo UML, CASE):

Každý konec asociace, vazby se nazývá role. Pro roli můžeme definovat řadu vlastností.

- Asociativní třída (Association Class) je vazba asociace, která je rozšířena přiřazením třídy pro zachycení informací nutných pro úplnou specifikaci této vazby.
- Asociativní třída se používá například v případě oboustranně násobné vazby N:M. Asociativní třída nemá vlastní identitu, identitu přejímá od „asociovaných“ tříd
- Vztah mezi třemi a více prvky popisují vícenásobné asociace (N-ary Association). Pro vyjádření vícenásobné asociace se používá element modelu asociativní třída.

Vazba agregace (Aggregation)

Agregace je vyjádřením abstrakce vztahu mezi objekty (instancemi tříd), který odpovídá vztahu celku a části .

Agregace je speciálním případem asociace. Jazyk UML rozlišuje mezi dvěma typy agregací:

- prostou agregací (Simple Aggregation)
- kompozicí (Composition).

Vazba kompozice je silnější než vazba prosté agregace, jedná o vlastnictví částí celkem. Pokud je celek existenčně (tj. svou logikou) závislý na částech a nemůže bez nich fungovat, jedná se o kompozici – pokud se bez nich obejde, jedná se o agregaci.

Agregace: Profesoři - Katedra - zruším katedru, profesoři zůstanou, mohou fungovat bez katedry

Kompozice: Fakulta - katedra, zruším fakultu, katedra je bezvýznamná

Vazba generalizace (Generalization)

Vazba generalizace je vyjádřením vztahu mezi obecným elementem (Parent) a specifickým elementem (Child), který je konzistentní s obecným elementem a přidává k jeho definici další informace, je tedy bližší specifikací (specializací) obecného elementu.

Vazba generalizace mezi třídami je vyjádřením vlastnosti dědičnosti, jedné ze základních vlastností objektově orientovaného přístupu.

Jazyk UML povoluje vyjádřit vícenásobnou dědičnost zavedením více vazeb generalizace.

Vazba závislosti (Dependency)

- umožňuje znázornit jistou závislost mezi elementy modelu.
- je určena svým názvem a obvykle se používá s určitým stereotypem, který blíže specifikuje formu závislosti, zavádí její typ.
- je znázorněna orientovanou přerušovanou čárou, kde orientace je vyjádřena šipkou ve směru závislosti.

Závislost obvykle vzniká pouze dočasně pro potřeby poskytnutí služby klientskému objektu a poté tato vazba zaniká (implementační rozdíl od asociace).

Změna jednoho (nezávislého) elementu ovlivní druhý (závislý) element.

Chování objektů, předávání zpráv

Objekt poskytuje služby prostřednictvím operací (metod).

Rozhraní objektu je množina operací, které nabízí objekt k použití pro jiné objekty (nebo externí agenty). Objekty jsou známy jiným objektům pouze prostřednictvím svého rozhraní. Objekt má i své vnitřní – interní operace, které slouží k udržení vnitřní konzistence (stavu) objektu.

Objekt může poskytovat více rozhraní – mít více rolí, podle kontextu ve kterém se nachází. Stejně jako v reálném světě člověk vystupuje v různých rolích podle toho, v jakém kontextu se právě nachází (v zaměstnání se nachází v roli pracovníka, doma manželem, v automobilu řidičem)

Objekty tak odbourávají nevýhodu strukturálních metod, spočívající ve vzájemné izolaci funkční a datové vrstvy.

Objekty spolupracují proto, aby společně mohly vykonávat funkce poskytované systémem, viz modely spolupráce. Operace určující chování jsou definovány a rozpoznávány svojí signaturou – názvem, seznamem parametrů a návratových hodnot. Objekt přijme zprávu a vykoná operaci, jejíž signatura je shodná se signaturou zprávy.

Pro lepší pochopení myšlenky OODB: http://www.linuxexpres.cz/business/objektove-databaze

Objektové databáze

Stejně tak jako lidé postoupili od strukturálního programování k objektovému, tak si řekli, že by nebylo od věci neukládat data do tabulek a relací, ale do objektů tak, jak s nimi pracujeme přímo v programu. Bylo by přeci velice pěkné, když bych mohl objekt tak, jak ho mám, prostě uložit do databáze a o nic víc se nemuset starat.

Nemusel bych přemýšlet nad strukturami tabulek (tak aby dodržovaly „dobré mravy“ dané normami) a tvořit ruční INSERTy a SELECTy, jen bych databázovému stroji přes nějaké API řekl, načti mi uživatele s číslem 451, a dostal bych ho se všemi atributy naplněnými.

A přesně takto objektové databáze fungují. Místo tabulek jsou zde uloženy přímo objekty, včetně svých vlastností, a místo řádků se ukládají samotné instance objektů. Každý takto vložený objekt je jednoznačně identifikován svým OID, které na logické úrovni odpovídá ukazateli do virtuální paměti počítače a stejně tak se chová (při přesunu v paměti se změní i OID). Není tedy potřeba vytvářet primární klíče na objektech ani normalizovat databázi.

Objektové databáze také nabízejí využití možností vícenásobné dědičnosti, zapouzdření a polymorfizmu. Navíc vlastnosti (datové hodnoty) objektů nemusí být jen primitivního typu, ale mohou být dále strukturované jako například objekt (pomocí reference), množina nebo seznam.

Pojem zapouzdření znamená, že každý objekt obsahuje nejen datové hodnoty (vlastnosti), ale i funkce, které definují, jak je možné s těmito vlastnostmi zacházet.

Polymorfizmus umožňuje objektům zastupovat své potomky (ve smyslu dědičnosti) při volání metod. Program nemusí znát přesný typ objektu, který volá, ale ten se zjistí až za běhu a zavolá se metoda na správné třídě.

Pro vytváření nových tříd v databázi je definován nový speciální jazyk ODL (Object Definition Language). Nicméně v dnešní době se využívá vlastností pokročilých programovacích jazyků, jako je reflexe. Například v db4o stačí zavolat metodu set na objektu databáze, předat jí jako parametr obyčejný objekt a zbytek už si zařídí knihovna sama.

```
void storePilot (string pilotName, int pilotsPoints)
{
  Pilot pilot1 = new Pilot(pilotName, pilotsPoints);
  db.Set(pilot1);
}
```

Pro načítání dat z objektové databáze existuje jazyk OQL (Object Query Language). Jak je vidět už podle názvu, jeho syntaxe je velice blízká SQL. V následujícím příkladu si povšimněte, že odpadla potřeba spojovat tabulky, protože vše je dostupné přes objektové vazby:

```
SELECT o.customer_id.get_name(), o.room_id.id
FROM orders o
WHERE o.check_day(o.room_id.id, datefrom, dateto) = 1;
```

Další velice zajímavou možností je vytvořit dotaz pomocí QBE (Query By Example). Princip tohoto přístupu spočívá v tom, že databázi předáme částečně naplněný objekt a ta nám ho dohledá ve svém zdroji a doplní ostatní vlastnosti. Přesněji řečeno vrátí nějaký kontejner naplněný objekty, které původnímu objektu odpovídají. Uvedu zde jeden ilustrační příklad:

```
Pilot retrievePilotByName (string pilotName)
{
  Pilot proto = new Pilot("Michael Schumacher", 0);
  IObjectSet result = db.Get(proto);
  if (result.HasNext())
    return (Pilot)result.Next();
  else
    return null;
}
```

11. „Vnější“ programování (přes rozhraní/knihovny) – rozhraní ODBC, JDBC, rozhraní podporující objektově-relační mapování (Java Hibernate)

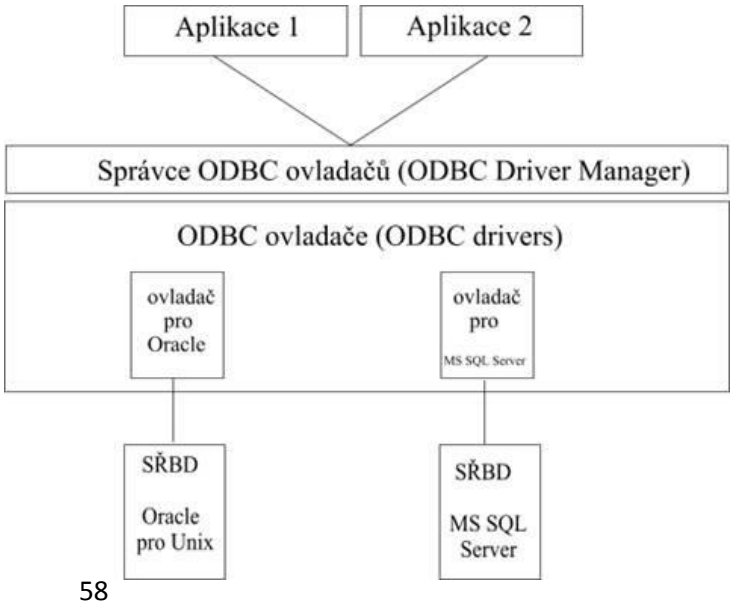
ODBC (Open DataBase Connectivity)

Open DataBase Connectivity. Je standardizované softwarové API pro přístup k databázovým systémům (DBMS). Snahou ODBC je poskytovat přístup nezávislý na programovacím jazyku, operačním systému a databázovém systému. Je to čistě C-čkové API, které nemá žádný objektový základ.
Navrženo Microsoftem, proto primárně přístupné pouze přes C/C++. Založeno na specifikaci X/Open a ISO: SQL Call Level Interface (SQL/CLI)

- Specifické API pro databáze
- Nezávislé na databázi a jazyce
- Databázově závislé ovladače
- Správce ovladačů – **Driver Manager**
- Založeno na SQL Call Level Interface (SQL/CLI)

Model struktury ODBC se dá znázornit pomocí čtyř vrstev:

1. **Aplikace** - V první nejvrchnější vrstvě se nachází samotná aplikace. Ta v případě, že potřebuje data, provede volání ODBC funkcí (ve formě SQL dotazu).
2. **Správce ODBC ovladačů** - Druhou vrstvou je tzv. "Správce ODBC ovladačů" (ODBC Driver Manager). Úkolem správce ovladačů je zajistit propojení mezi aplikací a příslušným ODBC ovladačem (ODBC ovladače tvoří třetí vrstvu modelu, podrobněji viz dále). Jakmile aplikace potřebuje data, správce ovladačů vyhledá a nahraje příslušný ovladač. (ve formě DLL knihovny). Správce ovladačů také zjistí, jaké konkrétní funkce jsou podporovány jednotlivými ovladači, a uschová si jejich adresy v paměti do tabulky. V případě, že aplikace volá konkrétní funkci, správce souborů zjistí, ke kterému ovladači funkce patří a zavolá ji. Tímto způsobem může být prováděn souběžný přístup k více ovladačům, což se hodí v případě programování aplikací přistupujících souběžně k několika zdrojům dat.
3. **ODBC ovladače** - Třetí vrstvou zde již zmíněnou vrstvou jsou ODBC ovladače. Ty provedou zpracování volané ODBC funkce, přeložení požadavku do SQL pro příslušný SŘBD (DBMS) a jeho následné poslání.
4. **SŘBD** - Poslední vrstvou je SŘBD, který provede zpracování operace požadované ODBC ovladačem a výsledky této operací mu vrátí.



Typy ovladačů – driverů

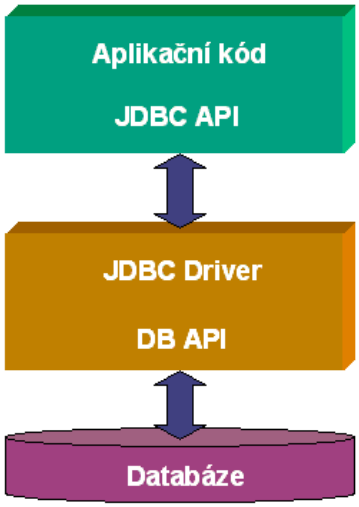
- **Ovladače založené na souborech**
  - přímý přístup k datům (ovladač = zdroj dat)
  - analýza a interpretace dotazů
- **Ovladače založené na SŘBD**
  - Dotazy se předávají ke zpracování SŘBD
  - Transformace ODBC SQL na konkrétní dialekt SQL

Postup

- Připojení k datovému zdroji (SŘBD)
- Inicializace
- Vytvoření a provedení dotazu
- Získání výsledku
- Ukončení transakce
- Odpojení od datového zdroje

JDBC (Java DataBase Connectivity)

- Jeho API poskytuje základní rozhraní pro unifikovaný přístup k databázím, aplikační programátor je tak odstíněn od specifického API databáze a může se naučit pouze jednotné rozhraní JDBC
- Použití i mimo databáze – data ve formě tabulek (CSV, XLS)
- Ovladače jsou k dispozici pro většinu databázových systémů
- Inspirováno rozhraním ODBC:
  - Objektové rozhraní
  - Možnost spolupráce s ODBC



JDBC ovladač

- zprostředkovává komunikaci aplikace s konkrétním typem databáze
- implementován obvykle výrobcem databáze
- dotazovací jazyk – SQL
  - předá se databázi
  - ovladač vyhodnotí přímo
- reprezentován specifickou třídou
  - sun.jdbc.odbc.JdbcOdbcDriver
  - com.mysql.jdbc.Driver

Typy JDBC ovladačů

Typ 1:

- využívá ODBC (pres JDBC-ODBC bridge)
- Obtížně konfigurovatelné

Typ 2:

- komunikace s nativním ovladačem nainstalovaným na počítači

Typ 3:

- komunikuje s centrálním serverem (Network Server) sítovým protokolem
- pro rozsáhlé heterogenní systémy, velmi efektivní i díky poolingu připojení

Typ 4:

- založen čistě na jazyce Java
- přímý přístup do databáze

Java Hibernate

Hibernate je framework napsaný v jazyce Java, který umožňuje tzv. ORM – Objektově-Relační mapování. Usnadňuje řešení otázky zachování dat z objektů i po ukončení běhu aplikace. Provádí podobné věci jako např. JPA – Java Persistence API.

Hibernate je nástroj poskytující techniku pro objektově-relační mapování tříd jazyka Java na tabulky relační databáze (ORM). Krom toho, Hibernate poskytuje nástroje, jak s těmito daty manipulovat včetně objektově orientovaného jazyka (HQL), který na základě objektové notace dokáže uložená data z databáze vybírat v podobě objektů.

Hibernate je konfigurován pomocí nastavení uvedených v `hibernate.properties` a pomocí XML souborů reprezentujících mapování tříd na tabulky relační databáze. Hibernate poté komunikuje přímo s databází pomocí JDBC spojení. (tohle byl případ mapování pomocí souboru viz. níže)

Mapování

A) Mapovací soubory

Jedná se soubory ve formátu XML s příponou „.hbm.xml“ (tedy např. Zajezd.hbm.xml), kdy pro jednu třídu máme jeden soubor, obvykle umístěný ve stejném balíčku jako samotná třída

B) Anotace

```
@Entity(access = AccessType.FIELD)
@Table (name="ZAJEZD")
public class Zajezd {

   @Id (generate = GeneratorType.AUTO)
   private Long id;
```

Co dělá hibernate

Hibernate poskytuje způsob, pomocí nějž je možné zachovat stav objektů mezi dvěma spuštěními aplikacemi. Říkáme tedy, že udržuje data persistentní. Dosahuje toho pomocí ORM, což znamená, že mapuje Javovské objekty na entity v relační databázi. K tomu používá tzv. mapovací soubory, ve kterých je popsáno, jakým způsobem se mají data z objektu transformovat do databáze a naopak, jakým způsobem se z databázových tabulek mají vytvořit objekty. Druhý způsob, jak mapovat objekty, je použít anotace místo mapovacích souborů. V Hibernate tedy pracujete se svými normálními business objekty, pouze pro každý atribut přidáte get/set metody a metody hashCode() a equals(). Nutno podotknout, že nelze použít EJB(viz.Java Bean), ale pouze tzv. POJO(Plain Old Java Object). Poté, co máte objekty uložené v databázi se na ně můžete dotazovat jazykem HQL (Hibernate Query Language), který je odvozen z SQL a je mu tedy velice podobný.

Objektovou struktura mého programu předhodím Hibernatu a základě tohoto objektového modelu (a označením objektů které tam požaduju) si Hibernate vytvoří vlastní schéma, aby věděl, kde jsou data uložena.

Výhody používání Hibernate

Hibernate, framework pro perzistentní vrstvu, usnadňuje programátorovi práci tím, že nemusí transformovat objekty do relací ručně, ale přenechá to perzistentní vrstvě. Zároveň jsou tím odstíněna specifika jednotlivých databází – programátor používá API Hibernate.

S relačními databázemi přichází i potřeba podpory **objektově relačního mapování**, což je proces překladu objektů na tabulkovou reprezentaci a překlad vazeb a vztahů mezi těmito objekty do dodatečných tabulek

12. Distribuované databáze – koncepce distribuovaného databázového systému, replikace a fragmentace dat, distribuovaná správa transakcí

Distribuovaná DB je množina databází, která je uložena na několika počítačích. Uživateli se může jevit jako jedna velká databáze. V DB neexistuje žádný centrální uzel nebo proces odpovědný za vrcholové řízení funkcí celého systému. Výrazně to zvyšuje odolnost systému proti výpadkům jeho částí. Data i funkce rozděleny mezi více počítačů

DDBMS – kolekce DBMS (SŘBD) propojených sítí – logicky tvoří jeden celek

- Lokální data jsou řízena lokálním DBMS
- Integrace dat bez nežádoucí centralizace
- Návrhy:
    - Zdola-nahoru – integrace existujících systémů
    - Shora-dolů – začíná se na „zelené louce“, návrh datového modelu, návrh distribuce dat

Výhody DDBMS

- Reflektuje organizační strukturu
- Data mohou být uložena blízko místa nečastějšího používání
- Oproti centrálnímu řešení zvyšuje výkon
    - Práce s lokálními daty je rychlá (lokální zpracování)
    - Skrytý paralelismus při zpracování SQL
- Zvyšuje spolehlivost – výpadek jednoho uzlu nezaství celý systém
- Umožňuje integraci existujících systémů
- Agregace informaci – z více bází dat lze získat informace nového typu

Nevýhody DDBMS

- Složitost DDBMS
- Bezpečnost – nutno zabezpečit více uzlů i síťové přenosy
- Náročná kontrola integrity dat – jen v rámci uzlu
- Neexistence standardů
- Složitý návrh, obtížní přechod z normální DB na Distribuovanou DB –neexistují konverzní nástroje

Klasifikace DDBMS

A) **Homogenní DDBMS –** v každém uzlu je stejný DBMS
B) **Heterogenní DDBMS –** v různých uzlech různé DBMS (různí výrobci, různé datové modely)

Klasifikace DDBMS

Dle stupně autonomie:

A) Plně autonomní (izolované, neví o sobě)
B) Částečně autonomní = federativní, lokální data řízena plně lokálně, možnost zpřístupnit vybraná data pro distribuované zpracování, možnost přistupovat na data z jiných vzdálených zdrojů
C) Těsná integrace – pro uživatele vypadá jako jeden DBMS, vše se zpracovává globálně

Požadavky na DDBMS

- Transparentnost – z pohledu uživatele jako by pracoval na lokální databázi, je odstíněn od definic dat, jejich struktur..
    - Logická – logická struktura databáze
    - Fyzická - konkrétní způsob uložení dat
    - Síťová – uživatel neví o síti
    - Replikační – neobtěžovat uživatele tím že pracuje s kopií
    - Fragmentační – dotaz směřován na celou tabulku ale vykonán na jejím fragmentu
- Autonomie
- Heterogennost
- Výkon

Problémy co DDBMS přináší

1. Distribuované SQL
2. Distribuované transakce
3. Podpora distribuování dat
    a. FRAGMENTACE DAT
    b. REPLIKACE DAT
4. Zotavení z nových druhů chyb – výpadek sítě, výpadek uzlu

Zpracování SQL

- SQL dotazy do lokálních tabulek jsou zpracovány lokálním DBMS (jako by db nebyl distribuovaná)
- SQL dotazy do vzdálených tabulek musí být rozloženy na dílčí operace v jednotlivých databázích (podle umístění dat)
- Nemělo by být nutné přenášet všechna data do jednoho uzlu a tam zpracovávat dotaz.

Další vlastnosti

Optimalizace v DDBMS zahrnuje: cenu I/O operací + CPU čas + **cenu komunikace**

Transakce

Musí být možné provést transakčně změnu dat ve více databázích najednou

2PC (Two Phase Commit) = protokol pro řešení distribuovaných transakcí

1. Někdo to musí řídit = **uzel koordinátor**
2. **Ostatní uzly poslouchají koordinátora**
3. Commit ve **2 fázích:**
    A) **Fáze PREPARE**
    - Koordinátor zašle všem zúčastněným (v té transakci) uzlům zprávu PREPARE a koordinátor čeká na výsledky.
    - Každý uzel se pokusí provést fázi PREPARE – uloží transakční log na disk, neprovádí commit
    - Každý uzel odpoví zpět PREPARED – pokud se vše povedlo, pokud se vyskytl problém, odpoví ABORT

B) **Fáze COMMIT**

- Koordinátor po obdržení vyhodnotí výsledky
- Pokud všechny PREPARED, provede potvrzení svých lokálních změn, tedy COMMIT a ostatním uzlům také pošle COMMIT
- Pokud alespoň jeden výsledek ABORT, povede ROLLBACK svých změn a odešle ROLLBACK i ostatním

Protokol je imunní proti výpadku sítě/uzlu – koordinátor si po celou dobu udržuje data o stavu transakce a je schopen transakci dokončit po obnovení spojení

Distribuování dat

- Data rozmístit do jednotlivých databází tak ,a by se minimalizovaly síťové přenosy a maximalizovalo lokální zpracování
- Pro správný návrh je nutná analýza používání dat v jednotlivých uzlech
- Základní jednotka alokace je tabulka
- Další techniky pro alokaci dat jsou fragmentace a replikace

Fragmentace

=jedna z technik alokace dat

- Vhodné pokud se v každém uzlu distribuovaného systému pracuje s částí tabulky
- Rozdělení relace na několik sub-relací, které se umístí do jednotlivých databází (relace =tabulka?)
- Data se umístí tam, kde se nejčastěji používají, ostatní data jsou dostupná ve vzdálených databázích -> omezují se tak síťové přenosy
- Fragmentace musí být: **úplná, disjunktní, rekonstruovatelná**
- **DĚLENÍ:**
  - **Horzitontální** – podmnožina řádků
  - **Vertikální** - podmnožina sloupců (projekce)
  - **Smíšná** – horizontálně fragmentovaný vertikální fragment, nebo vetikálně fragmentovaný horizontální fragment
  - **Odvozená horizontální** – podřízená tabulka fragmentovaná podle horizontální fragmentace rodičovské tabulky

Replikace

=další z technik alokace dat
= udržování kopií tabulek nebo fragmentů ve více databázích

- Omezení síťových přenosů
- Zvýšení dostupnosti dat při výpadku části distribuovaného systému
- Náročná údržba kopií
- **Vhodné pro často používaná data, která se málo aktualizují**
- **DĚLENÍ podle těsnosti spojení replik:**
  - **Synchronní –** aktualizace replik je součást transakce, degraduje výkon, spolehlivost
  - **Asynchronní –** na vyžádání nebo v pravidelných intervalech, data nemusí být úplně **aktuální**

- **DĚLENÍ podle způsobu provádění repliky**
  - **Pouze změněné řádky –** nutný log změněných řádků
  - **DML příkazy** – fronta odložených příkazů
  - **Úplná**
- **DĚLENÍ podle způsobu zacházení s replikami**
  - **Pouze pro čtení – MASTER-SLAVE** – změny je možné provádět jen v master, repliky jsou read-only
  - **Pro transakční zpracování – MULTI-MASTER –**změny je možno provádět i v replikách, nutno synchronizovat obousměrně – možnost vzniku konfliktů – řešení konfliktů: priority, časová razítka

12 pravidel DDBMS (dle Christopher Date)

1. Lokální autonomie – jednotlivé databáze by měly být maximálně samostatné. Lokální data řízena lokálně, operace řešeny lokálně.
2. Bez centrálního místa, na kterém by byl distribuovaný systém závislý (aby se SQL nezpracovávalo na jednom místě)
3. Nepřetržitá činnost – kvůli jedné databázi neodstavit celý systém
4. Nezávislost na umístění – uživatel nemusí vědět kde jsou data umístěna
5. Nezávislost na fragmentaci
6. Nezávislost na replikaci dat
7. Distribuované zpracování SQL
8. Distribuované zpracování transakcí
9. Nezávislost zpracování na HW
10. Nezávislost zpracování na OS
11. Nezávislost na síti
12. Nezávislost na konkrétním DBMS (nemusí jít ano o RDBMS)

DDBMS by se z uživatelského pohledu měl tvářit jako lokální DBMS

Oracle

Využívá koncept, kdy databázový server zprostředkovvává pro své klienty přístup na data jiných databázových serverů. Databázové servery je nutné propojit – používá se stejný protokol jako pro komunikaci mezi klientem a serverem – Net8. Základem koncemplu je **databázový link.**

DB Link:

- Databázový objekt (podobně jako View, Role)
- Definuje připojení do vzdálené databáze, obsahuje identifikaci vzdálené databáze a databázový účet, na který se připojí
- DB Link se zakládá v databázovém serveru, který má zprostředkovat svým klientům přístup do vzdálené databáze (pro symetrické připojení nutné 2 linky) – architektura Klient – Server
- **Zpřístupnění vzdálených dat řídí administrátor vzdálené databáze –** založení konta s oprávněním, na které vede db link

ORACLE Gateways – pro heterogenní prostředí (různé DBMS)
= SW který zajistí, že se non-Oracle DBMS tváří jako Oracle
Provádí: překlad SQL, datových typů, optimalizace SQL.

Charakteristické vlastnosti (z OneNote)

Distribuovaná DB je charakterizována:

- Transparentností – z pohledu klienta se zdá, že data jsou zpracována na jednom serveru v lokální databázi. Uživatel si rozdělení nemusí být vědom.
- Rozšiřitelností – zvýšení výkonu přidáním dalších počítačů.
- Robustností – výpadek jednoho počítače neovlivní funkci ostatních
- Jsou syntakticky shodné příkazy pro lokální i vzdálená data, nespecifikuje se místo uložení dat (řeší to distribuovaný SŘBD)
- Autonomností – s každou lokální bází dat zapojenou do distribuované databáze je možno pracovat nezávisle na ostatních databázích.
- Lokální Db je funkčně samostatná, připojení do jiné části distribuované db se v případě potřeby zřizují dynamicky.
- Nezávislost na počítačové síti – jsou podporovány různé typy architektur lokálních i globálních počítačových sítí (LAN, WAN)
- V distribuované databázi mohou být zapojeny počítače i počítačové sítě různých architektur, pro komunikaci se používá jazyk SQL.

Distribuované databáze jsou výhodné kvůli:

- Lokální autonomie – odpovídají struktuře decentralizovaných organizací. Data jsou uložena v místě nejčastějšího využití a zpracování – zlevnění provozu
- Zvýšení výkonu (rozdělení zátěže na více počítačů)
- Spolehlivosti (replikace dat, degradace služeb při výpadku uzlu, přesunutí na jiný uzel)
- Rozšiřitelnosti
- Schopnosti sdílet informace integrací podnikových zdrojů
- Agregaci informací – z více bází dat lze získat informace nového typu.

Naopak Distribuované Db mohou způsobit i pro ně specifické problémy:

- Složitost – distribuce db, distribuce zpracování dotazu a jeho optimalizace, složité globální transakční zpracování, distribuce katalogu, paralelismus a uvíznutí, složité zotavování z chyb
- Cena (komunikace je navíc)
- Bezpečnost
- Obtížný přechod – neexistuje automatický konverzní prostředek z centralizovaných DB na DDB

Taxonomie DDBS

- **Těsně integrované**
  - Uživatel vidí data centralizovaná v jediné databázi, DDB je vybudována nad lokálními DB, každé místo má úplnou znalost o datech v celém DDBS a může zpracovávat požadavky používající data z různých míst.
- **Semiautonomní**
  - Lokální DBMS pracují nezávisle a sdílejí svoje lokální data v celé federaci, jen část jejich dat je sdílena.
- **Zcela autonomní**
  - Izolované, lokální DBMS pracují nezávisle a neví o ostatních DBMS, pro vzájemnou komunikaci potřebují softwarovou vrstvu pracující nad jednotlivými DBMS.

Problémy replikace a fragmentace dat

U fragmentace a replikace bychom měli respektovat hlediska:

- Rozdělit relace do lokálních serverů tak, aby aplikace zatěžovaly servery stejnoměrně.
- Přístupnost a spolehlivost - Replikací zlepšíme spolehlivost a read-only dostupnost.
- Lokality zpracování (maximalizovat lokální).
- Dostupnosti a ceny paměti v jednotlivých uzlech

Replikace dat

Replikační transparentnost je neobtěžovat uživatele skutečností, že pracuje s daty existujícími ve více kopiích = uživatel neví o replikách.

Při replikaci dat se nachází kopie množiny objektů v každém uzlu, ve kterém je využívaná. V systému tedy existuje několik kopií každého objektu. Výhodou tohoto řešení je kvalitní dostupnost, rychlý přístup ke každému objektu a menší nároky na komunikaci mezi uzly. Nevýhodou je problém duplicity, díky níž je nutné trvale *zajišťovat konzistenci všech kopií*. Je nutné implementovat systém, který určí správné pořadí provedených operací a při replikaci rozdistribuuje správnou kopii. Také je nutné zabránit současné modifikaci dvou kopií objektu.

Správné pořadí provedených operací je určováno většinou časovými razítky, současné modifikaci dvou kopií jednoho objektu mohou zabránit klasické paralelní synchronizační prostředky (zámky, semafory apod.)

Jsou dva základní způsoby zacházení s replikami:

1. *Pouze pro čtení (master-slave)* - změny (zápis) může provádět jen master, repliky jsou read-only a periodicky se synchronizují s masterem; repliky tak mají lehce zastaralá, nekonzistentní data (tzv. eventual consistency, nakonec se dojde do konzistentního stavu)
2. *Pro transakční zpracování (multi-master)* - rovnocennější, změny možné provádět v replikách, je proto nutná obousměrná synchronizace, mohou tak vznikat konflikty, které se řeší různě (priority, časová razítka, manuálně, vlastní procedura)

Jiný zdroj:
Replikace = uchování kopií relací v různých uzlech.
výhody: porucha v jednom uzlu neznemožní přístup k jeho lokálním relacím, data v lokálních bázích jsou připravena k použití okamžitě, bez nutnosti přenosu
nevýhody: ztížení aktualizace, všechny kopie musí být aktualizovány současně a v průběhu aktualizace je nutno uzamknout aktualizovaná data ve všech uzlech sítě.
Proto se replikují data, ke kterým je potřebný rychlý přístup a která nejsou často aktualizována, příp. která jsou pro systém mimořádně důležitá (číselníky, registry ap.)

Fragmentace dat

Fragmentační transparentnost = uživatelův dotaz je specifikován na celou relaci, ale musí být vykonán na jejím fragmentu = uživatel neví o fragmentech.

Fragmentace dat se dělí na:

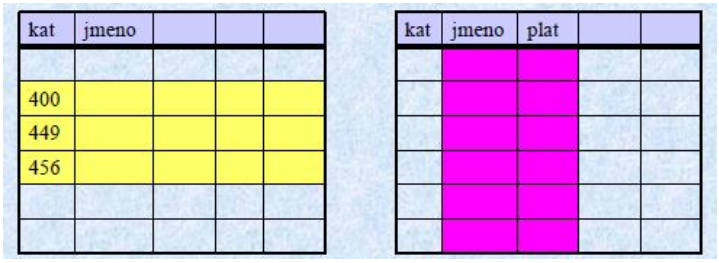
- Horizontální – dle selekční podmínky rozdělíme tabulku na 2 části horizontálním řezem, tedy např. na knihy s ISBN nižším než 122 a na knihy s ISBN větším nebo rovným 122.
- Odvozená horizontální – dochází k rozdělení tabulky na 2 a více částí horizontálním řezem, v tomto případě však je fragmentace založena na jiné relaci. Např. DODAVATELE a KNIHY. DODAVATELE

rozdělíme do fragmentů a na základě těchto fragmentů provedeme fragmentaci v tabulce KNIHY, která je spojena s DODAVATLE relací.

- Vertikální – rozdělení tabulky podle sloupců na 2 a více částí. V jedné skupině jsou jedny sloupce, v druhé jiné.
- Smíšená – tabulku např. rozdělíme dle sloupců (vertikální) a následně v jednotlivých částech provedeme horizontální fragmentaci (a nebo obráceně).

Jiný zdroj:

Fragmentace = rozložení relace na části (fragmenty), které jsou umístěny v různých uzlech sítě. Může jít o horizontální fragmentaci, kdy se v různých uzlech ukládají části relace rozložené do skupin řádků, nebo o vertikální fragmentaci, kdy se v uzlech ukládají různé projekce relace. Fragmentace se provádí tak, aby bylo možno z fragmentů získat původní relaci standardními operacemi nad relační databází (sjednocením nebo spojením).



Distribuovaná správa transakcí

- Pro provedení transakční změny dat ve více databázích najednou
- K tomu se využívá **dvoufázový commit** (2PC)
  - Jde o protokol řešení distribuovaných transakcí
  - Řídící uzel (koordinátor) řídí průběh celé distribuované transakce
  - Ostatní zainteresované uzly poslouchají
  - **Fáze PREPARE**
  - **Koordinátor** zašle zprávu PREPARE všem uzlům účastnících se distrib. transakce
  - Koordinátor v této fázi nic nedělá, čeká jen na výsledky od ostatních uzlů
  - **Každý podřízený uzel** se pokusí provést fázi PREPARE
    - Neprovádí vlastní commit, transakce je pořád neukončená, změněné řádky zůstavají stále zamčené
    - Jen se zajistí, aby aktuální stav rozpracované transakce byl schopný přežít výpadek
    - Každý uzel odpoví zpět koordinátorovi PREPARED (pokud vše ok) nebo ABORT (pokud nastal problém)
  - **Fáze COMMIT**
  - Pokud obdržel koordinátor od všech uzlů PREPARED, provede potvrzení svých lokálních změn (normální commit)
  - A všem uzlům zašle zprávu COMMIT
  - Pokud je aspoň 1 výsledek ABORT, provede ROLLBACK svých změn (standardní COMMIT) + všem uzlům zašle zprávu ROLLBACK
- Protokol je imunní vůči výpadku sítě/uzlu v libovolné fázi
- Koordinátor si po celou dobu zpracování globální transakce udržuje data o stavu transakce a všech podtransakcích
  - Je tedy schopen transakci dokončit např. při obnovení spojení se vzdáleným uzlem

13. Temporální databáze, porovnání klasických a temporálních databází, modely času ,vztah událostí a času (snaphsot), temporální SQL

Temporální databáze jsou databáze určitým způsobem podporujícím čas. Čas potřebujeme v databázích např. ve studijním informačním systému, účetních a bankovních systémech, docházkových systémech. **Hlavní cíl temporálního DM by měl být zachytit sémantiku dat měnící se v čase**. V praxi existuje mnoho nekompatibilních datových modelů s mnoha dotazovacími jazyky.

Temporální databáze (temporální databázový systém) je databáze (databázový systém) zohledňující časové vlastnosti ukládaných dat

Klasické vs. Temporální databáze

Klasický databázový systém – zachycuje stav systému v aktuálním časovém okamžiku, problém – co dělat se starými daty. (např. „jaký plat má Pepa“)

Temporální databázový systém – databáze určitým způsobem podporující čas, jednodušší dotazy, jednodušší udržování aplikací (např. „jaký je aktuální Pepovo plat“, „jaká je Pepova platová historie“)

Modely času

Temporální logika: čas je libovolná množina okamžiků s daným uspořádáním

Modely času podle uspořádání //modely těch databáze

- **Lineární –** jedna minulost až jedna budoucnost
- **Větvený (čas možných budoucností) –** do více budoucností
- **Cyklický**

Modely času podle hustoty

- **Diskrétní**
- **Hustý**
- **Spojitý**

Datové typy pro čas

- **Časový okamžik (instant)**
  - **Date**
  - **Time**
  - **Timestamp**
- **Časový úsek (time period)**
  - **Doba mezi dvěma časovými okamžiky (15.00 až 15.30)**
- **Časový interval (interval)**
  - **Doba o specifikované délce, ale bez konkrétních krajních bodů (30 minut)**
- **Množina časových okamžiků (instant set)**
- **Množina časových úseků (temporal elements)**

Vztah událostí a času

- **Čas platnosti (valid time)** //dostupnost jídla
  - Čas, kdy byla událost pravdivá v reálném světě
  - Může být v minulosti, přítomnosti i budoucnosti

- **Transakční čas (transaction time)** //kdy jsem to zadal do db
  - Čas, kdy byl fakt reprezentován v databázi
  - Nabývá pouze aktuální hodnoty
  - Monotónně roste

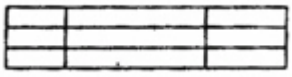
  Čas platnosti a transakční čas jsou ortogonální

Datové modely zohledňující časy (asi?)

- **Snapshot** – datový model nepodporující čas platnosti ani trasakční čas
- **Valid-time** – datový model podporující pouze transakční čas
- **Transaction-time** – datový model podporující pouze trasakční čas
- **Bitemporální** – datový model podporující čas platnosti i transakční čas
- **Temporální** – datový model podporující čas planosti nebo transakční čas

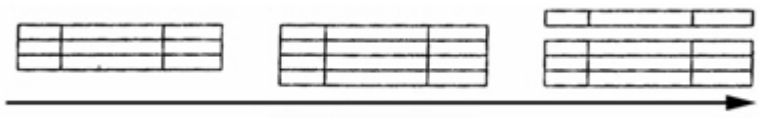
SNAPSHOT RELACE (tabulka)

- Datový model nepodporující čas planosti ani transakční čas
- => **klasický relační model**
- **Každá n-tice(záznam) je fakt planý v reálném světě**



TRANSACTION-TIME RELACE

- Dat model podporující pouze transakční čas
- **Platnost snapshotů indexované transakčním časem**
- **Umožňuje získat informaci ze stavu databáze v nějakém okamžiku minulosti**



VALID-TIME RELACE

TSQL2

- Temporální dotazovací jazyk – RELAČNÍ (mezi objektové patří třeba OQL)
- Temporal Structured Query Languge 2
- Nadmnožina SQL92
- **Je použit Bitemporální** Konceptuální **Datový model (BCDM)**
- **„kolik jsem toho prodal před 2 lety“**
- Umožňuje provádět dotazy se zohledněním času, reálného nebo transakčního

//doplnit

Další popis (z OneNote)

Temporální databáze (temporální databázový systém) je databáze (databázový systém) zohledňující časové vlastnosti ukládaných dat

Jméno	Plat	Funkce	Datum narození	Platí_od	Platí_do
Pepa	60000	Vrátný	1945-04-09	1995-01-01	1995-06-01
Pepa	70000	Vrátný	1945-04-09	1995-06-01	1995-10-01
Pepa	70000	Vrchní vrátný	1945-04-09	1995-10-01	1995-02-01
Pepa	70000	Ředitel bezpečnosti	1945-04-09	1996-02-01	1997-01-01

Porovnání klasických a temporálních DB

Klasický DB systém

Zachycuje stav systému v aktuálním časovém okamžiku. Problém: co dělat se starými daty. V případě, že se systém v čase vyvíjí, změny se v DB projeví přidáním nových informací a mazáním starých. Klasické DB neobsahují informaci o čase. V případě, že požadujeme uchování historie změn, či alespoň předchozího stavu, je nutné do DB doplnit informace o čase. Aktualizaci a operace s časem musí zajistit uživatel, což není triviální.

Temporální DB

Databáze určitým způsobem podporující čas. Poskytuje vhodný dotazovací jazyk zahrnující práci s časem – výhodou jsou jednodušší dotazy, v nichž se vyskytuje čas, což přináší méně chyb v aplikačním kódu a zajišťuje jednodušší udržování aplikací.

Temporální projekce – jako projekce v klasických databázích (z celé relace jsou vybrány hodnoty podle zadaných atributů), navíc bere v úvahu čas. V případě, že dvě n-tice výsledku mají stejné hodnoty všech svých atributů a překrývají se nebo dotýkají se časem, srostou tyto dvě n-tice s časem odpovídajícím sjednocení obou n-tic.

Temporální spojení – stejné jako spojení (JOIN) v klasických DB (zadáme sloupce a podmínku, která říká, kdy jsou dva řádky tabulky spojeny), navíc bere v úvahu čas události. V Temporálních DB nemusíme zadávat sloupce uchovávající čas, pouze podmínku na spojení pro čas.

Jiný zdroj:

Klasické databáze

- Neobsahují informaci o čase
- V databázi zachycen pouze aktuální stav systému. V případě, že se v čase systém vyvíjí, změny se v databázi projeví přidáváním nových informací a mazáním starých.
- V případě, že požadujeme uchovávání historie změn, či alespoň předchozího stavu, je nutné do databáze doplnit informaci o čase. Aktualizaci a operace s časem musí zajistit uživatel. Což (jak ilustrujeme dále) není triviální.
- Jako příklad poslouží SIS, kde chceme uchovávat informace o předcházejících semestrech. Řešením je přidání sloupce, který identifikuje konkrétní semestr. Nevýhodou tohoto řešení je, že s touto informací musí manipulovat uživatel sám.

Temporální databáze

- Konkrétní podporu času uvidíme později
- Vhodný dotazovací jazyk zahrnující práci s časem
- Výhodou jsou jednodušší dotazy v nichž se vyskytuje čas, což přináší méně chyb v aplikačním kódu

Modely času

Temporální logika: čas je libovolná množina okamžiků s daným uspořádáním. Modely času se rozlišují podle:

Dle uspořádání

- **Lineární** – čas roste od minulosti k budoucnosti lineárně
- **Větvený** (čas možných budoucností) – lineární minulost až do teď, pak se větví do několika časových linií reprezentujících možný sled událostí. Každá linie se může dále větvit.
- **Cyklický** – opakující se procesy. Př. týden, každý den se opakuje po sedmi dnech.

Dle hustoty

- **Diskrétní** – spolu s lineárním uspořádáním. Každý okamžik má právě jednoho následníka.
- **Hustý** – Mezi každými dvěma okamžiky existuje nějaký další
- **Spojitý** – každé reálné číslo odpovídá bodu v čase.

- Omezenost času – Omezený – nutnost zejména kvůli reprezentaci v počítači, Neomezený.
- Absolutní /relativní čas – Absolutní se vyjádří hodnotou, také ale potřebuje počátek. Relativní vyžaduje nějaký počátek, čas se pak vyjádří jako vzdálenost a směr od počátku.

Datové typy pro čas jsou:

- Časový okamžik (instant) – DATE, TIME, TIMESTAMP
- Časový úsek (time period) – doba mezi dvěma časovými okamžiky (15:30-16:00)
- Časový interval (interval) – doba o specifikované délce, ale bez konkrétních krajních bodů (30 minut)
- Množina časových okamžiků (instant set)
- Množina časových úseků (temporal elements)

Vztah událostí a času (snapshot)

Čas platnosti (valid time)

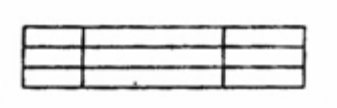
- Čas, kdy byla událost pravdivá v reálném světě. Může být v minulosti přítomnosti i budoucnosti.
- Reprezentace času platnosti – časový okamžik, doba, časový úsek, množina okamžiků
- Čas platnosti může být přidružen k atributům, množině atributů, celé n-tici nebo objektu

Transakční čas (transaction time)

- Čas, kdy byl fakt reprezentován v DB. Nabývá pouze aktuální hodnoty. Monotónně roste.
- Reprezentace transakčního času – časový okamžik (nová n-tice se stejným klíčem – logické odstranění původní), časový úsek (teď, dokud nezměněno), tři časové okamžiky (čas zaznamenání

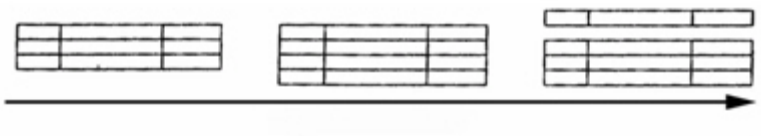
začátku v reálném světě, konce události v reálném světě, logického odstranění události z db), množina časových úseků

Snapshot
Datový model nepodporující čas platnosti ani transakční čas. Je to klasický relační model. Každá n-tice je fakt platný v reálném světě. Při změně reálného světa jsou do relace prvky přidávány nebo z ní odebírány.



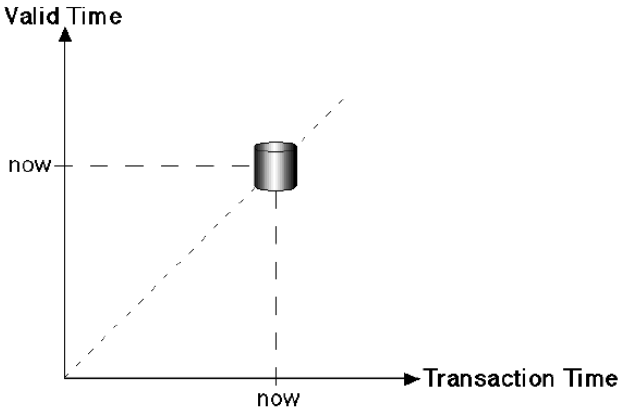
Další:

Další datové modely mohou být **valid-time** relace (podporuje čas platnosti, umožňuje klást dotazy o faktech z minulosti i budoucnosti), **transaction-time** relace (podporuje pouze transakční čas, umožňuje získat informaci ze stavu db v nějakém okamžiku v minulosti), bitemporální, temporální.

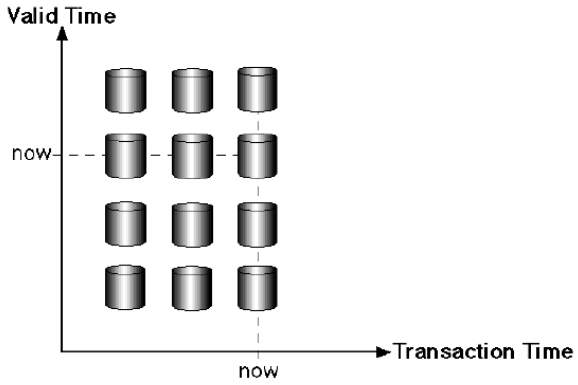


Obr – transaction-time

Snapshot database:



Temporal database: //lze se dotazovat na různé chvíle



Temporální SQL

Temporální datový model obsahuje objekty s přesně danou strukturou, omezení pro dané objekty a operace na daných objektech (temporální dotazovací jazyky). Temporálních dotazovacích jazyků je velké množství. Nejčastěji jsou založené na SQL. Typy jsou:

- Relační – HQL, HSQL, TDM, TQuel, TSQL, TSQL2
- Objektově orientované – Matisse, OSQL, OQL, TMQL

TSQL2

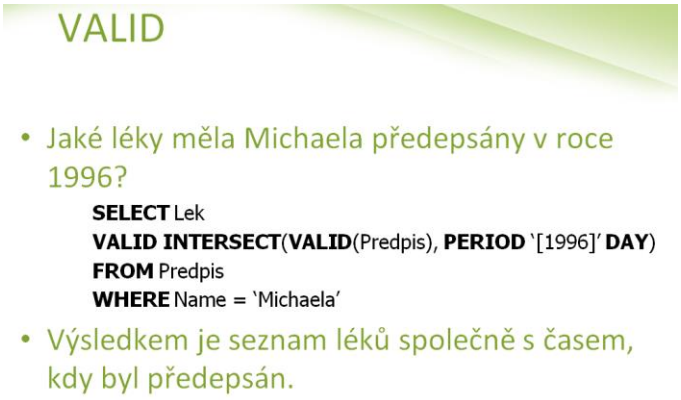
Temporal SQL 2 – měl sjednotit přístupy k temporálním datovým modelům. Je to nadmnožina SQL92.

V TSQL2 je časová osa na obou koncích omezena, ale dostatečně daleko (18 miliard let). U časových údajů jsou možné různé granularity. Časové typy: DATE, TIME, TIMESTAMP, INTERVAL, PERIOD.

Datový model je bitemporální. Řádek je orazítkován množinou bitemporálních chrononů. Bitemporální chronon je dvojice (chronon transakčního času, chronon času platnosti). Př. Relace ZAMESTNANEC – umístění lidí v odděleních určitého podniku. Schéma (Jméno, Oddělení) + časové razítko.

Př. SELECT – komu byl předepsán nějaký lék – výsledek bez podpory času

```
SELECT SNAPSHOT Jmeno FROM Predpis
```



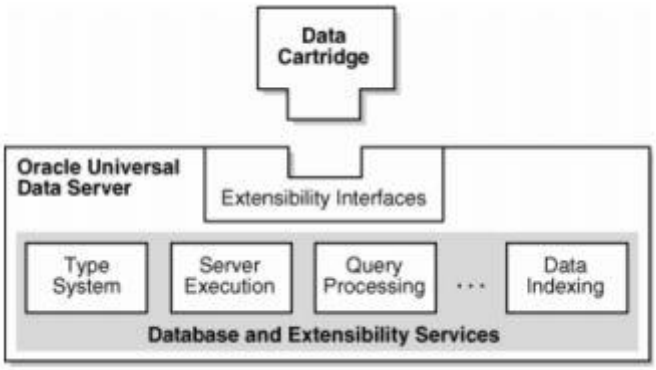
14. Uživatelské rozšíření databázových systémů – data cartridge, příklady použití

Asi sem patří i přednáška s SQL/MM

rozšiřitelnost = možnost přidávání nových datových typů a programů (funkcí) zabalených do speciálního modulu:

- ⇨ uživatelsky definované typy a funkce:
  - **typy = UDT**
  - **funkce = UDF**
- problém: zapojení do relačního SŘDB včetně SQL
  - DB/2 = relační extendery
  - Informix = DataBlades
  - Sybase = Component Integration Layer
  - **Oracle = cartridges**

Oracle Extensibility Framework



A) Data cartridge
- způsob uživatelského rozšíření databázového serveru Oracle
- rozšíření je pouze na straně serveru
- jsou integrované se serverem přes rozhraní
- jsou v podobně balíků = instalují se jako celek

B) Extensibility interface
- rozhraní, které umožňuje vytvářet cartridge jednotným způsobem
- poskytuje přesně definovaný způsob, jak se serverem komunikovat
- zpřístupňuje jednotlivé standardní služby, které lze v serveru rozšířit

C) Database and extensibility services
- sada standardních služeb, které poskytuje server
- lze je využívat a rozšiřovat pomocí cartridge
- jednotlivé služby jsou:
  - **Extensible type systém** -podpora pro uživatelské typy, kolekce, reference (REF), LOBy
  - **ExtensibleServer execution environment**- podpora vlastních procedur a funkcí
  - **Extensible Indexing** - vlastní způsob indexování = tzv. doménové indexování pro doménově specifická data
  - **Extensible Optimise** - podpora pro vytváření uživatelských funkcí a indexů pro vlastní sběr statistik pro CBO

Pracuje na principu tzv. cartridge (kdo??)

- způsob uživatelského rozšíření databáze Oracle
  - integruje se pomocí sady rozhraní pro jednotlivé přístupy (k datům, indexům,…)
- rozšíření může definovat
  - nové datové typy(standardní, pole, hnízděné tabulky, LOBy) a jejich funkce
  - nové typy indexů
  - nové operátory
- proč implementovat DataCartridge
  - nutnost zpracování komplexních dat, která neodpovídají standardním relačním informacím
  - nutnost snadné manipulace s takovými daty
- Příklady
  - multioborové = datové, statistické výpočty, prostorové databáze, multimédia
  - specializované, finanční a právní systémy

- **typická struktura cartridge**
  - definice nových objektových typů
  - implementace těl typů, balíky, procedury, funkce
  - případné DLL knihovny s implementací v C
  - operátory
  - doménové indexy pro podporu operátorů
- př. standardní cartridge: podpora indexování a vyhledávání v textech

Př. cartridge:

LOB - Large Objects

- standardní typy pro ukládání objemných dat na serveru (až 4 GB)
- typy
  - Externí
    - BFILE = samostatný binární soubor uložený vně databáze
  - Interní
    - CLOB = znakový typ
    - NCLOB = znakový typ v národní sadě
    - BLOB = binární typ

- ve sloupci tabulky uložen pouze deskriptor odkazující na samotná data
  - hodnoty pro xLOB slupce = NULL, EMPTY_CLOB(), EMPTY_BLOB(), EMPTY_NCLOB()
- manipulace
  - Oracle nabízí balík DBMS_LOB s řadou funkcí a procedur pro standardní manipulaci s daty
  - provádí se po částech pomocí bufferů
- indexace
  - není standardně indexováno, ale je možné implementovat svá vlastní rozhraní pro indexaci

Rozšíření serveru

- server dovoluje psát implementace procedur v řadě jazyků
  - nativním PL/SQL
  - Javě
  - v čemkoliv s konvencí jazyka C a kompilací DLL

Příklady rozšíření

- Matematické, finanční funkce
- Práce s audio, video objekty v reálném čase
- GIS a prostorové objekty
- Textová rozšíření (různé statistiky textu)

Cartridge Oracle Text (z nadpisu jeji prednasky, takze to bylo asi dulezite)

Oracle Text and Ultra Search

Oracle Text brings search engine-like full text search capabilities to the Oracle Database. Ultra Search provides a ready-to-use application, while Oracle Text provides a foundation for building your own search applications. Search regular columns, text inside various kinds of binary-format documents, and text, tags, and attributes inside XML documents.

15. Dokumentografické systémy, fulltextové vyhledávání, filtrace, disambiguace, lemmatizace, index, tezaury, dotazování.

Dokumentografické systémy (DIS)

- vznik 50. léta 20. stol. za účelem automatizace postupů používaných v knihovnictví
- Nyní samostatná podčást IS
  - **Faktografický IS** - informace s definovanou vnitřní strukturou (nejčastěji tabulky) //asi normální databáze
  - **Dokumentografický IS** - informace v podobě textu v přirozeném jazyce bez pevné vnitřní struktury

Práce s DIS:

- **Zadání dotazu**
- **Porovnání**
- **Získání seznamu odpovídajících dokumentů**
- **Ladění dotazu**
- **Vyžádání dokumentu**
- **Obdržení textu**

Struktura DIS:

- **Systém zpřístupnění dokumentu** – vrací sekundární informace o dokumentu (Autor, Název, ...)
- **Systém dodání dokumentu** - někdy není řešeno pomocí SW

Vyhodnocení dotazu

- přímé porovnávání náročné na čas

Ale při využití indexace a modelu dokumentu:

- nutné vytvořit model dokumentu
- ztrátový proces, založený na identifikaci slov v dokumentech
- výsledkem strukturovaná data vhodná pro porovnávání
- dotaz se upraví do odpovídající podoby a porovná se s modelem dokumentů

Text

Předzpracování

- vyhledávání nad modelem efektivnější, ale lze použít jen informace z modelu
- **cíl**: vytvořit model, zachovávající nejvíce info z původního textu
- Problém: nejednoznačností (**ambiguity**)
- dosud neřešené nároky na encyklopedické i asociativní znalosti

Porozumění textu

- Homonymie slov
  - jedno slovo může mít stejný tvar pro různé pády a další gram. jevy
    - kontroly: 1.p.m.č., 2.p.j.č. - není zřejmé jestli více kontrol nebo jedna kontrola
  - jeden tvar může mít různý význam
    - hnát - sloveso, podst. jm.
    - pět - číslovka, sloveso
- přiřazení je závislé na osobě, která dokument píše nebo čte
  - dva lidé mohou jednomu slovu přikládat zcela nebo částečně jiný význam
  - dva lidé si i pod stejným významem mohou představit jiný konkrétní předmět nebo množinu
    - máma, pokoj, ...
- výsledkem situace, kdy dva různí čtenáři nemusí přečtením získat stejnou informaci jako autor, ani navzájem.
- Homonymie a nejednoznačnosti narůstají při přechodu od slov k větám.

Přesnost a úplnost

- důsledek nejednoznačnosti: žádný existující DIS nedává ideální výsledky
- Pro zobrazení odpovědi na dotaz lze určit
  - Nv (počet vrácených dokumentů) - O nich si DB myslí, že jsou relevantní, odpovídající dotazu
  - Nvr (počet vrácených relevantních dok.) - o nich si tazatel myslí, že uspokojí jeho požadavky
  - Nr (počet všech relevantních dok. v DB) - problematické u velkých DB
- Kvalita výsledné množiny se měří na základě:
  - **Přesnost** (Precision): $P = N_{vr} / N_v$
    - pravděpodobnost, že dokument zařazený v odpovědi je skutečně relevantní
  - **Úplnost** (Recall): $R = N_{vr} / N_r$
    - pravděpodobnost, že skutečně relevantní dokument je zařazený v odpovědi
- koeficienty jsou opět závislé na subjektivním názoru tazatele
- dokument vrácený na výstupu může uspokojovat požadavky dvou uživatelů, kteří položili stejný dotaz, různou měrou
- ideální případ: P == R == 1

Kritéria

A) Kritérium predikce

- při formulaci dotazů je třeba uhádnout, které termy (slova) byly v dokumentu autorem použity pro vyjádření dané myšlenky
  - problémy způsobují
    - synonyma - autor používá synonyma, které si tazatel nemusí při dotazu uvědomit
    - překrývající se význam slov
    - opisy jedné situace jinými slovy
- částečné řešení - zařazení tezauru, který obsahuje
  - hierarchie slov a jejich významů
  - synonyma slov
  - asociace mezi slovy
- tazatel může tezaurus využít při formulaci svých dotazů
- při ladění dotazů má uživatel tendenci postupovat konzervativně

    - v dotazu často zůstávají ty části, které uživatele napadly na začátku a mění se jen podružné části, které nekvalitní výsledek nemusí zásadně ovlivnit
  - vhodné je uživateli pomoci s odstraněním nevhodných částí dotazu, které nepopisují relevantní dokumenty a naopak s přidáváním formulací, které relevantní dokumenty popisují

B) Kritérium maxima

- tazatel obvykle není schopen (ochoten) procházet příliš mnoho dokumentů do té míry, aby se rozhodl, zda jsou pro něj relevantní nebo ne
- obvykle 20-50 podle velikosti
- potřeba nejen dokumenty rozlišovat na odpovídající/neodpovídající dotazu, ale řadit je na výstupu podle míry předpokládané relevance
- v důsledku kritéria maxima se při ladění dotazu uživatel obvykle snaží zvýšit přesnost
  - malé množství dokumentů v odpovědi, obsahující co největší poměr relevantních dokumentů
- některé oblasti použití vyžadují co nejvyšší přesnost i úplnost
  - např. právnictví

Modely dokumentografických systémů

Úrovně modelů:

- rozlišují (ne)přítomnost slov v dokumentech
- rozlišují frekvence výskytů slov //booleovsky
- rozlišují pozice výskytů slov v dokumentech //asi vektorovy

1. Boolský model

- vznik 50. léta 20. stol., automatizace postupů používaných v knihovnictví
- Databáze obsahuje dokumenty, dokumenty popisovány pomocí termů, reprezentace dokumentu pomocí množiny termů (obsažených v dokumentu, popisujících význam dokumentu)
- **Indexace**
  - Přiřazení množiny termů, které jej popisují ke každému dokumentu
    - Ruční - nekonzistence
    - Automatická - konzistentní, ale bez porozumění textu
    - Řízená - předem daná množina termů
    - Neřízená - množina termů se mění s přibývajícími dokumenty
  - Tezaurus - vnitřně strukturovaná množina termů
    - Synonyma s preferovanými termy
    - Hierarchie užších/širších termů
    - Příbuzné termy
    - ...
  - Stop-list - nevýznamová slova
  - Příliš obecná slova nejsou pro identifikaci dokumentů vhodná, příliš specifická slova také ne
  - dotaz vyjádřen logickým výrazem: AND, OR, NOT
  - Příklad dotazu: počítač AND NOT osobní
  - Víceslovné termy: počítač AND NOT osobní počítač
  - Organizace indexu:
    - Invertovaný seznam - pro každý term je seznam dokumentů, ve kterých se vyskytuje
    - Zpracování dokumentů na vstupu - vznikne posloupnost dvojic <dok_id,term_id>
    - Setřídění dle term_id,dok_id

- **Nevýhody**
  - formulace dotazů je spíše uměním než vědou
  - nemožnost ohodnotit vhodnost vystupujících dokumentů
  - všechny termy v dotazu i v identifikaci dokumentu jsou chápány jako stejně důležité
  - nemožnost řízení velikosti výstupu
  - některé výsledky neodpovídají intuitivní představě

2. **Vektorový model**

- vznik 70. léta 20. stol., cca o 20 let mladší než Booleovské DIS
- snaha minimalizovat nebo odstranit nevýhody Booleovských DIS
- Struktura:
  - databáze obsahuje dokumenty
  - dokument popisován pomocí množiny termů
  - term je slovo nebo sousloví
  - reprezentace dokumentu pomocí vektoru vah termů

Tezaury

In Microsoft SQL Server 2005, full-text queries can use a thesaurus to find synonyms of search terms. For each supported language, there exists a single thesaurus file.

Tezaurus je vnitřně strukturovaná množina termů:
- synonyma s preferovanými termy
- hierarchie užších/širších termů
- asociace mezi slovy

Fulltextové vyhledávání

- Odlišné od principů běžného vyhledávání
  - neprohledávají se striktně strukturovaná data, kde má každý sloupec každé tabulky předem daný význam
  - prohledávají se volně psané texty, kde může být stejná událost popsána více autory rozdílně - různá slova stejného významu, různé slovní obraty a opisy

- DB systémy využívají svých prostředků rozšiřitelnosti a dodávají standardně prostředky, které vyhledávání v textových datech umožňují

- Rozdílné přístupy a možnosti
  - neexistuje objektivně nejlepší řešení
  - výsledky navíc podléhají subjektivním názorům tazatelů

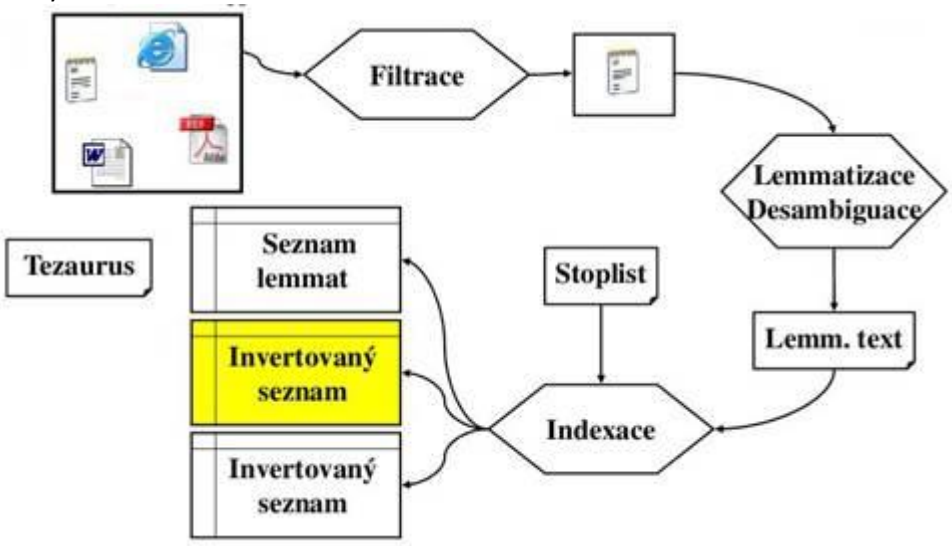
- Samotná formulace dotazu, který by vrátil všechny dokumenty, které tazatele zajímají a žádné jiné, obvykle nelze zformulovat
  - spolu s vyhovujícími - relevantními - odpověďmi se obvykle vrací i odpovědi nerelevantní

- **Problémy**
  - Homonyma - ptá se tazatel dotazem "koruna" na finanční, lesnické či panovnické dokumenty?
  - Synonyma
    - Vyhovuje dokument o "krychlích" dotazu na dokumenty o "kostkách"?
    - Vyhovuje dokument o "stromech" dotazu na "souvislé grafy bez cyklů"?
  - Hierarchie významů
    - Zvíře - Savec - Šelma - Medvěd
    - Tiskovina - Časopis
  - Ohebnost slov - Jít, Jde, Jdu, Jdou, ...

- Striktní booleovská logika není pro formulaci dotazů příliš vhodná
  - Dokument buďto vyhovuje nebo nevyhovuje
  - Dotazování v textech vyžaduje třídit odpovědi podle předpokládané vhodnosti pro tazatele - je potřebné mít možnost definovat míru shody dotazu s dokumentem

- Pozn. Možná dobré vědět něco o algoritmech prohledávání řetězců viz PT: Knuth-Morris-Pratt, Boyer-Moore, Brute Force, Rabin-Karp…

Předzpracování

- Databáze obvykle používají některý z booleovských modelů reprezentace dokumentů
  - nejlépe odpovídá běžným dotazům
  - relativně snadno se implementuje
  - dotazy jsou ve formě booleovských formulí, ve kterých operandy tvoří jednotlivá slova - řada různých modifikací
- 

Jednotlivé kroky předzpracování fulltextového vyhledávání:

Filtrace

- *Filtrace* - odstraní formátovací značky a nechá čistý ASCII text

Desambiguace

- *Desambiguace* - určí význam slova podle kontextu
  - "pět chválu" ... sloveso pět
  - "pět vozidel" ... číslovka pět

Lemmatizace

- *Lemmatizace* - určí základní tvar slova a gramatický tvar v dokumentu, často nahrazen pomocí stemmeru, který hledá kmen slova

Indexace

- *Indexace* - vytvoří pomocné seznamy lemmat a dokumentů a invertovaný soubor
  - dvojice *[id_dok, id_lemmatu]* setříděné dle id_lemmatu a zbavené duplicit
  - dnes obvykle více informací, např. pětice *[id_dok,č_odstavce,č_věty,č_slova,id_lemmatu]* - dovoluje vyhodnocování tzv. proximitních omezení na vzdálenost slov v dokumentu

Large OBjects (LOB)

- pro podporu vyhledávání je potřeba nad textovým sloupcem vytvořit index - invertovaný soubor
- běžné textové sloupce jsou pro tyto účely krátké a nevyhovující
  - obvykle se takto indexují sloupce některého z LOB (Large OBject) typů
- **LOBy**
  - standardní typy pro ukládání objemných dat na serveru, definováno v SQL-92 Full
  - až 4GB dat
    - BLOB - standardní binární typ
    - CLOB - znakový typ v univerzální znakové sadě
    - NCLOB - znakový typ v národní znakové sadě
  - v Oracle navíc externí typ BFILE
    - pouze pro čtení
    - samostatný binární soubor uložený vně databáe v OS
  - v MS SQL
    - Image - binární data do velikosti 2 GB
    - Text - textová data do velikosti 2 GB
    - NText - textová data v národní znakové sadě do vel. 1 GB
- Ve sloupcích tabulky je uložen pouze deskriptor (tzv. LOB lokátor), odkazující na samostatně uložená data

Dotazování

Oracle fulltext

- Filtrování vstupních dokumentů
  - **NULL_FILTER** - pro textové dokumenty TXT, HTML, XML
  - **INSO_FILTER** - pro binární dokumenty
  - **CHARSET_FILTER** - pro konverzi získaných dokumentů do znakové sady databáze
- Druhy fulltextových indexů
  - **CONTEXT**
    - základní typ indexu pro vyhledávání v textových datech
    - vhodný pro větší dokumenty
    - synchronizace indexu s daty je nutno provést explicitně

    - **CTXCAT**
        - vhodný pro menší dokumenty a jejich úryvky
        - může být zkombinován s dalšími netextovými sloupci pro kombinované dotazování
        - synchronizace indexu s daty se provádí automaticky se změnami v tabulce
    - **CTXRULE**
        - postaven na množině předdefinovaných dotazů
        - slouží pro klasifikaci dokumentů do skupin podle toho, kterým dotazům vyhovuje
- Uložení dokumentů
    - **NORMAL_DATASTORE** - text je v jednom sloupci jednoho řádků
    - **MULTI_COLUMN_DATASTORE** - text ve více sloupcích jednoho řádku
    - **URL_DATASTORE** - text je na internetu, dostupný přes URL ve sloupci
- Příklad vytvoření indexu nad textovým sloupcem:

```
CREATE INDEX myindex ON doc(htmlfile)
INDEXTYPE IS ctxsys.context
PARAMETERS('datastore ctxsys.default_datastore
filter ctxsys.null_filter
section group ctxsys.html_section_group');
```

- Spolu s novými typy indexů databáze implementují nové operátory pro porovnávání dotazu s textem
- Operátory vrací číslo - očekávanou míru shody obsahu textu s tazatelovými požadavky

  Příklad dotazu:

```
strSql = "SELECT SCORE(1), file_name, filesize FROM my_doc " & _
    WHERE CONTAINS(content," & Search.Value & ", 1) > 0 " & _
    ORDER BY SCORE(1) DESC"
```

16. Možnosti tvorby datových skaldů a metody dolování znalostí

Základní problémy u běžných transakčních databázových systémů:

- nedosažitelnost dat skrytých v transakčních systémech
- dlouhá odezva při plnění komplikovaných dotazů
- složitá, uživatelsky nepříjemná rozhraní k databázovému softwaru
- cena v administrativě a složitost v podpoře vzdálených uživatelů
- soutěžení o počítačové zdroje mezi transakčními systémy a systémy podporujícími rozhodování

Cesta k řešení těchto problémů = datové sklady, tzv. Data Warehouse – DW

Datawarehouse

DW označují db architekturu používanou pro údržbu historických dat, která jsou získána z jedné nebo více operativních db. Typicky, tato data jsou vyčištěna a restrukturována pro podporu dotazů, agregací a analýz.

Klíčové: integrace vlastních + externích dat

Modely & Operátory warehousu
Datové Modely

- relační
- hvězdice & vločky
- krychle

Operátory

- slice & dice (řez & výřez)
- roll-up, drill-down (srolování, zavrtání)
- pivoting
- další

Komponenty DW

• **akvizice dat a jejich integrace** do DW (generátory kódu, replikace dat, middleware, kopírování)

• **řízení dat** (databázový server + služby: archivace, autorizace, zálohování a zotavení z chyb, provoz, monitorování a ladění, řízení zdrojů)

• **slovník informací** (metadata a přístup k nim)

• **přístup k datům** a komponenty dodání dat (db middleware, OLAP, multidimenzionální data, data řízená časem a událostmi)

Velká diskuze: E-R vs. multidimenzionální přístupy

2 přístupy k Datovému Modelování:

- konceptuální struktury založené **na tabulkách** (dimenzionální a tabulky faktů) organizovaných do tzv. hvězdicových schémat,
- konceptuální struktury jsou založeny **na hyperkostkách** (kostkách, multidimenzionálních polích), které reprezentují data jako multidimenzionální strukturu.

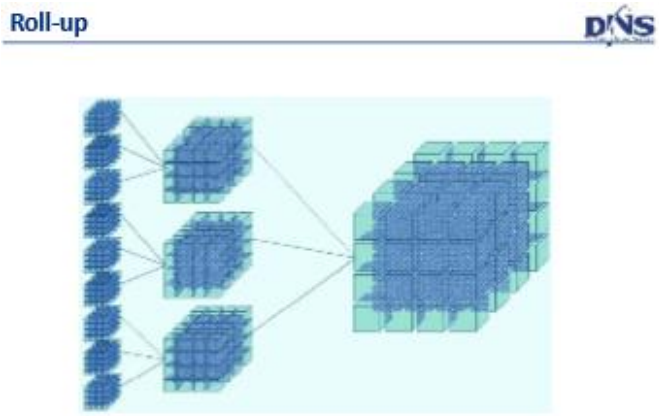
Dále o DW

- samostatný informační systém postaven na již pořízených datech, určen především k jejich analýze
- architektura založená na ~~relačním~~ SŘBD, která se používá pro údržbu historických dat získaných z databází operativních dat, jenž byla sjednocena a zkontrolována před jejich použitím v databázi DW
- data z DW jsou aktualizována v delších časových intervalech, jsou vyjádřena v jednoduchých uživatelských pojmech a jsou sumarizována pro rychlou analýzu
- DW je obrovská databáze obsahující data za dlouhé časové období
- často slučuje data z více rozdílných zdrojů, které mohou obsahovat data různé kvality nebo používat nejednotné formáty a reprezentace
- objemově zabírá stovky GB až několik TB
- nemusí být databází v běžném smyslu, tj. pro přesné provádění transakcí
- je určen pro rychlé vyhledávání
- nejsou kladeny nijak důrazné požadavky na správnost a úplnost dat

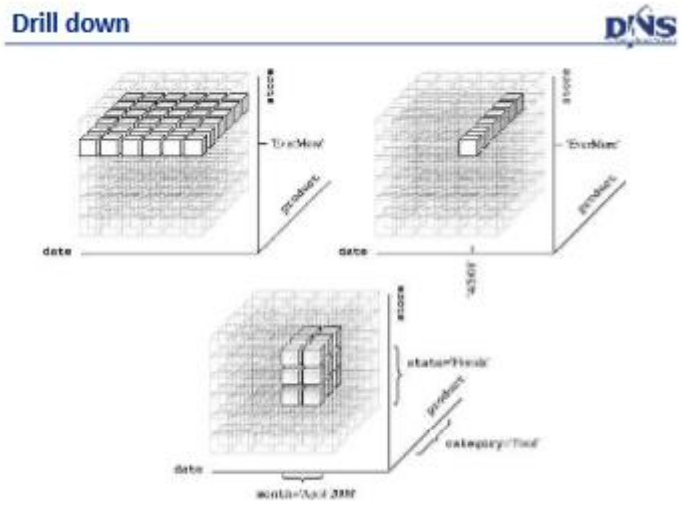
Charakteristika

- data jsou uložena na různých místech ve formě relačních tabulek
  - uživatelé mohou tabulky jen číst
  - zapisovat může aktualizační program pravidelně udržující tabulky
- dotazy jsou většinou komplexní
  - podporují tzv. on-line analytické zpracování (OLAP)
    - výrazně se liší od on-line transakčního zpracování (OLTP)
    - operační databáze je přizpůsobena pro podporu OLTP
      - složité OLAP dotazy by vyústily do nepřijatelné odezvy
    - **<u>typické OLAP operace:</u>**

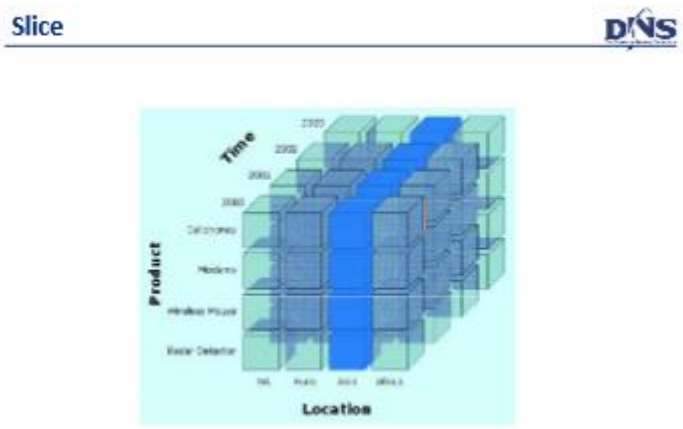
A) roll-up (zvýšení stupně agregace)

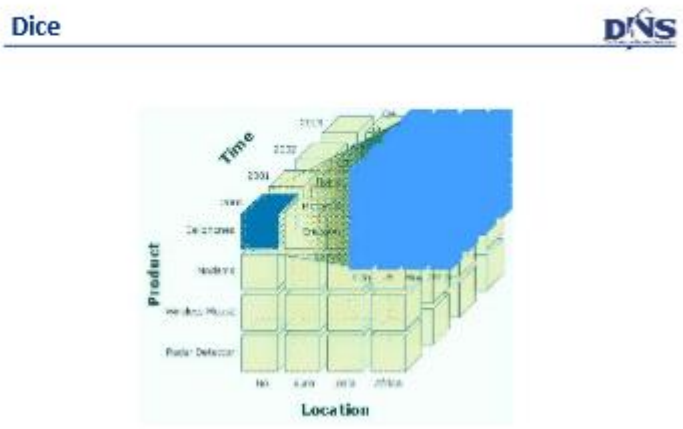


B) drill-down (snížení stupně agregace)

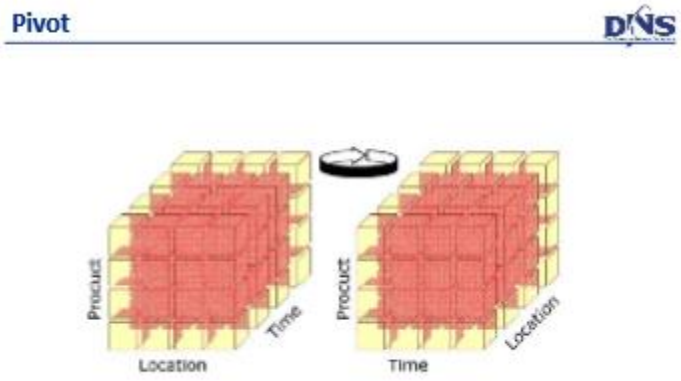


C) slice-and-dice (selekce a projekce)





D) pivot (přeorientování vícerozměrného pohledu na data)



- na základě dotazu se pospojují potřebná data do vícerozměrné tabulky (nebo více tabulek), do kterých lze klást SQL dotazy
- pro častější dotazy si uchovávají předem připravené vícerozměrné tabulky
- zátěž je většinou způsobena složitými dotazy, jež přistupují k miliónům záznamů a provádějí množství operací
- data bývají modelována vícerozměrně
    - v obchodním data warehouse mohou těmito rozměry být např. čas prodeje, místo prodeje, prodavač, výrobek, …
    - rozměry mohou být i hierarchické např. čas prodeje jako den-měsíc-čtvrtletí-rok, zboží jako výrobek-kategorie-průmysl
    - spojení více tabulek pomocí odkazu na řádky jednotlivých tabulek
    - používají speciální organizaci dat, přístupové a implementační metody, jež obecně nejsou v komerčních databázových systémech určených pro OLTP podporovány

Databázový systém – OLTP (Online Transaction Processing Systems)

- zákaznicky orientovaný
- aktuální data -- lze považovat i za slabinu, při výpadku (chybě), vznikají ztráty pro byznys
- ER schéma
- sofistikované atomické transakce i přes několik systémů(bank, po síti,...)
- velikost DB až několik GB
- jednoduché a efektivní
- příkladem je bankomat

DataWarehouse – OLAP (Online analytical Processing)

- orientovaný na trh, rychlé (oproti OLTP) získání výsledků na analytické dotazy
- historická data, multidimenzionální datový model
- agregovaná data (nenormalizovaná=redundantní)
- schéma hvězdy či vločky
- převážně pouze čtení
- velikost až TB
- použití: byznys reporty o prodeji, marketing, management reporty, rozpočty, finanční předpovědi a reporty

OLAP (**Online Analytical Processing**) je technologie uložení dat v databázi, která umožňuje uspořádat velké objemy dat tak, aby byla data přístupná a srozumitelná uživatelům zabývajícím se analýzou obchodních trendů a výsledků (Business Intelligence). Způsob uložení dat se svým zaměřením liší od běžněji užívaného **OLTP** (**Online Transaction Processing**), kde je důraz kladen především na snadné a bezpečné ukládání změn v datech v konkurenčním (víceuživatelském) prostředí.

Použití DW

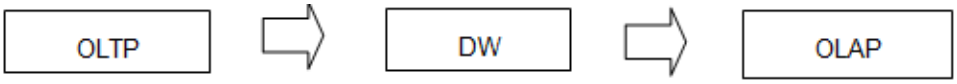
- prezentace dat
- testování hypotéz
- objevování nových informací

Architektura DataWarehouse

tři úrovně:

- klient
- OLAP server (MOLAP/ROLAP server)
- databázový server DW

- data lze organizovat v tzv. multidimenzionálním datovém modelu
  - odlišný od modelu relačního
  - odpovídá mu specializovaný software, multidimenzionální SŘBD (MDD)
  - model připomíná techniku spreadsheet ve více než dvou rozměrech
  - data jsou implementována pomocí vícerozměrných polí, jejichž dimenze odpovídají dimenzím podnikání organizace

- navržení a vytvoření DW je proces skládající se z následujících bodů:
  - definovat architekturu, umístění a rozčlenění dat a fyzickou organizaci
  - naplánovat kapacitu, vybrat OLAP servery a nástroje
  - spojit servery, klientské nástroje, zdroje přes gatewaye, drivery ODBC, ...
  - navrhnout schéma a pohledy, přístupové metody, některé složité dotazy
  - mít skripty pro získávání, čištění, transformaci, ukládání a aktualizaci dat
  - vytvořit koncové uživatelské aplikace
  - spustit data warehouse i aplikace
- vytvoření je složitý proces trvající mnohdy i několik let
- mnoho organizací proto používá Data Mart umožňující rychlejší práci
- 

Datová tržiště (Data Mart)

- DW slouží jako základna pro extrakci množin dat, resp. jejich agregaci do dílčích (replikovaných) MDD (Multidimenzionální DB)
  - MDD může pro DW sloužit ve dvou rolích
    - ”front-end” pro DW a poskytovat uživateli služby pro realizaci analytického zpracování (DW/OLAP)
    - “front-end” jednomu (několika) systémům OLTP - alternativa za DW, tj. poskytnout uživateli s OLTP data analytickým způsobem (OLTP/OLAP) – jde vlastně o datové tržiště

Systém OLAP (OnLine Analytical Processing)

- na databázové stroje jsou kladeny specifické požadavky
- objem zpracovávaných dat
- transakční systém o velikosti gigabajtů dosáhne použitím jen jedné dimense velikosti desítek či stovek gigabajtů
- rychlost odezvy analytického systému je důležitá
- počet uživatelů současně pracujících s databází není zajímavý
  - počet pracovníků vyššího managementu je omezen
  - pro pracovníky nižších stupňů bývají údaje z datových skladů převedeny do menších specializovaných databází – datových tržišť
- s těmito omezeními se vyrovnává dvojím způsobem
- uzpůsobení stávajících systémů pro práci s vícerozměrovými daty

- přidáním modulu, který to zajišťuje a prostředků pro jeho ovládání
- v lepším případě mění způsob uložení dat, v horším“překládá” operace s vícedimenzionálními daty na operace s daty relačními
- vytvoření speciálního systému správy dat, určeného pouze pro OLAP
- umožňuje provést maximum optimalizací vzhledem k nárokům, jež jsou kladené analytickým způsobem práce - převažující způsob

Programy pro vytváření a plnění databáze (ETL - viz SI)

- převodní programy
  - načtení data z několika databází, či souborů a udělat z nich novou databázi, agregace se musí naprogramovat
- systémy znázorňující převodu dat graficky a administrátor dat namapuje zdrojová data do struktur vytvářeného datového skladu
  - výsledkem jsou buď programy (scripty) nebo přímo vykonání funkce
- moduly pro plánování jednotlivých akcí

Nástroje pro práci s daty - poslední trendy v architektuře klient/server

- nabízejí variantu tenkého klienta v podobě HTML prohlížeče

Reporting, monitorování, ad-hoc dotazy

- programy umožňující kladení dotazů a formátování odpovědí
  - nejčastěji jde o vizuální dotazovací nástroje
  - makra v tabulkovém procesoru
  - uživatelské rozhraní různě propracované:
    - zadání seskupení výsledku podle různých kritérií
    - formální kontrola dotazů
    - vytváření slovníků a metadat

MOLAP - Multidimenzionální OLAP

- datová krychle (obsahuje fakta)
- hierarchické dimenze (částečné či totální uspořádání)
  - **vločkové schéma** -- hlavní tabulka faktů je v relaci s dimezionálními tabulkami, přes cizí klíče, dimenzionální tabulky mohou být také v relaci s dalšími subdimenzionálními tabulkami podobně jako hlavní tabulka faktů; vytváří hierarchie dimenzí
  - **hvězdové schéma** -- je speciální případ vločkového, dimenzionální tabulky již nejsou v relaci s dalšími subdimenzionálními tabulkami; žádné hierarchie, jednodušší

ROLAP – Relační OLAP

- na relační architektuře založený model DW strukturou propojených DB tabulek - Relační OLAP (ROLAP) – pomalejší zpracování než MOLAP
- užívá relační nebo rozšířený relační DBMS, např server METACUBE Informix, pracuje s relačními tabulkami uspořádanými do hvězdy/vločky, adresuje pomocí klíče, data jsou neagregovaná)

Metody dolování -dataminingu (jiný zdroj, ne z přednášek)

1. popis dat (asi asociace – asociační pravidla)
    - najít charakteristiky dat
    - často ve spojení s dalšími metodami, např. segmentací (hledám charakteristiky segmentů)
2. shlukování (clustering)
    - hledání a vytváření kategorií, do kterých jsou data uspořádána
    - nemusí být disjunktní, jeden objekt klidně ve více skupinách
    - tím se shlukování liší od klasifikace!!
    - **"mám shluk dat se společnými vlastnostmi a celý tento shluk se označní jako nějaká kategorie" tzn kategorie nejsou předem známy**

3. klasifikace
    - rozdělování objektu do jednotlivých tříd podle cílového atributu (nespojitá veličina)
    - **předpokládá se, že jednotlivé třídy, do kterých objekty rozděluji, jsou předem známé**
    - **"škatulkování" dat do předem daných tříd**
4. regresní analýza
    - obdoba klasifikace; cílový atribut je spojitá veličina
5. analýza závislosti
    - hledám takový model, který popisuje vztahy mezi daty
    - např. analýza spotřebního koše (co se s čím často prodává)
    - hrozí, že naleznu neexistující závislost

Zavádění datového skladu

strategie velkého třesku (hub architecture)

- buduji celý sklad naráz
- vytvoření celopodnikového datového modelu
- datová tržiště modelována dimenzionálně a napojena na samotný DW

strategie postupného budování datových tržišť (bus architecture)

- z nich pak dohromady vznikne datový sklad
- výhodou je jednoduchost a srozumitelnost datového modelu, iterativnost procesu

//vynechana jedna část metod dolovani