

## Relační databáze, entity, ER diagramy

*Procesy návrhu databáze (báze dat):*

- stanovení účelu databáze (které údaje budou evidovány)
- stanovení potřebných tabulek (rozdělení subjektů do tabulek)
- stanovení názvu polí

*Databázové modely* - relační databázový model, objektový databázový model, deduktivní model, síťový model, hierarchický model, dokumentové databáze

*Relační databázový model*

- datové prvky jsou organizovány v tabulkách
- každá tabulka reprezentuje entitu aplikace
- každý řádek je instance dané entity, relace spojují řádky ve dvou tabulkách
- SQL tvoří jednotné uživatelské rozhraní pro přístup a práci s daty

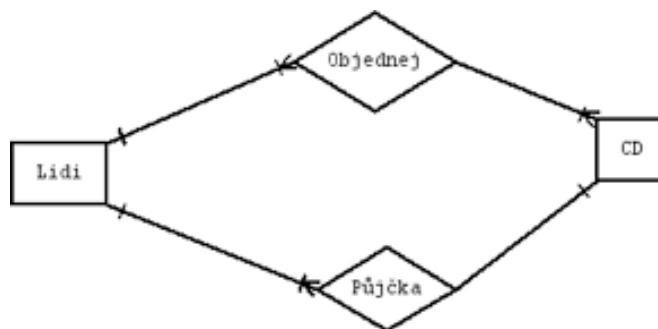
př. kočky – entita; 1 kočka – 1 řádek; studenti- druhá entita, relace – student vlastník kočky

*Objektový databázový model*

Entita (jako třída) - objekt, který popisujeme (vlastnosti a chování); Data - konkrétní údaje v tabulce

*ER diagram*

Pro návrh a zápis vztahů mezi jednotlivými entitami databáze byl vytvořen model E-R diagramů (Entity Relationship Diagrams). Typový E-R diagram je obvykle obrázek podobný klasickému vývojovému diagramu, v němž jsou entity znázorněny obdélníky, vztahy kosočtverci a mezi nimi vedou čáry, aby se poznalo, co k čemu patří.



*Kardinalita vztahu*

- 1:1 - do vztahu vstupuje nejvýše jedna hodnota obou zúčastněných entit
- 1:N - nejvýše jedna hodnota jedné z entit a neomezený počet hodnot druhé entity
- M:N - do vztahu vstupuje neomezený počet hodnot obou zúčastněných entit

## DDL a základní datové typy

DDL – Data Definition Language – slouží pro vytvoření struktury databáze

TYP – numerické (integer), datové (char(255), varchar(255), blob), datum a čas (date, time, timestamp), uživatelské

## DDL p íkazy

CREATE TABLE jmeno\_tabulky (jmeno\_sloupce TYP, integritní omezení)

Integritní omezení jsou nutná pro udržení konzistence dat. Je nutné zajistit, aby se do ní dostali jen data, která tam patří a neztratila se data, která nemají.

DEFAULT – implicitní hodnota; NOT NULL; PRIMARY KEY

DROP TABLE jmeno\_tabulky

ALTER TABLE jmeno\_tabulky ADD jmeno\_sloupce TYP integritní\_omezení  
DROP jmeno\_sloupce  
ALTER jmeno\_sloupce

## DML, jazyk SQL, SELECT, funkce, spojování tabulek, vnořené dotazy a sekvence

SQL – structur query language, deklarativní jazyk

DML – data manipulation language, příkazy SELECT, INSERT, DELETE, UPDATE

*SELECT* - výběr dat z databáze, na straně klienta se zadává a odesílá se na server, projekce – omezení na určité sloupce ( \* , určité sloupce)

SELECT <projekce> FROM <tabulky> WHERE <podminka> GROUP BY <seskupování sloupce> HAVING <podmínky> ORDER BY <řazení sloupce>

FROM – *spojování tabulek* – ZAMESTNANCI, PLATY

jedná se o přirozený součin – kartézský součin, na výpisu budou všechny údaje z jedné i druhé tabulky (vnitřní spojení)

vnější spojení – zaměstnanci LEFT JOIN plat ON id=id2

z první tabulky se vezmou všechny řádky a z druhé se přiřadí vše co odpovídá id=id2, kde se neshodě, přiřadí se NULL

WHERE – které sloupce by měly být na výstupu  
může obsahovat vnořené SELECT  
LIKE, IN, ALL, SOME

GROUP BY – seskupování dotazů

uvádíme jména sloupců, podle kterých chceme seskupovat  
COUNT, SUM, MIN, MAX, AVG

*Sekvence* – aby každý vložený záznam měl unikátní číslo

CREATE SEQUENCE nazev [START WITH cislo][INCREMENT cislo][MIN VALUE cislo]

-aktuální hodnota v sekvenci

SELECT \* FROM tabulka WHERE ID<SEQ.CURVAL

INSERT INTO ZAKAZNICI (ID,jmeno) VALUES (SEQ.NEXTVAL,'franta')

DROP SEQUENCE jmeno

## Klíče, primární klíče, indexy

KLÍČ – identifikuje záznam v tabulce - může i více řádků najednou

SUPERKLÍČ – taková množina atributů, která jednoznačně identifikuje řádek v tabulce

KANDIDÁTNÍ KLÍČ – minimální superklíč

takový superklíč, že když odebereme jeden z atributů, ztrácíme superklíč

PRIMÁRNÍ KLÍČ – jeden vybraný z kandidátních klíčů

ALTERNATIVNÍ KLÍČ – všechny ostatní klíče, které nejsou primární

ALTER TABLE jméno tabulky ADD PRIMARY KEY (jméno\_sloupec)

### Indexy

pokud je vytvořen index, v první řadě se prohledávají pouze ty sloupce, které jsou označeny indexy

zrychlují výběr, ale zpomalují vkládání  
projevují se až při větším počtu dat

CREATE [unique] INDEX jmeno ON tabulka (sloupec)

DROP INDEX jmeno

## NORMÁLNÍ FORMY

uspořádání dat, abychom se vyvarovali různým anomáliím

*0NF* (nultá normální forma):

Tabulka je v nulté normální formě právě tehdy, existuje-li alespoň jedno pole, které obsahuje více než jednu hodnotu.

*1NF* (první normální forma):

První, nejjednodušší, normální forma (značíme 1NF) říká, že všechny atributy jsou atomické, tj. dále již nedělitelné (jinými slovy, hodnotou nesmí být relace). Mějme např. tabulku ADRESA, která bude mít sloupce JMÉNO, PŘÍJMENÍ a BYDLIŠTĚ. Naplnění tabulky nechť odpovídá reálnému světu:

JMÉNO	PŘÍJMENÍ	BYDLIŠTĚ
jan	novák	Ostravská 16, Praha 16000
petr	nový	Svitavská 8, Brno 61400
jan	nováček	Na bradlech 1147, Ostrava 79002

Pokud bychom v této tabulce chtěli vypsát všechny pracovníky, jejichž PSČ je rovno určité hodnotě - atribut BYDLIŠTĚ není atomický, skládá se z několika částí: ULICE, ČÍSLO, MĚSTO a PSČ. Správný návrh tabulky v 1NF:

JMÉNO	PŘÍJMENÍ	ULICE	ČÍSLO	MĚSTO	PSČ
jan	novák	Ostravská	16	Praha	16000
petr	nový	Svitavská	8	Brno	61400
jan	nováček	Na bradlech	1147	Ostrava	79002

### 2NF (druhá normální forma)

Tabulka splňuje 2NF, právě když splňuje 1NF a navíc každý atribut, který není primárním klíčem, je na primárním klíči úplně závislý. To znamená, že se nesmí v řádku tabulky objevit položka, která by byla závislá jen na části primárního klíče. Z definice vyplývá, že problém 2NF se týká jenom tabulek, kde volíme za primární klíč více položek než jednu. Jinými slovy, pokud má tabulka jako primární klíč jenom jeden sloupec, pak 2NF je splněna triviálně.

Nechť máme tabulku PRACOVNÍK, která bude vypadat následovně: (atribut ČÍS\_PRAC značí číslo pracoviště, kde daný pracovník pracuje, atribut NÁZEV\_PRAC uvádí jméno daného pracoviště)

ČÍSLO	JMÉNO	PŘÍJMENÍ	ČÍS_PRAC	NÁZEV_PRAC
1	jan	novák	10	studovna
2	petr	nový	15	centrála
3	jan	nováček	10	studovna

Jaký primární klíč zvolíme v této tabulce? Pokud zvolíme pouze ČÍSLO, je to špatně, neboť zcela určitě název pracoviště, kde zaměstnanec pracuje, není závislý na číslu pracovníka. Takže za primární klíč musíme vzít dvojici (ČÍSLO, ČÍS\_PRAC). Tím nám ovšem vznikl nový problém. Položky JMÉNO, PŘÍJMENÍ a NÁZEV\_PRAC nejsou úplně závislé na dvojici zvoleného primární klíče. Ať tedy děláme, co děláme, nejsme schopni vybrat takový primární klíč, aby tabulka splňovala 2NF. Jak z tohoto problému ven? Obecně převedení do tabulky, která již bude splňovat 2NF, znamená rozpad na dvě a více tabulek, kde každá už bude splňovat 2NF. Takovému "rozpadu" na více tabulek se odborně říká dekompozice relačního schématu. Správně navržené tabulky splňující 2NF budou vypadat takto: (tabulka PRACOVNÍK a PRACOVIŠTĚ)

ČÍSLO	JMÉNO	PŘÍJMENÍ	ČÍS_PRAC	ČÍSLO	NÁZEV
1	jan	novák	10	10	studovna
2	petr	nový	15	15	centrála
3	jan	nováček	10		

Dále si všimněte, že pokud tabulka nespĺňuje 2NF, dochází často k redundanci. Konkrétně v původní tabulce informace, že pracoviště číslo 10 se jmenuje "studovna", byla obsažena celkem dvakrát. Redundance je jev, který obvykle nesplnění 2NF doprovází. O tom, že redundance je nežádoucí, netřeba pochybovat.

### 3NF (druhá normální forma)

Relační tabulky splňují třetí normální formu (3NF), jestliže splňují 2NF a žádný atribut, který není primárním klíčem, není *tranzitivně závislý* na žádném klíči. Nejlépe to opět vysvětlí následující příklad. Mějme tabulku PLATY, která bude vypadat takto:

ČÍSLO	JMÉNO	PŘÍJMENÍ	FUNKCE	PLAT
1	jan	novák	technik	15000
2	petr	nový	vedoucí	21500
3	jan	nováček	správce	17500

Pomineme zatím fakt, že tato tabulka nespĺňuje ani 2NF, což je základní předpoklad pro 3NF. Chci zde jen vysvětlit pojem *tranzitivní závislosti*. Nebudeme přemýšlet, co je primární klíč, na první pohled vidíme, že konkrétně atributy *JMÉNO*, *PŘÍJMENÍ* a *FUNKCE* závisí na atributu *ČÍSLO* (ten by nejspíš byl primárním klíčem). Dále můžeme vidět, že atribut *PLAT* z *ejm* je funkce závislý na atributu *FUNKCE* a pokud vezmeme v úvahu, že *ČÍSLO*->*FUNKCE* a *FUNKCE*->*PLAT*, dostaneme díky tranzitivitě, že *ČÍSLO*->*PLAT*.

ČÍSLO	JMÉNO	PŘÍJMENÍ	FUNKCE	FUNKCE	PLAT
1	jan	novák	technik	vedoucí	21500
2	petr	nový	vedoucí	správce	17500
3	jan	nováček	správce	technik	15000

Z hlediska základních tří normálních forem, jsou tyto dvě tabulky již v pořádku. Z praktického hlediska je vhodnější použít nějaký číselník funkcí, abychom splnili podmínku, že primární klíč v tabulkách má být co nejkratší délky. Nejlepší zápis je tedy následující:

ČÍSLO	JMÉNO	PŘÍJMENÍ	CIS_FUN	ČÍSLO	FUNKCE	PLAT
1	jan	novák	127	121	vedoucí	21500
2	petr	nový	121	156	správce	17500
3	jan	nováček	156	127	technik	15000

## JAZYK PL/SQL – kurzory, procedury, funkce, triggerry

procedure language - rozšíření pro ORACLE

PL/SQL je strukturovaný jazyk, možnost si tedy psát procedury a funkce v souvislých logických blocích

```

DECLARE {nepovinné}
    -deklarace, definice
BEGIN
    -výkonová část
EXCEPTION {nepovinné}
    -vyjímky
END;
/          -uložit a zkompilovat
    
```

```

DECLARE
    -identifikátor TYP
    -typy stejné jako v SQL
    -operátor přiřazení :=
    -odkaz na typ v tabulce -   proměnná TABULKA.SLOUPEC%TYPE
    
```

```

BEGIN
    -možnost dalších bloků DECLARE, BEGIN, END
    
```

```

        SELECT projekce INTO seznam_promennych FROM tabulka
    
```

```

        IF podmínka THEN
    
```

```

        ...
    
```

```

        END IF;
    
```

```

        CASE vyraz
    
```

```

            WHEN hodnota1 THEN
    
```

```

            WHEN hodnota2 THEN
    
```

```

        ELSE ...
    
```

```

        END CASE;
    
```

## Databázové systémy

```
LOOP
  (EXIT,EXIT WHEN podmínka)
END LOOP;

FOR pocitadlo IN min..max LOOP
...
END LOOP;
```

### *kurzory*

-SELECT v PL pro více řádků

```
DECLARE
CURSOR nazev IS SELECT jmeno,plat FROM tabulka;
radek_kurzoru nazev%ROWTYPE;

BEGIN
OPEN nazev;
FETCH nazev INTO promenna1,promenna2;  -nutné deklarovat na začátku

FOR radek_kurzoru IN nazev LOOP
-- příkazy zde uvedené se provedou pro každý řádek
-- kurzoru
END LOOP;

CLOSE nazev;
```

### *Procedura*

```
CREATE [OR REPLACE] PROCEDURE nazev [(parametry)] AS
[DECLARE]
BEGIN
[EXCEPTION]
END;
/
```

parametry - jmeno IN,OUT,IN OUT datovy\_typ

```
EXECUTE nazevprocedury
EXECUTE nazevprocedury (1,2,3)
```

### *Funkce*

```
CREATE [OR REPLACE] PROCEDURE nazev [(parametry)] RETURN typ_navr_hodn AS
[DECLARE]
BEGIN
RETURN hodnota
[EXCEPTION]
END;
/
```

parametry funkce mohou byt pouze IN

*TRIGGER* - spouští procedury nebo funkce, kdy chceme

Databázový trigger je procedura PL/SQL spojená s tabulkou. Trigger se automaticky provede při provádění některého příkazu SQL, je-li splněna podmínka triggeru. Použití – zálohování tabulky před použitím DELETE.

```
CREATE [OR REPLACE] TRIGGER nazev {BEFORE|AFTER} [INSTEAD OF (namísto)]
udalost ON tabulka [FOR EACH ROW][WHEN omezeni]
[DECLARE]
BEGIN
[EXCEPTION]
END;
/
```

```
CREATE OR REPLACE TRIGGER nazev BEFORE DELETE ON tabulka FOR EACH
ROW
BEGIN
INSERT INTO zaloha VALUES(:OLD.id,:OLD.jmeno);
END;
/
```