Konceptuální modelování

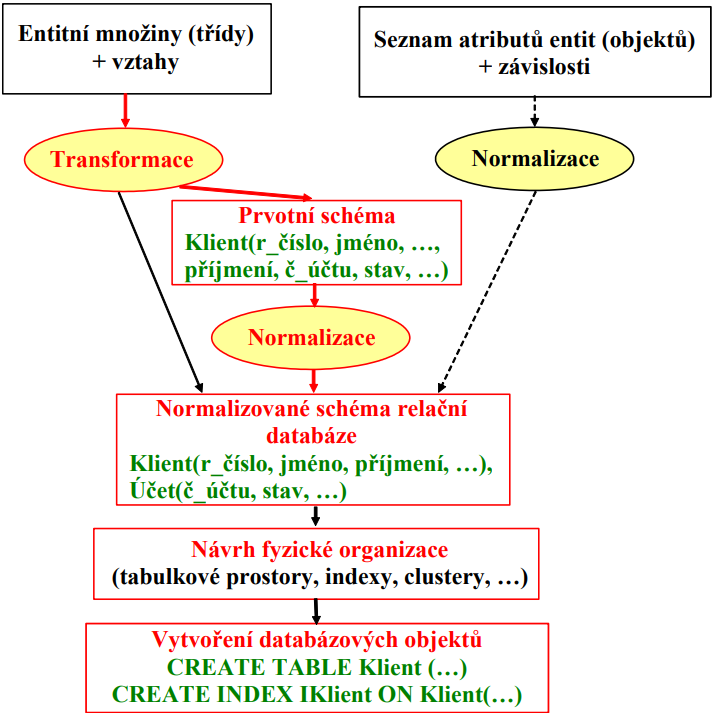
Co to je a proč to děláme?

Fáze datové/objektové analýzy, která využívá modelů založených na objektech **aplikační domény** – první musím dobře prozkoumat, co daný program (zákazník…) potřebuje ukládat.

Požadavky -> Konceptuální návrh -> ER diagram (konceptuální schéma) -> Logický návrh -> Tabulky (logické schéma) -> Fyzický návrh -> Fyzické schéma

Dva různé přístupy k modelování:

**Strukturovaný přístup** (*klasický*): východiskem pro návrh databáze je **ER model**.  
**Objektový přístup**: východiskem pro návrh databáze je **diagram tříd** (který vychází zase z jiných modelů – případů užití, interakce, stavů).



Dvě možnosti strukturovaného návrhu:  
ERD → transformace → normalizace,  
nebo seznam atributů → normalizace.

# ER modely

Založen na dvou množinách základních objektů: **entit** a **vztahů** mezi nimi. Tyto objekty mají navíc **atributy** (ale nazývat ty modely ERA se nikomu nechce).  
Popisuje data **v klidu** – neukazuje, jaké operace budou probíhat.

**Entita**: *jeden konkrétní* objekt *z reálného světa*, typicky rozlišitelná od jiných objektů, chceme o nich informace v databázi. Entity „stejného druhu“ modelujeme pomocí **typu entity** (= entitní množina).

**Atribut**: vlastnost entity nebo vztahu, která nás v kontextu problému zajímá a jejíž hodnotu chceme mít uloženou v databázi.  
**Doména atributu**: obor hodnot atributu.

**Vztah**: *jedna konkrétní* asociace mezi dvěma nebo více entitami. Vztahy „stejného druhu“ modelujeme pomocí **typu vztahu** (= vztahová množina) – definuje typ vztahů jednoho nebo více typů entit.

**Klíč** (v oblasti databází pak *primární klíč*): atribut, který má v rámci dané množiny entit u každé entity unikátní hodnotu.

### Vlastnosti atributů

Mohou být **jednoduché** (*simple*) nebo **složené** (*composite*).  
V ERD je v pořádku, že tam jsou – při tvorbě databáze samotné ale chceme, aby byly atributy **atomické**. Pokud mi hodnota stačí „v celku“ a nepotřebuji s jednotlivými částmi složeného atributu pracovat zvlášť, mohu ho tam mít jako složený, jinak se rozděluje na jednotlivé složky. (Dále viz normalizace.)

Mohou být **jednohodnotové** (single-valued) a **vícehodnotové** (multiple-valued).  
Při tvorbě databáze opět třeba zajistit atomicitu – vícehodnotové atributy se obvykle mění na provázané tabulky.

Mohou povolovat prázdnou hodnotu – být **nullable**: může mít význam chybějící (existuje, ale neznáme ji), nebo neznámé (nevíme, jestli existuje) hodnoty.

Mohou být **odvozené**.

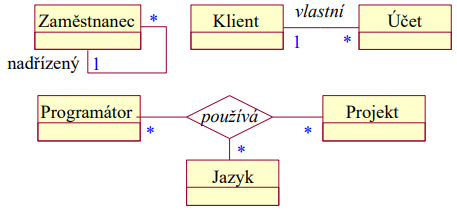
Mohou být **klíčem**: mít unikátní hodnotu, v teorii ještě *minimální* (viz dále). Můžou být jednoduché i složené.

### Vlastnosti (typů) vztahů

V modelech určujeme **jméno typu vztahu** – vyjadřují význam vztahu („klient *vlastní* účet“). Občas se mohou hodit **jména role** („klient *vlastní* účet“ → u klienta role „vlastník“) – většinou jsou ale jasné z jména typu vztahu.

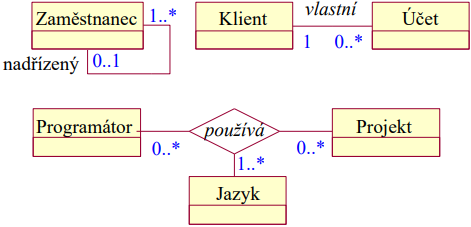
U vztahu určujeme **stupeň**: typické jsou vztahy **binární**, případně **unární** (reflexivní). Občas se vyskytnou i **ternární** vztahy a teoreticky mohou být jakkoliv *n*-nární, ale v praxi se to nehodí.

Určujeme (maximální) **kardinalitu**: maximální počet vztahů daného typu, ve kterých může participovat jedna entita. Určujeme ji pro každý „konec vztahu“. Její správné určení je obvykle **velmi důležité**!



U unárních vztahů se hodí použít to jméno role, abychom věděli, z jakého konce se na kardinality dívat.

Určit můžeme také **členství** (= **minimální kardinalitu**). Obvykle bývá buď 0, nebo 1.



Vztahy mohou mít **atributy**.

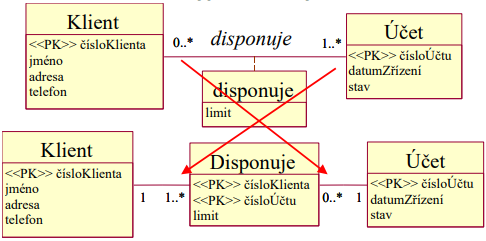
### Rozšíření klasického ERD

Můžeme rozlišovat **silné** a **slabé entity** (typy entit):  
Typ **silné** entity může existovat nezávisle na ostatních, typ **slabé** entity je závislý na **jednom** jiném typu entity. Slabá entita je určena **klíčem dominantního typu** a **diskriminátorem** (dílčím klíčem).

Můžeme využívat vztahu **generalizace** a **specializace** (vychází z dědičnosti v objektově orientovaném návrhu) – občas se označuje jako **ISA vztah** (sth *is a* sth, „spořicí účet *is a* účet“). Klíč typů entit nižší úrovně je stejný jako klíč vyššího typu.  
Mezi jednotlivými typy entit nižší úrovně mohou být dva typy omezení:  
**příslušnost – disjunktní** (účet může být jen spořicí, nebo jen běžný), nebo **překrývající se**;  
a **úplnost** – **totální**, nebo **částečná** (může existovat i entita vyšší úrovně, která není ani jednou z entit nižší úrovně – účet, který není ani spořicí, ani běžný).

### Doporučení

Celkový systém nikdy nezahrnujeme do ERD.  
Typ entity, nebo atribut? (Atribut *barva* u výrobku, nebo tabulka s *číselníkem barev*?) Je-li hodnota důležitá, i když neexistuje žádná entity s touto hodnotou jako vlastností, pak bychom ji měli modelovat jako entitu.  
Vztahy many-to-many můžeme rovnou v ERD nahradit *vazebním typem entity*. Kardinality se propisují do kříže:



### Časté chyby

* Uvádění „přesné“ kardinality.
* Vazební typ entity jako typ slabé entity (typ slabé entity je vždy závislý **pouze na jednom** dominantním typu.
* Generalizace a specializace vyjádřená pomocí tabulek a vazeb (ne, to se dělá až při transformaci na schéma databáze, může se to udělat různými způsoby).
* Atribut vs. typ entity, specializace bez atributů navíc.
* Vztah vs. atribut reprezentující vztah:  
    
  Má smysl mít tam ten atribut *vedoucí*? Už ho dostatečně reprezentuje ten vztah. (Ten tam naopak být určitě musí!) Není to ani vyloženě chyba.

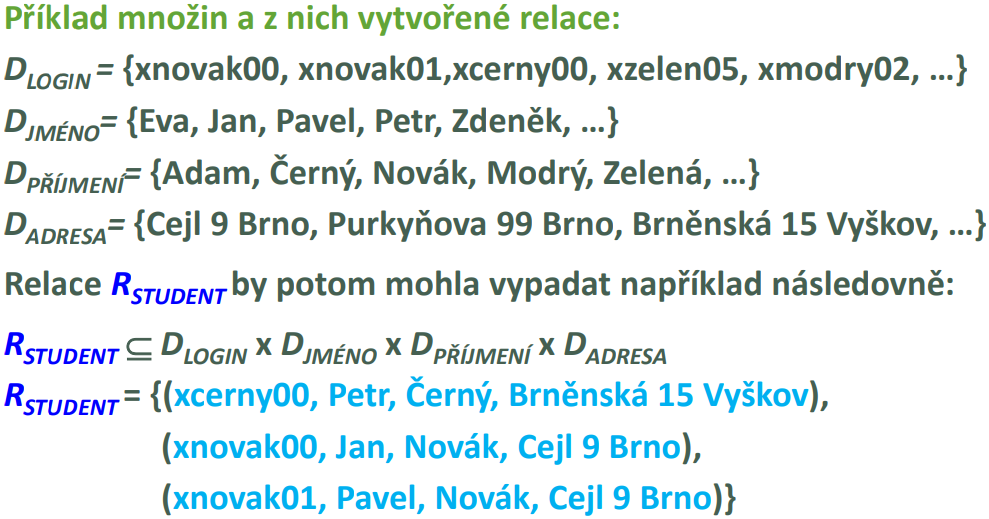
Relační model dat

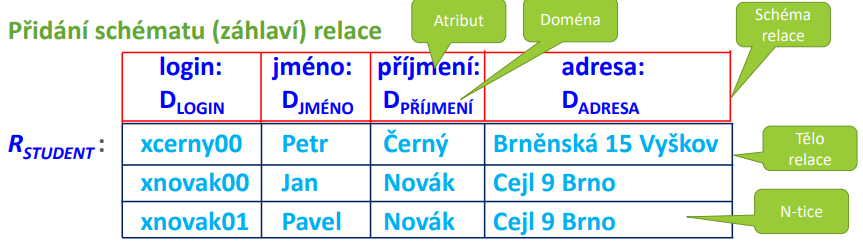
Poprvé navržen už v roce 1970, první implementace 1975, využíván dodnes. Přínosy:

* Oddělení logické struktury dat od implementace (viz první přednášku);
* Poskytnutí matematické podpory pro manipulaci s daty (**relační algebra**);
* Poskytnutí matematické podpory k omezení redundance při návrhu struktury (**normalizace** – teorie závislostí a normální formy).

# Relační struktura dat

Relační databáze je v podstatě kolekce **relací –** podmnožinou kartézského součinu několika množin.





**Doména**: pojmenovaná množina **skalárních hodnot** téhož typu.  
**Skalární hodnota**: nejmenší sémantická jednotka dat, **atomická** (vnitřně nestrukturovaná).  
**Složená doména**: doména složená z několika jednoduchých domén (hodnoty jsou *n*-tice).

Definice **relace v relačním modelu**:  
Relace na doménách D1, D2, …, Dn je dvojice , kde je **schéma relace** (v podstatě záhlaví tabulky, „název atributu“ + doména) a je **tělo relace**. Počet atributů ***n***se nazývá **stupeň (řád) relace**, kardinalita těla relace se označuje **kardinalita relace**.

### Vlastnosti (teoretické) relace:

* Neexistují duplicitní *n*-tice. (V praxi taková databáze obvykle vytvořit jde, ale i tam je to značně nepraktické.)
* *n*-tice jsou vzájemně neuspořádané.
* Hodnoty jednoduchých atributů jsou **atomické**. (Relace je normalizovaná v **1NF**, viz dále.)

### Pravidla integrity relačního modelu

Omezení, která říkají, jak to do těch tabulek ukládat.

**Specifická omezení**: platí jen pro konkrétní aplikaci (třeba rodné číslo, by spec).  
**Obecná omezení**: platí v každé databázi daného typu (v našem případě v každé relační databázi).

### Integrita entit

**Kandidátní klíč**: atribut, případně nedělitelný složený atribut (*n*-tice atributů), který jednoznačně identifikuje *n*-tici v relaci. Má dvě časově nezávislé vlastnosti: **jednoznačnost** a **neredukovatelnost** (vždy musí být *minimální* – třeba dvojice [ID, jméno] je jednoznačná, ale to jméno tam je zbytečné).

**Klíčový atribut**: atribut, který je součástí nějakého kandidátního klíče.

Každá relace v teorii relačního modelu má alespoň jeden kandidátní klíč! (V extrémním případě je to celá ta *n*-tice.) Relaci lze s pomocí kandidátních klíčů chápat jako paměť s asociativním výběrem.

**Primární klíč**: jeden vybraný kandidátní klíč (který za primární prohlásím). V praxi volíme nejjednodušší z kandidátních klíčů. Slouží jako základní prostředek „adresace“ *n*-tic – jednoznačně identifikuje řádek tabulky.

**Pravidlo integrity entit**: U žádné **komponenty** **primárního** **klíče** (tedy ani části složeného klíče) nesmí nikdy chybět hodnota – nesmí být NULL.

Primární klíč je tedy jednoznačný, minimální a nesmí obsahovat prázdné hodnoty. (**!**)

### Referenční integrita

Atribut *FK* bázové relace se nazývá **cizí klíč**, právě když splňuje tyto časově nezávislé vlastnosti:

* Každá hodnota *FK* je buď plně zadaná, nebo plně nezadaná. (Pokud by jím *náhodou* byl složený atribut, víc hodnot, musí být buď všechny vyplněné, nebo všechny prázdné, NULL.)
* Existuje relace s kandidátním klíčem *CK* takovým, že každá zadaná hodnota *FK* je identická s hodnotou *CK* nějaké *n*-tice relace .

Soulad hodnot cizích a primárních klíčů představuje *vztahy mezi řádky tabulek*.

**Pravidlo referenční integrity**: v podstatě druhá vlastnost původní definice – databáze nesmí obsahovat žádnou nesouhlasnou hodnotu cizího klíče.

## Relační algebra

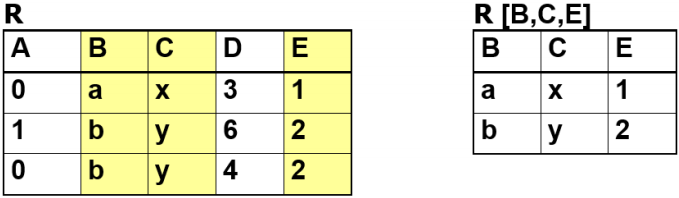
**Relační algebra** je dvojice , kde **nosičem** je množina relací a je množina **operací**, která zahrnuje tradiční množinové operace a speciální relační operace (***projekce, selekce, spojení, dělení***).

**Sjednocení relací**: – stejné schéma , sjednocení obsahů (těl relací).  
**Průnik**, **rozdíl relací** analogicky.

**Kartézský součin relací**: .

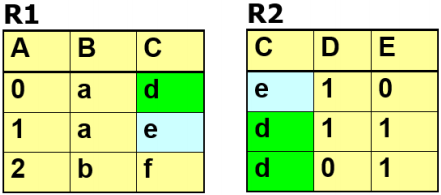
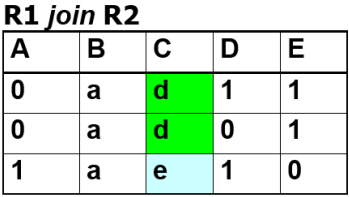
**Projekce**: projekce relace *na atributy* je relace se *schématem* a tělem zahrnující všechny *n*-tice .

Význam projekce je tedy **výběr jen některých sloupců z tabulky**.



**Selekce** (také **restrikce**): Nechť je operátor porovnání dvou hodnot: .  
-selekce relace *na atributech* je relace . Toto je základní definice, ve praxi může být , nebo literál.

**Přirozené spojení**: Nechť je relace se schématem a je relace se schématem , přičemž považujeme za složený atribut (a podobně pro ). Přirozeným spojením těchto relací je relace , jejíž tělo zahrnuje všechny *n*-tice takové, že .

Minimální množina operací relační algebry (aby se s tím systémem dalo „slušně pracovat“): **sjednocení, rozdíl, kartézský součin, projekce, selekce**. Těch operací ale může být velmi mnoho.

Relační algebra je základem pro **optimalizaci** zpracování dotazů. SQL dotazy se převádějí do relační algebry a pak se dají rozumně optimalizovat. Taky slouží jako referenční prostředek pro porovnávání relačních dotazovacích jazyků. Databázový jazyk je *relačně úplný*, je-li alespoň tak mocný, jako je relační algebra.

Transformace konceptuálního modelu na schéma

### Hlavní problémy špatného návrhu

* **Redundance** – opakující se informace.
* Nemožnost reprezentovat požadovanou informaci.
* Ztráta nějaké informace.
* Složitá kontrola integritních omezení.

### Pravidla transformace

1. Převod do 1NF:
   1. **Odstranění složených atributů** – převod na několik jednoduchých (pokud je nemůžeme považovat za atomické).
   2. **Odstranění vícehodnotových atributů** – vytvořením nového typu entity nebo nahrazením pevným počtem atributů (telefony -> telefon1, telefon2, telefon3).
2. Reprezentace **typů silných entit**: vytvoření tabulky se stejným primárním klíčem, návrh datových typů.
3. Reprezentace **typů vztahů** podle maximální kardinality:
   1. Vztah 1-1: vztah reprezentovaný pouze cizím klíčem. Mohu si vybrat, do které ze dvou tabulek jej dám – ideálně podle minimální kardinality – tam, kde ty hodnoty častěji budou zadané. Případné atributy vztahu přidám do tabulky, kde mám cizí klíč.
   2. Vztah 1-M: vztah reprezentovaný cizím klíčem. Cizí klíč musím dát do tabulky, u které má vztah kardinalitu M. (Oddělení 1-M zaměstnanců, *FK* dám k zaměstnanci.)
   3. Vztah M-M: vztah reprezentovaný pomocí vazební tabulky a dvou cizích klíčů. Klíčem vazební tabulky je složený klíč odpovídající klíčům dvou zdrojových tabulek.
4. Reprezentace **typů slabých entit**: jako silné se vztahem 1-M. U slabých entit bude primární klíč **složen** z cizího klíče odkazujícího na dominantní tabulku a z diskriminátoru.
5. Reprezentace **ternárních typů vztahů**: téměř vždy je potřeba přidat tabulku s třemi cizími klíči, které dohromady tvoří složený primární klíč.
6. Reprezentace **generalizace/specializace**:
   1. Pro všechny typy vytvořit tabulku. Podtypy mají jako primární klíč stejný klíč jako nadtyp, zároveň je to cizí klíč odkazující na tabulku nadtypu.
      * Nelze zajistit disjunktnost.
      * Vhodné pro překrývající se.
      * Nevhodné pro totální specializace.
   2. Vytvořit tabulky pouze pro podtypy, přidat jim atributy nadtypu.
      * Nelze zajistit disjunktnost
      * Vhodné pro překrývající se.
      * Vhodné pro totální specializace (vždy to musí být jedna ze specializací).
   3. Sloučit atributy ze všech podtypů a pak podobně jako v a).
      * Vhodné pro disjunktní specializace.
      * Nevhodné pro totální specializace.
   4. Sloučit atributy všech podtypů i nadtypu – vytvořit jednu tabulku.
      * Vhodné pro disjunktní specializace.
      * Vhodné pro totální specializace.

Vždycky je nutné zamyslet se, jak se s daty bude pracovat. (Jen specializace? I samotná generalizace?) Je potřeba brát v potaz také počty a typy rozdílných atributů; typy vztahů, které mohou existovat jen mezi nějakou specializací…

### Transformace objektového modelu

Viz slidy.

Normalizace návrhu databáze

Normalizace je významná teoretická podpora pro relační databáze. Napomáhá konzistenci dat.  
Je založena na třech typech závislostí mezi relacemi: **funkčních**, **vícehodnotových** a **závislostech na spojení**. My se omezíme na funkční závislosti – nejpodstatnější.

### Funkční závislost

Znamená to, že hodnota *atributu**relace* jednoznačně určuje hodnotu jiného *atributu* ***téže*** *relace*. Zapisujeme to jako . Vyplývá z významu atributů, představuje **integritní omezení**.

Rodné číslo jednoznačně identifikuje hodnoty ostatních atributů, v tomto případě je to tedy *zároveň také* primární klíč

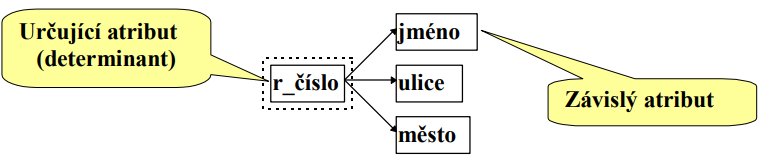
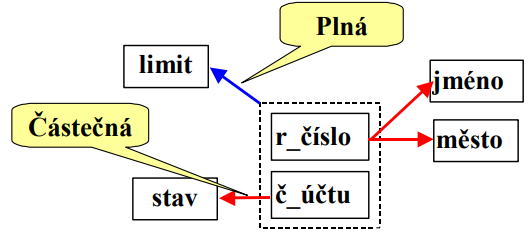


Diagram funkčních závislostí

Praktickým důsledkem je, že když se v relaci opakuje stejná hodnota determinantu, musí se opakovat i odpovídající stejné hodnoty závislých atributů.

**Plná funkční závislost**: atribut *Y* je plně funkčně závislý na (celém složeném) atributu *X*, když je funkčně závislý na *X* a není funkčně závislý na žádném atributu .



*(r\_číslo, č\_účtu)* je kandidátní klíč. *limit* na něm závisí plně, ostatní atributy na něm závisí částečně (jen na některých jeho složkách).

**Tranzitivní závislost**: vzniká, když je atribut funkčně závislý na jiném funkčně závislém atributu (prostě matematická tranzitivita). V databázích je **nechceme**!

### Proces návrhu

Teoretický postup:

1. *Univerzální relace* (jedna obří tabulka se vším – seznam atributů).
2. Postupná dekompozice na tabulky, které jsou korektní vůči **normálním formám**.

Praktický postup:

1. Datový model (ER diagram).
2. Transformace na schéma relační databáze.
3. Normalizace.

Bod 2 a 3 se v praxi obvykle přehazuje – „normalizujeme“ ten model a pak už to v podstatě 1:1 odpovídá požadovanému schématu databáze.

# Normalizace

Normalizace je postupná transformace – **dekompozice** – tabulky do vhodnějšího tvaru.  
Měla by splňovat a zajišťovat:

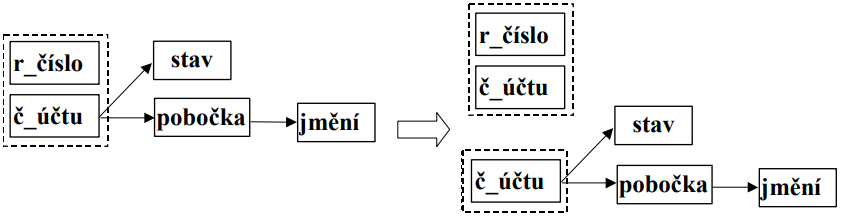
* Bezztrátovost (při zpětném spojení vzniklých tabulek).
* Zachování funkčních závislostí.
* Odstranění redundance (opakování informací) – to je ten její hlavní účel.

**Podmínka bezztrátové dekompozice:** společný atribut dvou vzniklých tabulek musí být kandidátním klíčem alespoň jedné z tabulek.

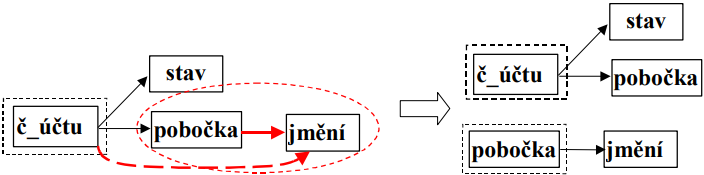
**Zachování závislostí**: musí být zachovány *všechny* původní závislosti.

Když nad těmi tabulkami potom udělám JOIN, musím dostat to, z čeho jsem (teoreticky) vyšel.

1. Rozdělíme všechny částečné funkční závislosti (**2NF**).



1. Odstraníme tranzitivní závislosti (**3NF**).



1. Boyce-Coddova normální forma (**BCNF**): všechny netriviální funkční závislosti jsou dány závislostí na kandidátních klíčích. (Viz dále.)

## Normální formy

Definují, co by měly v jednotlivých krocích normalizace splňovat všechny relace. Tvoří hierarchii, aplikují se postupně, každá splňuje podmínky předchozí a přidává něco navíc.

Návrh založený na normalizaci by měl dosahovat minimálně BCNF, případně 3NF.

**Klíčový atribut:** atribut, který je *součástí* nějakého kandidátního klíče.  
Ostatní atributy jsou **neklíčové**.

**Superklíč**: nadmnožina kandidátního klíče.

### První NF

Všechny jednoduché domény relace obsahují pouze atomické hodnoty.

### Druhá NF

Relace je v 1NF a navíc je **každý** **neklíčový** atribut **plně** funkčně závislý na **každém** **kandidátním** klíči.

To znamená, že tam musí nějaký kandidátní klíč být a *nesmí* tam být *částečná* závislost. Pokud je tam kandidátních klíčů víc, musí to platit pro **všechny**.

V databázovém schématu při dekompozicích podle 2NF vznikají cizí klíče.

### Třetí NF

Relace je ve 2NF a navíc **neexistuje** **žádný** **neklíčový** atribut, který by byl **tranzitivně** závislý na **nějakém** **kandidátním** klíči.

Znamená to, že všechny atributy jsou funkčně závislé pouze na (k.) klíči a na ničem jiném: „každý neklíčový atribut musí poskytovat informaci o klíči, o celém klíči a o ničem jiném než o klíči“.

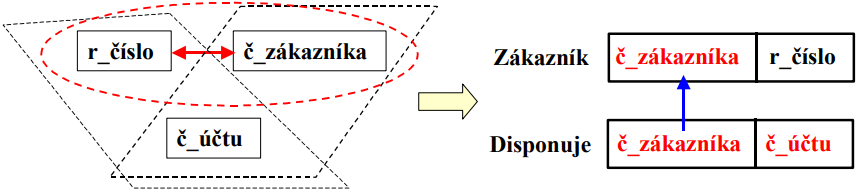
### Boyce-Coddova NF

Občas se označuje jako „3,5NF“ – automaticky splňuje všechny ostatní.

Relace je v BCNF, jestliže pro **každou netriviální funkční závislost** je **superklíčem**.

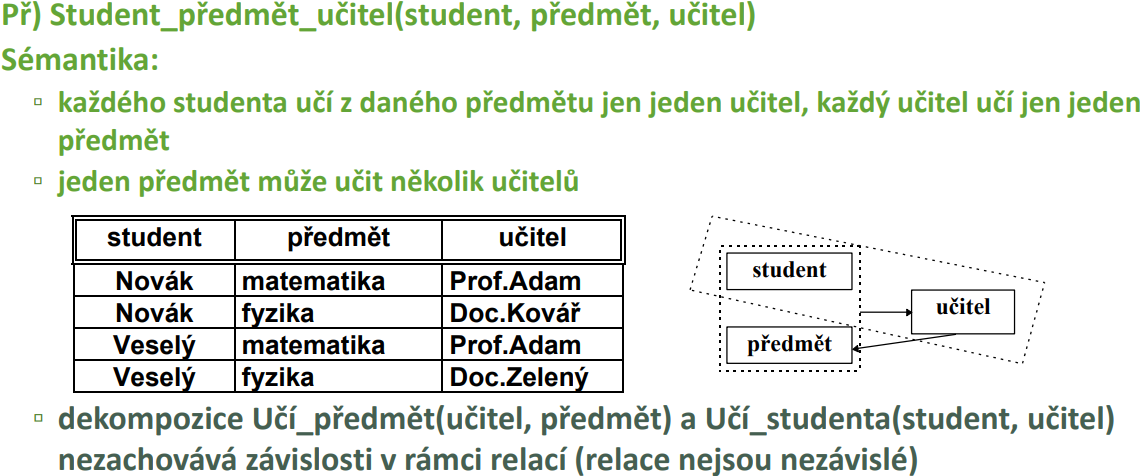
Využívá se, pokud může existovat **několik** kandidátních klíčů, kandidátní klíče mohou být **složené** a mohou se **překrývat**. To 3NF neřeší – připouští závislosti mezi klíčovými atributy.

Příklad – relace *Disponuje(r\_číslo, č\_klienta, č\_účtu):*



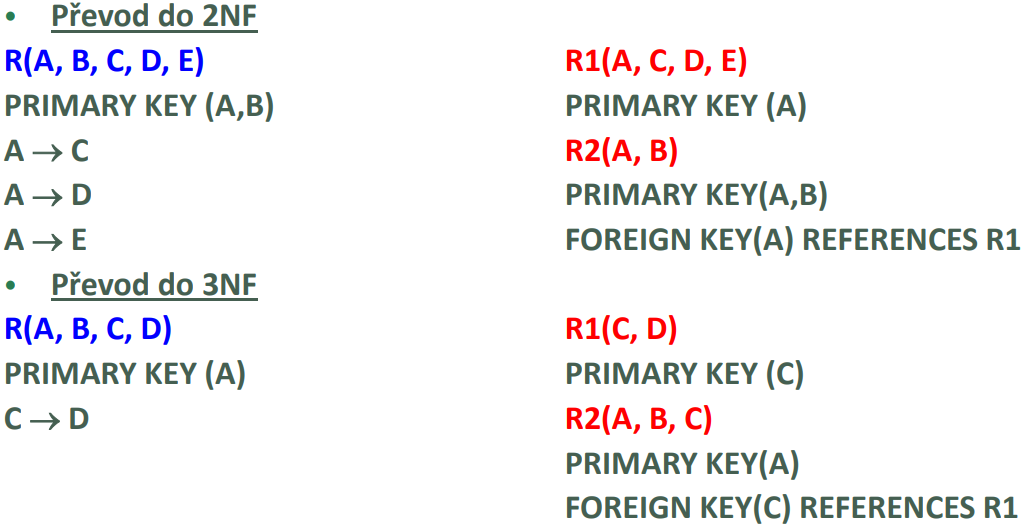
Zákazník je identifikovaný dvěma atributy – rodným číslem a číslem zákazníka.

Je to dobrý cíl normalizace – zajišťuje, že ve schématu **neexistuje žádná redundance kvůli funkčním závislostem**, ale **nemusí být dosažitelná** – některé korektní relace nejdou normalizovat do BCNF (tak, aby byly zachovány všechny závislosti; stává se zejména u některých ternárních vztahů). Nejde převést např. relace s funkčními závislostmi typu :



### Mechanický postup

Pro 2NF a 3NF v podstatě nepotřebujeme „přímo“ znát sémantiku tabulek (když známe ty závislosti).



Bez znalosti sémantiky ale nemůžeme dobře určit, na kterou stranu dát cizí klíč – to závisí na kardinalitě vazby (ve vztahu 1:M ho dáváme vždy na stranu jedničky).

Jazyk SQL

První standard ISO-SQL/86, dominantní byl dialekt IBM. Dnes je nejrozšířenějším základem, který podporují prakticky všechny databázové systémy, **SQL/92**. (Navíc to má tři úrovně souladu, takže to nemusí podporovat všechno…) V novějších normách existují i různé libůstky – objektové rysy, XML, JSON apod.

Tři možné kontexty použití – *binding styles*:

* Direct SQL – napíšu dotaz, pošlu ho do databáze, dostanu odpověď.
* Embedded SQL – „hostitelská verze“, v podstatě komunikace nějakým protokolem, nabízí například *kurzory* pro procházení výsledků.
* Jazyk modulů – SQL se prolíná s běžným programovacím jazykem. (Už se moc nepoužívá.)

Kategorie příkazů:

* Data Definition Language (DDL): definují data a pohledy.
* Data Manipulation Language (DML): manipulace s daty.
* Data Control Language (DCL): řízení přístupových práv, autorizace apod.
* Příkazy pro zavádění integritních omezení.
* Příkazy pro řízení transakcí.
* A další…

SQL **nerozlišuje** **velikosti** písmen (mimo řetězce, of course)!

# Definice dat

#### Vytvoření objektu:

CREATE TABLE název (  
 jméno\_sloupce typ\_sloupce [impl\_hodnota] [integritní\_omezení\_sloupce],  
 sloupec\_2, ...,  
 [integritní\_omezení\_1, ...]  
)

### Definice integritních omezení pro sloupce

[CONSTRAINT jméno\_omezení] definice\_omezení

#### Prázdné hodnoty:

NULL  
NOT NULL

#### Omezení:

CHECK (podmíněný\_výraz)

#### Primární a alternativní klíče:

PRIMARY KEY  
UNIQUE -- unikátnost hodnot v jednom sloupci

#### Cizí klíče:

REFERENCES cizí\_tabulka [(jméno\_odkazovaného\_sloupce)] [MATCH FULL|PARTIAL|SIMPLE] [událost ref\_akce]

Zavádí odkaz na jinou tabulku. Jména odkazovaných sloupců jsou volitelná, pokud se neuvedou, vezme se primární klíč cizí tabulky.

Referenční akce viz dále.

**Match Full**: everything must fully match or everything must be null.  
**Match Simple**: if one thing is NULL, the constraint is *simply* ignored.  
**Match Partial**: Dovoluje odkazovat se na klíče cizí tabulky, které jsou taky NULL – nemá tedy smysl pro primární klíče (ty nemohou být nikdy NULL), ale může se hodit u nějakých alternativních kandidátních klíčů.

### Definice integritních omezení pro celé tabulky

V podstatě stejně jako u sloupců, ale musí se uvést ten sloupec, kterého se to týká. Notably:

FOREIGN KEY (zdroj\_sloupec\_1, ...) REFERENCES cizí\_tabulka [(cíl\_sloupec\_1, ...)] [MATCH ...] [událost ref\_akce]

### Datové typy

* Znakové: CHARACTER(n) = CHAR(n), CHARACTER VARYING(n) = VARCHAR(n)
* Přesné číselné: NUMERIC(precision, scale), DECIMAL(precision, scale)
  + Precision = maximální počet číslic
  + Scale = kolik je číslic za tečkou
* Klasické (přesné) číselné: INTEGER, SMALLINT
* Přibližné: FLOAT(precision), REAL, DOUBLE PRECISION
* DATE, TIME, (DATETIME – někde), TIMESTAMP
* INTERVAL
* CHARACTER LARGE OBJECT, BINARY LARGE OBJECT
* NATIONAL CHARACTER(n) = NCHAR(n)
* BOOLEAN

### Referenční akce

Některé operace (modifikace dat) mohou porušit různá **integritní omezení** (referenční integritu, omezení primárních, alternativních, not null), případně **referenční integritu** (cizí klíče – např. odstranění záznamu, na který se odkazuje jiná tabulka) – v tomto případě lze předepsat, jak se databázový systém má zachovat.

Referenční události: **ON UPDATE** a **ON DELETE**.

Referenční akce:

* NO ACTION – operace se zakáže.
* CASCADE – bude propagovat změnu do závislých tabulek – „když smažeme klienta, smažou se i všechny jeho účty“.
* SET DEFAULT – nastaví se na nějakou výchozí hodnotu.
* SET NULL – nastaví se na NULL.

### Změny tabulek

ALTER TABLE jméno\_tabulky akce

ADD COLUMN jméno\_nového\_sloupce  
DROP COLUMN jméno\_rušeného\_sloupce  
ALTER COLUMN jméno\_měněného\_sloupce akce  
ADD/DROP CONSTRAINT název\_omezení

Často se používá v situacích, kdy se nedají integritní/referenční omezení nastavit přímo při vytváření tabulek – pokud je nutný cyklus referencí, reference sama na sebe…

DROP TABLE jméno\_tabulky

### Indexy

Pomocné struktury, které umožňují rychlejší vyhledávání v určených sloupcích.

CREATE [UNIQUE] INDEX jméno\_indexu ON jméno\_tabulky (jméno\_sloupce [**ASC**|DESC], ...)

Nikdy nevytvářejte index nad primárním klíčem, už existují…  
Indexy viz dále.

### Synonyma

Trvalé **odkazy** na jiné databázové **objekty**.

# Manipulace s daty

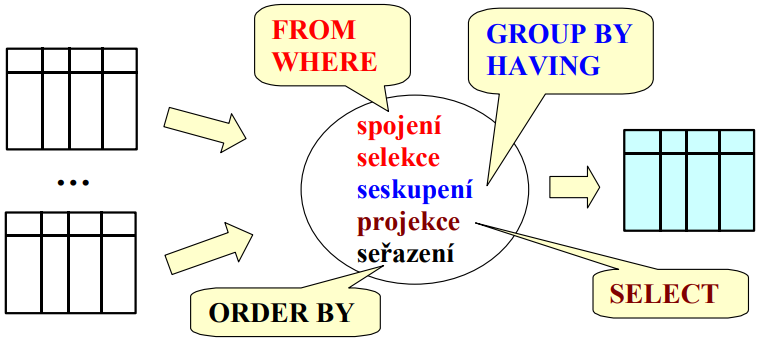
Čtyři základní příkazy: SELECT, UPDATE, DELETE, INSERT.

SELECT [ALL|DISTINCT] položka [[AS] alias\_sloupce], položka\_2 [...], ...  
FROM tabulkový\_výraz [[AS] [alias\_tabulky]], ...  
[WHERE podmínka]  
[GROUP BY jméno\_sloupce\_z\_FROM|číslo, ...]  
[HAVING podmínka]  
[ORDER BY jméno\_sloupce\_z\_SELECT|číslo [**ASC**|DESC], ...]

Rozdíl mezi WHERE a HAVING: WHERE se aplikuje na jednotlivé záznamy z výchozí tabulky; HAVING udává podmínku, která se aplikuje na skupiny vytvořené pomocí GROUP BY (provádí se tedy o něco „později“).

Zavedené aliasy ***sloupců*** lze použít **pouze** v klauzuli ORDER BY, ale aliasy ***tabulek*** lze použít **všude**! (Kvůli pořadí vyhodnocování, optimalizacím.)

SELECT se dvěma tabulkami bez nějakého omezení na spojení = kartézský součin.



Souvislost SQL s relační algebrou.

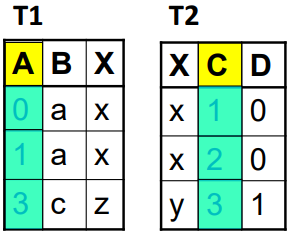
### Spojování

Tabulkový výraz může být buď jedna tabulka, nebo nějaký výraz spojující více tabulek.

#### Vnitřní spojení = inner join:

Záznamy, které nesplňují tu podmínku, **nenajdou protějška ve druhé tabulce**, se ve výsledku **neobjeví**.

* Obecné – na základě podmínky = **théta-join**: T1 JOIN T2 ON **A < C**

* Na základě rovnosti = **equijoin**: T1 JOIN T2 ON **A = D**
* Přirozené = **natural join** (na základě rovnosti hodnot *stejně pojmenovaných atributů)*:  
  T1 NATURAL JOIN T2

#### Vnější spojení = outer join:

* Levé vnější spojení – vždy se objeví *všechny* záznamy z *levé* tabulky, i pokud k nim nejsou žádné záznamy vpravo.

Výrazy spojení mají tvar:

tabulka\_1 CROSS JOIN tabulka\_2

tabulka\_1 [NATURAL] [**INNER**|LEFT|RIGHT|FULL|*UNION*] JOIN tabulka\_2  
[ON podmínka | USING (sloupec\_1, ...)]

### Agregační funkce

COUNT(\*) | AVG | MAX | MIN | SUM | COUNT([ALL|DISTINCT] výraz)

Standard říká, že se agregační funkce nedají zanořovat, ale konkrétně v Oracle to někdy jde (obvykle v kombinaci s GROUP BY).

### Seskupování záznamů

Klauzule GROUP BY seskupuje řádky podle nějakého atributu.  
Z jejího použití vyvstává omezení na to, co vybíráme pomocí SELECT – můžeme tam dát agregační funkce, prvky seznamu v GROUP BY nebo konstanty.

„Jaká je částka na účtech **v jednotlivých** pobočkách?“

SELECT pobocka, SUM(stav) AS celkem\_na\_uctech FROM Ucet GROUP BY pobocka

„Kolik mají **jednotliví klienti** účtů a celkem na nich peněz?“

SELECT K.jmeno, K.r\_cislo, COUNT(U.c\_uctu) AS pocet, SUM(U.stav) celkem  
FROM Klient K NATURAL *LEFT* JOIN Ucet U  
GROUP BY K.r\_cislo, *K.jmeno*

(V group klauzuli musí být i K.jmeno, protože jej selektujeme. Je to funkčně závislý atribut, groupování podle rodného čísla a rodného čísla se jménem dá stejné výsledky. Pokud použijeme LEFT join, budeme tam mít i klienty, kteří nemají žádný účet, jinak by ve výsledku vůbec nebyli.)

Použití klauzule HAVING pro výběr klientů s celkovým obnosem větším než 100 000:

HAVING SUM(stav) > 100000

Není možné tam dát ten alias celkem, s tím je možné pracovat pouze v ORDER BY.

### Podmíněný výraz

Klauzule WHERE může obsahovat různé predikáty, logické spojky a **poddotazy**.

Viz slidy! (#42)

Exists – téměř vždycky je třeba jej *korelovat* s výrazem výš – někde se tam vyskytne tabulka z toho vyššího výrazu.

### Další užitečné výrazy

* CASE WHEN podmínka THEN výraz ELSE výraz END
* COALESCE (x, y, z, ...) – vybere první ne-NULL hodnotu
* NULLIF (x, y) == CASE WHEN x = y THEN NULL ELSE x END

Klauzule WITH – vytváří „podtabulku“, zadefinovaný sub-select, který lze použít v dotazu – zjednodušuje složité dotazy:

WITH x AS (  
 SELECT a, b, c FROM d ...  
) SELECT x.a, y.a FROM y LEFT JOIN x ON x.b = y.b

Pomocí WITH se dá udělat rekurzivní dotaz.

## Vkládání

INSERT INTO jméno\_tabulky [(sloupec\_1, sloupec\_2, ...)] zdroj

Vkládá do tabulky jeden řádek nebo **obsah jiné tabulky**. Pokud nejsou uvedeny sloupce, vkládá se postupně tak, jak je tabulka původně zadefinovaná.

*Zdrojem* může být:

* DEFAULT\_VALUES
* VALUES (skalární\_výraz|NULL|DEFAULT, …)
* Tabulkový výraz (vloží se výsledek poddotazu)

## Mazání

DELETE FROM jméno\_tabulky [WHERE podmínka]

Bez uvedení podmínky smaže vše. Novější SQL umožňuje přidat ještě část TRUNCATE TABLE, která maže jen nějakou část tabulky (podle velikosti).

## Aktualizace

UPDATE jméno\_tabulky SET sloupec = výraz|NULL|DEFAULT, ... [WHERE podmínka]

# Pohledy

Pohled je pojmenovaná „virtuální tabulka“ odvozená od nějakých bázových tabulek. Je to jakýsi uložený dotaz, nad kterým se dá dále operovat, jako by to byla běžná tabulka.

CREATE VIEW jméno\_pohledu [(jméno\_sloupce, ...)] AS tabulkový\_výraz [WITH CHECK OPTION]

Sloupce jsou běžně odvozená od původních názvů (musí být unikátní!), mohou být přejmenované při vytvoření pohledu.

Nad pohledy se dají nastavovat různá práva pro uživatele – můžeme mít pohled, který pracuje pouze s klienty z Brna, uživatel spravující brněnskou pobočku tak může dělat všechny operace nad nimi, ale ne nad ostatními klienty (nemá přístup k bázové tabulce).

**WITH CHECK OPTION**: pokud aplikujeme aplikaci na pohled, výsledek operace musí opět skončit v pohledu – nedochází k porušení definice pohledu. Zamezuje například provést aktualizaci, která by daný řádek vyhodila z pohledu ven (ale přitom by data v těch tabulkách zůstala) – pohled s WHERE mesto='Brno', nastavím mesto='Praha'.

### Rušení pohledu

DROP VIEW jméno\_pohledu [RESTRICT|CASCADE]

Restrict – pokud na pohledu něco závisí, zrušení se neprovede. Cascade – zruší i závislé objekty.

## Manipulace nad pohledy

Řeší se „ve dvou krocích“: základem je, že se vezme původní tabulkový výraz a použije se jako subselect v dotazu, kde je pohled použit – jde o jakousi substituci; ve skutečnosti to ale spíš ty dotazy zkombinuje, *slije* je to dohromady.

CREATE VIEW Janska AS  
 SELECT K.\* FROM Klient K, Ucet U  
 WHERE K.r\_cislo=U.r\_cislo AND U.pobočka='Jánská'  
WITH CHECK OPTION

SELECT \* FROM Janska **WHERE mesto = 'Brno'**

se přetransformuje na:

SELECT K.\*   
FROM Klient K, Ucet U   
**WHERE K.mesto = 'Brno'** AND K.r\_cislo=U.r\_cislo AND U.pobocka='Jánská

Pohledy se ale tváří jako běžná tabulka a umožňuje aktualizovatelnost – jde nad tím dělat insert, delete, update. Má to podmínku: musí to jít na zdrojové tabulky nějak zpátky převést – pokud **pohled** obsahuje například **agregační funkce**, distinct, group by, **nelze jej upravovat** (logicky).

Pohledy jsou tedy **selektivní**, **projektivní** a **agregační**.

## Materializované pohledy

Pohledy, u nichž je výsledek dotazu fyzicky uložen v databázi (je zaručena synchronizace s původními tabulkami – automatická či manuální).

CREATE MATERIALIZED VIEW jméno\_pohledu [REFRESH ON COMMIT] AS ...

Výhoda – může to být výrazně rychlejší.

# Systémový katalog

SQL není jen o datech, ale i o metadatech. U relačních systémů má obvykle katalog (*slovník dat,* nese informace o struktuře databáze) stejné rozhraní jako uživatelské databáze.

Některé databázové systémy na to používají speciální příkazy.

# Práce s chybějící informací

Operace v SQL mohou operovat nad **tříhodnotovou logikou** – možné výsledky logické operace jsou true, false a unknown. Pravidla:

* Skalární výrazy: pokud je jakýkoliv z operandů NULL, výsledkem je NULL.
* Logické operace (porovnávání): pokud je jakýkoliv z operandů NULL, výsledkem je *unknown*.
* Agregační funkce: pokud je ve sloupci NULL, bere se jako neutrální hodnota vůči prováděné operaci (příp. datovému typu). Výjimkou je COUNT(\*), kde se neignoruje (počítá celé řádky).
* WHERE a HAVING vybírá pouze řádky s hodnotou *true*.

Nelze tedy porovnat dva řádky – pro = i <> vyjde *unknown*.

Testování na NULL je možné provádět jen pomocí predikátu IS [NOT] NULL.

# Hostitelská verze SQL

SQL embedded v nějakém jazyce. (Wtf.)

**Kurzor:** objekt, pomocí kterého se dá výsledek nějakého dotazu procházet řádek po řádku – ukazatel na *result set*.

**Dynamické SQL**: možnost vytvářet SQL příkazy z řetězců za běhu.

**Připravitelný příkaz**: příkaz, ve kterém jsou zástupné symboly „?“, za které se doplňuje až za běhu. Může se to hodit pro optimalizace – databáze z dotazů vytváří *execution plan*, který může dál optimalizovat. V některých případech nemusí být vytvořený *execution plan* výrazně závislý na tom, co bude dosazeno místo toho otazníku – dotaz je tak i po dosazení předem neznámé části rychlejší. (Dá se použít třeba pro opakované INSERTy!)

# Další relační jazyky

**Query By Example** (MS Access).  
**Datalog**: neprocedurální dotazovací jazyk vycházející z Prologu.

Architektury databázových aplikací, rozdělení zátěže

Architektura klient/server a třívrstvá architektura

Klient/server architektura je základem **kooperativního zpracování** – k jedné databázi může přistupovat více klientů.

Architektury ovlivňují různé faktory: požadavky na interoperabilitu, růst velikosti zdrojů, růst počtu klientů.

### Skupiny služeb

* **Prezentační služby**: příjem vstupů od uživatele, zobrazování výsledků – komunikace s uživatelem.
* **Prezentační logika**: řízení interakce s uživatelem (přepínání mezi okny, hierarchie obrazovek, menu…).
* **Logika aplikace**: operace realizující algoritmus aplikace.
* **Logika dat**: specifikace různých integritních omezení, datových typů apod. (Část může řešit databázový server, část může řešit až aplikace samotná – některé požadavky nelze vyřešit deklarativně.)
* **Datové služby**: surové akce s databází – definice dat, manipulace s daty, transakce apod.
* **Služby ovládání souborů**: I/O operace se soubory reprezentující databázi.

Vždy řešíme, co implementovat v klientské části a co v serverové části. Rozhodovat se můžeme zejména pro **logiku aplikace** a **dat**.

## Varianty architektury

### Klient/server se vzdálenými daty

Všechno, co může být umístěno na klientovi, je na klientovi – „tlustý klient“.  
Na serveru leží pouze **datové služby** a **ovládání souborů**.

Vede na vysokou **komunikační zátěž** a **vysoké zatížení klientské stanice**.

### Klient/server se vzdálenou prezentací

Všechno, co může být umístěno na serveru, je na serveru – „tenký klient“.  
Na klientovi zůstává pouze **prezentace**.

Vede na vysoké **zatížení serveru**.

### Klient/server s rozdělenou logikou

Část logiky aplikace a dat na klientovi, část na serveru.

**Rozkládá zátěž**, ale zvyšuje náročnost údržby klientů.

### Třívrstvá architektura

Klient ⇋ aplikační server ⇋ databázový server (⇋ databáze).

Tenký klient – jednoduchá udržitelnost – pouze prezentační služby a logika.  
Aplikační server – obsahuje nějakou nebo veškerou logiku aplikací a logikou dat.

Přínosy: pružnější rozdělení práce, škálovatelnost, standardizovaný přístup k různým zdrojům, centralizace dat (ochrana dat, možnost využití sdílených objektů několika aplikacemi).

# Rozdělování zátěže

Prostředky rozdělování zátěže jsou zejména:

* Deklarativní integritní omezení (primární klíče, cizí klíče, CHECK klauzule)
* **Uložené podprogramy** – databázové procedury a funkce; spouštěny explicitně
* **Databázové triggery** – spouštěny při provedení specifikované databázové operace

## Uložené podprogramy

Databázový objekt obsahující kód prováděny na straně databázového serveru. Nějaká standardizace v SQL-92/PSM, v Oracle rozšířeno jejich PL/SQL.

CREATE PROCEDURE název (arg\_1 [**IN**|OUT|IN OUT] typ\_1, ...) IS tělo

Uložená funkce navíc vrací hodnotu:

CREATE FUNCTION název (...) RETURN typ IS tělo

## Databázové triggery

Databázový objekt obsahující kód spouštěný specifikovanou událostí v databázi.

### Typy triggerů

Podle typu/času provedení:

* BEFORE, AFTER: příkazy těla triggeru se provedou před/po provedení spouštějící události.
* INSTEAD OF: příkazy těla triggeru se provedou místo přímé modifikace **neaktualizovatelného** **pohledu** – takový trigger je navázaný na pohled, *ne* na tabulku.

Podle počtu provedení:

* **Příkazový**: je spuštěn jedenkrát pro příkaz, který způsobí zadanou událost.  
  FOR EACH STATEMENT
* **Řádkový**: je spuštěn jedenkrát pro každý řádek ovlivněný danou událostí. V těle triggeru se mohu odkázat na hodnoty, které jsou zrovna modifikovány.  
  FOR EACH ROW

### Typická využití

* Složitá integritní omezení.
* Složitá bezpečnostní omezení.
* Implementace auditu – automatické ukládání informací o změnách.
* Výpočet hodnot pro sloupec s odvozenými hodnotami.

### Zásady

* Nepoužívat trigger, pokud je možné použít deklarativní integritní omezení.
* Nevytvářet rekurzivní triggery.
* Pro *rozsáhlé* triggery vytvořit radši uloženou proceduru, která se z těla triggeru zavolá.
* Používat triggery jen pro centralizované globální operace, které mají být spuštěny bez ohledu na to, odkud (uživatel, aplikace) požadavek na jejich provedení přijde.

**DOPLNIT PŘÍKLAD** [**https://docs.oracle.com/cd/B19306\_01/server.102/b14200/statements\_7004.htm**](https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_7004.htm)

Bezpečnost dat

Hlavní cíle:

* **Důvěrnost** (secrecy): informace by neměly být přístupné neautorizovaným uživatelům.
* **Integrita**: modifikovat data může pouze autorizovaný uživatel.
* **Dostupnost**: autorizovaným uživatelům nesmí být bráněno v přístupu.

Všechno se to týká **bezpečnostní politiky** (říká, kdo může co s jakými daty dělat), která je zajištována **bezpečnostními mechanismy**.

## Bezpečnostní mechanismy

* **Pohledy**: umožňuje různým uživatelům zpřístupňovat pouze omezené části dat; navíc je možné zajistit jejich neaktualizovatelnost.
* **Volitelné/nepovinné řízení přístupu** (DAC – discretionary access control): založeno na *přístupových právech*, která jsou přiřazována uživatelům a umožňují nějaký druh přístupu k databázovým objektům.
* **Povinné řízení přístupu** (mandatory): založeno na *bezpečnostních třídách*– stupních utajení objektů, *stupních prověření* subjektů (uživatelů) a *pravidlech provádění operací*.

SQL poskytuje podporu pro **nepovinné** řízení přístupu; některé SŘDB poskytují i povinné řízení.

## Typické prostředky řízení přístupových práv

### Objektová oprávnění

GRANT seznam\_oprávnění ON databázový\_objekt TO {seznam\_uživatelů | PUBLIC} [WITH GRANT OPTION]

### Systémová oprávnění

**Uživatelé**:

CREATE USER jméno IDENTIFIED {BY heslo | EXTERNALLY}

**Role**:

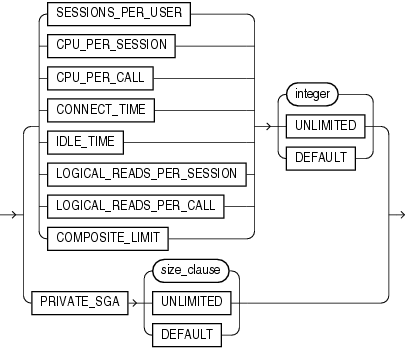
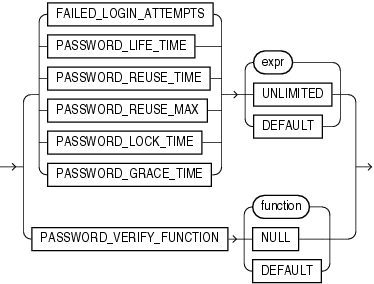
CREATE ROLE jméno [NOT IDENTIFIED | IDENTIFIED {BY heslo | EXTERNALLY}}]

**Oprávnění**:

GRANT seznam\_oprávnění\_a\_rolí TO seznam\_uživatelů\_nebo\_rolí [WITH ADMIN OPTION]  
REVOKE ... FROM ...

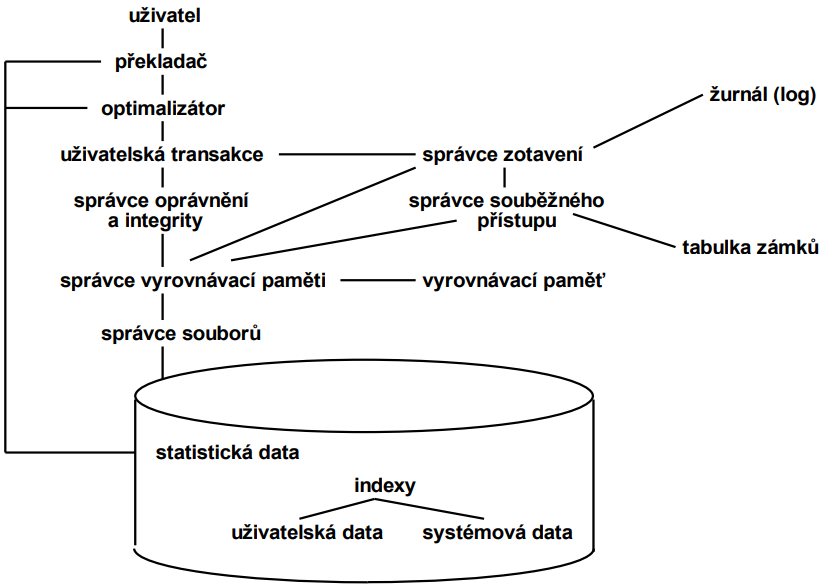
**Parametry hesel a přístupu k objektům**:

CREATE PROFILE název LIMIT pravidla

Fyzická úroveň databázového systému

* **Uživatel** zadá dotaz.
* **Překladač** přeloží dotaz do interní reprezentace.
* **Optimalizátor** převede dotaz do efektivní podoby.
* A pak nastává celkem maškaráda.



# Přístup k datům v databázi

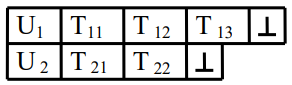
Cílem je vždy minimalizace přístupů k disku.

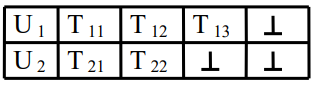
## Organizace záznamů

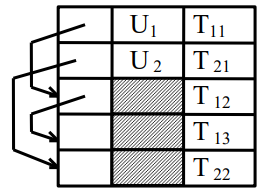
**Záznamy pevné délky**: obvykle se v praxi nevyskytují. Můžeme je uložit jeden za druhým – pak ale vznikají prázdná místa a zabírá to místo navíc. Možno vylepšit přidáním adresáře – lineárního seznamu volných míst.

**Záznamy proměnné délky**:

* Shlukované záznamy: na disku jsou blízko sebe související data (třeba záznamy *účet* a příslušné *transakce*). Pak je otázkou, jak je uložit – nejjednodušší je zase převést je na záznamy pevné délky:

  
Za sebe bez zarovnání

  
Za sebe s pevnou délkou a se zarovnáním.



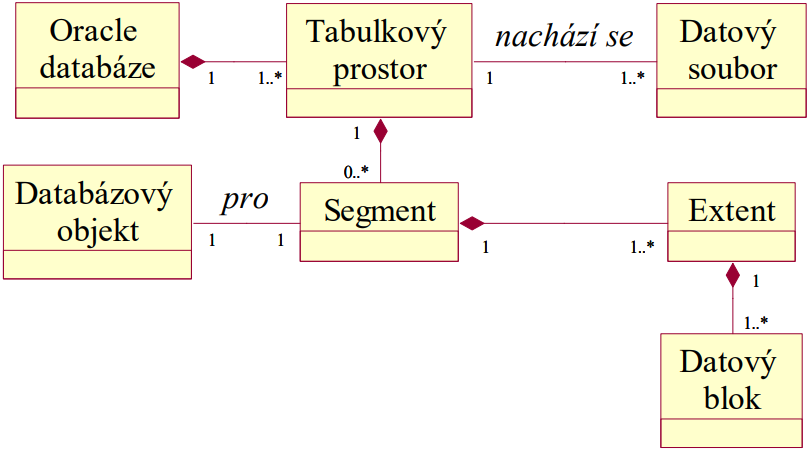
Za sebe s pevnou délkou a přetokovou oblastí.

* **Struktura s adresářem bloku**: každý *záznam* má svou pevnou adresu. Každý blok (diskový) je opatřen *adresářem*, ve kterém jsou adresy jednotlivých záznamů. V rámci bloku je tak záznam uložen pevně.

Jak zařazovat záznamy do bloků souboru?

* **Neuspořádaný soubor**: záznam je umístěn na první volné místo (halda).
* **Sekvenční soubor**: záznamy uspořádány podle hodnoty *vyhledávacího klíče*. Podle způsobu vytvoření tohoto klíče navíc může vést k tomu shlukování – logicky svázané záznamy mohou být umístěny blízko sebe.
* **Hashovaný soubor**: záznamy jsou umisťovány do bloků na základě nějaké hashovací funkce.

## Databázové soubory serveru Oracle



Databáze se skládá z množiny tabulkových prostorů. Tabulkový prostor je nějaký kus místa na disku, kde jsou uloženy objekty. Každý tabulkový prostor má jeden nebo více datových souborů.

Segment je prostor na disku vytvořený pro právě jeden databázový objekt. Dělí se na menší části – extenty. S narůstající velikostí objektu (tabulky) se přidávají nové extenty, které jsou rozprostřeny přes bloky.

Struktura datového bloku:

* Hlavička
* Table Directory
* Row Directory
* Volné místo
* Data

V datovém bloku se vždy drží nějaké volné místo – po dosažení hranice PCTFREE, typicky 20 %, se přepne blok do módu „pouze aktualizace záznamů, žádné vkládání nových“. Z tohoto módu ale vyjde jen po uvolnění *větší* části místa – podle hranice PCTUSED, typicky 40 %.

# Správa vyrovnávací paměti

**Požadavky**:

1. Co nejlepší strategie výměny bloků. (LRU? MRU?)
2. Omezení času, kdy nelze blok zapsat na disk. Souvisí s transakčním zpracováním – občas potřebuji bloky z mezipaměti zapsat na disk v určitém pořadí.
3. Možnost vynuceného zápisu bloků na disk – opět souvisí s transakcemi – kontrolní body.
4. Preference často používaných bloků (katalog, indexy).

Můžeme nechat správu vyrovnávací paměti na operačním systému, ale typicky mi to nesplňuje požadavek 2 a 3 – je tedy lepší, když si **vyrovnávací** **paměť** **spravuje** **sám** **SŘBD**.

# Indexování a hashování

Index je **vyhledávací struktura**, která na základě hodnoty **vyhledávacího klíče** umožní **rychle** přistoupit k příslušnému záznamu v **primárním souboru** (souboru se záznamy tabulky).

Vyhledávacím klíčem je nějaký sloupec tabulky, může být i složený. Může jich být víc, záleží na použití (kdybychom ho měli nad každou.

**Primární vyhledávací klíč**: vyhledávací klíč, podle jehož hodnot je **seřazen primární soubor**.  
**Sekundární vyhledávací klíč**: jiný vyhledávací klíč než ten primární.

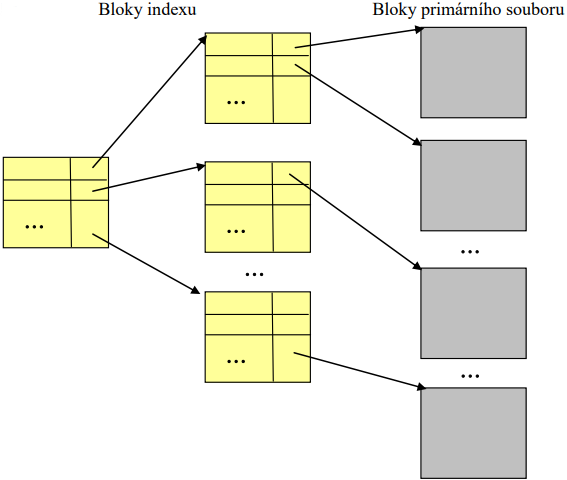
**Uspořádaný index**: založený na seřazení podle hodnot vyhledávacího klíče (např. B+ stromy).  
**Hashovaný index**: využívá hashovací funkci. Záznamy nebo ukazatele na ně jsou zatříděny v *sektorech*, které jsou určeny hodnotou hashovací funkce pro jejich vyhledávací klíče.

**Husté indexy**: v indexu jsou uloženy všechny hodnoty vyhledávacího klíče, které se objevují v primárním souboru.  
**Řídké indexy**: v indexu jsou uloženy jen některé vyhledávací klíče, pokud se do té hodnoty netrefíme, dostaneme se k nějaké nejbližší a dál hledáme **sekvenčně**. V *primárním řídkém indexu* je typicky jedna položka pro každý *blok* primárního souboru. Řídký index může být pro jednu tabulku **jen jeden**.

**Index s unikátními hodnotami**: index pro vyhledávací klíč, jehož hodnoty jsou v primárním souboru unikátní.

## Víceúrovňové indexy

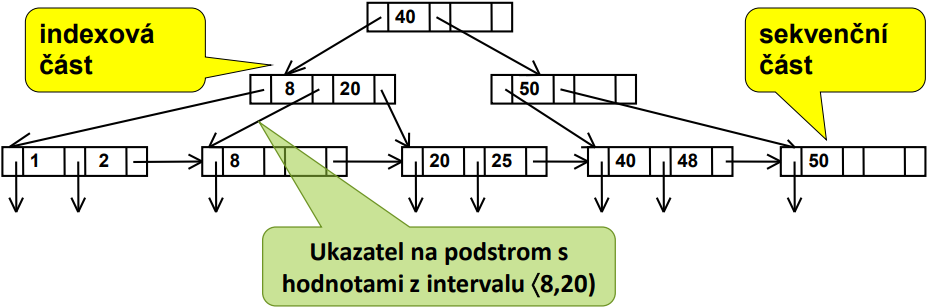
Indexy s více úrovněmi, kde každá úroveň kromě poslední je řídkým primárním indexem následující úrovně. Poslední úroveň je pak indexem primárního souboru.



### B+ stromy

Nejpoužívanější indexové struktury v relačních databázových systémech.  
Jsou to **vyvážené n-ární stromy** – je maximální počet následníků!

* Každý uzel kromě kořene a listu má až následníků.
* Kořen (není-li listem) má nejméně 2 následníky.
* List obsahuje až hodnot vyhledávacího klíče.
* Listové uzly **tvoří jednosměrný uspořádaný seznam**.
* Každý uzel má typicky velikost jednoho diskového bloku.
* Pro hodnot vyhledávacího klíče je cesta od kořene k listu maximálně .



, tedy každý uzel má **tři** **ukazatele**. Zaplněnost uzlu musí být aspoň poloviční – zde splněno.

Za index kromě místa platíme zejména režií při vkládání a mazání záznamů.

**TODO: vyzkoušet si to. Ad příklad v prezentaci. IMPORTANT.**

## Hashování

Hashovací funkce převádí hodnotu vyhledávacího klíče na adresu sektoru záznamů (typicky adresu bloku). Hlavní nevýhodou je potřeba lineárního procházení při shodě – přetečení sektorů.

**Požadavky**: rovnoměrné rozložení hodnot, náhodný výskyt hodnot.

Hashování může být velmi výhodné, když jsem schopen přibližně říct, kolik bude v tabulce záznamů – to ale bohužel obvykle nevím.

Přímým hashováním může být přístupný jen jeden (primární) vyhledávací klíč.

### Dynamické hashování

Modifikace hashovací funkce, která odráží změny velikosti tabulky.

TODO?

# Shlukování

Podstata: umisťování záznamů se „stejnými vlastnostmi“ fyzicky blízko sebe. Touto vlastností je často nějaký **shlukovací klíč – cluster key**. Nemusí to nutně shlukovat víc různých tabulek.

V Oracle se vytvoří pojmenovaný cluster pomocí CREATE CLUSTER a následně nad tím vytvoříme index (ON CLUSTER).

# Bitmapové indexy

Vhodné pro indexování podle atributů, které nabývají **několika málo různých hodnot**. V souboru se pak uloží informace o tom, ve kterých řádcích se vyskytuje daná hodnota klíče, ve formě *bitových vektorů*.  
*Velká tabulka indexovaná podle takového atributu pomocí B+ stromu by vedla na naprosto neefektivní B+ strom s jen několika málo úrovněmi a obrovským počtem listů.*

# Fyzický návrh databáze

Důležité je zanalyzovat zátěž systému a podle kritických míst vytvořit indexy:

* Vytvořit seznam dotazů a jejich četnosti
  + Zjistit, které tabulky se často čtou
  + Které atributy jsou často vybírány
  + A především které atributy se vyskytují v podmínkách (WHERE, JOIN)
* Vytvořit seznam aktualizací a jejich četností
  + Opět atributy, které se vyskytují v podmínkách
  + Jaké tabulky a sloupce se často modifikují

Jaké přístupové metody použít a pro které sloupce/kombinace sloupců? (Uspořádaný index, hashování, bitmapové indexy, shluky…)

Základní zásady rozhodování:

* Použít vůbec nějakou takovou metodu?
  + Nevytvářet index, pokud *nepřispěje* ke zrychlení: pokud nemají dotazy vysokou **selektivitu** (hodně různých hodnot, málo opakujících se hodnot), je zbytečné dělat index.
* Jak zvolit vyhledávací klíč?
  + Obvykle sloupce ve WHERE a JOIN.
  + Záleží na typu: pro dotazy na **rovnost** nebo **přirozené spojování** je výhodnější **hashování**; pro dotazy na **rozsah** je výrazně výhodnější B+ strom.
  + *Hashování je výhodné, pokud lze určit prostor potřebný pro statické hashování.*
* Použít složené vyhledávací klíče?
  + Pokud WHERE obsahuje podmínky pro více než jeden sloupec tabulky a je to kritický dotaz.
* Použít shlukování?
  + Může přispět při častém spojování řádků shlukovaných tabulek.
  + Shluk se nemusí vlézt do bloku na souboru – pokud se dá čekat, že záznamů pro jeden shlukovací klíč bude *velké* množství, nemusí nakonec přinést moc výrazné zlepšení
* A co režie indexu?
  + Pokud se často provádí nějaká významná aktualizace, údržba indexu je může nepříjemně zpomalovat.

Další možnosti zefektivnění mohou ležet ve změně návrhu tabulek:

* Alternativní normalizované schéma.
* V extrémních případech občas i denormalizace (snižuje nutnost spojování, ale má nebezpečné důsledky).
* Vertikální rozčlenění – pokud má tabulka spoustu sloupců, ale často se pracuje jen s malým počtem z nich (řádově menším), může se hodit rozdělit tu tabulku na dvě.
* Replikace dat.

Optimalizace dotazů

1. Převod do vnitřní reprezentace
2. Nalezení ekvivalentního, ale efektivnějšího výrazu
   * Využívá různé heuristiky (např. předřazování selekcí před spojení)
   * Sémantické optimalizace
3. Výběr kandidátů procedur pro provedení operací (podle **statistik**, indexů…)
4. Generování plánu (plánů) vyhodnocení (*execution plan*) a výběr nejlepšího podle nějaké ceny – obvykle podle počtu diskových operací, případně podle CPU času

## Využití databázových statistik k odhadu ceny

SŘBD uchovává různé statistiky o tabulkách, sloupcích v nich a o dalších objektech.

**Základní statistiky**: : počet n-tic v relaci, : velikost n-tice, počet různých hodnot atributu.  
**Selektivita atributu X relace R**: průměrný počet n-tic připadajících na jednu hodnotu atributu X v relaci R.

Viz slidy.

## Optimalizace v Oracle

**Režimy optimalizace**: minimální doba odezvy (FIRST\_ROWS), maximální propustnost (ALL\_ROWS).

ALTER SESSION SET OPTIMIZER\_MOD = ...

**Zjištění výsledného plánu vyhodnocení:**

EXPLAIN PLAN

**Ruční aktualizace statistik:**

ANALYZE objekt COMPUTE STATISTICS

Transakční zpracování

**Databázová transakce** je jednotka provádění programu, která zpřístupňuje anebo modifikuje data v databázi. Má čtyři základní vlastnosti – **ACID** model:

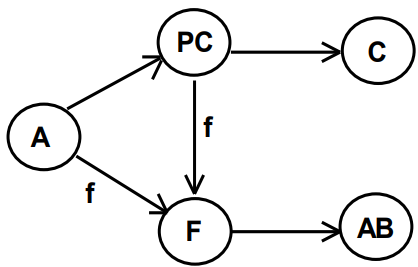
**Atomicity**: buď je provedena celá transakce, nebo *žádná* z databázových operací, které ji tvoří.

**Consistency**: izolovaná transakce zachovává *konzistenci* databáze. Pokud je transakce správně naprogramovaná, spuštěná a ukončená, nemělo by nikdy dojít ke stavu datové nekonzistence.

**Isolation**: při *souběžném* běhu více než jedné transakce je zajištěno, že pro každou dvojici transakcí A a B se v transakci B jeví, že A skončila dříve, než B zahájila provádění; nebo že A zahájí provádění až poté, co B skončí.

**Durability**: poté, co transakce úspěšně skončí, budou všechny změny v databázi mít *trvalý charakter*, a to i při výpadku systému (v kterémkoliv momentu).

## Stavy transakce



**Aktivní** (A): stav během provádění transakce.  
**Částečně potvrzená** (PC – partially commited): po provedení posledního příkazu, změny ještě nejsou trvalé.  
**Chyba** (F): po zjištění, že normální provádění není možné (třeba kvůli výpadku). Může do něj dojít i z PC – například nedošlo k uložení do žurnálu.  
**Zrušená** (AB): poté, co byly změny provedené transakcí anulovány (rollbacked); databáze je ve stavu před zahájením transakce.  
**Potvrzená** (C): transakce je úplně dokončená a uložená v databázi,

Transakce může být zahájena implicitně, nebo explicitně. Implicitně se transakčně provádí každý SQL příkaz. Nelze je zanořovat.

**Částečný rollback**: SAVEPOINT název, ROLLBACK název.

# Zotavení

**Zotavení**: obnovení konzistentního stavu databáze po výpadku systému (způsobeném naprosto čímkoliv).

Z pohledu zotavování rozlišujme paměti:

* Energeticky závislé – volatile (RAM)
* Nezávislé – non-volatile (HDD, SSD)
* *Stabilní* – stable (typicky nějaká zálohovací média)

K poruše dochází běžně na úrovni volatile pamětí (výpadek proudu…) i non-volatile (disk umře).

### Klasifikace výpadků

* **Výpadek transakce**
  + Logická chyba
  + Systémová chyba – vnitřní chyba transakčního zpracování, například deadlock
* Zhroucení systému
* Porucha disku

## Model přístup transakce k datům

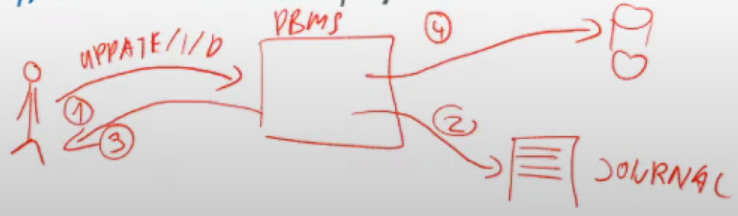
Transakce provádí operace *read* a *write* nad **mezipamětí** v RAM (dělat je přímo nad diskem by bylo pomalé). Caching probíhá typicky **nad celými bloky**.

Jak poznat, že došlo chybě (a data jsou nekonzistentní)? Jak poznat, *kde* k té chybě došlo?

Zavádíme **žurnálování**. K zajištění atomičnosti transakce je nutné **před** modifikací databáze uložit do stabilní paměti informace o této modifikaci.

**Žurnál** (deník transakcí, log file) je posloupnost záznamů, které zachycují všechny modifikace databáze. Záznamy mají různé typy:

* Ti start: zahájení transakce
* Ti, Xi, H1, H2: zápis datové položky Xi, původní hodnota H1, nová hodnota H2
* Ti commit: potvrzení změny, úspěšné dokončení transakce
* Ti abort: transakce byla zrušena



Uživatel dostane potvrzení o provedení transakce vždy až poté, co je uložena do žurnálu.

### Okamžitá modifikace databáze

Každá operace zápisu je provedena *ihned* poté, co je zapsána do žurnálu. V případě detekce výpadku je pak nutné u nedokončených transakcí vrátit původní hodnoty a zapsat nové hodnoty – operace **undo** a **redo**.

Pokud žurnál obsahuje **Ti start**, ale ne **Ti commit**, provde se **undo(Ti)** – jsou zrušeny všechny změny, které transakce provedla.

Pokud žurnál obsahuje **Ti start** i **Ti commit**, provede se **redo(Ti)** – data nemusela být z cache úspěšně zapsána na disk, preventivně je tedy zapíšeme znovu.

Pokud uživatel neudělá commit, ale **rollback**, provede se **undo**! Máme okamžitý zápis na disk, takže to z něj musíme dostat zase zpátky. Proto je to ve srovnání s commitem celkem drahá operace.

### Kontrolní body

Jakmile je na disk skutečně vyprázdněna vyrovnávací paměť, do žurnálu je uložen **checkpoint**. Při následné kontrole databáze podle žurnálu je nutné provést pouze transakce, které byly *úspěšně dokončeny*, commitnuty, **před** checkpointem. (Ale je nutné provést **redo** i u transakcí, které byly commitnuty až po checkpointu.)

Také můžeme z žurnálu mazat kusy před posledním checkpointem – abychom neměli na disku obrovský žurnál.

## Správa vyrovnávacích pamětí

Datové položky se nezapisují přímo na disk a ideálně nechceme okamžitě do stabilní paměti zapisovat ani záznamy žurnálu.

Máme tedy několik zásad:

* Transakce se dostává do stavu potvrzeno až po uložení **záznamu žurnálu** **Ti commit** do stabilní paměti.
* Před záznamem **Ti commit** musí být uloženy všechny **záznamy žurnálu** týkající se transakce Ti.
* Před uložením bloku **dat** na disk musí být uloženy do stabilní paměti uloženy všechny **záznamy žurnálu o tomto bloku**.

## Poruchy non-volatile paměti

Zálohovat, zálohovat, zálohovat.  
**Archivace** (backup): ukládání obsahu databáze do stabilní paměti (typicky pravidelné).  
**Obnova** (restore): obnovení databáze do stavu před poslední archivací.

Typický postup archivace:

* Uložit záznamy **žurnálu** do **stabilní** paměti,
* uložit modifikované **bloky** **DB** z paměti na **disk**,
* uložit **kopii** **DB** z **disku** do **stabilní** paměti,
* vytvořit záznam **dump** v **žurnálu** a zapsat jej do **stabilní** paměti.

Pokud je tedy ve stabilní paměti zazálohován stav DB v nějakém bodě a žurnál od tohoto bodu, stačí obnovit tento stav, projet žurnál a databáze je tím obnovena.

# Řízení souběžného přístupu

Databáze vytváří pro souběžné transakce, respektive pro jejich jednotlivé *instrukce*, **plán provádění**.

Nejjednodušším typem plánu je **sériový plán**: instrukce jedné transakce bezprostředně za sebou. Ten samozřejmě triviálně *zachovává konzistenci*, ale z praktického hlediska je **nepoužitelný** – takový databázový server by byl extrémně pomalý.

Typické problémy při řízení souběžného přístupu:

* Ztráta aktualizace (**lost update**): přepis zápisu zápisem jiné transakce.
* Závislost na potvrzení načtené hodnoty (**dirty read**): transakce přečte data, která zapsala jiná transakce, která potom ale bude rollbacknuta.
* **Nekonzistentní analýza**: některé věci (*analytická data*,například při zjišťování součtu všech zůstatků napříč účty…) budou započítány několikrát nebo ani jednou.

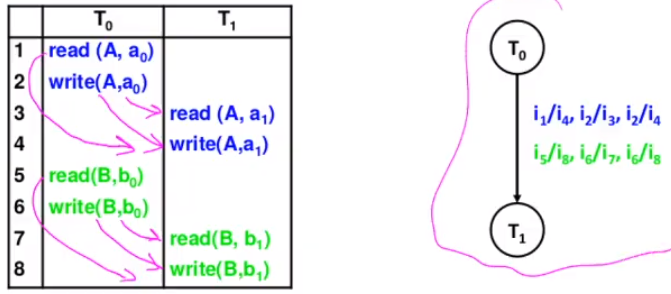
Musíme tedy zajistit, aby byl plán provádění „funkčně ekvivalentní“ se sériovým plánem – zavádíme **uspořádatelné plány**.

Zavádíme relaci **je v konfliktu**: dvě transakce jsou v konfliktu, pokud v množině instrukcí jedné a v množině instrukcí druhé najdeme dvojici, která přistupuje ke stejnému objektu a alespoň jednou z nich je instrukce zápisu.

Plány S a S‘ označujeme jako **ekvivalentní vzhledem ke konfliktům**, lze-li plán S transformovat na plán S‘ přehozením **nekonfliktních** instrukcí.

Plán S je **uspořádatelný vzhledem ke konfliktům**, pokud existuje sériový plán, který je *ekvivalentní* *s* S *vzhledem* *ke* *konfliktům*. Označuje se také jako **serializovatelný plán**.

**Graf relace precedence** je graf reprezentující relaci **Ti předchází Tj** implikovanou konfliktními instrukcemi. Plán je **serializovatelný**, právě když je odpovídající graf precedence **acyklický**.



Šipky představují konfliktní instrukce.

## Uzamykací protokoly

Podstata: transakce před přístupem k *objektu* databáze požádá o přidělení *zámku* pro tento objekt. Typicky existují dva druhy zámků: **sdílený zámek** a **výlučný zámek**. Kompatibilní jsou pouze dvojice sdílených zámků – pokud existuje nějaký výlučný zámek, neexistuje pro stejný objekt **žádný jiný zámek**.

Zamykání samo o sobě nám nestačí – musíme inteligentně vymyslet, kdy to zase odemknout! Špatné sekvence zamykání a odemykání povedou na nekonzistentní analýzu nebo na deadlocky.

**Uzamykací protokol** je soustava pravidel stanovující, kdy může transakce uzamknout či odemknout záznam. Často se používá dvoufázový uzamykací protokol:

**Fáze růstu – growing**: transakce uzamyká podle potřeby objekty, ale žádný neodemyká.  
**Fáze zmenšování – shrinking**: transakce odemyká objekty, ale žádný nesmí uzamknout.

Zajišťuje *uspořádatelnost*, ale **nevylučuje deadlocky**.

Typicky jsou všechny zámky uvolněny až poté, co se transakce potvrdí, nebo když je zrušena.

V databázových systémech je implementován **správce uzamykání** – používá tabulku zámků a čekajících transakcí.

**Granularita uzamykání**: udává, jak velká část podléhá uzamykací operaci (řádek/blok/tabulka/databáze…)

## Deadlocky

**Zablokování** (uváznutí, deadlock) je stav, kdy dvě nebo více transakcí, které jsou v aktivním stavu, nemohou pokračovat v provádění, protože si navzájem blokují požadované systémové prostředky – zámky.

Varianty řešení: buď prevencí (použitím protokolu zabraňujícího zablokování – například získáním *všech* zámků hned na začátku), nebo ex-post (potom, co nastane – timeout; nebo průběžnou analýzou grafu závislostí). Když databáze detekuje deadlock, typicky se některá transakce **zabije** a je naplánovaná na později.