Kapitola 1: Úvod

- Účel databázových systémů
- Pohled na data
- Modely dat
- Jazyk pro definici dat (Data Definition Language; DDL)
- Jazyk pro manipulaci s daty (Data Manipulation Language; DML)
- Správa transakcí
- Správa ukládání dat
- Správce databáze
- Uživatelé databáze
- Celková struktura systému

Systém pro správu databáze (Database Management Systém DBMS)

- Soubor dat ve vzájemném vztahu
- Sada programů pro přístup k datům
- DBMS obsahuje informace o jednom konkrétním „podnikání“
- DBMS poskytuje prostředí, které je pohodlné i účinné

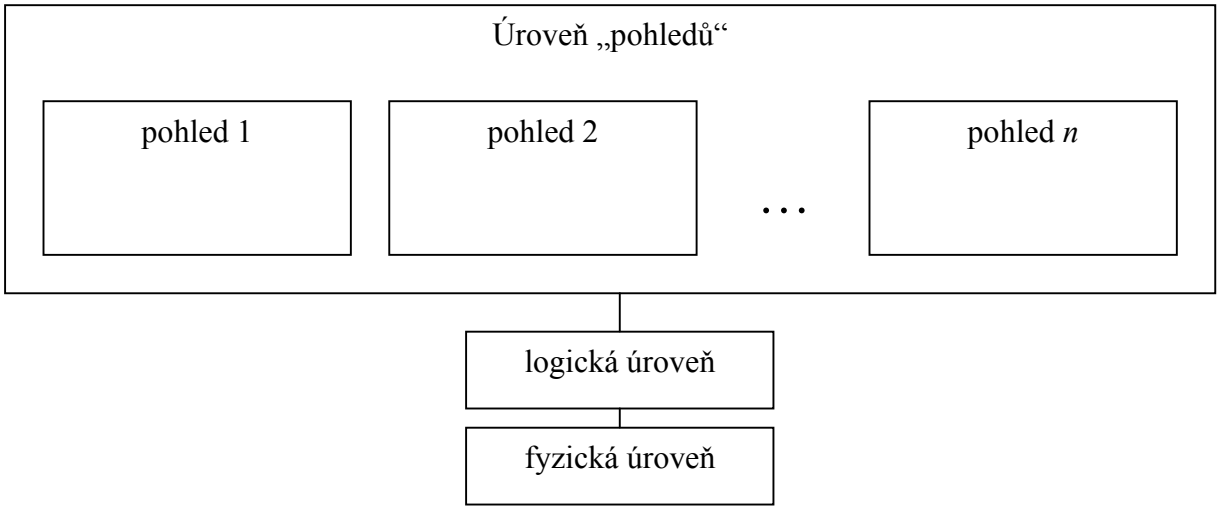
Účel databázových systémů

Systémy pro správu databází byly vyvinuty kvůli zvládnutí následujících problémů při zpracování souborů v tradičních operačních systémech:

- Redundance a inkonsistence dat
- Problémy s přístupy k datům
- Izolace dat – různé soubory a formáty
- Problémy s integritou
- Jedinečnost (atomicita) aktualizací
- Současný přístup více uživatelů
- Bezpečnostní problémy

Pohled na data

Architektura databázového systému:



Úrovně abstrakce

- Fyzická úroveň: popisuje, jak je záznam (např. *customer*) uložen
- Logická úroveň: popisuje data uložená v databázi a vztahy mezi nimi.

```
type customer = record
        name: string;
        street: string;
        city: integer;
end;
```

- Úroveň „pohledů“: aplikační programy skrývají detaily o typech dat. *Pohledy* mohou také skrývat informace (jako např. *plat*) – např. z bezpečnostních důvodů.

Instance a schémata

- Analogie s *typy* a *proměnnými* v programovacích jazycích
- *Schéma* – logická struktura databáze (např. množina *zákazníků* a *účtů* + vztah mezi nimi)
- *Instance* – aktuální obsah databáze v konkrétním časovém okamžiku

Nezávislost dat

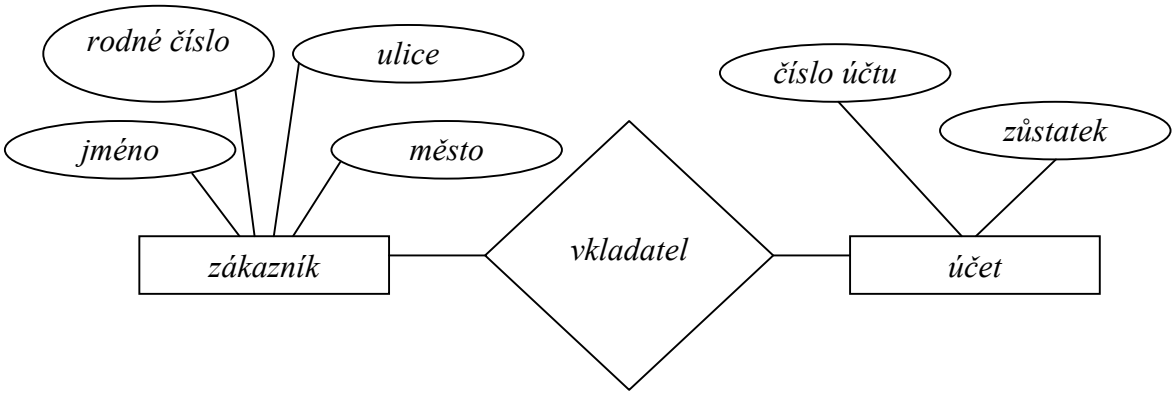
- Možnost měnit *definici schématu* na jedné úrovni bez vlivu na *definici schématu* na jiné vyšší úrovni.
- *Rozhraní* mezi různými úrovněmi a komponentami by měla být dobře definována, aby změny v některých částech neměly významný vliv na jiné části.
- Dva stupně nezávislosti dat:
  - Fyzická nezávislost dat
  - Logická nezávislost dat

Modely dat

- Sada nástrojů pro popis:
  - dat
  - vztahů mezi daty
  - sémantiky dat
  - omezení dat (*data constraints*)
- Logické modely založené na objektech
  - *entitně-vztahový model* (entity-relationship model)
  - *objektově orientovaný model*
  - sémantický model
  - funkcionální model
- Logické modely založené na záznamech
  - *relační model* (např. MySQL, Oracle)
  - síťový model
  - hierarchický model (např. IMS)

Entitně-vztahový model (Entity-relationship model)

Příklad entitně-vztahového diagramu



Relační model

Příklad dat v tabulce v relačním modelu:

jméno zákazníka	*rodné číslo*	*ulice*	*město*	*číslo účtu*
Johnson	800327/6655	Alma	Palo Alto	A-101
Smith	735203/5312	North	Rye	A-215
Johnson	800327/6655	Alma	Palo Alto	A-201
Jones	605229/2929	Main	Harrison	A-217
Smith	735203/5312	North	Rye	A-201

číslo účtu	*zůstatek*
A-101	500
A-201	900
A-215	700
A-217	750

Jazyk pro definici dat (Data definition language; DDL)

- Značení pro specifikaci definice schématu databáze
- DDL kompilátor generuje množinu tabulek uložených v *datovém slovníku* (*Data dictionary*)
- Datový slovník obsahuje *metadata* (data o datech)
- *Jazyk pro ukládání a definice dat* (*Data storage and definition language*) – speciální typ DDL, pomocí kterého se specifikuje struktura *uložení dat* a metody použité databázovým systémem k *přístupu k datům*

Jazyk manipulace s daty (Data manipulation language; DML)

- Jazyk pro zpřístupnění a manipulaci s daty organizovanými příslušným datovým modelem
- Dvě třídy jazyků
  - Procedurální – uživatel specifikuje, která data jsou vyžadována a jak je získat
  - Neprocedurální (deklarativní) – uživatel specifikuje, která data jsou vyžadována, ale ne, jak je získat

Správa transakcí (Transaction Management)

- *Transakce* je sada operací, které představují jednu logickou funkci v databázové aplikaci
- *Správa transakcí* zajišťuje, že databáze zůstává v konzistentním stavu nezávisle na selhání systému (např. výpadky energie a pády OS) nebo selhání transakcí.
- *Správa souběžnosti* (*Concurrency-control manager*) dohlíží na vzájemné ovlivňování mezi současně probíhajícími transakcemi, aby byla zajištěna konzistence databáze.

Správa ukládání dat (Storage Management)

- *Správce ukládání* je programový modul, který poskytuje rozhraní mezi daty uloženými v databázi na nízké úrovni a aplikacemi a dotazy posílanými systému.
- Správce ukládání je zodpovědný za následující úlohy:
  - interakce se správcem souborů
  - efektivní ukládání, získávání a aktualizace dat

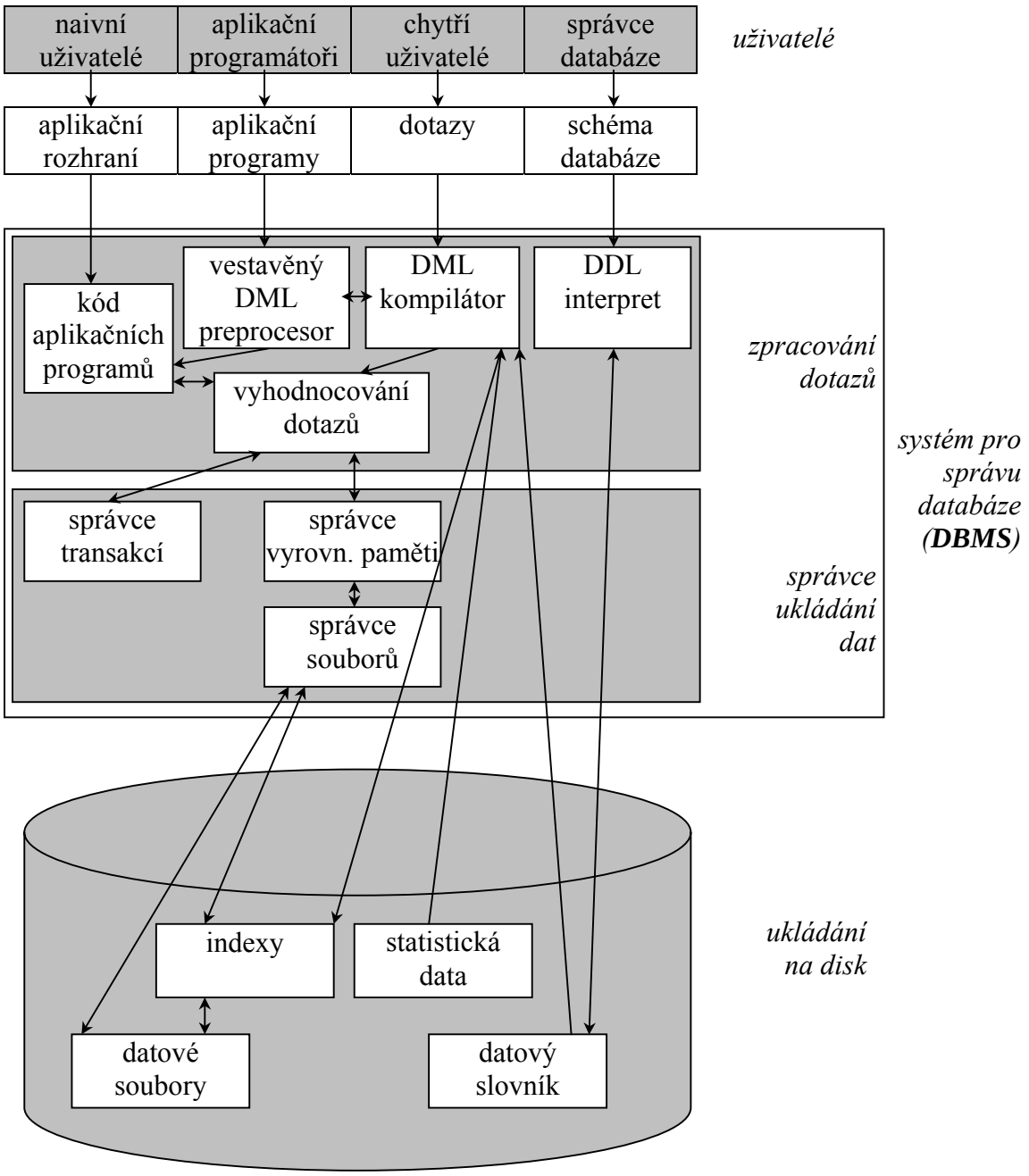
Správce databáze (Database Administrator)

- Koordinuje všechny aktivity v databázovém systému; má dobré znalosti o informačních zdrojích a potřebách podniku.
- Povinnosti správce databáze zahrnují:
  - Definice *schématu*
  - Definice *struktury ukládání* a *metody přístupu*
  - *Modifikace* schématu a fyzické organizace
  - Přiděluje uživatelům *práva přístupu* k databázi
  - Specifikuje *integritní omezení*
  - Styčný bod s uživateli
  - Monitoruje *výkon* a zodpovídá za změny v požadavcích

Uživatelé databáze (Database Users)

- Uživatelé jsou rozděleni podle toho, jakým způsobem spolupracují se systémem
- *Aplikační programátoři* – pracují se systémem pomocí volání DML
- *Chytří uživatelé* – formulují dotazy v databázovém dotazovacím jazyku
- *Speciální uživatelé* – píší speciální databázové aplikace, které nespadají do klasického zpracování dat
- *Naivní uživatelé* – spouští jeden z předem napsaných aplikačních programů

Celková struktura systému



Kapitola 2: Entitně-vztahový model (Entity-Relationship model)

- Množiny entit
- Množiny vztahů
- Otázky návrhu
- Plánování mezí
- Klíče
- E-R diagram
- Rozšířené E-R rysy
- Návrh E-R databázového schématu
- Redukce E-R schématu na tabulky

Množiny entit (entitní množiny)

- *Databáze* může být modelována jako:
  - množina entit
  - vztahy mezi entitami
- *Entita* je objekt, který existuje a je odlišitelný od ostatních objektů.
  Např.: nějaká osoba, společnost, událost, rostlina
- *Množina entit* je skupina entit stejného typu, které sdílejí stejné vlastnosti.
  Např.: skupina všech osob, firem, stromů

Atributy

- Entita je reprezentována množinou atributů, to jsou popisné vlastnosti všech členů množiny entit.
  Např.:
  zákazník = (*jméno, rodné_číslo, ulice, město*)
  účet = (*číslo účtu, zůstatek*)
  tj. *entita* = (*atributy, …*)
- *Doména* – množina povolených hodnot pro každý atribut
- Typy atributů:
  - *Jednoduché* atributy (*jméno*) a *složené* atributy (*datum*).
  - Atributy s *jednoduchou hodnotou* (*single-valued*) (např. *jméno*) a s *násobnou hodnotou* (*vícehodnotové*) (*multi-valued*) (např. *telefonní čísla*)
  - *Nulové* atributy (např.: *nemá telefon*) (**null**)
  - *Odvozené* atributy (např.: *věk*)

Množiny vztahů

- *Vztah* je spojení mezi několika entitami
  Např.:

Novák	*vkladatel*	A-102
entita *zákazník*	množina vztahů	entita *účet*

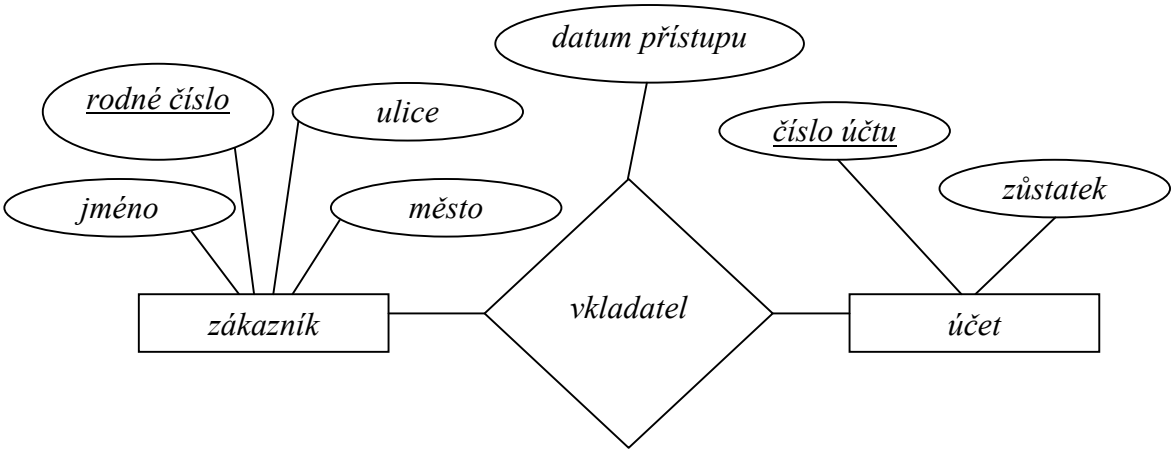
- *Množina vztahů* je matematická relace mezi $n \geq 2$ entitami, každá je braná z konkrétní množiny entit

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

kde $(e_1, e_2, \ldots, e_n)$ je *vztah*, $e_1, e_2, \ldots, e_n$ jsou entity a $E_1, E_2, \ldots, E_n$ entitní množiny např.:

$$(\text{Novák, A-102}) \in \textit{vkladatel}$$

- Množina vztahů může mít také atributy. Např. množina vztahů *vkladatel* mezi množinami entit *zákazník* a *účet* může mít atribut *(poslední) datum přístupu*.

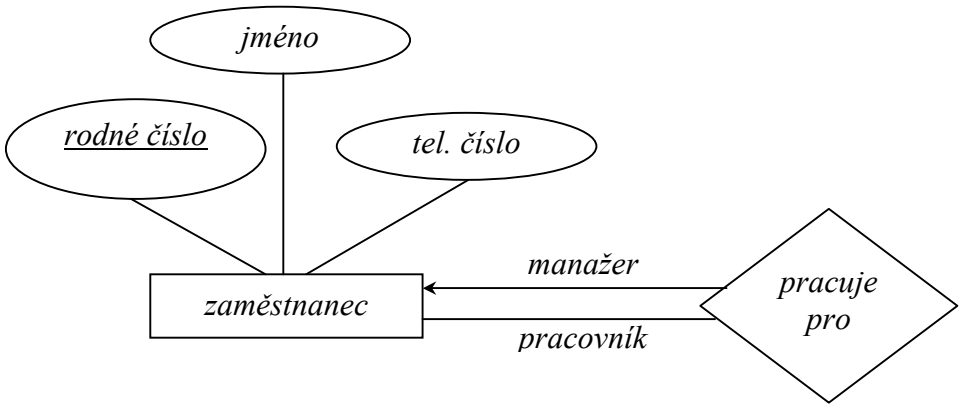


Stupeň vztahu

- Ukazuje počet množin entit, které jsou součástí množiny vztahů.
- Množiny vztahů, které zahrnují 2 množiny entit, se nazývají *binární* (nebo stupně 2). Obecně, většina vztahů v databázovém systému je binární.
- Množiny vztahů můžou zahrnovat více než 2 množiny entit. Např. množiny entit *zákazník*, *půjčka* a *pobočka* můžou být spojeny ternární (stupně 3) množinou vztahů *CLB*.

Role

Množiny entit u vztahů nemusí být rozdílné



- Popisky *manažer* a *pracovník* jsou nazývány *role*; specifikují, jak na sebe entity typu *zaměstnanec* vzájemně působí přes množinu vztahů *pracuje pro*.
- Role jsou v E-R diagramech znázorněny popisky u čar, které spojují kosočtverce s obdélníky.
- Popisky rolí jsou dobrovolné a jsou používány pro zvýraznění *sémantiky* (významu) vztahu

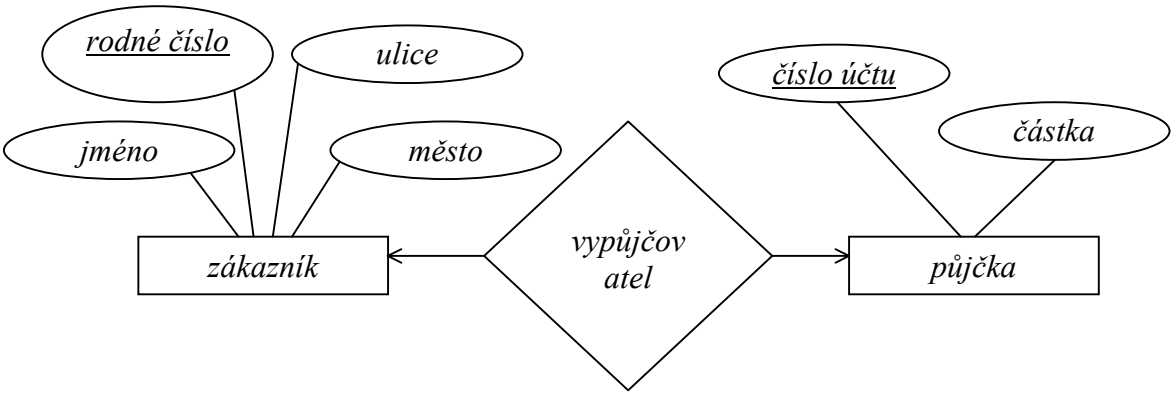
Otázky návrhu (Design Issues)

- Použití *entitní množiny* vs. *atributu*
  Výběr závisí zejména na struktuře podniku a na významu daného atributu. Např.: *student – ročník*.
- Použití *entitní množiny* vs. *množiny vztahů*
  Možným vodítkem může být sestrojení množiny vztahů pro popis akce, která se odehrává mezi entitami. Např.: *vkladatel*
- *Binární* vs. *n*-ární množiny vztahů
  Přestože je možné nahradit ne-binární (*n*-ární, pro n > 2) množinu vztahů několika různými binárními množinami vztahů, *n*-ární ukazuje mnohem jasněji, že několik entit je součástí jednoho vztahu. Např. *zákazník*, *půjčka*, *pobočka*.

Četnosti vztahů (Mapping Cardinalities)

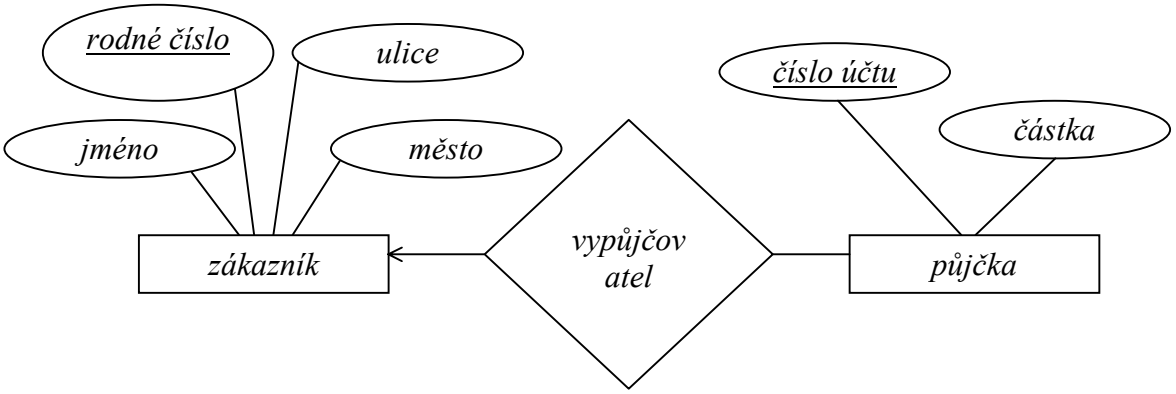
- Označuje počet entit, se kterými mohou být ostatní entity propojeny pomocí množiny vztahů.
- Nejužitečnější je v popisu binárních množin vztahů.
- Pro binární množinu vztahů musí být četnost jednoho z následujících typů:
  - jedna na jednu
  - jedna na mnoho
  - mnoho na jednu
  - mnoho na mnoho
- Mezi těmito typy rozlišujeme kreslením buď šipky (→) značící *jeden* nebo normální čáry (–) značící *mnoho* mezi množinou entit a vztahů.

Vztah jedna na jednu (One-to-one)

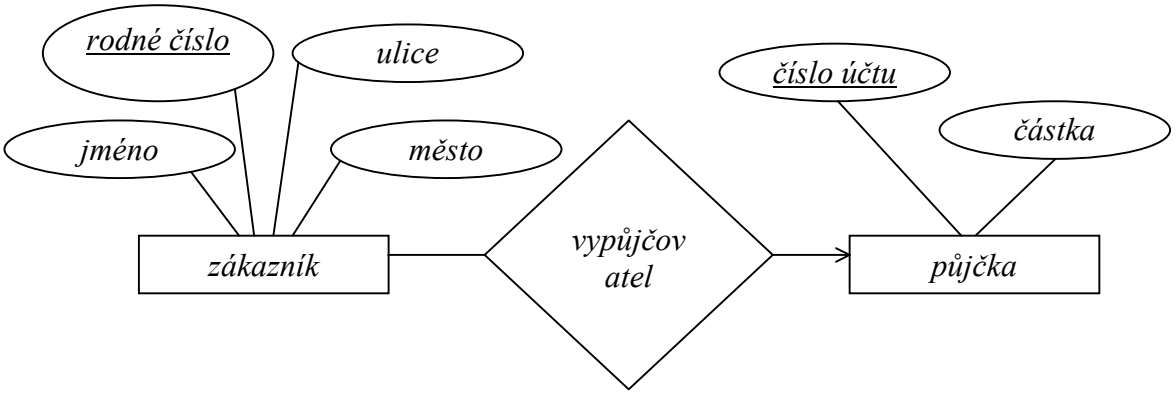


- Zákazník je spojen s nejvýše jednou půjčkou vztahem *vypůjčovatel*.
- Půjčka je spojena s nejvýše jedním zákazníkem vztahem *vypůjčovatel*.

Vztahy jedna na mnoho a mnoho na jednu (One-to-many a many-to-one)

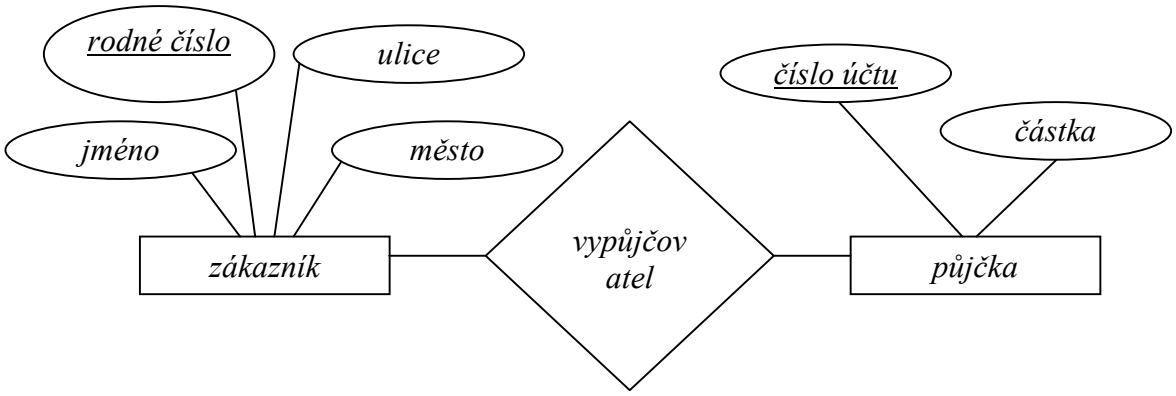


- Ve vztahu jedna na mnoho je půjčka spojena s nejvýše jedním zákazníkem a zákazník je spojen s žádnou nebo několika půjčkami vztahem *vypůjčovatel*.



- Ve vztahu mnoho na jednu je půjčka spojena s žádným nebo několika zákazníky a zákazník je spojen s nejvýše jednou půjčkou vztahem *vypůjčovatel*.

Vztah mnoho na mnoho (Many-to-many)



- Zákazník je spojen s žádnou nebo několika půjčkami vztahem *vypůjčovatel*
- Půjčka je spojena s žádným nebo několika zákazníky vztahem *vypůjčovatel*

Existenční závislost

- Závisí-li existence entity *x* na existenci entity *y*, pak *x* se nazývá existenčně závislé (*existence dependent*) na *y*.
  - *y* je *dominantní entita* (v příkladu níže *půjčka*)
  - *x* je *podřízená entita* (v příkladu níže *splátka*)

- Je-li entita *půjčka* smazána, pak všechny s ní spojené entity *splátka* musí být smazány také.

Klíče

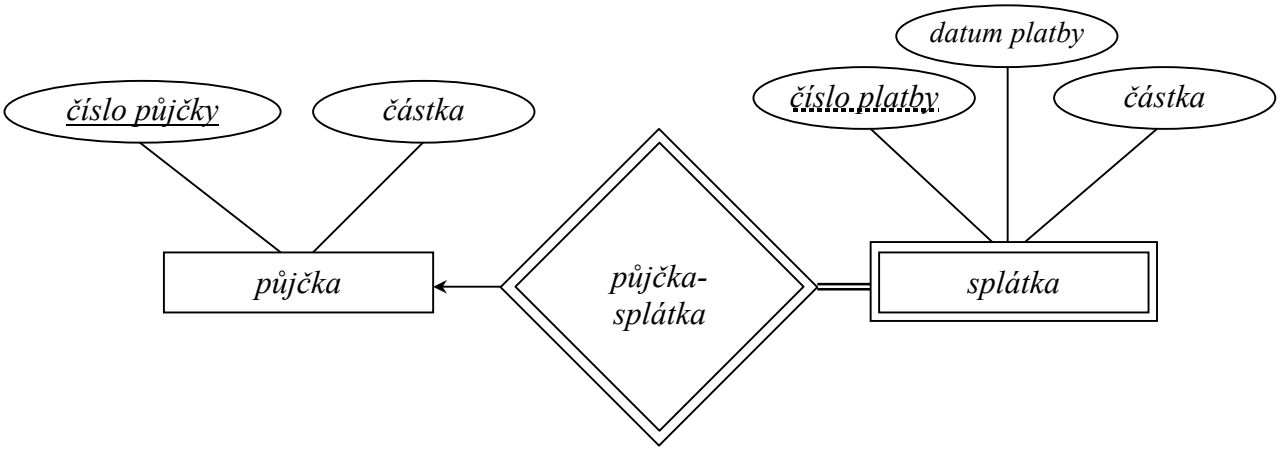
- *Super klíč* množiny entit je množina jednoho nebo více atributů, jejichž hodnoty jednoznačně určují entitu (tedy *klíč* je *podmnožina atributů* – např. všechny atributy).
- *Kandidátní klíč* množiny entit je minimální super klíč.
  - *rodné číslo* je kandidátní klíč entity *zákazník*
  - *číslo účtu* je kandidátní klíč je kandidátní klíč entity *účet*
- Protože může existovat několik kandidátních klíčů, jeden z nich je vybrán jako *primární klíč*.
- Sjednocení primárních klíčů zúčastněných entitních množin určuje kandidátní klíč pro množinu vztahů.
  - při výběru *primárního klíče* musíme dávat pozor na četnosti vztahů a sémantiku množiny vztahů
  - např. (*rodné číslo*, *číslo účtu*) je primární klíč množiny vztahů *vkladatel*

Komponenty E-R diagramu

- **Obdélníky** reprezentují množiny entit.
- **Elipsy** reprezentují atributy.
- **Kosočtverce** reprezentují množiny vztahů.
- **Čáry** spojují atributy s množinami entit a množiny entit s množinami vztahů.
- **Dvojité elipsy** reprezentují vícehodnotové atributy.
- **Čárkované elipsy** označují odvozené atributy.
- Atributy primárního klíče jsou podtržené.

Slabé množiny entit

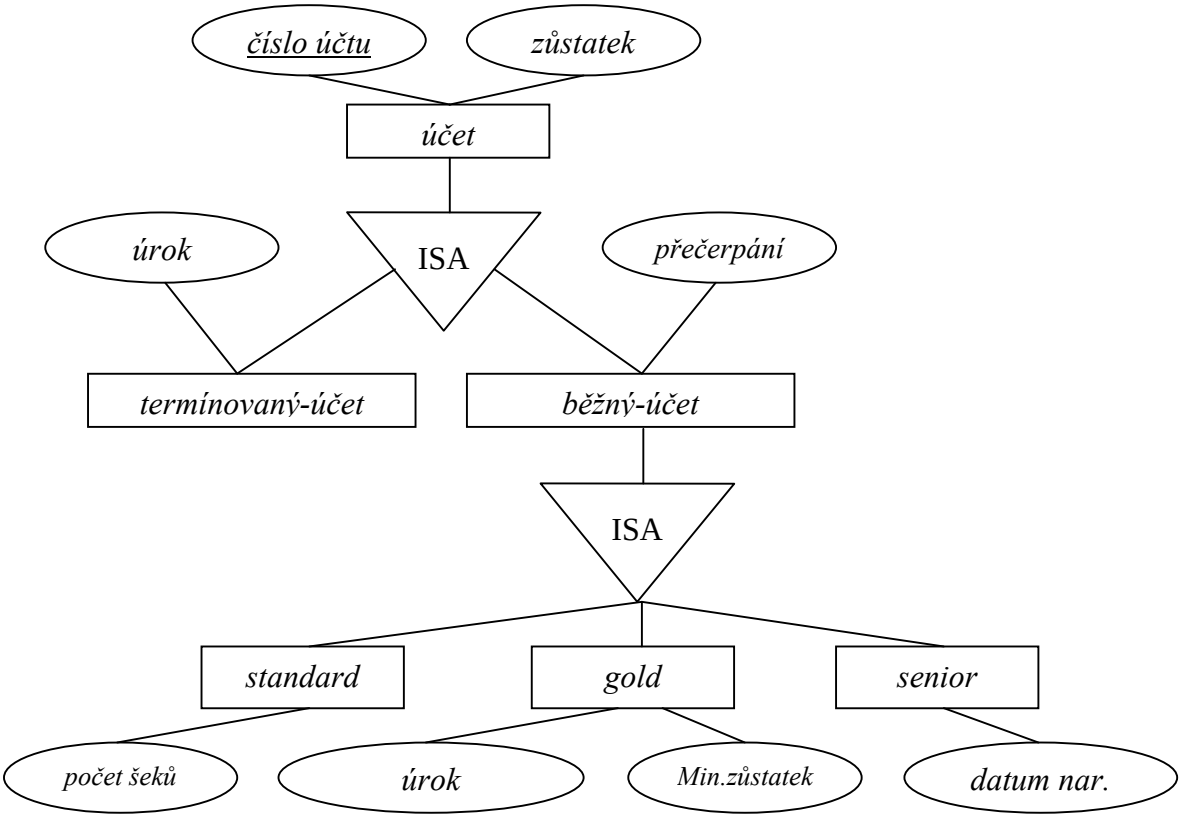
- Množina entit, která nemá primární klíč, se nazývá *slabá množina entit*.
- Existence slabé množiny entit závisí na existenci silné množiny entit; musí být spojena se silnou množinou vztahem mnoho na jednu.
- *Diskriminátor* (*parciální klíč*) slabé množiny entit je množina atributů, která od sebe odlišuje entity slabé množiny
- Primární klíč slabé množiny je tvořen *primárním klíčem silné množiny*, na níž je tato množina závislá a *parciálním klíčem* této slabé množiny.



- Slabé množiny entity znázorňujeme dvojitým obdélníkem.
- Parciální klíč slabé množiny entit se podtrhává přerušovanou čárou.
- *číslo splátky* – parciální klíč množiny entit *splátka*
- Primární klíč pro množinu *splátka* je (*číslo půjčky*, *číslo splátky*)

Specializace

- Tvoříme podskupiny v množině entit, které jsou různé od ostatních entit v množině (proces seshora dolů)
- Tyto podskupiny se stávají množinami entit nižší úrovně, které mají atributy nebo jsou součástí množin vztahů, které se nepromítají do množiny vztahů vyšší úrovně.
- Znázorňujeme trojúhelníkovou komponentou označenou ISA (*termínovaný vklad* „je (*is a*)“ *účet*)



Generalizace (Zobecnění)

- Kombinujeme několik množin entit, které sdílejí stejné rysy do množiny entit vyšší úrovně (proces zezdola nahoru)
- Specializace a generalizace jsou jednoduše vzájemně inverzní; jsou reprezentovány E-R diagramem stejným způsobem.
- **Dědičnost atributů** – entita nižší úrovně dědí všechny atributy a účasti ve vztazích z množiny entit vyšší úrovně.

Omezení pro specializaci/generalizaci

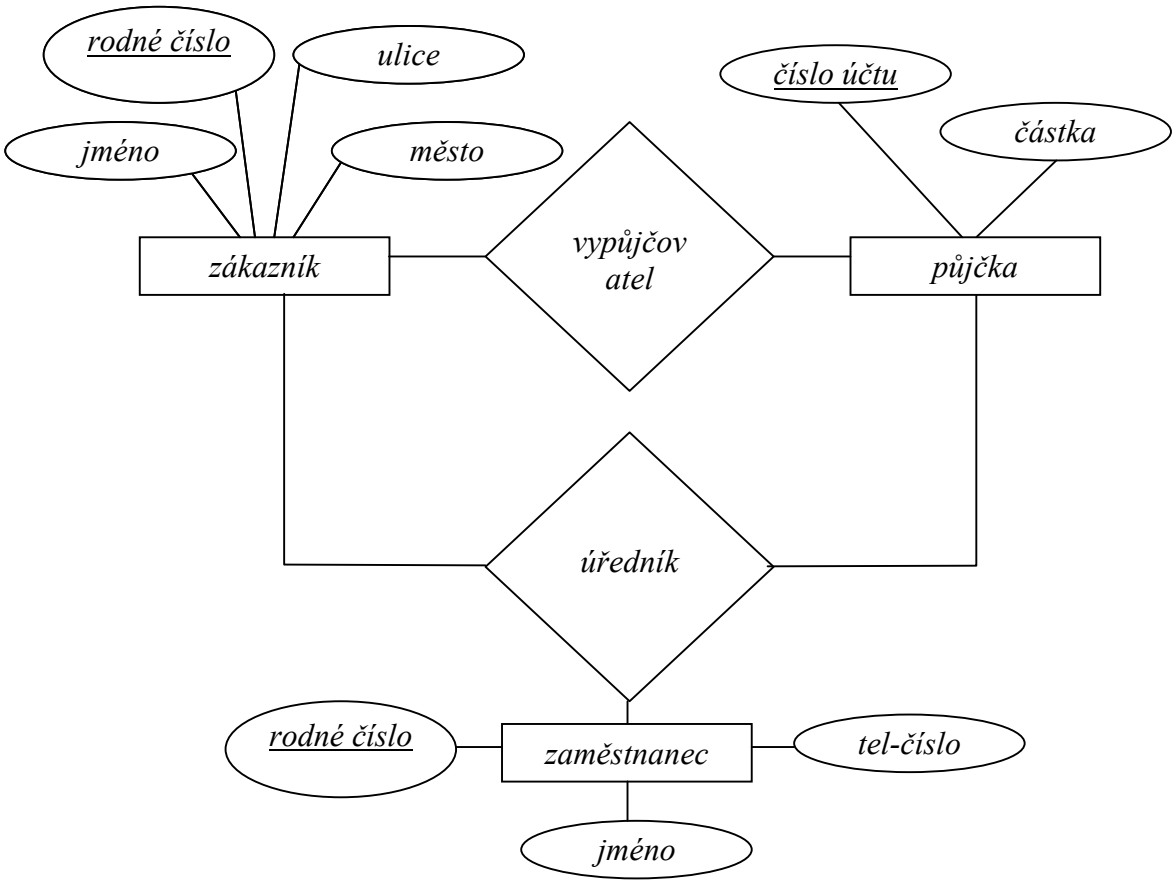
Omezení na to, které entity mohou být prvky entitní množiny na nižší úrovni:
- omezení daná nějakou *podmínkou*
- omezení definovaná *uživatelem* (pro každou entitu zvlášť)

Omezení na to, jestli může entita patřit jen do jedné nebo do více entitních množin na jedné úrovni jedné generalizace:
- disjunktní
- překrývající se

Omezení na to, jestli každá entita z vyšší třídy musí nebo nemusí patřit do jedné z entitních množin na nižší úrovni:
- úplná specializace
- částečná specializace

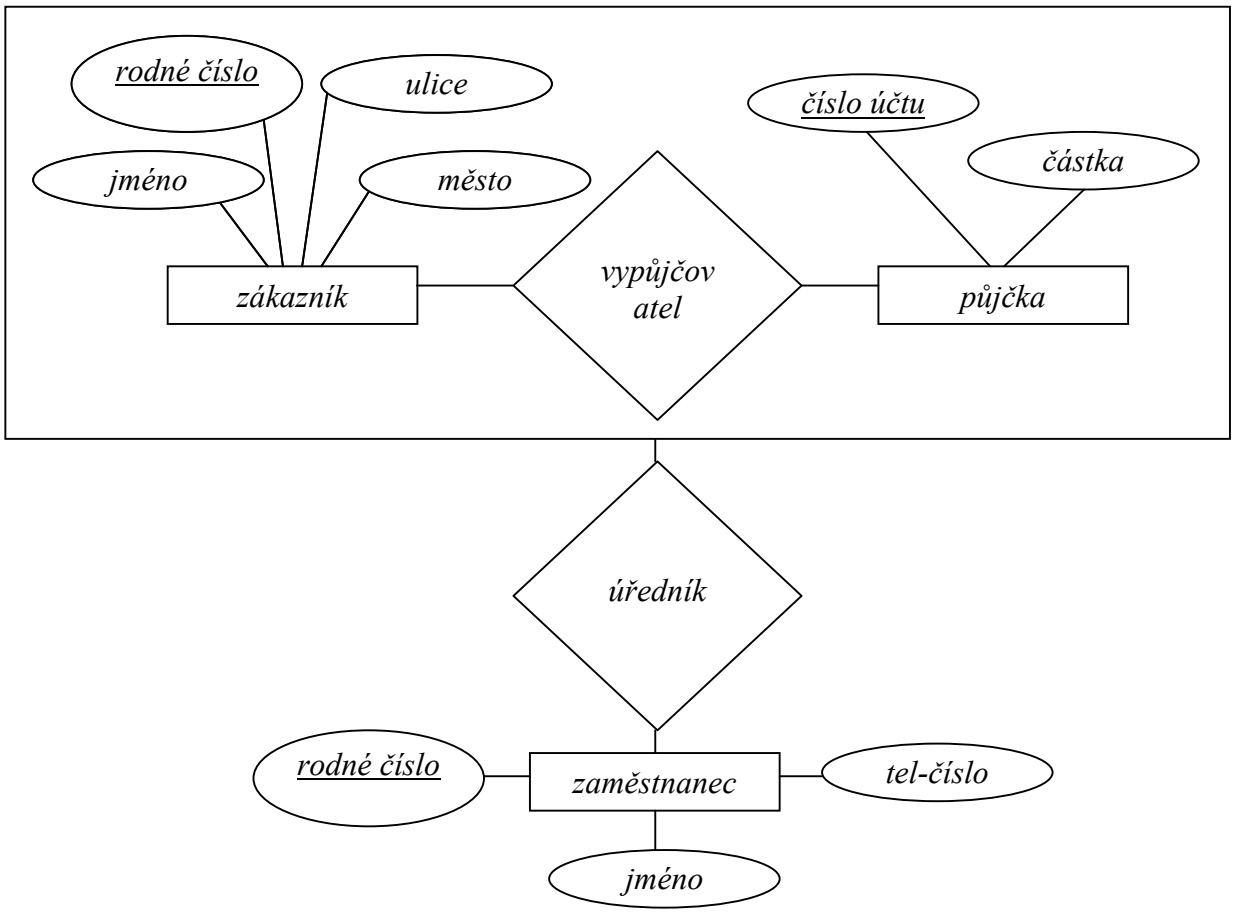
Agregace

- Dlužník (*zákazník*) může být kontrolován *úředníkem*.



- Množiny vztahů *vypůjčovatel* a *úředník* reprezentují stejnou informaci.
- Tuto redundanci eliminujeme agregací
  - Se *vztahem* zacházíme jako s abstraktní entitou
  - Umožňuje vztahy mezi vztahy
  - Abstrakce vztahu do nové entity

- Následující diagram reprezentuje:
  - Zákazník si vezme půjčku
  - *Zaměstnanec* může být *úředníkem* pro dvojici *zákazník-půjčka*



Rozhodnutí o návrhu E-R schématu

- Použití atributu nebo množiny entit pro reprezentaci objektu.
- Je význam reálného světa lépe vyjádřen entitní množinou nebo množinou vztahů?
- Použití ternárního vztahu vs. páru binárních vztahů.
- Použití silných nebo slabých množin entit.
- Použití generalizace – přispívá k modularitě návrhu.
- Použití agregace – můžeme zacházet s agregovanou množinou entit jako s jednotkou bez ohledu na podrobnosti a její vnitřní strukturu.

Převod E-R schématu na tabulky

- Primární klíče umožňují vyjádřit množiny entit a vztahů jako tabulky reprezentující obsah databáze.
- Databáze, která odpovídá E-R diagramu, může být reprezentována jako kolekce tabulek.
- Pro každou *množinu entit* a *vztahů* je jedinečná tabulka, která je spojená se jménem příslušné množiny entit nebo vztahů.
- Každá tabulka má počet sloupců odpovídající atributům dané množiny entit/vztahů. Sloupce mají jedinečná *jména* v rámci jedné tabulky.
- Převod E-R diagramu na tabulky je základ pro odvozování návrhu relační databáze z E-R diagramu.

Reprezentace množin entit tabulkami

- Silná množina entit se převede na tabulku se stejnými atributy.

jméno	*rodné číslo*	*ulice*	*město*
Starý	800327/6655	Lidická	Brno
Slavík	645326/1258	Úzká	Klatovy
Novák	891117/1111	Lidická	Brno

Tabulka *zákazník*

- Slabá množina entit se převede na tabulku, která obsahuje i sloupec pro primární klíč identifikační silné entitní množiny.

číslo půjčky	*číslo splátky*	*datum splátky*	*částka splátky*
L-17	5	10.5.1996	50
L-23	11	17.5.1996	75
L-15	22	23.5.1996	300

Tabulka *splátka*

Reprezentace množin vztahů tabulkami

- Množina vztahů *mnoho na mnoho* je reprezentována jako tabulka se sloupci pro primární klíče dvou zúčastněných entitních množin a sloupce pro popisné atributy množiny vztahů.

rodné číslo	*číslo účtu*	*datum přístupu*
…	…	…

Tabulka *vkladatel*

- Tabulka odpovídající množině vztahů spojující slabou množinu entit s její identifikační silnou množinou je zbytečná. Tabulka *splátka* již obsahuje informace, které by se objevily v tabulce *půjčka-splátka* (tj. sloupce *číslo půjčky* a *číslo splátky*)

Reprezentace generalizace tabulkami

- Metoda č. 1: Sestrojíme tabulku pro nadřazenou entitu *účet*. Sestrojíme tabulku pro každou entitní množinu, která odvozena (na nižší úrovni) – zahrneme do těchto tabulek i primární klíč generalizované množiny.

tabulka	atributy tabulky
účet	*číslo účtu, zůstatek, typ účtu*
termínovaný účet	*číslo účtu, úroková sazba*
běžný účet	*číslo účtu, překročení*

- Metoda č. 2: Sestrojíme tabulku jen pro každou entitní množinu na nižší úrovni.

tabulka	atributy tabulky
termínovaný účet	*číslo účtu, zůstatek, úroková sazba*
běžný účet	*číslo účtu, zůstatek, překročení*

Metoda č. 2 netvoří žádnou tabulku pro generalizovanou entitu *účet*.

Vztahy odpovídající agregaci

zákazník

jméno	*rodné číslo*	*ulice*	*město*

půjčka

číslo půjčky	*částka*

vypůjčovatel

rodné číslo	*číslo půjčky*

zaměstnanec

rodné číslo	*jméno*	*tel. číslo*

úředník

rodné číslo (zaměstnanec)	*rodné číslo (zákazník)*	*číslo půjčky*

Kapitola 3: Relační model

- Struktura relačních databází
- Relační algebra
- n-ticový relační kalkul
- Doménový relační kalkul
- Rozšířené operace relační algebry
- Modifikace databáze
- Pohledy

Základní struktura

- Mějme množiny A_1, A_2, …, A_n *relace* ***r*** je podmnožina kartézského součinu $A_1 \times A_2 \times \ldots \times A_n$. Tedy relace ***r*** je *množina n-tic* (a_1, a_2, …, a_n), kde $a_i \in A_i$
- Příklad: je-li
  jméno = {Novák, Starý, Coufal, Liška}
  ulice = {Hlavní, Severní, Sadová}
  město = {Hradec, Rájec, Polná}
  pak ***r*** = {(Novák, Hlavní, Harrison), (Starý, Severní, Rájec), (Coufal, Severní, Rájec), (Liška, Sadová, Polná)} je relace na *jméno* × *ulice* × *město*

Relační schéma

- A_1, A_2, …, A_n jsou *atributy*
- $R = (A_1, A_2, \ldots, A_n)$ je *relační schéma*
  zákazník-schéma = (*jméno*, *ulice*, *město*)
- $r(R)$ je *relace* (pojmenování) na relačním schématu R
  zákazník (zákazník-schéma)
  zákazník (jméno, ulice, město)

Instance relace

- Současné hodnoty (*instance relace*) jsou specifikovány tabulkou.
- Element *t* relace *r* je *n-tice* reprezentovaná *řádkem* v tabulce.

jméno	*Ulice*	*město*
Novák	Hlavní	Hradec
Starý	Severní	Rájec
Coufal	Severní	Rájec
Liška	Sadová	Polná

zákazník

Klíče

- Nechť $K \subseteq R$
- K je *superklíč* schématu R, jestliže hodnoty K jsou dostatečné pro jednoznačnou identifikaci n-tice každé možné relace $r(R)$.
  Např.: {*jméno*, *ulice*} a {*jméno*} jsou superklíče schématu *zákazník*, jestliže dva zákazníci nemají shodné jméno.
- K je *kandidátní klíč*, jestliže K je minimální.
  Např.: {*jméno*} je kandidátní klíč schématu *zákazník*, jestliže dva zákazníci nemají shodné jméno a žádná jeho podmnožina není superklíč.

Odvozování klíčů z E-R množin

- **Silná entitní množina.** Primární klíč množiny se stává primárním klíčem relace.
- **Slabá entitní množina.** Primární klíč relace je sjednocení primárního klíče silné množiny entit a parciálního klíče slabé množiny.
- **Množina vztahů.** Sjednocení primárních klíčů spojených množin entit se stává superklíčem relace. Pro binární vztahy mnoho na mnoho je superklíč též primární klíč. Pro binární vztahy mnoho na jednu se stává primární klíč množiny entit „mnoho“ primárním klíčem relace. Pro vztahy jedna na jednu může být primární klíč relace z každé množiny entit.

Dotazovací jazyky

- Jazyk, kterým uživatel žádá informace z databáze.
- Kategorie jazyků:
  - *Procedurální* = uživatel specifikuje jaké operace má systém provést pro získání výsledku; uživatel určí posloupnost kroků k výpočtu výsledku
  - *Neprocedurální* = uživatel určí jak má vypadat výsledek, ale neříká jak výsledek vytvořit
- Čisté jazyky („*pure*“):
  - Relační algebra (procedurální)
  - *N*-ticový relační kalkul (neprocedurální)
  - Doménový relační kalkul (neprocedurální)
- Čisté jazyky jsou stručné, formální, bez jakéhokoli syntaktického cukru.
- Čisté jazyky jsou základem dotazovacích jazyků, které se běžně používají.

Relační algebra

- Procedurální jazyk
- 6 základních operátorů
  - výběr (selekce)
  - projekce
  - sjednocení
  - rozdíl množin
  - kartézský součin
  - přejmenování
- Operátory berou jednu nebo více relací jako vstup a vrací novou relaci jako výsledek.

Operace výběr (selekce)

- Značení: $\sigma_P(r)$
- Definováno jako:

$$\sigma_P(r) = \{t \mid t \in r \text{ and } P(t)\}$$

Kde P je výraz v propozičním kalkulu, skládající se z podmínek ve formě:

<atribut> $=$ <atribut> nebo <konstanta>
$\neq$
$>$
$\geq$
$<$
$\leq$

„spojené pomocí“: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)

Příklad selekce

- Relace *r*:

A	*B*	*C*	*D*
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A = B \wedge D > 5}(r)$ – vybere řádky, kde A = B a zároveň D > 5

A	*B*	*C*	*D*
α	α	1	7
β	β	23	10

Operace projekce

- Značení: $\Pi_{A_1, A_2, \ldots, A_k}(r)$

kde A_1, $A_{2,\ldots,}$ A_k jsou jména atributů a r je jméno relace.

- Výsledek je definován jako relace k sloupců, které dostaneme smazáním sloupců, které nejsou vyjmenovány
- Duplicitní řádky jsou z výsledku odstraněny, protože relace je množina.

Příklad projekce

- Relace r:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\Pi_{A, C}(r)$

A	C
α	1
~~α~~	~~1~~
β	1
β	2

=

A	C
α	1
β	1
β	2

Operace sjednocení

- Značení: $r \cup s$
- Definováno jako:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- Aby bylo sjednocení $r \cup s$ platné,
  - r a s musí mít *stejnou aritu* (stejný počet atributů)
  - Domény atributů musí být kompatibilní (např. 2. sloupec r obsahuje hodnoty stejného typu jako 2. sloupec s)

Příklad sjednocení

- Relace r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cup s$

A	B
α	1
α	2
β	1
β	3

Operace rozdíl množin

- Značení: $r - s$
- Definováno jako:

$$r - s = \{t \mid t \in r \text{ and } t \notin s\}$$

- Rozdíly množin musí být brány mezi kompatibilními relacemi.
  - r a s musí mít stejnou aritu
  - domény atributů relací r a s musí být kompatibilní

Příklad rozdílu množin

- Relace r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r - s$

A	B
α	1
β	1

Operace kartézského součinu

- Značení: $r \times s$
- Definováno jako:

$$r \times s = \{t\, q \mid t \in r \text{ and } q \in s\}$$

- Předpokládá se, že atributy relací $r(R)$ a $s(S)$ jsou disjunktní (tj. $R \cap S = \varnothing$).
- Pokud atributy relací $r(R)$ a $s(S)$ nejsou disjunktní, musí se přejmenovat.

Příklad kartézského součinu

- Relace *r*, *s*:

A	*B*
α	1
β	2

r

C	*D*	*E*
α	10	+
β	10	+
β	20	-
γ	10	-

s

- $r \times s$

A	*B*	*C*	*D*	*E*
α	1	α	10	+
α	1	β	10	+
α	*1*	β	20	-
α	1	γ	10	-
β	2	α	10	+
β	2	β	10	+
β	2	β	20	-
β	2	γ	10	-

Skládání operací

- Můžeme tvořit výrazy skládáním operací
- Příklad: $\sigma_{A=C}\ (r \times s)$
- Operace *přirozené spojení* (natural join):
  - Značení: $r \bowtie s$
  - Nechť *r* a *s* jsou relace na schématech *R* a *S*. Výsledek je relace na schématu $R \cup S$, kterou získáme uvažováním každého páru n-tic t_r z *r* a t_s z *s*.
  - Mají-li t_r a t_s stejné hodnoty na každém atributu z $R \cap S$, je k výsledku přidána n-tice *t* taková, že
    * *t* má na atributech z *R* stejnou hodnotu jako t_r
    * *t* má na atributech z *S* stejnou hodnotu jako t_s

Příklad:
R = (*A, B, C, D*)
S = (*E, B, D*)

- Výsledné schéma = (*A, B, C, D, E*)
- $r \bowtie s$ je definováno jako:

$$r \bowtie s = \Pi_{r.A,r.B,r.C,r.D,s.E}\ (\sigma_{r.B=s.B \wedge r.D=s.D}\ (r \times s))$$

Příklad přirozeného spojení (Natural join operation)

- Relace *r*, *s*:

A	*B*	*C*	*D*
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	*D*	*E*
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- $r \bowtie s$

A	*B*	*C*	*D*	*E*
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Operace dělení

- Značení: $r \div s$
- Odpovídá dotazům, které obsahují frázi „pro všechny“.
- Nechť *r* a *s* jsou relace na schématech *R* a *S*, kde
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$

  Výsledek operace $r \div s$ je relace na schématu $R - S = (A_1, \ldots, A_m)$

  $$r \div s = \{t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s: tu \in r\}$$
- Vlastnost
  - Nechť $q = r \div s$
  - Pak *q* je největší relace, pro kterou platí: $q \times s \subseteq r$
- Definice pomocí podmínek základních operací algebry. Nechť $r(R)$ a $s(S)$ jsou relace a nechť $S \subseteq R$

  $$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$
- Vysvětlení:
  - $\Pi_{R-S,S}(r)$ jednoduše přeřadí atributy *r*
  - $\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$ dává ty n-tice *t* v $\Pi_{R-S}(r)$, že pro nějakou n-tici $u \in s$, $tu \notin r$

Příklad dělení

- Relace *r*, *s*:

A	*B*
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
δ	6
ε	1
ε	2

r

B
1
2

s

- *r* ÷ *s*

A
α
ε

Jiný příklad dělení

- Relace *r*, *s*:

A	*B*	*C*	*D*	*E*
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	*E*
a	1
b	1

s

- *r* ÷ *s*

A	*B*	*C*
α	a	γ
γ	a	γ

Operace přiřazení

- Operace přiřazení ($\leftarrow$) představuje vhodný způsob, jak vyjádřit komplexní dotazy; dotazy se píší jako sekvenční program skládající se ze skupiny přiřazení následovaných výrazem, jehož hodnota je zobrazena jako výsledek dotazu.
- Přiřazení musí být vždy prováděno na dočasné relační proměnné.
- Příklad: Napišme $r \div s$ jako

  $temp1 \leftarrow \Pi_{R\text{-}S}(r)$

  $temp2 \leftarrow \Pi_{R\text{-}S}((temp1 \times s) - \Pi_{R\text{-}S,S}(r))$

  $result = temp1 - temp2$

  – Výsledek vpravo od $\leftarrow$ je přiřazen do relační proměnné vlevo od $\leftarrow$.

Příklady dotazů

- Najděte všechny zákazníky, kteří mají (zároveň) účet aspoň v pobočkách „Praha-Vršovice“ a „Praha-Spořilov“.
  – Dotaz 1

$\Pi_{JZAK}(\sigma_{JPOB = \text{„Praha-Vršovice“}}(vkladatel \bowtie účet)) \cap \Pi_{JZAK}(\sigma_{JPOB = \text{„Praha-Spořilov“}}(vkladatel \bowtie účet))$

  kde *JZAK* značí *jméno-zákazníka*, *JPOB* značí *jméno-pobočky* a *MPOB* značí *město-pobočky*

  – Dotaz 2

$\Pi_{JZAK,\ JPOB}(vkladatel \bowtie účet) \div \rho_{temp(JPOB)}(\{(\text{„}Praha\text{-}Vršovice\text{“}), (\text{„}Praha\text{-}Spořilov\text{“})\})$

- Najděte všechny zákazníky, kteří mají účet ve všech pobočkách v Praze.

$\Pi_{JZAK,\ JPOB}(vkladatel \bowtie účet) \div \Pi_{JPOB}(\sigma_{MPOB = \text{„Praha“}}(pobočka))$

n-ticový relační kalkul (Tuple relational calculus)

- Neprocedurální dotazovací jazyk, kde každý dotaz je ve tvaru

$$\{t \mid P(t)\}$$

- Je to množina všech n-tic t takových, že predikát P je pravdivý pro t
- t je n-ticová proměnná (*tuple variable*); $t[A]$ značí hodnotu n-tice t na atributu A
- $t \in r$ značí, že n-tice t je v relaci r
- P je výraz (*formula*) podobný výrazu v predikátové logice (predikátovém kalkulu)

Výraz v predikátovém kalkulu

- Množina atributů a konstant
- Množina operátorů porovnání: (např.: $<$, $\leq$, $=$, $\neq$, $>$, $\geq$)
- Množina spojek: and ($\wedge$), or ($\vee$), not ($\neg$)
- Implikace ($\Rightarrow$): $x \Rightarrow y$, jestliže x je pravdivé, pak y také

$$x \Rightarrow y \equiv \neg x \vee y$$

- Množina kvantifikátorů:
  – $\exists\ t \in r\ (Q(t)) \equiv$ existuje n-tice t z relace r taková, že platí $Q(t)$
  – $\forall\ t \in r\ (Q(t)) \equiv$ pro všechny n-tice t v relaci r platí $Q(t)$

Příklad (banka)

pobočka (pobočka-jméno, pobočka-město, aktivum)
zákazník (zákazník-jméno, zákazník-ulice, zákazník-město)
účet (pobočka-jméno, číslo-účtu, částka)
půjčka (pobočka-jméno, půjčka-číslo, částka)
vkladatel (zákazník-jméno, číslo-účtu)
dlužník (zákazník-jméno, půjčka-číslo)

Příklady dotazů

- Najděte *pobočka-jméno, půjčka-číslo* a *částka* pro půjčky přes 1200

$$\{t \mid t \in \textit{půjčka} \wedge t[\textit{částka}] > 1200\}$$

- Najděte číslo půjčky pro každou půjčku s částkou větší než 1200

$$\{t \mid \exists\, s \in \textit{půjčka}\ (t[\textit{půjčka-číslo}] = s[\textit{půjčka-číslo}] \wedge s[\textit{částka}] > 1200)\}$$

Poznámka: *t* je relace na schématu *(půjčka-číslo)* protože to je jediný *zmíněný* atribut n-tice *t*.

- Najděte jména všech zákazníků, kteří mají v bance půjčku, účet nebo obojí

$$\{t \mid \exists s \in \textit{dlužník}(t[\textit{zákazník-jméno}] = s[\textit{zákazník-jméno}]) \vee$$

$$\exists\, u \in \textit{vkladatel}(t[\textit{zákazník-jméno}] = u[\textit{zákazník-jméno}])\}$$

- Najděte jména všech zákazníků, kteří mají v bance půjčku a účet

$$\{t \mid \exists s \in \textit{dlužník}(t[\textit{zákazník-jméno}] = s[\textit{zákazník-jméno}]) \wedge$$

$$\exists\, u \in \textit{vkladatel}(t[\textit{zákazník-jméno}] = u[\textit{zákazník-jméno}])\}$$

- Najděte jména všech zákazníků, kteří mají půjčku v pobočce Praha-Spořilov

$$\{t \mid \exists s \in \textit{dlužník}(t[\textit{zákazník-jméno}] = s[\textit{zákazník-jméno}] \wedge$$

$$\exists\, u \in \textit{půjčka}(u[\textit{pobočka-jméno}] = \text{„Praha-Spořilov“} \wedge u[\textit{půjčka-číslo}] = s[\textit{půjčka-číslo}]))\}$$

- Najděte jména všech zákazníků, kteří mají půjčku v pobočce Praha-Spořilov, ale nemají účet v žádné pobočce banky.

$$\{t \mid \exists s \in \textit{dlužník}(t[\textit{zákazník-jméno}] = s[\textit{zákazník-jméno}] \wedge$$

$$\exists\, u \in \textit{půjčka}(u[\textit{pobočka-jméno}] = \text{„Praha-Spořilov“} \wedge u[\textit{půjčka-číslo}] = s[\textit{půjčka-číslo}])$$

$$\wedge$$

$$\neg\, \exists v \in \textit{vkladatel}\ (v[\textit{zákazník-jméno}] = t[\textit{zákazník-jméno}]))\}$$

- Najděte jména všech zákazníků, kteří mají půjčku v pobočce Praha-Spořilov a města, ze kterých jsou

$$\{t \mid \exists\, s \in \textit{půjčka}(s[\textit{pobočka-jméno}] = \text{„Praha-Spořilov“} \wedge$$

$$\exists u \in \textit{dlužník}\ (u[\textit{půjčka-číslo}] = s[\textit{půjčka-číslo}] \wedge$$

$$t[\textit{zákazník-jméno}] = u[\textit{zákazník-jméno}] \wedge$$

$$\exists v \in \textit{zákazník}\ (u[\textit{zákazník-jméno}] = v[\textit{zákazník-jméno}] \wedge$$

$$t[\textit{zákazník-město}] = v[\textit{zákazník-město}]))\}$$

- Najděte jména všech zákazníků, kteří mají účet ve všech pobočkách umístěných v Praze:

$$\{t \mid \forall s \in \textit{pobočka}\ (s[\textit{pobočka-město}] = \text{„Praha“} \Rightarrow$$

$$\exists\, u \in \textit{účet}\ (s[\textit{pobočka-jméno}] = u[\textit{pobočka-jméno}] \wedge$$

$$\exists\, s \in \textit{vkladatel}\ (t[\textit{zákazník-jméno}] = s[\textit{zákazník-jméno}] \wedge$$

$$s[\textit{číslo-účtu}] = u[\textit{číslo-účtu}]))\}$$

Bezpečnost výrazů

- V kalkulu n-tic je možné psát výrazy, které generují nekonečné relace
- Např. $\{t \mid \neg(t \in r)\}$ vede k nekonečné relaci, je-li doména aspoň jednoho atributu relace r nekonečná
- Abychom tomuto zamezili, omezíme množinu použitelných výrazů na *bezpečné* výrazy
- Výraz $\{t \mid P(t)\}$ v relačním kalkulu n-tic je *bezpečný*, jestliže se každá komponenta t objeví v $dom(P)$.
- $dom(P)$ je doména výrazu P – hodnoty konstant a n-tic vyskytujících se v P a také všechny hodnoty relací jejichž názvy se vyskytují v P.

Doménový relační kalkul

- Neprocedurální dotazovací jazyk, který je, co se týče síly, ekvivalentní n-ticovému relačnímu kalkulu.
- Každý dotaz je výraz ve tvaru:

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

  - $x_1, x_2, \ldots, x_n$ reprezentují doménové proměnné
  - P reprezentuje formuli podobnou formuli predikátového kalkulu

Příklady dotazů

- Najděte *pobočka-jméno (b), půjčka-číslo (l)* a *částka (a)* pro půjčky přes 1200:

$$\{< b, l, a > \mid < b, l, a > \in \textit{půjčka} \wedge a > 1200\}$$

- Najděte jména všech zákazníků *(c)*, kteří mají půjčku přes 1200:

$$\{< c > \mid \exists\, b, l, a\ (< c, l > \in \textit{dlužník} \wedge < b, l, a > \in \textit{půjčka} \wedge a > 1200)\}$$

- Najděte jména všech zákazníků, kteří mají půjčku u pobočky Praha-Spořilov a částku půjčky:

$$\{< c, a > \mid \exists\, l\ (< c, l > \in \textit{dlužník} \wedge \exists\, b\ (< b, l, a > \in \textit{půjčka} \wedge b = \text{„Praha-Spořilov“}))\}$$

- Najděte jména všech zákazníků, kteří mají půjčku, účet nebo obojí u pobočky Praha-Spořilov:

$$\{< c > \mid \exists\, l\ (< c, l > \in \textit{dlužník} \wedge \exists\, b, a\ (< b, l, a > \exists\, \textit{půjčka} \wedge b = \text{„Praha-Spořilov“})$$

$$\vee \exists\, a\ (< c, a > \in \textit{dlužník} \wedge \exists\, b, n\ (< b, a, n > \in \textit{účet} \wedge b = \text{„Praha-Spořilov“}))\}$$

- Najděte jména všech zákazníků, kteří mají účet ve všech pobočkách umístěných v Praze:

$$\{< c > \mid \forall\, x, y, z\ ((< x, y, z > \in \textit{pobočka} \wedge y = \text{„Praha“}) \Rightarrow$$

$$\exists\, a, b\ (< x, a, b > \in \textit{účet} \wedge < c, a > \in \textit{vkladatel}))\}$$

Bezpečnost výrazů

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

je bezpečný výraz, platí-li následující:

- Všechny hodnoty, které se objeví ve všech n-ticích celého výrazu, jsou hodnoty z $dom(P)$.
- Pro každou podformuli „existuje“ tvaru $\exists\, x\ (P_1(x))$ platí, že tato podformule je pravdivá jen tehdy, je-li hodnota x v $dom(P_1)$, takže je $P_1(x)$ pravdivá.
- Pro každou podformuli „pro všechny“ tvaru $\forall\, x\ (P_1(x))$ platí, že tato podformule je pravdivá jen tehdy, je-li $P_1(x)$ pravdivá pro všechny hodnoty x z $dom(P_1)$.

Rozšířené operace relační algebry

- Zobecněná projekce
- Vnější spojení (Outer join)
- Souhrnné funkce

Zobecněná projekce

- Rozšiřuje operaci projekce tak, že umožňuje použití aritmetických funkcí v seznamu projekce.

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

- E je jakýkoliv výraz relační algebry
- Každá z F_1, F_2, …, F_n jsou aritmetické výrazy zahrnující konstanty a atributy ve schématu E.
- Daná relace *úvěr-info (zákazník_jméno, limit, stav*) najde, kolik může každá osoba utratit:

$$\Pi_{\textit{zákazník\_jméno, limit – stav}}(\textit{úvěr-info})$$

Vnější spojení (Outer join)

- Rozšíření operace přirozené spojení, které zabraňuje ztrátě informací.
- Spočítá operaci spojení a přidá n-tice z jedné relace, které neodpovídají n-ticím v druhé relaci k výsledkům operace spojení.
- Používá hodnotu *null*:
  - *null* značí, že hodnota je neznámá nebo neexistuje
  - všechna porovnávání zahrnující *null* mají z definice hodnotu **false**.

Příklad

- Relace *půjčka*

pobočka-jméno	*půjčka-číslo*	*částka*
Děčín	L-170	3000
Rájec	L-230	4000
Praha-Spořilov	L-260	1700

- Relace *dlužník*

zákazník-jméno	*půjčka-číslo*
Novák	L-170
Starý	L-230
Hájek	L-155

- *půjčka* ⋈ *dlužník*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Děčín	L-170	3000	Novák
Rájec	L-230	4000	Starý

- *půjčka* ⟖ *dlužník*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Děčín	L-170	3000	Novák
Rájec	L-230	4000	Starý
Praha-Spořilov	L-260	1700	*null*

- *půjčka* ⟕ *dlužník*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Děčín	L-170	3000	Novák
Rájec	L-230	4000	Starý
null	L-155	*null*	Hájek

- *půjčka* ⋈ *dlužník*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Děčín	L-170	3000	Novák
Rájec	L-230	4000	Starý
Praha-Spořilov	L-260	1700	*null*
null	L-155	*null*	Hájek

Souhrnné funkce

- Souhrnný operátor $\mathcal{G}$ vezme kolekci hodnot a vrátí jednoduchou hodnotu jako výsledek.

  avg: průměrná hodnota
  min: minimální hodnota
  max: maximální hodnota
  sum: součet hodnot
  count: počet hodnot

$$G_1, G_2, \ldots, G_n \mathcal{G} F_1 A_1, F_2 A_2, \ldots, F_m A_m (E)$$

  - E je jakýkoliv výraz relační algebry
  - $G_1, G_2, \ldots, G_n$ je seznam atributů podle kterých se n-tice seskupují
  - F_i je souhrnná funkce
  - A_i je jméno atributu
  - Výsledkem je relace složená z atributů:
    (i) všechny použité G_i atributy
    (ii) jeden atribut pro každou použitou souhrnnou funkci

Příklad

- Relace *r*

A	*B*	*C*
α	α	7
α	β	7
β	β	3
β	β	10

- $\text{sum}_C (r)$: 27
- Relace *účet* seskupená podle *pobočka-jméno*:

Pobočka-jméno	*číslo-účtu*	*zůstatek*
Praha-Spořilov	A-102	400
Praha-Spořilov	A-201	900
Brno	A-217	750
Brno	A-215	750
Rájec	A-222	700

- *pobočka-jméno* $\mathcal{G}$ sum *zůstatek* (*účet*)

pobočka-jméno	*suma-zůstatek*
Praha-Spořilov	1300
Brno	1500
Rájec	700

Modifikace databáze

- Obsah databáze lze měnit následujícími operacemi:
  - Mazání
  - Vložení
  - Aktualizace
- Tyto operace jsou vyjadřovány operátorem přiřazení.

Mazání

- Požadavek na mazání je vyjádřen podobně jako dotaz, ale n-tice nejsou poslány uživateli, ale smazány z databáze.
- Lze mazat pouze celé n-tice, nikoli pouze hodnoty ve vybraných atributech.
- V relační algebře je mazání vyjádřeno takto:

$$r \leftarrow r - E$$

kde r je relace a E je dotaz relační algebry.

Příklady mazání

- Smaže všechny záznamy o účtech v pobočce Praha-Spořilov.

$$\textit{účet} \leftarrow \textit{účet} - \sigma_{\textit{pobočka-jméno} = \text{„Praha-Spořilov“}} (\textit{účet})$$

- Smaže všechny záznamy o půjčkách s částkou v rozmezí 0–50.

$$\textit{půjčka} \leftarrow \textit{půjčka} - \sigma_{\textit{částka} \geq 0 \text{ and } \textit{částka} \leq 50} (\textit{půjčka})$$

- Smaže všechny účty a vkladatele v pobočkách umístěných v Brně.

$$r_1 \leftarrow \sigma_{\textit{pobočka-město} = \text{„Brno“}} (\textit{účet} \bowtie \textit{pobočka})$$
$$r_2 \leftarrow \Pi_{\textit{pobočka-jméno, číslo-účtu, zůstatek}} (r_1)$$
$$r_3 \leftarrow \Pi_{\textit{zákazník-jméno, číslo-účtu}} (r_2 \bowtie \textit{vkladatel})$$
$$\textit{účet} \leftarrow \textit{účet} - r_2$$
$$\textit{vkladatel} \leftarrow \textit{vkladatel} - r_3$$

Vložení

- Abychom přidali data do relace, buď:
  - specifikujeme n-tici, která má být přidána
  - napíšeme dotaz, jehož výsledek je množina n-tic, která má být přidána
- V relační algebře je vložení vyjádřeno takto:

$$r \leftarrow r \cup E$$

kde r je relace a E je dotaz relační algebry.

- Vložení jednoduché n-tice je vyjádřeno tak, že E může být konstantní relace obsahující jednu n-tici.

Příklady vložení

- Vložte informaci do databáze: Novák má 1200 na účtu A-973 v pobočce Praha-Spořilov.

$$účet \leftarrow účet \cup \{(\text{„Praha-Spořilov“, A-973, 1200})\}$$
$$vkladatel \leftarrow vkladatel \cup \{(\text{„Novák“, A-973})\}$$

- Jako dárek poskytněte všem zákazníkům půjček v pobočce Praha-Spořilov účet se zůstatkem 200. Nechť číslo půjčky slouží jako číslo účtu pro nový účet.

$$r_1 \leftarrow (\sigma_{pobočka\text{-}jméno = \text{„Praha-Spořilov“}} (dlužník \bowtie půjčka))$$
$$účet \leftarrow účet \cup \Pi_{pobočka\text{-}jméno,\ půjčka\text{-}číslo,\ 200} (r_1)$$
$$vkladatel \leftarrow vkladatel \cup \Pi_{zákazník\text{-}jméno,\ půjčka\text{-}číslo} (r_1)$$

Aktualizace

- Mechanismus jak změnit hodnoty v n-tici, aniž by musely být změněny *všechny*
- K tomu se používá operátor zobecněné projekce

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n} (r)$$

  - F_i je *i*-tý atribut relace *r*, jestliže *i*-tý atribut není aktualizován

  nebo
  - F_i je výraz obsahující pouze konstanty a atributy *r*, který dává novou hodnotu atributu, jestliže má být *i*-tý atribut aktualitován

Příklady aktualizace

- Zvyšte všechny zůstatky o 5 %.

$$účet \leftarrow \Pi_{JPOB,\ ČÚ,\ ZůST \leftarrow ZůST * 1.05} (účet)$$

  kde *ZůST, JPOB* a *ČÚ* jsou *zůstatek, pobočka-jméno* a *číslo-účtu*.
- Vyplaťte všechny účty se zůstatkem přes $10,000 6% úrokem, ostatní 5% úrokem.

$$účet \leftarrow \Pi_{JPOB,\ ČÚ,\ ZůST \leftarrow ZůST * 1.06} (\sigma_{BAL > 10000} (účet)) \cup$$
$$\Pi_{JPOB,\ ČÚ,\ ZůST \leftarrow ZůST * 1.05} (\sigma_{BAL \leq 10000} (účet))$$

Pohledy

- V některých případech není vhodné, aby všichni uživatelé viděli celý logický model databáze (tj. všechny relace aktuálně uložené v databázi).
- Představme si osobu, která potřebuje vědět o počtu půjček zákazníkům, ale nemusí vidět částku půjčky. Tato osoba by měla vidět relaci popsanou v relační algebře takto:

$$\Pi_{zákazník\text{-}jméno,\ půjčka\text{-}číslo} (dlužník \bowtie půjčka)$$

- Jakákoli relace, která není součástí koncepčního modelu, ale je viditelná uživateli jako „virtuální relace“, se nazývá *pohled*.

Definice pohledu

- Pohled je definován příkazem **create view**, který má tvar
  create view *v* **as** <výraz dotazu>
  kde <výraz dotazu> je jakýkoliv správný výraz relační algebry. Jméno pohledu je reprezentováno proměnnou *v*.
- Jakmile je pohled definován, jeho jméno může být použito pro odkazování na virtuální relaci, která pohled generuje.
- Definice pohledu není to samé jako vytvoření nové relace vyhodnocením dotazovacího výrazu. Definice pohledu způsobuje uložení výrazu, aby mohl být substituován do dotazů, které používají tento pohled.

Příklady pohledů

- Představme si pohled (pojmenovaný *všichni-zákazníci*) skládající se z poboček a jejich zákazníků.

  create view *všichni-zákazníci* **as**
  $\Pi_{\textit{pobočka-jméno, zákazník-jméno}}$ (*vkladatel* ⋈ *účet*)
  $\cup\ \Pi_{\textit{pobočka-jméno, zákazník-jméno}}$ (*dlužník* ⋈ *účet*)
- Nyní můžeme najít všechny zákazníky pobočky Praha-Spořilov napsáním:
  $\Pi_{\textit{zákazník-jméno}}$ ($\sigma_{\textit{pobočka-jméno = „Praha-Spořilov“}}$ (*všichni-zákazníci*))

Aktualizace přes pohledy

- Modifikace databáze s použitím pohledů musí být překládány na modifikace aktuálních relací v databázi.
- Představme si uživatele, který potřebuje vidět všechna data o půjčkách v relaci *půjčka* s výjimkou *částky*. Pohled, který ji poskytneme (*pobočka-půjčka*), je definován takto:

  create view *pobočka-půjčka* **as** $\Pi_{\textit{pobočka-jméno, půjčka-číslo}}$ (*půjčka*)

  Umožníme-li, aby se jméno pohledu objevilo kdekoliv, kde je akceptováno jméno relace, uživatel může psát:

  pobočka-půjčka ← *pobočka-půjčka* ∪ {(„Praha-Spořilov“, L-37)}
- Předchozí vkládání musí být reprezentováno vkládáním do aktuální relace *půjčka*, ze které byl pohled *pobočka-půjčka* vytvořen.
- Vkládání do *půjčka* vyžaduje hodnotu pro *částka*. Vkládání se může chovat takto:
  - odmítne vkládání a vrátí chybové hlášení

  nebo
  - vloží n-tici („Praha-Spořilov“, L-37, *null*) do relace *půjčka*

Pohledy definované použitím jiných pohledů

- V definování pohledu je možné použít jen jeden jiný pohled.
- Relace pohledu v_1 *závisí přímo na* relaci pohledu v_2, jestliže v_2 je použit ve výrazu, který definuje v_1.
- Relace pohledu v_1 *závisí na* relaci pohledu v_2, jestliže v grafu závislostí je cesta z v_2 do v_1.
- Relace pohledu *v* je *rekurzivní*, jestliže závisí sama na sobě.

Expanze pohledu

- Cesta, jak definovat význam pohledů definovaných podmínkami ostatních pohledů.
- Nechť pohled v_1 je definován výrazem e_1, který může sám obsahovat použití relací pohledů.
- Expanze pohledu výrazu opakuje následující nahrazovací krok:
  repeat
    najdi jakoukoli relaci v_i v e_1
    nahraď relaci pohledu v_i výrazem definujícím v_i
  until v e_1 nejsou přítomny další relace pohledů
- Pokud definice pohledů nebudou rekurzivní, tato smyčka skončí.

Kapitola 4: SQL

- Základní struktura
- Množinové operace
- Souhrnné funkce
- Nulové hodnoty
- Vnořené poddotazy (Nested sub-queries)
- Odvozené relace
- Pohledy
- Modifikace databáze
- Spojené relace
- Jazyk definice dat (Data definition language)
- Vložený SQL

Základní struktura

- SQL je založen na množinových a relačních operacích s několika modifikacemi a vylepšeními
- Typický SQL dotaz má tvar:

  select A_1, A_2, …, A_n
  from r_1, r_2, …, r_m
  where P

  – A_i reprezentuje atributy
  – r_i reprezentuje relace
  – P je predikát
- Tento dotaz je ekvivalentní následujícímu výrazu relační algebry:

$$\Pi_{A_1, A_2, \ldots, A_n} (\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

- Výsledek každého SQL dotazu je relace.

Klauzule select

- Klauzule **select** odpovídá operaci projekce v relační algebře. Je používána k vypsání požadovaných atributů ve výsledku dotazu.
- Najděte jména všech poboček v relaci *půjčka*

  select *pobočka-jméno*
  from *půjčka*

  V syntaxi „čisté“ relační algebry se tento dotaz zapíše takto:

$$\Pi_{pobočka\text{-}jméno} (půjčka)$$

- Hvězdička značí „všechny atributy“

  select *
  from *půjčka*
- SQL umožňuje duplikáty v relacích i ve výsledcích dotazů.
- Pro eliminaci duplikátů vložte klíčové slovo **distinct** za **select**.
- Najděte jména všech poboček v relaci *půjčka* a odstraňte duplikáty

  select distinct *pobočka-jméno*
  from *půjčka*

- Klíčové slovo **all** specifikuje, že duplikáty odstraněny nebudou.
  select all *pobočka-jméno*
  from *půjčka*
- Klauzule **select** může obsahovat aritmetické výrazy s operátory +, -, * a / na konstantách nebo atributech n-tic.
- Dotaz
  select *pobočka-jméno, půjčka-číslo, částka* * 100
  from *půjčka*
  vrátí relaci shodnou s relací *půjčka*, jen atribut *částka* je vynásoben číslem 100

Klauzule where

- Klauzule **where** odpovídá operaci výběr v relační algebře. Skládá se z predikátu zahrnujícího atributy relací, které jsou v klauzuli **from**.
- Najděte všechna čísla půjček pro půjčky v pobočce Praha-Spořilov s částkami většími než $1200.
  select *půjčka-číslo*
  from *půjčka*
  where *pobočka-jméno* = „Praha-Spořilov“ **and** *částka* > 1200
- SQL používá logické spojky **and**, **or** a **not**. Umožňuje tak použití aritmetických výrazů jako operandů pro operátory porovnání.
- SQL zahrnuje operátor porovnání **between** pro zjednodušení klauzule **where**, který specifikuje hodnotu z určitého intervalu.
- Najděte čísla půjček s částkami mezi $90,000 a $100,000 (tj. ≥$90,000 a ≤$100,000)
  select *půjčka-číslo*
  from *půjčka*
  where *částka* **between** 90000 **and** 100000

Klauzule from

- Klauzule **from** odpovídá kartézskému součinu z relační algebry. Vypíše relace, které mají být procházeny při vyhodnocování výrazu.
- Najděte kartézský součin *půjčovatel* × *půjčka*
  select *
  from *půjčovatel, půjčka*
- Najděte jméno a číslo půjčky všech zákazníků, kteří mají půjčku v pobočce Praha-Spořilov.
  select distinct *zákazník-jméno, půjčovatel.půjčka-číslo*
  from *půjčovatel, půjčka*
  where *půjčovatel.půjčka*-číslo = *půjčka.půjčka-číslo*
  and pobočka-*jméno* = „Praha-Spořilov“

Operace přejmenování

- Mechanismus pro přejmenovávání relací je v SQL řešen pomocí klauzule **as**:
  staré-jméno **as** *nové-jméno*
- Najděte jméno a číslo půjčky všech zákazníků, kteří mají půjčku v pobočce Praha-Spořilov; nahraďte jméno sloupce *půjčka-číslo* jménem *půjčka-id*.
  select distinct *zákazník-jméno, půjčovatel.půjčka číslo* **as** *půjčka-id*
  from *půjčovatel, půjčka*
  where *půjčovatel.půjčka-číslo* = *půjčka.půjčka-číslo*
  and *pobočka-jméno* = „Praha-Spořilov“

Proměnné n-tic (Tuple variables)

- Proměnné n-tic jsou definovány v klauzuli **from** pomocí klauzule **as**.
- Najděte jména zákazníků a čísla jejich půjček pro všechny zákazníky, kteří mají půjčku ve stejné pobočce.
  select distinct *zákazník-jméno, T.půjčka-číslo*
  from *půjčovatel* **as** *T*, *půjčka* **as** *S*
  where *T.půjčka-číslo* = *S.půjčka-číslo*
- Najděte jména všech poboček, které mají větší aktiva než nějaká pobočka v Praze.
  select distinct *T.pobočka-jméno*
  from *pobočka* **as** *T*, *pobočka* **as** *S*
  where *T.aktiva* > *S.aktiva* **and** *S.pobočka-město* = „Praha“

Operace s řetězci

- SQL zahrnuje operátor pro porovnávání řetězců znaků. Vzorky jsou popsány použitím dvou speciálních znaků:
  - procento (%). Znak % vyhovuje jakémukoliv podřetězci.
  - podtržítko (_). Znak _ vyhovuje jakémukoliv znaku.
- Najděte jména všech zákazníků, jejichž ulice obsahuje podřetězec ‚Main‘.
  select *zákazník-jméno*
  from *zákazník*
  where *zákazník-ulice* **like** „%Main%“
- Co vyhovuje jménu „Main%“?
  like „Main\%“ **escape** „\“

Uspořádání zobrazení n-tic

- Výpis všech zákazníků, kteří mají půjčku v pobočce Praha-Spořilov seřazený podle abecedy

  select distinct *zákazník-jméno*
  from *půjčovatel, půjčka*
  where *půjčovatel.půjčka-číslo* = *půjčka.půjčka-číslo* **and**
  pobočka-jméno = „Praha-Spořilov“
  order by *zákazník-jméno*
- Můžeme ještě specifikovat **desc** pro sestupné pořadí nebo **asc** pro vzestupné pořadí přidáním na konec dotazu; vzestupné pořadí je implicitní.
- SQL musí provést řazení, aby vyhověl požadavku **order by**. Protože řazení velkého množství n-tic může být drahé, je vhodné řadit pouze v nezbytných případech.

Duplikáty

- V relaci s duplikáty může SQL definovat kolik kopií n-tic se objeví ve výsledku.
- *Více-množinové* verze některých operátorů relační algebry – mějme více-množinové relace r_1 a r_2:
  - Je-li c_1 kopií n-tice t_1 v r_1 a t_1 vyhovuje výběru σ_θ, pak je c_1 kopií n-tice t_1 ve výsledku $\sigma_\theta(r_1)$
  - Pro každou kopii n-tice t_1 v r_1 je kopie n-tice $\Pi_A(t_1)$ v $\Pi_A(r_1)$, kde $\Pi_A(t_1)$ značí projekci jednoduché n-tice t_1.
  - Je-li c_1 kopií n-tice t_1 v r_1 a c_2 kopií n-tice t_2 v r_2, pak je $c_1 \times c_2$ kopií n-tice $t_1.t_2$ v $r_1 \times r_2$.
- Předpokládejme, že relace r_1 se schématem (*A, B*) a r_2 se schématem (C) jsou následující více-množiny:

  $r_1 = \{(1, a), (2, a)\}$ $r_2 = \{(2), (3), (3)\}$
- Pak $\Pi_B(r_1)$ bude $\{(a), (a)\}$, zatímco $\Pi_B(r_1) \times r_2$ bude

  $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$
- SQL sémantika duplikátů:

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  je ekvivalentní *více-množinové* verzi výrazu:

  $$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Množinové operace

- Množinové operace **union**, **intersect** a **except** operují na relacích a odpovídají operacím relační algebry $\cup$, $\cap$ a –.
- Každá z těchto operací automaticky eliminuje duplikáty; chceme-li i duplikáty, použijeme více-množinové verze **union all**, **intersect all** a **except all**. Předpokládejme, že se n-tice objeví *m*-krát v *r* a *n*-krát v *s*, pak se objeví:
  - *(m + n)*-krát v *r* **union all** *s*
  - *min(m, n)*-krát v *r* **intersect all** *s*
  - *max(0, m – n)*-krát v *r* **except all** *s*

- Najděte všechny zákazníky, kteří mají půjčku, účet nebo obojí:
  (**select** *zákazník-jméno* **from** *vkladatel*)
  union
  (**select** *zákazník-jméno* **from** *půjčovatel*)
- Najděte všechny zákazníky, kteří mají půjčku i účet:
  (**select** *zákazník-jméno* **from** *vkladatel*)
  intersect
  (**select** *zákazník-jméno* **from** *půjčovatel*)
- Najděte všechny zákazníky, kteří mají účet, ale ne půjčku:
  (**select** *zákazník-jméno* **from** *vkladatel*)
  except
  (**select** *zákazník-jméno* **from** *půjčovatel*)

Souhrnné funkce

Tyto funkce operují na více-množinových hodnotách sloupců relace a vracejí hodnotu
avg: průměrná hodnota
min: minimální hodnota
max: maximální hodnota
sum: suma hodnot
count: počet hodnot

- Najděte průměrnou hodnotu zůstatku účtu v pobočce Praha-Spořilov.
  select avg (*zůstatek*)
  from *účet*
  where *pobočka-jméno* = „Praha-Spořilov“
- Najděte počet n-tic v relaci *zákazník*.
  select count (*)
  from *zákazník*
- Najděte počet vkladatelů v bance.
  select count (**distinct** *zákazník-jméno*)
  from *vkladatel*

Souhrnné funkce – Group by

- Najděte počet vkladatelů v každé pobočce.
  select *pobočka-jméno*, **count** (**distinct** *zákazník-jméno*)
  from *vkladatel, účet*
  where *vkladatel.číslo-účtu = účet.číslo-účtu*
  group by *pobočka-jméno*

  Poznámka: Atributy v klauzuli **select** mimo souhrnné funkce se musí objevit v seznamu **group by**!!!

Souhrnné funkce – klauzule having

- Najděte jména všech poboček, jejichž průměrný zůstatek účtu je větší než $1,200.

  select *pobočka-jméno*, **avg** (*zůstatek*)
  from *účet*
  group by *pobočka-jméno*
  having avg (*zůstatek*) > 1200

  Poznámka: predikáty v klauzuli **having** jsou po zformování skupin

Nulové hodnoty

- N-tice můžou mít nulovou hodnotu, značenou *null*, ta znamená neznámou hodnotu nebo hodnotu, která neexistuje.
- Výsledek jakékoliv aritmetické operace zahrnující *null* je *null*.
- Všechna porovnávání zahrnující *null* vrací *false*. Precizněji:
  - Jakékoliv porovnání s *null* vrací *unknown*
  - (*true* or *unknown*) = *true* (*false* or *unknown*) = *unknown*
    (*unknown* or *unknown*) = *unknown* (*true* and *unknown*) = *unknown*
    (*false* and *unknown*) = *false* (*unknown* and *unknown*) = *unknown*
  - Výsledek klauzule **where** je brán jako *false*, je-li *unknown*
  - „*P* **is unknown**“ se vyhodnotí jako *true*, jestliže predikát *P* je vyhodnocen jako *unknown*
- Najděte všechna čísla půjček, které se objeví v relaci *půjčka* s nulovou hodnotou *částka*.

  select *půjčka-číslo*
  from *půjčka*
  where *částka* **is null**
- Součet všech částek půjček

  select sum (*částka*)
  from *půjčka*

  Tyto výrazy ignorují nulové částky; výsledek je nula, jestliže zde není žádná nenulová částka.
- Všechny souhrnné operace s výjimkou **count(*)** ignorují n-tice s nulovými hodnotami na souhrnných atributech.

Vnořené poddotazy (Nested subqueries)

- SQL poskytuje mechanismus pro vnořování poddotazů.
- Poddotaz je výraz **select-from-where**, který je vnořen do jiného dotazu.
- Obvyklé použití poddotazů je provádění testů na členství v množině, porovnávání množin a kardinalitu množin.

Členství v množině

- F **in** $r \Leftrightarrow \exists\, t \in r\ (t = \mathrm{F})$

(5 **in** | 0 / 4 / 5 |) = true

(5 **in** | 0 / 4 / 6 |) = false

(5 **not in** | 0 / 4 / 6 |) = true

Příklady dotazů

- Najděte všechny zákazníky, kteří mají v bance účet i půjčku.

```
select distinct zákazník-jméno
from půjčovatel
where zákazník-jméno in (select zákazník-jméno
                         from vkladatel)
```

- Najděte všechny zákazníky, kteří mají v bance půjčku, ale ne účet.

```
select distinct zákazník-jméno
from půjčovatel
where zákazník-jméno not in (select zákazník-jméno
                             from vkladatel)
```

- Najděte všechny zákazníky, kteří mají účet i půjčku v pobočce Praha-Spořilov.

```
select distinct zákazník-jméno
from půjčovatel, půjčka
where půjčovatel.půjčka-číslo = půjčka.půjčka-číslo and
      pobočka-jméno = „Praha-Spořilov“ and
      (pobočka-jméno, zákazník-jméno) in
            (select pobočka-jméno, zákazník-jméno
            from vkladatel, účet
            where vkladatel.číslo-účtu = účet.číslo-účtu)
```

Porovnávání množin

- Najděte všechny pobočky, které mají větší aktiva než nějaká pobočka v Praze.

```
select distinct T.pobočka-jméno
from pobočka as T, pobočka as S
where T.aktiva > S.aktiva and S.pobočka-město = „Praha“
```

Klauzule some

- F <comp> **some** $r \Leftrightarrow \exists\, t\, (t \in r \wedge$ [F <comp> t])

kde <comp> může být: <, ≤, >, ≥, =, ≠

(5 < **some** | 0 / 5 / 6 |) = true (čteme: 5 je menší než nějaká n-tice z relace

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (neboť 0 ≠ 5)

- (= **some**) je stejné jako **in**
- Ale: (≠ **some**) je různé od **not in** !!!

Příklad dotazu

- Najděte pobočky, které mají větší aktiva než nějaká pobočka v Praze.

```
select pobočka-jméno
from pobočka
where aktiva > some
        (select aktiva
        from pobočka
        where pobočka-město = „Praha“)
```

Klauzule all

- F <comp> **all** $r \Leftrightarrow \forall\, t\, (t \in r \wedge$ [F <comp> t])

(5 < **all** | 0 / 5 / 6 |) = false

(5 < **all** | 6 / 10 |) = true

(5 = **all** | 4 / 5 |) = false

(5 ≠ **all** | 4 / 6 |) = true (neboť 5 ≠ 4 **and** 5 ≠ 6)

- (≠ **all**) je stejné jako **not in**
- Ale: (**= all**) je různé od **in** !!!

Příklad dotazu

- Najděte pobočky, které mají větší aktiva než všechny pobočky v Praze.

```
select pobočka-jméno
from pobočka
where aktiva > all
        (select aktiva
        from pobočka
        where pobočka-město = „Praha“)
```

Test na prázdné relace

- Konstrukce **exists** vrací hodnotu **true**, je-li poddotaz v argumentu neprázdný.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Příklad dotazu

- Najděte všechny zákazníky, kteří mají účet ve všech pobočkách v Praze.

```
select dictinct S.zákazník-jméno
from vkladatel as S
where not exists (
        (select pobočka-jméno
        from pobočka
        where pobočka-město = „Praha“)
        except
        (select R.pobočka-jméno
        from vkladatel as T, účet as R
        where T.číslo-účtu = R.číslo-účtu and
                S.zákazník-jméno = T.zákazník-jméno))
```

- Poznámka: $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

Test na nepřítomnost duplikátních n-tic

- Konstrukce **unique** testuje, zda poddotaz má ve svém výsledku nějaké duplikátní n-tice.
- Najděte všechny zákazníky, kteří mají pouze jeden účet v pobočce Praha-Spořilov.

```
select T.zákazník-jméno
from vkladatel as T
where unique (
        select R.zákazník-jméno
        from účet, vkladatel as R
        where T.zákazník-jméno = R.zákazník-jméno and
                R.číslo-účtu = účet.číslo-účtu and
                účet.pobočka-jméno = „Praha-Spořilov“)
```

Příklad dotazu

- Najděte všechny zákazníky, kteří mají alespoň dva účty v pobočce Praha-Spořilov.

```
select T.zákazník-jméno
from vkladatel as T
where not unique (
        select R.zákazník-jméno
        from účet, vkladatel as R
        where T.zákazník-jméno = R.zákazník-jméno and
                R.číslo-účtu = účet.číslo-účtu and
                účet.pobočka-jméno = „Praha-Spořilov“)
```

Odvozené relace

- Najděte průměrný zůstatek účtu v těch pobočkách, kde je průměrný zůstatek větší než $1200.

```
select pobočka-jméno, prům-zůstatek
from (select pobočka-jméno, avg (zůstatek)
        from účet
        group by pobočka-jméno)
        as result (pobočka-jméno, prům-zůstatek)
where prům-zůstatek > 1200
```

Poznámka: nepotřebujeme použít klausuli **having**, jestliže v klauzuli **from** spočítáme dočasnou relaci *result* a její atributy můžeme použít přímo v klauzuli **where**.

Pohledy

- Poskytují mechanismus, jak skrýt nějaká data před nějakými uživateli. Pro vytvoření pohledu použijeme příkaz:

  create view *v* **as** <výraz dotazu>

  kde:
  - <výraz dotazu> je jakýkoliv dovolený výraz
  - *v* reprezentuje jméno pohledu

Příklady dotazů

- Pohled skládající se z poboček a jejich zákazníků

```
create view všichni-zákazníci as
(select pobočka-jméno, zákazník-jméno
from vkladatel, účet
where vkladatel.číslo-účtu = účet.číslo-účtu)
union
(select pobočka-jméno, zákazník-jméno
from půjčovatel, účet
where půjčovatel.půjčka-číslo = půjčka.půjčka-číslo)
```

Modifikace databáze – mazání

- Smažte všechny záznamy o účtech z pobočky Praha-Spořilov

```
delete from účet
where pobočka-jméno = „Praha-Spořilov“
```

- Smažte všechny účty z každé pobočky v Brně

```
delete from účet
where pobočka-jméno in (
        select pobočka-jméno
        from pobočka
        where pobočka-město = „Brno“)
delete from vkladatel
where číslo-účtu in (
        select číslo-účtu
        from pobočka, účet
        where pobočka-město = „Brno“
        and pobočka.pobočka-jméno = účet.pobočka-jméno)
```

- Smažte záznamy o všech účtech se zůstatkem pod průměrem banky

```
delete from účet
where zůstatek < (
        select avg (zůstatek)
        from účet)
```

  - Problém: když smažeme n-tice z *vklad*, průměrný zůstatek se změní
  - Řešení použité v SQL:
    * Nejprve spočítáme průměrný zůstatek a najdeme všechny n-tice ke smazání
    * Potom smažeme všechny n-tice nalezené výše

Modifikace databáze – vkládání

- Vložení nové n-tice do relace *účet*

```
insert into účet
        values („Praha-Spořilov“, A-9732, $1200)
```

  nebo ekvivalentně

```
insert into účet (pobočka-jméno, zůstatek, číslo-účtu)
        values („Praha-Spořilov“, $1200, A-9732)
```

- Přidání nové n-tice do relace *účet* se *zůstatkem* nastaveným na nulu.

```
insert into účet
        values („Praha-Spořilov“, A-9732, null)
```

- Jako dárek poskytněte všem zákazníkům půjček v pobočce Praha-Spořilov $200 termínovaný účet. Nechť číslo půjčky slouží jako číslo účtu pro nový termínovaný účet.

```
insert into účet
    select pobočka-jméno, půjčka-číslo, 200
    from půjčka
    where pobočka-jméno = „Praha-Spořilov“
insert into vkladatel
    select zákazník-jméno, půjčka-číslo
    from půjčka, půjčovatel
    where pobočka-jméno = „Praha-Spořilov“
    and půjčka.číslo-účtu = půjčovatel.číslo-účtu
```

Modifikace databáze – aktualizace

- Všechny účty se zůstatkem přes $10,000 o 6 %, ostatní o 5 %.
  - Napíšeme dva výrazy **update**:

```
update účet
set zůstatek = zůstatek * 1.06
where zůstatek > 10000

update účet
set zůstatek = zůstatek * 1.05
where zůstatek ≤ 10000
```

  - pořadí je důležité !!! Co se stane v případě opačného pořadí?
  - lépe lze vyřešit pomocí výrazu **case**

Aktualizace pohledu

- Vytvoříme pohled na všechna data v relaci *půjčka*, zakryjte atribut *částka*

```
create view pobočka-půjčka as
    select pobočka-jméno, půjčka-číslo
    from půjčka
```

- Přidáme novou n-tici do pohledu

```
insert into pobočka-půjčka
    values („Praha-Spořilov“, „L-307“)
```

Toto vkládání musí být reprezentováno vložením n-tice („Praha-Spořilov“, „L-307“, *null*) do relace *půjčka*.

- Aktualizaci komplexnějších pohledů je náročné nebo nemožné přeložit a proto nejsou povoleny.

Spojené relace (Joined relations)

- Operace spojení vezme dvě relace a jako výsledek vrátí jednu relaci.
- Tyto operace jsou typicky používány jako poddotazy v klauzuli **from**.
- Podmínka spojení (join condition) – definuje, které n-tice ve dvou relacích si odpovídají a které atributy jsou ve výsledku spojení.
- Typ spojení (join type) – definuje kolik je n-tic v každé relaci, které neodpovídají žádné n-tici v jiné relaci.

Typy spojení
vnitřní spojení (inner join)
levé vnější spojení (left outer join)
pravé vnější spojení (right outer join)
plné vnější spojení (full outer join)

Podmínky spojení
přirozené (natural)
na <predikát>
on <predicate>
použitím (A_1, A_2, …, A_n)
using (A_1, A_2, …, A_n)

Příklady

- Relace *půjčka*

pobočka-jméno	*půjčka-číslo*	*částka*
Brno-Vinohrady	L-170	3000
Praha-Vršovice	L-230	4000
Praha-Spořilov	L-260	1700

- Relace *půjčovatel*

zákazník-jméno	*půjčka-číslo*
Janda	L-170
Starý	L-230
Horák	L-155

- *půjčka* **inner join** *půjčovatel* **on** *půjčka.půjčka-číslo* = *půjčovatel.půjčka-číslo*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*	*půjčka-číslo*
Brno-Vinohrady	L-170	3000	Janda	L-170
Praha-Vršovice	L-230	4000	Starý	L-230

- *půjčka* **left outer join** *půjčovatel* **on** *půjčka.půjčka-číslo* = *půjčovatel.půjčka-číslo*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*	*půjčka-číslo*
Brno-Vinohrady	L-170	3000	Janda	L-170
Praha-Vršovice	L-230	4000	Starý	L-230
Praha-Spořilov	L-260	1700	*null*	*null*

- *půjčka* **natural join** *půjčovatel*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Brno-Vinohrady	L-170	3000	Janda
Praha-Vršovice	L-230	4000	Starý

- *půjčka* **natural right outer join** *půjčovatel*

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Brno-Vinohrady	L-170	3000	Janda
Praha-Vršovice	L-230	4000	Starý
null	L-155	*null*	Horák

- *půjčka* **full outer join** *půjčovatel* **using** (*půjčka-číslo*)

pobočka-jméno	*půjčka-číslo*	*částka*	*zákazník-jméno*
Brno-Vinohrady	L-170	3000	Janda
Praha-Vršovice	L-230	4000	Starý
Praha-Spořilov	L-260	1700	*null*
null	L-155	*null*	Horák

- Najděte všechny zákazníky, kteří mají v bance buď účet nebo půjčku (ale ne obojí).

select *zákazník-jméno*
from (*vkladatel* **natural full outer join** *půjčovatel*)
where *číslo-účtu* **is** *null* **or** *půjčka-číslo* **is** *null*

Jazyk definice dat (Data definition language; DDL)

Umožňuje specifikaci nejen množin atributů, ale také informaci o každé relaci, zahrnující:

- Schéma každé relace.
- Doménu hodnot spojenou s každým atributem.
- Omezení integrity.
- Množinu indexů, které budou udržovány pro každou relaci.
- Bezpečnostní a autorizační informace o každé relaci.
- Fyzickou strukturu uložení na disk pro každou relaci.

Doménové typy v SQL

- **char(n)**. Řetězec znaků s pevnou délkou *n*.
- **varchar(n)**. Řetězec znaků s proměnlivou délkou (maximálně *n*).
- **int**. Celé číslo; závislé na implementaci.
- **smallint**. Krátké celé číslo; závislé na implementaci.
- **numeric(p,d)**. Číslo s pevnou desetinnou čárkou s přesností na *p* míst s *d* místy vpravo od desetinné tečky.
- **real, double precision**. Čísla s plovoucí desetinnou čárkou; závislé na implementaci.
- **float(n)**. Číslo s plovoucí desetinnou čárkou s přesností nejméně na *n* míst.
- **date**. Datumy obsahující (4bitový) rok, měsíc a den.
- **time**. Čas v hodinách, minutách a sekundách.
  - Ve všech doménových typech jsou povoleny nulové hodnoty. Deklarování atributu **not null** je zakazuje nulové hodnoty pro daný atribut.
  - V SQL-92 vytvoří konstrukce **create domain** uživatelské doménové typy

create domain *osoba-jméno* **char**(20) **not null**

Konstrukce create table

- SQL relace je definována použitím příkazu **create table**:

  create table *r* (A_1 D_1, A_2 D_2, …, A_n D_n,
  <omezení-integrity$_1$>,
  …,
  <omezení-integrity$_k$>)

  - *r* je jméno relace
  - každé A_i je jméno atributu ve schématu relace *r*
  - D_i je datový typ hodnot v doméně atributu A_i
- Příklad:

```
create table pobočka
        (pobočka-jméno      char(15) not null,
        pobočka-město       char(30),
        aktiva              integer)
```

Omezení integrity v konstrukci create table

- **not null**
- **primary key** (A_1, …, A_n)
- **check** (*P*), kde *P* je predikát

Příklad: Deklarace *pobočka-jméno* jako primárního klíče pro *pobočka* a zajištění, že hodnota atributu *aktiva* bude nezáporná.

```
create table pobočka
        (pobočka-jméno      char(15) not null,
        pobočka-město       char(30),
        aktiva              integer,
        primary key (pobočka-jméno),
        check (aktiva >= 0))
```

- Deklarace atributu jako **primary key** zajišťuje v SQL-92 automaticky i **not null**.

Konstrukce drop table a alter table

- Příkaz **drop table** smaže všechny informace o dané relaci z databáze.
- Příkaz **alter table** přidá atributy k existující relaci. Ke všem n-ticím v relaci přiřadí *null* jako hodnotu po nové atributy. Tvar příkazu je následující

  alter table *r* **add** *A D*

  kde *A* je jméno atributu, který je přidán do relace *r,* a *D* je jeho doména.
- Příkaz **alter table** je často používán též k odstranění (drop) atributů z relace:

  alter table *r* **drop** *A*

  kde *A* je jméno atributu relace *r*.

Zabudovaný SQL

- Standard SQL definuje zabudování SQL do řady programovacích jazyků jako jsou Pascal, PL/I, Fortran, C a Cobol.
- Jazyk, ve kterém jsou zabudovány SQL dotazy, je nazýván *hostitelský* jazyk a SQL struktury povolené v hostitelském jazyce vytvářejí *zabudovaný* SQL.
- Výraz EXEC SQL je používán pro identifikaci vloženého SQL dotazu pro preprocesor

EXEC SQL <vložený SQL výraz> END EXEC

Příklad dotazu

Z hostitelského jazyka najděte jména a čísla účtu s více než *částka* dolary na nějakém účtu.

- Specifikujte dotaz v SQL a deklarujete pro ni *kurzor*

```
EXEC SQL
    declare c cursor for
    select zákazník-jméno, číslo-účtu
    from vkladatel, účet
    where vkladatel.číslo-účtu = účet.číslo-účtu
        and účet.zůstatek > :částka
END-EXEC
```

Zabudovaný SQL (pokračování)

- Výraz **open** způsobí vyhodnocení dotazu

EXEC SQL **open** *c* END-EXEC

- Výraz **fetch** způsobí, že hodnoty jedné n-tice budou vloženy do proměnných hostitelského jazyka.

EXEC SQL **fetch** *c* **into** :*cn* :*an* END-EXEC

- Výraz **close** zavře databázový systém, takže se smaže dočasná relace, která obsahuje výsledek dotazu

EXEC SQL **close** *c* END-EXEC

Dynamický SQL

- Umožňuje programům konstruovat SQL dotazy za běhu.
- Příklad použití dynamického SQL z programu v jazyce C.

```
char *sqlprog = „update účet set zůstatek = zůstatek * 1.05
                where číslo-účtu = ?“
EXEC SQL prepare dynprog from :sqlprog;
char účet[10] = „A-101“;
EXEC SQL execute dynprog using :účet;
```

- Dynamický SQL program obsahuje znak ?, který je místem pro hodnotu, která je poskytována před spuštěním SQL programu.

Další rysy SQL

- *Jazyky čtvrté generace* –speciální jazyky asistující aplikačním programátorům při tvorbě uživatelského rozhraní a formátování výstupu
- SQL sezení (SQL session) – poskytuje abstrakci klienta a serveru
  - klient se *připojí* k SQL serveru, zahájí sezení
  - spustí sérii výrazů
  - *odpojí* sezení
  - může odevzdat nebo vrátit práci uskutečněnou v sezení
- SQL prostředí obsahuje několik komponent (identifikátor uživatele a schéma, které identifikuje, které z několika schémat sezení používá).

Kapitola 6: Omezení integrity

- Omezení domény
- Referenční integrita
- Aserce
- Spouštěče (Triggers)
- Funkční závislosti

Omezení domény

- Omezení integrity zabraňují poškození databáze; zajišťují, že autorizované zásahy do databáze nezpůsobí ztrátu konzistence dat.
- Omezení domény je základní formou omezení integrity.
- Omezení integrity testují hodnoty vkládané do databáze a kontroluje dotazy, aby bylo zajištěno, že porovnávání dávají smysl.
- Klauzule **check** dovoluje v SQL-92 omezit domény:
  - Použijte klauzuli **check** k zajištění, že doména *hodinová-mzda* je větší než specifikovaná hodnota

    create domain *hodinová-mzda* **numeric(5,2)**
    constraint *value-test* **check** (*value* >= 4.00)
  - Doména *hodinová-mzda* je deklarována jako dekadické číslo s pěti číslicemi, z nichž 2 jsou za desetinnou čárkou
  - Doména má omezení, které zaručuje, že *hodinová-mzda* je větší než nebo rovno 4.00
  - Klauzule **constraint** *value-test* je volitelná; užitečná pro indikaci, které omezení bylo při aktualizaci porušeno.

Referenční integrita

- Zajišťuje, že hodnota, která se objeví v jedné relaci pro danou množinu atributů, se také objeví v určitých množinách atributů v jiné relaci.
  - Příklad: jestliže „Praha-Spořilov“ je jméno pobočky, které se objeví v jedné z n-tic v relaci *účet*, pak zde existuje n-tice v relaci *pobočka* pro pobočku „Praha-Spořilov“.
- Formální definice
  - Nechť r_1 (R_1) a r_2 (R_2) jsou relace s primárními klíči K_1 a K_2.
  - Podmnožina α z R_2 je *cizí klíč* odkazující na K_1 v relaci r_1, jestliže pro každé t_2 v r_2 musí být n-tice v taková, že $t_1[K_1] = t_2[\alpha]$.
  - Omezení referenční integrity: $\Pi_\alpha (r_2) \subseteq \Pi_{K1} (r_1)$

Referenční integrita v E-R modelu

- Uvažujme množinu vztahů R mezi množinami entit E_1 a E_2. Relační schéma pro R zahrnuje primární klíče K_1 z E_1 a K_2 z E_2. Pak K_1 a K_2 jsou v R cizí klíče pro relační schémata E_1 a E_2.
- Slabé množiny entit jsou též zdroje pro omezení referenční integrity. Proto musí relační schéma pro každou slabou množinu entit zahrnovat primární klíč množiny vztahů, na které závisí, tj. nadřazené entitní množiny

Modifikace databáze

- Tyto testy musí být provedeny, aby se zachovala následující mez referenční integrity:

$$\Pi_{\alpha}(r_2) \subseteq \Pi_K(r_1)$$

- **Vkládání.** Je-li n-tice t_2 vkládána do r_2, systém musí zajistit, že v r_1 je n-tice t_1 taková, že $t_1[K_1] = t_2[\alpha]$. Tj.

$$t_2[\alpha] \in \Pi_K(r_1)$$

- **Mazání.** Je-li n-tice t_1 mazána z r_1, systém musí spočítat množinu n-tic v r_2, která odkazuje na t_1:

$$\sigma_{\alpha = t1[K]}(r_2)$$

  Jestliže tato množina není prázdná, příkaz mazání skončí s chybou nebo se n-tice, které odkazují na t_1, musí samy smazat (rekurzivní mazání je možné).
- **Aktualizace.** Jsou dvě možnosti:
  - Je-li n-tice t_2 aktualizován v relaci r_2 a aktualizace modifikuje hodnoty pro cizí klíč α, pak je proveden podobný test jako při operaci vkládání. Nechť t_2‘ značí novou hodnotu n-tice t_2. Systém musí zajistit, že

$$t_2'[\alpha] \in \Pi_K(r_1)$$

  - Je-li n-tice t_1 aktualizována v r_1 a aktualizace modifikuje hodnoty primárního klíče (K), pak je prováděn podobný test jako při operaci mazání. Systém musí spočítat

$$\sigma_{\alpha = t1[K]}(r_2)$$

    s použitím staré hodnoty t_1. Není-li tato množina prázdná, aktualizace skončí s chybou nebo je změna kaskádovitě provedena na těchto n-ticích nebo mohou být tyto n-tice v relaci smazány.

Referenční integrita v SQL

- Primární a kandidátní klíče a cizí klíče můžou být specifikovány jako část SQL příkazu **create table**:
  - Klauzule **primary key** v příkazu **create table** zahrnuje seznam atributů, které tvoří primární klíč.
  - Klauzule **unique key** v příkazu **create table** zahrnuje seznam atributů, které tvoří kandidátní klíč.
  - Klauzule **foreign key** v příkazu **create table** zahrnuje seznam atributů, které tvoří cizí klíč a jméno relace odkazované cizím klíčem.

Příklad referenční integrity v SQL

create table *zákazník*
(*zákazník-jméno* char(20) **not null**,
zákazník-ulice char(30),
zákazník-město char(30),
primary key (*zákazník-jméno*))

create table *pobočka*
(*pobočka-jméno* char(15) **not null**,
pobočka-město char(30),
aktiva integer,
primary key (*pobočka-jméno*))

create table *účet*
(*pobočka-jméno* char(15),
číslo-účtu char(10) **not null**,
zůstatek integer,
primary key (*číslo-účtu*),
foreign key (*pobočka-jméno*) **references** *pobočka*)

create table *vkladatel*
(*zákazník-jméno* char(20) **not null**,
číslo-účtu char(10) **not null**,
primary key (*zákazník-jméno, číslo-účtu*),
foreign key (*číslo-účtu*) **references** *účet*,
foreign key (*zákazník-jméno*) **references** *zákazník*)

Kaskádové akce v SQL

create table *účet*
…
foreign key () **references** *pobočka*
on delete cascade
on update cascade,
…)

- Díky klauzuli **on delete cascade**, vyvolá-li mazání n-tice v relaci *pobočka* porušení referenční integrity, mazání „přeteče“ do relace *účet* a smaže n-tice, které odkazují na n-tice v relaci *pobočka*, které jsou mazány.
- Kaskádové aktualizace jsou obdobné.
- Je-li mezi více relacemi sada závislostí cizích klíčů (každý se specifikací **on delete cascade**), mazání nebo aktualizace se může rozšířit do celé této sady.
- Jestliže kaskádová aktualizace či mazání vyvolá porušení integritních omezení, které nemůže zachytit dřívější kaskádová operace, systém zruší transakci. Výsledek: všechny změny způsobené transakcí a jejími kaskádovými akcemi jsou zrušeny.

Tvrzení (Assertion)

- Tvrzení (aserce, prosazování) je predikát vyjadřující podmínku, kterou musí databáze vždy splňovat.
- V SQL-92 má tvar
  create assertion <jméno-tvrzení> **check** <predikát>
- Když je tvrzení vytvořeno, systém ji otestuje, je-li platná.

Příklad

- Suma všech částek půjček pro každou pobočku musí být menší než suma všech zůstatků účtů v pobočce.

```
create assertion mez-sumy check
    (not exists (select * from pobočka
       where (select sum (částka) from půjčka
         where půjčka.pobočka-jméno = pobočka.pobočka-jméno)
             >= (select sum (částka) from účet
         where půjčka.pobočka-jméno = pobočka.pobočka-jméno)))
```

- Každá půjčka má alespoň jednoho dlužníka, který udržuje účet s minimálním zůstatkem $1000.00.

```
create assertion mez-zůstatku check
    (not exists (select * from půjčka
    where not exists ( select *
       from dlužník, vkladatel, účet
       where půjčka.půjčka-číslo = dlužník.půjčka-číslo
          and dlužník.zákazník-jméno = vkladatel.zákazník-jméno
          and vkladatel.účet-číslo = účet.účet-číslo
          and účet.zůstatek >= 1000)))
```

Spouště (Triggers)

- *Spouštěč (Trigger,* spoušť) je příkaz, který systém automaticky spouští jako vedlejší efekt modifikace databáze.
- Pro aktivaci tohoto mechanismu musíme:
  – Specifikovat podmínky, za jakých je spouštěč prováděn.
  – Specifikovat akce, které se budou dít při jeho spuštění.
- Standard SQL-92 nezahrnuje spouště, ale mnoho implementací je podporuje.

Příklad triggeru

- Předpokládejme, že místo povolení záporných zůstatků účtů banka zachází se saldem (přečerpáním) takto:
  – nastaví zůstatek účtu na nulu
  – vytvoří půjčku s částkou salda
  – dá této půjčce číslo tohoto účtu
- Podmínka spuštění této spouště je aktualizace relace *účet*, která vrátí zápornou hodnotu *zůstatek*.

```
define trigger saldo on update of účet as T
      (if new T.zůstatek < 0
      then (insert into půjčka values
              (T.pobočka-jméno, T.číslo-účtu, - new T.zůstatek)
            insert into půjčovatel
              (select zákazník-jméno, číslo-účtu
              from vkladatel
              where T.číslo-účtu = vkladatel.číslo-účtu)
            update účet as S
            set S.zůstatek = 0
            where S.číslo-účtu = T.číslo-účtu)
       )
```

Klíčové slovo **new** před *T.zůstatek* značí, že by měla být použita hodnota *T.zůstatek* po aktualizaci; je-li vynecháno, je použita hodnota před aktualizací

Funkční závislosti

- Omezení na množině povolených (legálních) relací.
- Vyžaduje, že hodnota jisté množiny atributů jednoznačně určuje hodnotu jiné množiny atributů.
- Funkční závislost je zobecnění představy *klíče*.
- Nechť *R* je relační schéma

$$\alpha \subseteq R,\ \beta \subseteq R$$

- Funkční závislost $\alpha \rightarrow \beta$

je platná pro schéma *R* právě tehdy, když pro jakoukoli povolenou relaci *r*(*R*), jakékoli dvě n-tice t_1 a t_2 z *r* jsou stejné na atributech α, souhlasí též na atributech β. Tj.

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- *K* je superklíč pro relační schéma *R* právě tehdy, když $K \rightarrow R$
- *K* je kandidátní klíč *R* právě tehdy, když
  – $K \rightarrow R$, a
  – pro žádné $\alpha \subset K$, $\alpha \rightarrow R$
- Funkční závislosti umožňují vyjádřit omezení, které nelze vyjádřit pomocí superklíčů. Představme si schéma:

  info-půjčka = (*pobočka-jméno, půjčka-číslo, zákazník-jméno, částka*).

Očekáváme, že se bude platit tato množina funkčních závislostí:

půjčka-číslo $\rightarrow$ *částka*

půjčka-číslo $\rightarrow$ *pobočka-jméno*

ale nechceme splňovat následující:

půjčka-číslo $\rightarrow$ *zákazník-jméno*

Použití funkčních závislostí

- Funkční závislosti používáme k:
  - testování relací, jsou-li povolené na dané množině funkčních závislostí. Je-li relace *r* povolená na množině *F* funkčních závislostí, říkáme, že *r splňuje F*.
  - definování omezení na množině povolených relací; říkáme, že *F je platná na R*, když všechny povolené relace na *R* splňují množinu *F*.
- Poznámka: Některá instance relačního schématu (tj. relace) může splňovat funkční závislosti, i když tyto závislosti nejsou platné pro všechny povolené instance. Např. specifická instance *info-půjčka* může (náhodou) vyhovovat *půjčka-číslo* $\rightarrow$ *zákazník-jméno*

Uzávěr množiny funkčních závislostí

- Pro danou množinu funkčních závislostí *F* existují další funkční závislosti, které *F* logicky implikuje (tzv. *uzávěr* množiny *F*).
- *Uzávěr F* značíme F^+.
- Všechny F^+ můžeme najít pomocí Armstrongových axiomů:
  - je-li $\beta \subseteq \alpha$, pak $\alpha \rightarrow \beta$ (**reflexivita**)
  - je-li $\alpha \rightarrow \beta$, pak $\gamma\alpha \rightarrow \gamma\beta$ (**rozšíření**)
  - je-li $\alpha \rightarrow \beta$ a $\beta \rightarrow \gamma$, pak $\alpha \rightarrow \gamma$ (**tranzitivita**)

  Tyto axiomy tvoří minimální a úplnou množinu axiomů.
- Výpočet F^+ můžeme zjednodušit použitím následujících doplňkových pravidel.
  - je-li $\alpha \rightarrow \beta$ a $\alpha \rightarrow \gamma$, pak $\alpha \rightarrow \beta\gamma$ (**sjednocení**)
  - je-li $\alpha \rightarrow \beta\gamma$, pak $\alpha \rightarrow \beta$ a $\alpha \rightarrow \gamma$ (**rozklad**)
  - je-li $\alpha \rightarrow \beta$ a $\gamma\beta \rightarrow \delta$, pak $\alpha\gamma \rightarrow \delta$ (**pseudotranzitivita**)

  Tato pravidla jsou odvozena z Armstrongových axiomů.

Příklad

- *R = (A, B, C, G, H, I)*
- *F = {A* $\rightarrow$ *B*
  A $\rightarrow$ *C*
  CG $\rightarrow$ *H*
  CG $\rightarrow$ *I*
  B $\rightarrow$ *H}*
- některé prvky F^+:
  - *A* $\rightarrow$ *H*
  - *AG* $\rightarrow$ *I*
  - *CG* $\rightarrow$ *HI*

Uzávěr množiny atributů

- *Uzávěr* atributu α pod *F* (značíme α^+) definujeme jako množinu atributů, které jsou funkčními závislostmi *F* určeny z α:

$$\alpha \rightarrow \beta \text{ je z } F^+ \Leftrightarrow \beta \subseteq \alpha^+$$

- Algoritmus pro výpočet α^+

```
result := α;
while (byla změna v result) do
    for each β → γ in F do begin
        if β ⊆ result then result := result ∪ γ;
    end
```

Příklad

- *R = (A, B, C, G, H, I)*
- *F = {A → B*
  A → C
  CG → H
  CG → I
  B → H}
- (AG^+)
  - 1. krok: *result = AG*
  - 2. krok: *result = ABCG* ($A \rightarrow C$ a $A \subseteq AGB$)
  - 3. krok: *result = ABCGH* ($CG \rightarrow H$ a $CG \subseteq AGBC$)
  - 4. krok: *result = ABCGHI* ($CG \rightarrow I$ a $CG \subseteq AGBCH$)
- Je *AG* kandidátní klíč?
  - 1. $AG \rightarrow R$
  - 2. je $A^+ \rightarrow R$?
  - 3. je $G^+ \rightarrow R$?

Kanonický obal (Canonical Cover)

- Uvažujme množinu funkčních závislostí *F* a funkční závislost $\alpha \rightarrow \beta$ z *F*.
  - Atribut *A* je nadbytečný v α, jestliže $A \in \alpha$ a *F* logicky implikuje
    $$(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$$
  - Atribut *A* je nadbytečný v β, jestliže $A \in \beta$ a množina funkčních vztahů
    $$(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$ logicky implikuje *F*.
- Kanonický obal F_C pro *F* je množina závislostí taková, že *F* logicky implikuje všechny závislosti v F_C a F_C logicky implikuje všechny závislosti ve *F* a navíc
  - žádná funkční závislost v F_C neobsahuje nadbytečný atribut
  - každá levá strana funkční závislosti z F_C je jedinečná

- Spočítejte kanonický obal pro *F*:
  repeat
    Použijeme pravidlo sjednocení pro nahrazení všech závislostí
      $\alpha_1 \rightarrow \beta_1$ a $\alpha_1 \rightarrow \beta_2$ z *F* jednou závislostí $\alpha_1 \rightarrow \beta_1 \beta_2$
    Najdeme funkční závislost $\alpha \rightarrow \beta$ s nadbytečným atributem buď
      v α nebo v β
    Je-li tento atribut nalezen, smažeme ho z $\alpha \rightarrow \beta$
  until *F* se nezměnilo

Příklad

- *R = (A, B, C)*
- *F = {*$A \rightarrow BC$
  $B \rightarrow C$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C$*}*
- Zkombinujeme $A \rightarrow BC$ a $A \rightarrow B$ do závislosti $A \rightarrow BC$.
- *A* je nadbytečné v $AB \rightarrow C$, protože $B \rightarrow C$ logicky implikuje $AB \rightarrow C$.
- *C* je nadbytečné v $A \rightarrow BC$, je-li $A \rightarrow BC$ logicky implikováno z $A \rightarrow B$ a $B \rightarrow C$.
- Kanonický obal je:

$$A \rightarrow B$$
$$B \rightarrow C$$

Kapitola 7: Návrh relačních databází

- Nástrahy návrhu relačních databází
- Dekompozice (rozklad)
- Normalizace použitím funkčních závislostí

Nástrahy relačního návrhu

- Návrh relačních databází vyžaduje nalézt nějakou dobrou množinu relačních schémat. Špatný návrh může vést k:
  - Opakování stejné informace
  - Nemožnosti vyjádřit nějakou informaci
- Cíle návrhu:
  - Zabránit redundanci (opakování) dat
  - Zajistit vyjádření všech vztahů mezi atributy
  - Usnadnit kontrolu porušení integritních omezení databáze při změnách dat

Příklad

- Uvažujme relační schéma:
  schéma-půjček = (jméno-pobočky, město-pobočky, aktiva, jméno-zákazníka, číslo-půjčky, zůstatek)
- Redundance (opakování, nadbytečnost):
  - Data o *jméno-pobočky, město-pobočky, aktiva* jsou opakována pro každou půjčku, kterou pobočka vytvoří
  - Plýtvá místem a komplikuje provádění změn
- Null hodnoty
  - Nelze uložit informace o pobočce, pokud neexistuje žádná půjčka
  - Lze použít prázdné (null) hodnoty, ale je obtížné s nimi pracovat

Rozklad (dekompozice)

- Rozložit relační schéma *schéma-půjček* do:
  pobočka-zákazník-schéma = (jméno-pobočky, město-pobočky, aktiva, jméno-zákazníka)
  zákazník-půjčka-schéma = (jméno-zákazníka, číslo-půjčky, zůstatek)
- Všechny atributy původního schématu R se musí objevit v rozkladu (R_1,R_2):
  $R = R_1 \cup R_2$
- Rozklad bezeztrátového spojení
  Pro všechny možné relace r na schématu R platí
  $r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$

Příklad bezeztrátového rozkladu

- Dekompozice schématu $R=(A,B)$

$R_1=(A)$ $R_2=(B)$

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_B(r)$

- $\Pi_A(r) \bowtie \Pi_B(r)$

A	B
α	1
α	2
β	1
β	2

Cíl – navrhnout teorii

- Rozhodnout, zda-li určité relační schéma je v „dobrém“ tvaru.
- V případě, že schéma R není v „dobrém“ tvaru, rozlož jej na množinu $\{R_1,R_2,\ldots,R_n\}$ takovou, že
  - Každá relace R_i je v „dobrém“ tvaru
  - Rozklad je bezeztrátový
- Tato teorie je založená na:
  - Funkčních závislostech
  - Vícehodnotových závislostech

Normalizace pomocí funkčních závislostí

Pokud dekomponujeme relační schéma R s množinou funkčních závislostí F do schémat R_1 a R_2, požadujeme aby:

- Bezeztrátovost spojení: nejméně jedna z následujících závislostí je v F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- Žádná redundance: relační schémata R_1 a R_2 by nejlépe měly být v Boyce-Coddově normální formě nebo ve třetí normální formě.
- Uchování závislostí: nechť F_i je množina závislostí v F^+, která obsahuje pouze atributy ve schématu R_i, ověříme, že platí:
  - $(F_1 \cup F_2)^+ = F^+$

  Jinak je test na porušení funkčních závislostí příliš drahý.

Příklad

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - Rozklad pomocí bezeztrátového spojení:
    $R_1 \cap R_2 = \{B\}$ a $B \rightarrow BC$
  - Zachovává závislosti
- $R_1 = (A, B)$, $R_2 = (A, C)$
  - Rozklad pomocí bezeztrátového spojení:
    $R_1 \cap R_2 = \{A\}$ a $A \rightarrow AB$
  - Nezachovává závislosti
    (nelze otestovat $B \rightarrow C$ bez výpočtu spojení $R_1 \bowtie R_2$)

Boyce-Coddova normální forma

Relační schéma R je v BCNF vzhledem k množině funkčních závislostí F, pokud pro všechny funkční závislosti v F^+ tvaru $\alpha \rightarrow \beta$, kde $\alpha \subseteq R$ a $\beta \subseteq R$, je splněna alespoň jedna z podmínek:

- $\alpha \rightarrow \beta$ je triviální (tj. $\beta \subseteq \alpha$)
- α je superklíč schématu R

Příklad

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$, klíč = $\{A\}$
- R není v BCNF
- Rozklad na $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 a R_2 je v BCNF
  - Splněna podmínka bezeztrátovosti spojení
  - Zachovává závislosti

BCNF – algoritmus rozkladu

result := {*R*};
done := false;
vypočítej F^+;
while (**not** *done*) **do**
  if (existuje schéma R_i v *result*, které není v BCNF) **then**
    nechť $\alpha \rightarrow \beta$ je netriviální funkční závislost, která je splněna na R_i
      taková, že $\alpha \rightarrow R_i$ není v F^+ a $\alpha \cap \beta = \emptyset$;
    result := (*result* - R_i) $\cup$ (R_i - β) $\cup$ (α,β);
  else
    done := true;
  fi

Poznámka: každé schéma R_i je v BCNF a rozklad je bezeztrátový podle spojení.

Příklad rozkladu do BCNF

- *R = (jméno-pobočky, město-pobočky, aktiva, jméno-zákazníka, číslo-půjčky, zůstatek)*
  F = {jméno-pobočky → aktiva město-pobočky
  číslo-půjčky → zůstatek jméno-pobočky}
  Key = {číslo-půjčky, jméno-zákazníka}
- První rozklad
  - R_1 *= (jméno-pobočky, město-pobočky, aktiva)*
  - R_2 *= (jméno-pobočky, jméno-zákazníka, číslo-půjčky, zůstatek)*
- Druhý rozklad
  - R_1 *= (jméno-pobočky, město-pobočky, aktiva)*
  - R_3 *= (jméno-pobočky, číslo-půjčky, zůstatek)*
  - R_4 *= (jméno-zákazníka, číslo-půjčky)*
- Výsledný rozklad:

  R_1, R_3, R_4

BCNF a zachování závislostí

Vždy není možné vytvořit rozklad do BCNF normální formy, který zachovává funkční závislosti.

- *R = (J, K, L)*
  F = {JK → L
  L → K}
  Dva kandidátní klíče *JK* a *JL*
- *R* není v BCNF
- Jakýkoli rozklad schématu *R* nebude splňovat závislost:

  $JK \rightarrow L$

Třetí normální forma

- Relační schéma *R* je ve třetí normální formě (3NF), jestliže pro všechny závislosti

  $\alpha \rightarrow \beta$ z F^+

  alespoň jedna podmínka z následujících platí:
  - $\alpha \rightarrow \beta$ je triviální (tj. $\beta \subseteq \alpha$)
  - α je superklíč schématu *R*
  - každý atribut *A* v $\beta - \alpha$ je obsažený v kandidátním klíči schématu *R*
- Pokud je relace v BCNF, pak je i v 3NF (protože v BCNF musí platit alespoň jedna z prvních dvou podmínek).
- Příklad:
  - *R = (J, K, L)*
    F = {JK → L, L → K}
  - Dva kandidátní klíče: *JK* a *JL*
  - *R* je ve 3NF

$JK \rightarrow L$	*JK* je superklíč
$L \rightarrow K$	*K* je obsažený v kandidátním klíči

- Algoritmus pro rozklad relačního schématu *R* do množiny relačních schémat {R_1, R_2, ..., R_n} zaručuje následující:
  - Každé relační schéma R_i je ve 3NF
  - Rozklad splňuje bezeztrátovost spojení
  - Zachovává závislosti

3NF algoritmus pro rozklad

Nechť F_C je kanonický uzávěr množiny *F*;
i := 0;
for each funkční závislost $\alpha \rightarrow \beta$ z F_C **do**
  if žádné ze schémat R_j, $1 \leq j \leq i$ neobsahuje $\alpha\beta$ **then**
    i := *i* + 1;
    $R_i := \alpha\beta$;
if žádné ze schémat R_j, $1 \leq j \leq i$ neobsahuje kandidátní klíč pro *R* **then**
  i := *i* + 1;
  R_i := libovolný kandidátní klíč pro *R*;
return (R_1, R_2, ..., R_n)

Příklad

- Relační schéma:
  schéma-bankéř-info = (jméno-pobočky, jméno-zákazníka, jméno-bankéře, číslo-kanceláře)
- Funkční závislosti pro toto schéma jsou:
  jméno-bankéře $\rightarrow$ *jméno-pobočky číslo-kanceláře*
  jméno-zákazníka jméno-pobočky $\rightarrow$ *jméno-bankéře*
- Klíč schématu je:
  {jméno-zákazníka, jméno-pobočky}
- **For** cyklus v algoritmu způsobí vytvoření schémat:
  schéma-bankéř-kancelář = (jméno-bankéře, jméno-pobočky, číslo-kanceláře)
  schéma-bankéř = (jméno-zákazníka, jméno-pobočky, jméno-bankéře)
- Protože *schéma-bankéř* obsahuje kandidátní klíč pro *schéma-bankéř-info*, jsme hotovi s rozkladem.

Porovnání BCNF a 3NF

- Vždy je možné provést rozklad schématu do více schématů, které jsou v 3NF, a
  - rozklad je bezeztrátový
  - závislosti jsou zachovány
- Vždy je možné provést rozklad schématu do více schématů, které jsou v BCNF, a
  - rozklad je bezeztrátový
  - všechny závislosti nemusí být zachovány

- $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$
- Uvažujme následující relaci:

J	***L***	***K***
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

- Schéma, které je v 3NF ale ne v BCNF, má následující problémy:
  - opakování informací (např. vztah l_1,k_1)
  - potřebuje používat prázdné hodnoty (null) (např. pro vyjádření vztahu l_2,k_2 nemáme odpovídající hodnotu pro atribut *J*).

Cíle návrhu

- Cíle návrhu relačních databází jsou:
  - BCNF
  - bezeztrátové spojení
  - zachování závislostí
- Pokud předchozího nemůžeme dosáhnout, stačí nám:
  - 3NF
  - bezeztrátové spojení
  - zachování závislostí

Další normální formy

- Existují i další normální formy, zejména:
  - 1. normální forma
    * Relační schéma je v 1NF tehdy a jen tehdy, pokud každý atribut je atomický (tj. není vícehodnotový)
  - 2. normální forma
    * Relační schéma je v 2NF právě, když je v 1NF a každý atribut závisí na klíči (primárním klíči), pozn. závislost může být i tranzitivní, závislost musí být na celém klíči nikoli jen na některé jeho části.

Kapitola 10: Diskové a souborové struktury

- Přehled fyzických ukládacích médií
- Magnetické disky
- RAID (Redundant Array of Inexpensive Disks)
- Terciární úložiště
- Přístup k médiu
- Souborové organizace

Klasifikace fyzických médií

- Rychlost, se kterou jsou data přístupná
- Náklady na jednotku dat
- Spolehlivost
  - Ztráta dat při výpadku proudu nebo při pádu systému
  - Fyzická chyba na ukládacím médiu

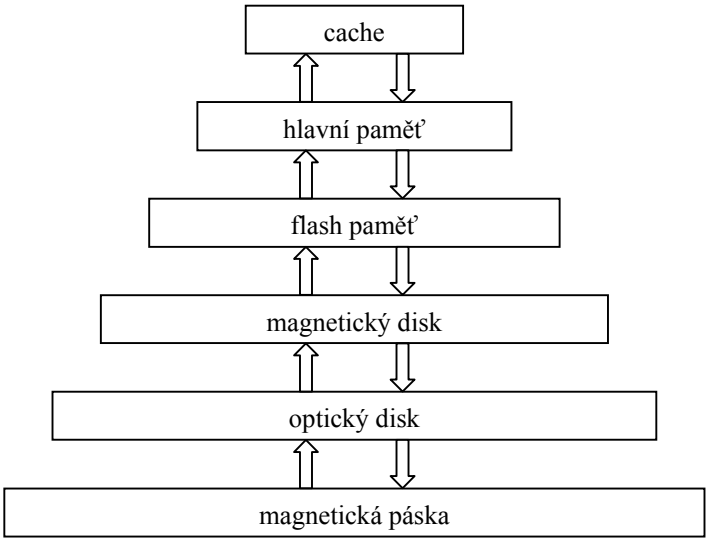
  Média lze rozdělit do skupin:
  - **Nestálá paměť** – obsah paměti se ztratí při přerušení proudu
  - **Stálá paměť** – obsah je uchován, když je napájení odpojeno. Zahrnuje sekundární a terciární úložiště, stejně jako hlavní paměť zálohovaná baterií.

Fyzická média

- Cache (keš) – nejrychlejší a nejdražší paměť, nestálá, obsluhována operačním systémem nebo hardwarem
- Hlavní paměť:
  - Strojové instrukce pracují s daty uloženými v hlavní paměti
  - Rychlý přístup, ale obecně stále příliš malá pro uložení celé databáze
  - Někdy též nazývaná jako *operační* paměť
  - Nestálá – obsah paměti je ztracený při pádu systému nebo výpadku napájení
- Flash paměti – čtení je téměř stejně rychlé jako hlavní paměť, obsah paměti je nezávislý na napájení, v některých případech může být omezený počet přepisovacích cyklů
- Magnetické disky – primární místo pro dlouhodobé ukládání dat, typicky lze uložit celou databázi.
  - Data musí být přesouvána z disku do hlavní paměti při zpracování a zapsána zpět pro uskladnění
  - **Přímý přístup** – obvykle lze data číst v libovolném pořadí (náhodný přístup)
  - Obsah většinou přežijí pád systému a výpadky napájení; chyba disku může způsobit ztrátu dat, ale tyto chyby jsou mnohem méně časté než pád systému.
- Optické disky – stálá paměť, nejznámější je CD-ROM, DVD. Většinou ve formě médií pro jeden zápis a vícenásobné čtení, které se většinou používají pro archivaci. Vyskytují se i média pro několikanásobné přepsání (počet přepisů se omezený).
- Pásková paměti – stálé, primární využití je archivace a zálohování

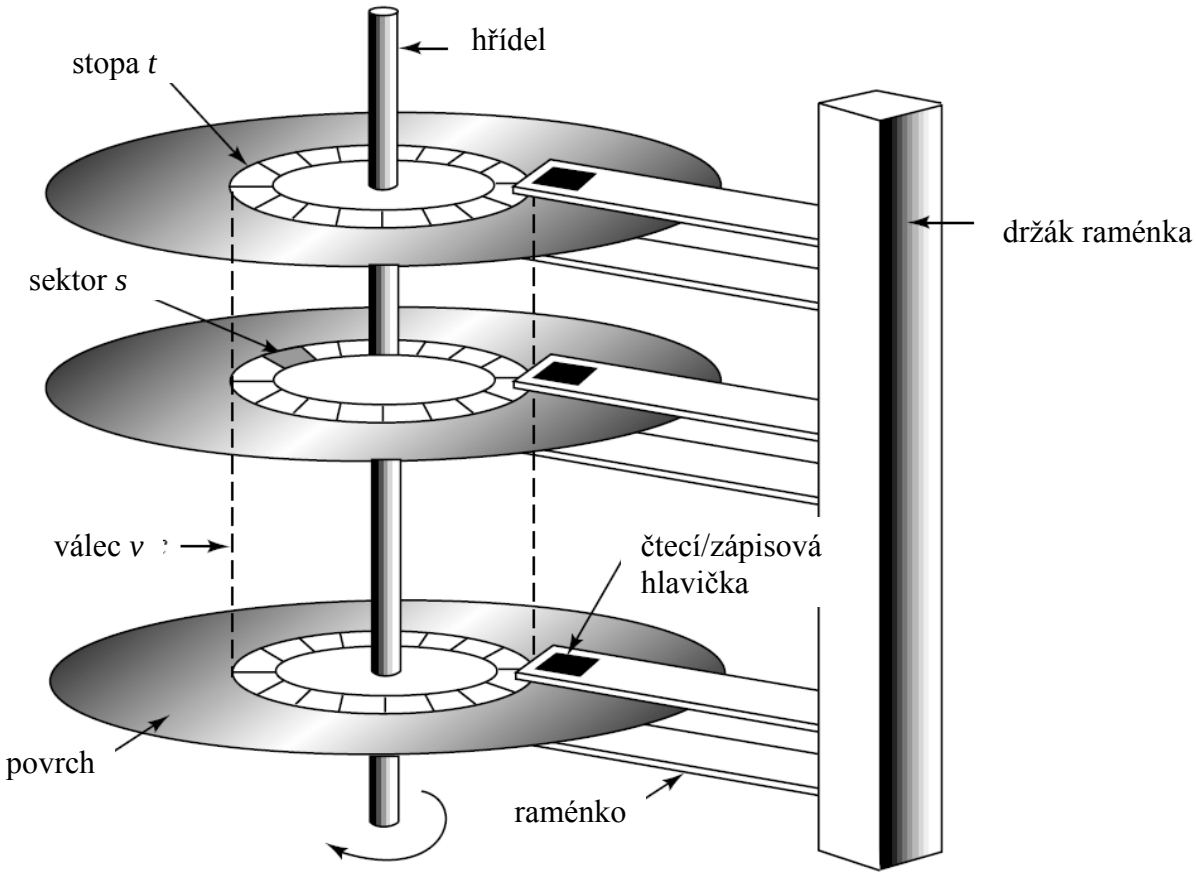
- Sekvenční přístup – mnohonásobně pomalejší než disk
- Velmi vysoká kapacita – i terabytové pásky
- Pásky lze v mechanice měnit, levnější médium než disk.

Hierarchie pamětí



- **Primární paměti:** nejrychlejší ale nestálé (cache, hlavní paměť)
- **Sekundární paměti:** další úroveň v hierarchii, stálé, poměrně rychlý přístup k datům, též nazývané jako *on-line paměti* (flash paměť, magnetický disk)
- **Terciární paměti:** nejnižší úroveň v hierarchii, stálé, pomalý přístup, též nazývané jako *off-line paměti* (magnetické pásky, optické disky)

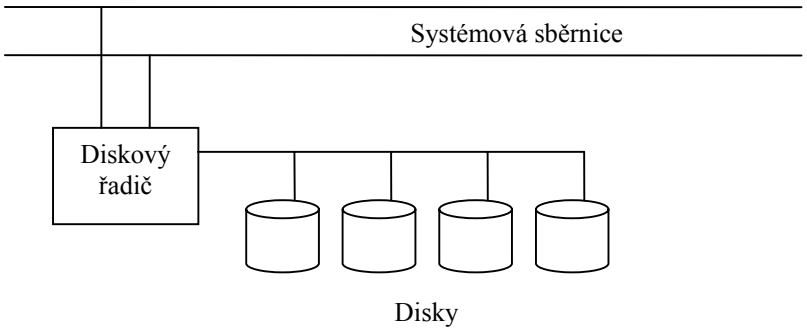
Mechanizmus magnetických disků



Magnetické disky

- Čtecí/zápisová hlavička – zařízení pohybující se v těsné blízkosti nad povrchem plotny disku, slouží k magnetickému čtení a zápisu informací na disk.
- Povrch plotny je rozdělený do kruhových *stop* (tracks), každá stopa je dělena do *sektorů.* Sektor je nejmenší datová jednotka, kterou lze číst a zapisovat.
- Pro čtení/zápis sektoru:
  - Raménko je přesune hlavičku na správnou stopu
  - Plotna se nepřetržitě otáčí, data jsou čtena/zapisována, když se sektor dostane pod hlavičku.
- Způsob montáže: více diskových ploten na jedné hřídeli s více hlavičkami (jedna pro plotnu) na společném raménku.
- **Válec** *i* se skládá z *i-té* stopy na každé plotně

Diskový podsystém



- Diskový řadič (controller) – rozhraní mezi počítačovým systémem a hardware disku.
  - Přijímá vyšší příkazy (high-level commands) pro čtení/zápis sektoru
  - Spouští akce typu vystavení raménka na správnou stopu a vlastní čtení a zápis dat.

Výkonnostní míry disků

- **Přístupová doba** – doba potřebná ke čtení/zápisu (od vyslání požadavku na čtení/zápis po zahájení přenosu dat). Skládá se z:
  - **Čas vystavení** – doba potřebná pro vystavení raménka na požadovanou stopu. Průměrná doba vystavení je třetina nejhorší hodnoty.
  - **Rotační zpoždění** – doba potřebná na otočení plotny, aby se požadovaný sektor dostal pod hlavičku. Průměrná hodnota je polovina nejhoršího času (tj. polovina doby jedné otáčky)
- **Rychlost přenosu dat** – rychlost s jakou jsou data přenášena z disku nebo na disk.
- **Střední doba poruchy** (MTTF) – průměrná doba mezi dvěma výpadky disku.

Optimalizace blokového přístupu k disku

- **Blok** – souvislá posloupnost sektorů na jedné stopě
  - Data jsou mezi diskem a hlavní pamětí přenášena v blocích
  - Velikost se pohybuje od 512 bytů do několika kilobitů
- Algoritmy pro vystavování raménka na stopy řadí požadavky tak, aby byl pohyb raménka minimalizovaný (algoritmus *výtah* je často používaný)
- Organizace souborů – bloky jsou organizovány tak, aby pořadí odpovídalo pořadí v jakém budou data čtena. Blízké informace jsou uložené na stejném válci nebo sousedním válci.
- **Stálé vyrovnávací paměti** – urychlují zápis dat na disk tím, že jsou data uložena do vyrovnávací paměti (stálá paměť); řadič zapíše data na disk ve chvíli, když disk nemá práci.

RAID

- **Nezávislá pole levných disků** (Redundant Arrays of Inexpensive Disks) – technika organizace disků, která spojuje více běžně dostupných disků do jednoho systému.
- Původně poměrně levná alternativa k velkým a velmi drahým diskům
- Dnes jsou RAID používány pro jejich vysokou spolehlivost a rychlost spíše než z ekonomických důvodů. Proto je „I“ v názvu chápáno jako **independent** (nezávislý) než levný.

Zvýšení spolehlivosti pomocí redundance

- Pravděpodobnost, že některý disk z množiny *N* disků je chybný je mnohem vyšší než pravděpodobnost havárie jednoho určitého disku. Např. systém se 100 disků, každý s MTTF 100 tisíc hodin (cca 11 let) bude mít MTTF rovnu 1000 hodinám (cca 41 dnů)!!!
- **Redundance** (nadbytečnost) – ukládej zvláštní informaci, kterou lze využít pro opětovné vytvoření informace ztracené při výpadku disku
- Např. **zrcadlení**
  - Zdvojení každého disku, logický disk se skládá ze dvou fyzických disků
  - Každý zápis musí být proveden na obou discích
  - Pokud jeden z disků je vadný, data jsou stále k dispozici na druhém.

Zvýšení výkonnosti pomocí paralelizace

- Hlavní cíle paralelizace diskových systémů:
  - Vyvažování zátěže pro zvýšení propustnosti
  - Paralelizace požadavků na velké objemy pro snížení času odezvy
- Zlepšení přenosové rychlosti pomocí dělení dat na více disků
- **Dělení na bitové úrovni** (bit-level striping) – rozděl každý byte na bity, které se uloží na různé disky
  - v poli s osmi disky jde každý bit na zvláštní disk
  - každý přístup může číst data 8x rychleji než disk jeden
  - ale přístupová doba je stejná jako v případě jednoho disku (všechny disky jsou využity pro jeden přístup)
- **Dělení na blokové úrovni** (block-level striping) – rozděl data na bloky a každý blok jde na disk *(i mod n)+1*
  - Při čtení pouze jednoho bloku je zaměstnán pouze jeden disk

Úrovně RAIDu

- Schémata poskytující redundanci při nízkých nákladech pomocí dělení dat na disky spolu s paritními bity.
- Různé úrovně RAIDu mají různé náklady, výkon a spolehlivost
- **Úroveň 0:** dělení na úrovni bloků; žádná redundance.
  - Používané pro vysoké rychlosti čtení/zápis, ztráta dat není kritická
- **Úroveň 1:** zrcadlené disky, rychlé čtení, zápis rychlý jako při použití jednoho disku, redundance - vyšší spolehlivost, nízká kapacita
- **Úroveň 2:** použití paritní informace pro zvýšení spolehlivosti, která je schopna opravit chybu v jednom bitu => pro každý bit jeden disk, navíc paritní disky

- **Úroveň 3:** bitově prokládaná parita (bit-interleaved parity), dělení na úrovni bitů, ale jediný bit stačí pro detekci i opravu.
  - při zápisu je paritní bit vypočítán a uložen na zvláštní disk
  - rychlejší přenosová rychlost než má jeden disk, ale nelze vyřizovat několik požadavků současně (všechny disky se podílejí na čtení)
  - nahrazuje druhou úroveň, protože používá pouze jeden paritní disk
- **Úroveň 4:** blokově prokládaná parita (block-interleaved parity), dělení na úrovni bloků, paritní blok je opět na zvláštním disku.
  - Vyšší propustnost požadavků, jedno čtení nepoužívá všechny disk
  - rychlejší přenosová rychlost než má jeden disk
  - paritní disk je slabé místo, protože každý zápis znamená zápis na paritní disk
- **Úroveň 5:** blokově prokládaná parity rozložená po discích
  - v případě pěti disků, je paritní blok uložen na disk *(i mod n)+1* a data na ostatní disky
  - lze vyřizovat více požadavků současně, i zápisů
  - nahrazuje čtvrtou úroveň
- **Úroveň 6:** stejné jako pátá úroveň, ale ukládá další informaci pro možnost kompletní obnovy při výpadku více disků (ne jenom jednoho). Není moc rozšířené.

Přístup k úložnému prostoru

- Databázový soubor je rozdělený na bloky pevné délky, blok je základní jednotkou přenosu i alokace místa.
- Databázový systém se snaží minimalizovat počet přenášených bloků mezi diskem a hlavní pamětí. Toho lze dosáhnout pomocí uchovávání co možná největšího počtu bloků v hlavní paměti.
- **Vyrovnávací paměť (mezipaměť, buffer)** – část hlavní paměti použitelné pro ukládání kopií diskových bloků
- **Správce mezipaměti (buffer manager)** – podsystém odpovědný pro alokaci vyrovnávací paměti v hlavní paměti
- Program, který vyžaduje nějaký blok z disku, zavolá správce mezipaměti:
  - Pokud je blok v hlavní paměti, správce mu vrátí jeho adresu
  - Pokud blok není v paměti, správce alokuje volné místo, pokud není k dispozici nahradí některý již načtený blok
  - Nahrazovaný (vyhazovaný) blok je uložen na disk, pokud byl změněn.
  - Když je volné místo, správce přečte blok z disku a uloží ho do mezipaměti a aplikaci vrátí jeho adresu.

Nahrazování bufferů

- Většina operačních systému nahrazuje blok, který byl nejdéle nepoužitý (LRU – least recently used)
- LRU může být nevhodná strategie, pokud jsou data opakovaně procházena
- Optimalizátor dotazů používá různé kombinované strategie pro lepší správu mezipaměti.
- **Přišpendlený blok** (pinned block) – daný blok nelze vyhodit z mezipaměti
- **Ihned zahoď** (toss-immediately) – ihned po ukončení zpracování bloku je blok z mezipaměti uvolněn.

- MRU (most recently used) – blok naposledy použitý je uvolněn, během zpracování bloku je blok označen jako přišpendlený a nemůže být uvolněn, po ukončení zpracování se stává naposledy použitím blokem.
- Správce mezipaměti může používat různé statistické informace, např. nějaká relace je často používána => ponechej ji v paměti

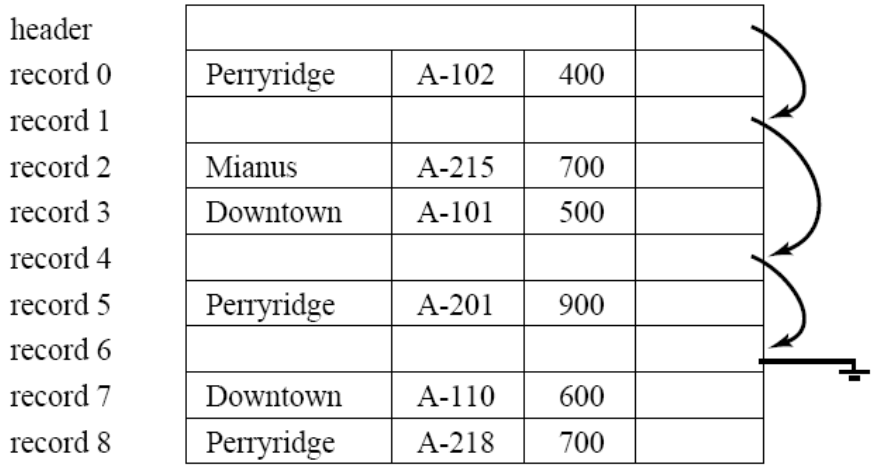
Souborové organizace

- Databáze je uložena v kolekci souborů. Každý soubor je tvořen posloupností záznamů (records). Záznam se skládá z jednotlivých atributů (polí).
- Jeden přístup:
  - Délka záznamu je pevná
  - Každý soubor má záznamy pouze stejného typu
  - Jeden soubor pro jednu relaci

  Tento případ je nejjednodušší na implementaci.

Záznamy s pevnou délkou

- Jednoduchý přístup:
  - Záznam *i* je uložen na pozici *l*(i-1)* v souboru, kde *l* je délka záznamu
  - Přístup k záznamu je jednoduchý, ale záznam může přesahovat bloky
- Mazání záznamu *i* – možnosti:
  - Všechny následující záznamy (*i+1,…,n*) o jeden záznam níže
  - Přesuň poslední záznam *n* na místo *i-tého* záznamu
  - Udržuj si seznam prázdných záznamů
    * V hlavičce souboru si ulož číslo prvního smazaného záznamu
    * V prvním smazaném záznamu si ulož číslo dalšího smazaného záznamu
    * Tato čísla mohou být chápána jako ukazatele na další volnou paměť

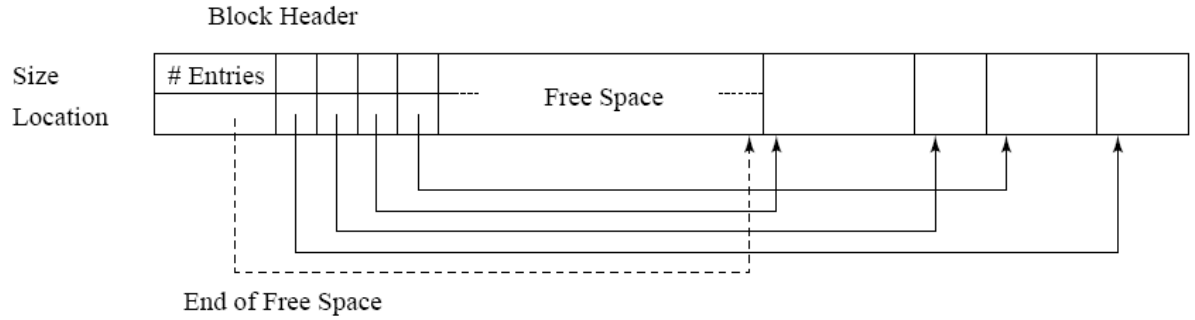


    * Prostorově nenáročné řešení, prázdné atributy lze využít pro uložení ukazatele

Záznamy s proměnnou délkou

- Záznamy s proměnnou délkou vznikají v DB systémech několika způsoby:
  - Ukládání více různých typů záznamů v jednom souboru
  - Záznamy obsahující atributy s proměnnou délkou

- Řešení pomocí řetězcové reprezentace:
  - Na konec záznamu je připojen zvláštní znak pro ukončení záznamu ⊥
  - Problémy s mazáním záznamů a jejich zvětšováním
- Dělený soubor (slotted page structure):



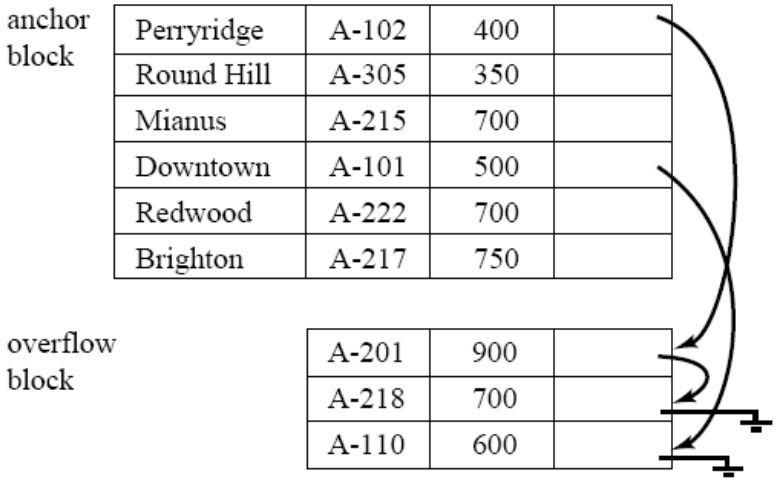
  - Hlavička obsahuje počet záznamů, ukazatel na konec volného místa a dvojici (ukazatel, délka) pro každý záznam
  - Při mazání jsou ostatní záznamy přesunuty tak, aby vzniklo souvislé volné místo, a hlavička je aktualizována
- Řešení s pevnou délkou pomocí vyhrazeného (rezervovaného) místa:
  - Lze použít pokud známe maximální délku záznamu
  - Každý záznam je pevně dlouhý (stejně jako maximálně dlouhý záznam)
  - Nevyužité místo je vyplněno NULL hodnotou nebo koncem záznamu ⊥

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

- Řešení s pevnou délkou pomocí ukazatelů:
  - Maximální délka záznamu není známa
  - Proměnná délka záznamu je vyjádřena pomocí zřetězeného seznamu záznamů pevné délky

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

  - Nevýhodou je plýtvání místem ve všech záznamech seznamu kromě prvního
  - Lze řešit pomocí bloků dvou druhů:
    * Kotvící blok (anchor block) – obsahuje první záznam v řetězci
    * Přetokový blok (overflow block) – obsahuje záznam, které ukládají pouze proměnné atributy.

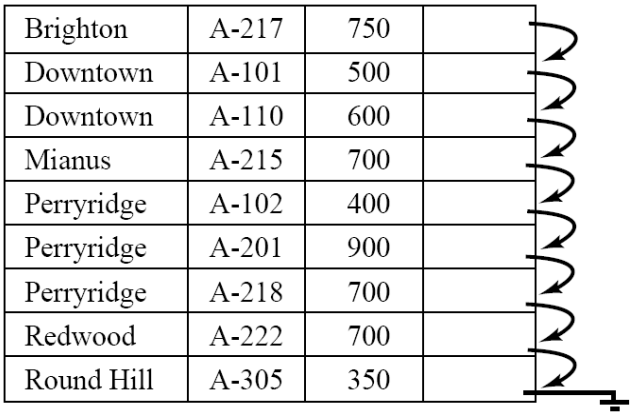


Organizace záznamů v souboru

- **Halda** (heap) – záznam je uložen kdekoli na volné místo v souboru
- **Sekvenční** – ukládáme záznamy za sebou uspořádané podle vyhledávacího atributu
- **Hešování** (hashing) – používá se hešovací funkce pro výpočet čísla bloku, kde má být záznam uložen. Toto číslo je vypočítáno na základě hodnot vybraných atributů.
- **Shlukování** – záznamy různých relací mohou být uloženy v jednom souboru a příbuzné záznamy jsou uloženy ve stejném bloku

Sekvenční soubor

- Vhodný pro aplikace, které postupně procházejí celý soubor
- Záznamy jsou obvykle uspořádány podle vyhledávacího klíče (atributu)



- Mazání pomocí řetězení volných záznamů
- Vkládání – nejprve nalézt místo pro místo pro vkládaný záznam
  - Pokud je tam volné místo, ulož
  - Pokud ne, vlož nový záznam do přetokového bloku
  - V obou případech se musí aktualizovat seznam volného místa
- Občas je nutná reorganizace souboru

Shlukování

- Vložení více relací v jednom souboru
- Příklad uložení relací *zákazník* a *účet* do jednoho souboru

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

  - Vhodná organizace pro dotazy zahrnující spojení relací a pro dotazy, které vypisují účty pro jednoho zákazníka
  - Nevhodné pro dotazy zpracovávající pouze relaci *zákazník*
  - Důsledkem této metody je soubor s proměnnou délkou záznamu

Kapitola 11: Indexování a hešování

- Základní představa
- Řazené indexy (ordered indices)
- B+-strom indexový soubor
- B-strom indexový soubor
- Hešování
- Porovnání řazených indexů a hešování
- Definice indexů v SQL

Základní představa

- Indexové mechanizmy se používají pro zrychlení přístupu k požadovaným datům
  - Např. katalog autorů v knihovně
- **Vyhledávací klíč** (search key) – atribut nebo množina atributů používaný pro vyhledávání záznamů v souboru
- **Indexový soubor** se skládá ze záznamů (index entries) ve tvaru

Vyhledávací klíč	ukazatel

- Indexový soubor je typicky mnohem menší než původní soubor
- Dva základní typy indexů:
  - **Řazené indexy** – vyhledávací klíče jsou uspořádané
  - **Hešovací indexy** – vyhledávací klíče jsou rovnoměrně rozprostřeny po adresovacím prostoru hešovací funkce

Metriky pro porovnávání indexů

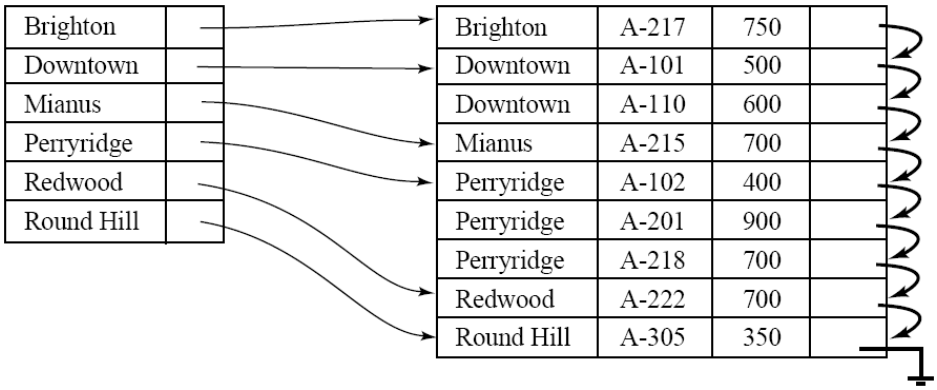
- Typy efektivně (rychle) podporovaných přístupů:
  - Dotazy na přesnou shodu ve vyhledávacím klíči
  - Dotazy na rozsah vyhledávacích klíčů
- Přístupová doba
- Doba pro vložení záznamu
- Doba pro smazání záznamu
- Prostorová režie (kolik místa je potřeba pro index)

Řazené indexy

- Indexové záznamy jsou uloženy uspořádané podle vyhledávacího klíče, např. katalog autorů v knihovně
- **Primární index** – seřazený sekvenční soubor; index, jehož vyhledávací klíč určuje pořadí záznamů v souboru
  - často nazývaný jako **shlukující soubor**
  - vyhledávací klíč primárního indexu nemusí být ve všech případech primárním klíčem
- **Sekundární index** – index, jehož vyhledávací klíč určuje jiné pořadí než v seřazeném sekvenčním souboru; nazývaný **neshlukující index**
- **Index-sekvenční soubor** – seřazený sekvenční soubor s primárním indexem

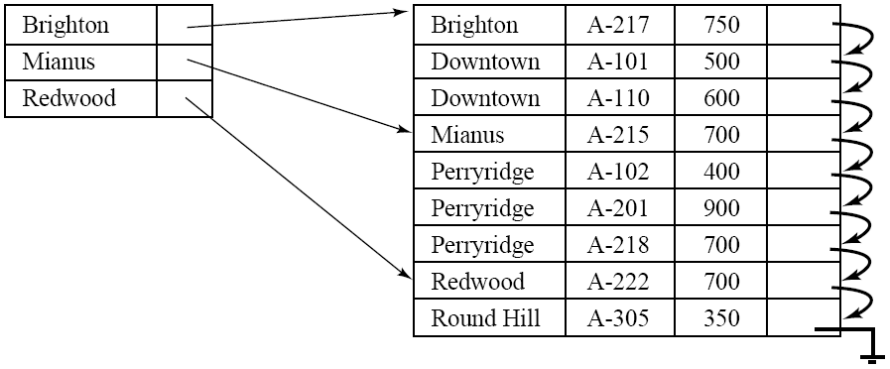
Hustý indexový soubor

- Hustý index – indexový záznam je uložený pouze pro každou hodnotu vyhledávacího klíče (ale stejné hodnoty klíče se v indexu neopakují)



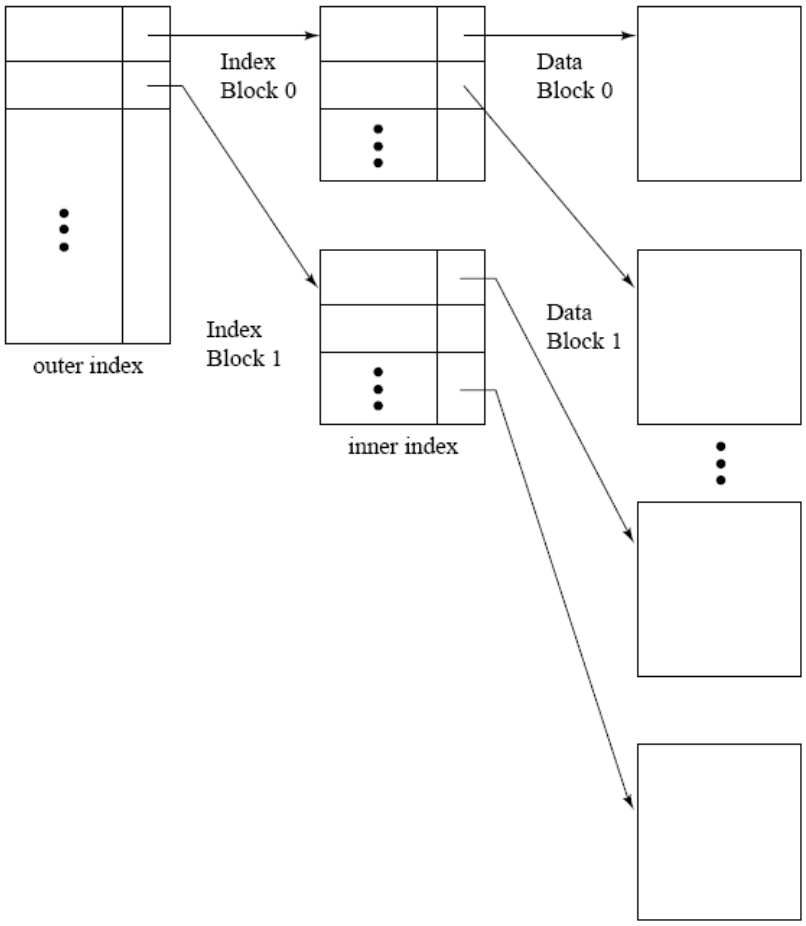
Řídký indexový soubor

- Indexové záznamy jsou pouze pro některé hodnoty vyhledávacího klíče
- Pro nalezení záznamu s vyhledávacím klíčem *K* musíme:
  - Nalézt indexový záznam s největším vyhledávacím klíčem menším než *K*
  - Prohledat sekvenční soubor od tohoto záznamu
- Méně prostoru pro uložení indexu a méně udržujících operací při vkládání a mazání záznamu
- Obecně je ale při vyhledávání pomalejší než hustý index
- Vhodné řešení je řídký indexový soubor pro každý blok v souboru



Víceúrovňový index

- Pokud se primární index nevejde do operační paměti mohou být přístupy pomalé (tím pádem drahé)
- Pro snížení počtu diskových přístupů k indexovému souboru se primární index považuje za sekvenční soubor na disku a vytvoří se pro něj řídký index.
  - Vnější (outer) index je řídký index na primárním indexu
  - Vnitřní (inner) index je primární index na souboru
- Pokud se i vnější index nevejde do paměti, můžeme vytvořit další úroveň
- Indexy na všech úrovních musí být aktualizovány při vkládání nebo mazání



Aktualizace indexu – mazání

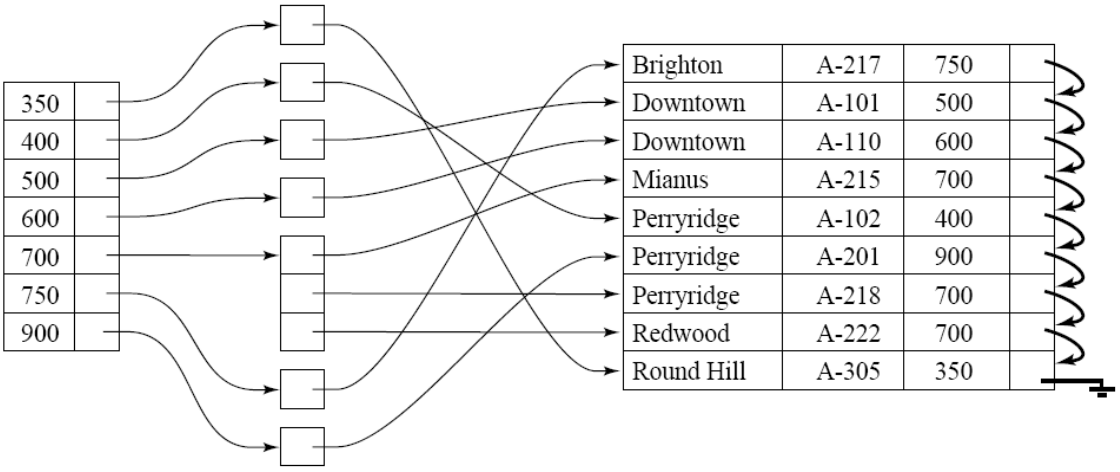
- Pokud je mazaný záznam jediným záznamem s daným vyhledávacím klíčem, vyhledávací klíč je také vymazán z indexu
- Jednoúrovňové mazání:
  - Hustý index – mazání vyhledávacího klíče je podobné jako mazání záznamu
  - Řídký index – pokud existuje indexový záznam pro hodnotu vyhledávacího klíče, je tento záznam nahrazen vyhledávacím klíčem, který bezprostředně následuje mazaný klíč v pořadí řazení podle vyhledávacího klíče. Pokud takový klíč již v indexu existuje, je hodnota pro mazaný záznam v indexu smazána.

Aktualizace indexu – vkládání

- Jednoúrovňové mazání:
  - Proveď vyhledávání s hodnotou vyhledávacího klíče vkládaného záznamu
  - Hustý index – pokud není klíč v indexu nalezen, je tam vložen
  - Řídký index – pokud je v indexu uložen klíč pro každý blok souboru, žádné změny není nutno provádět tehdy, když není vytvořen nový blok. Pokud je nový blok vytvořen je vložen klíč, který ho bude zastupovat.

Sekundární index

- Často jsou kladeny vyhledávací dotazy, které hledají všechny záznamy, které na určitém atributu (který není součástí primárního indexu) nabývají požadované hodnoty.
  - Např.: v relaci *účet* jsou záznamy uloženy v sekvenčním souboru a upořádány podle čísla účtu, ale my vyhledáváme všechny účty vedené danou pobočkou.
  - Např.: stejný dotaz, ale navíc chceme vypsat pouze účty, které mají určitý zůstatek nebo mající zůstatek v nějakém intervalu.
- Můžeme definovat sekundární index se záznamy pro každou hodnotu vyhledávacího klíče; takový indexový záznam určuje blok, který obsahuje ukazatele na všechny záznamy mají danou hodnotu vyhledávacího klíče.
- Sekundární index na relaci *účet* s atributem *zůstatek*



Primární a sekundární indexy

- Sekundární indexy musí být husté.
- Indexy obecně nabízejí podstatné výhody při vyhledávání záznamů.
- Pokud je soubor změněn, každý index na tomto souboru musí být změněn také. Aktualizace indexů ale přináší další režii.
- Sekvenční prohledávání s použitím primárního indexu je výkonné, ale sekvenční hledání s použitím sekundárního indexu je drahé (přístup ke každému záznamu může vést ke čtení nového bloku, protože pořadí záznamů v sekvenčním souboru se může významně lišit od pořadí podle sekundárního indexu).

B^+-stromy

B^+-stromy jsou jinou možností k index-sekvenčním souborům

- Nevýhody index-sekvenčních souborů: výkonnost klesá s rostoucím souborem, protože mnoho bloků přeteče a je nutné provádět opakované reorganizace.

- Výhoda indexů s B^+-stromy: při vkládání/mazání se provádí automatická reorganizace pouze s malými, lokálními změnami. Reorganizace celého souboru nejsou nutné pro opětovné zvýšení výkonnosti.
- Nevýhoda B^+-stromů: režie při vkládání/mazání záznamů, zvýšené prostorové nároky.
- Ovšem výhody B^+-stromů převažují nevýhody a jsou nejčastěji používanou indexovou organizací v databázích.

B^+-strom je strom s jedním kořenem splňující následující podmínky:

- Všechny cesty od kořene k listům mají stejnou délku
- Každý uzel, který není kořenem nebo listem, má potomků $\lceil n/2 \rceil$ až n
- Listový uzel má $\lceil (n-1)/2 \rceil$ až $n-1$ hodnot (klíčů)
- Zvláštní případy:
  - Pokud kořen stromu není zároveň list, potom má nejméně 2 potomky.
  - Pokud je kořen současně listem (strom má 1 uzel), počet uložených hodnot je mezi 0 a $(n-1)$
- Typický uzel

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

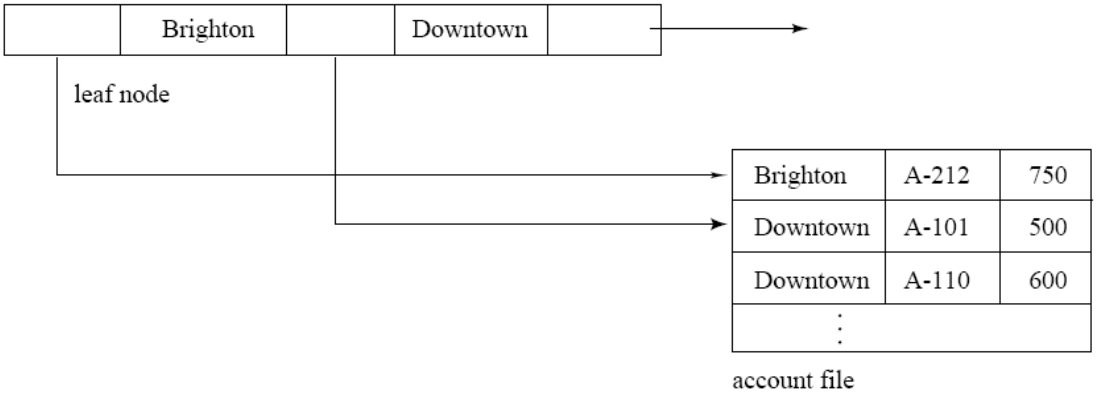
  - K_i jsou hodnoty vyhledávacího klíče
  - P_i jsou ukazatele na potomky (platí pro vnitřní uzly) nebo ukazatele na záznamy nebo bloky záznamů (pro listové uzly)
- Vyhledávací klíče jsou uspořádány

$$K_1 < K_2 < \ldots < K_{n-1}$$

Listy v B^+-stromech

Vlastnosti každého listu:

- Pro každé $i=1,2,\ldots,n-1$ ukazatel P_i odkazuje buď na záznam s klíčem K_i nebo na blok ukazatelů na záznamy, kde každý záznam má klíč K_i. Pokud index není primární, potom jsou uloženy odkazy na bloky.
- Pokud L_i, L_j jsou listy a $i < j$, potom klíče v L_i jsou menší než klíče v L_j.
- P_n ukazuje na další list



Vnitřní uzly v B^+-stromech

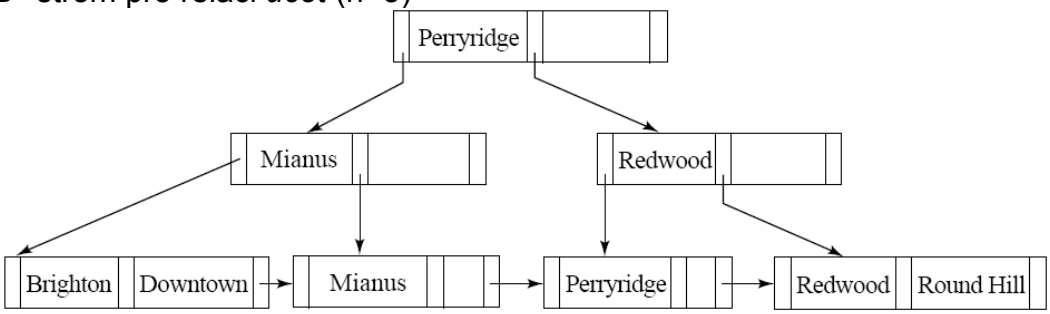
- Nelistové uzly vytvářejí víceúrovňový řídký index na listových uzlech. Pro každý nelistový uzel s m ukazateli platí:

- Všechny vyhledávací klíče v podstromu, na který P_1 ukazuje, jsou menší než K_1
- Pro $2 \le i \le n-1$, všechny vyhledávací klíče v podstromu, na který ukazuje P_i, mají hodnoty větší nebo rovny než K_{i-1} a menší než K_i
- Všechny vyhledávací klíče v podstromu, na který ukazuje P_m, jsou větší nebo rovny než K_{m-1}

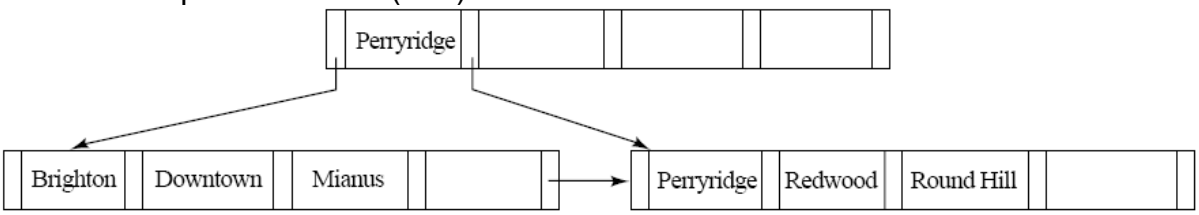
P_1	K_1	P_2	…	P_{m-1}	K_{m-1}	P_m

Příklad B$^+$-stromu

- B$^+$-strom pro relaci *účet* (n=3)



- B$^+$-strom pro relaci *účet* (n=5)



- Listy musí mít 2 až 4 hodnoty (pro n=5)
- Nelistový uzel kromě kořene musí mít 3 až 5 potomků (pro n=5)
- Kořen musí mít alespoň 2 potomky

B$^+$-strom - pozorování

- Protože spojení vnitřních uzlů je pomocí ukazatelů, nemůžeme předpokládat, že bloky „logicky“ blízké v B$^+$-stromu jsou také blízké „fyzicky“
- Nelistové úrovně stromu tvoří hierarchii řídkých indexů
- B$^+$-strom obsahuje relativně malý počet úrovní (logaritmický k velikosti souboru), tedy vyhledávání jsou prováděny efektivně.
- Vkládání a mazání v souboru lze řešit také efektivně, protože index je aktualizovaný v logaritmickém čase.

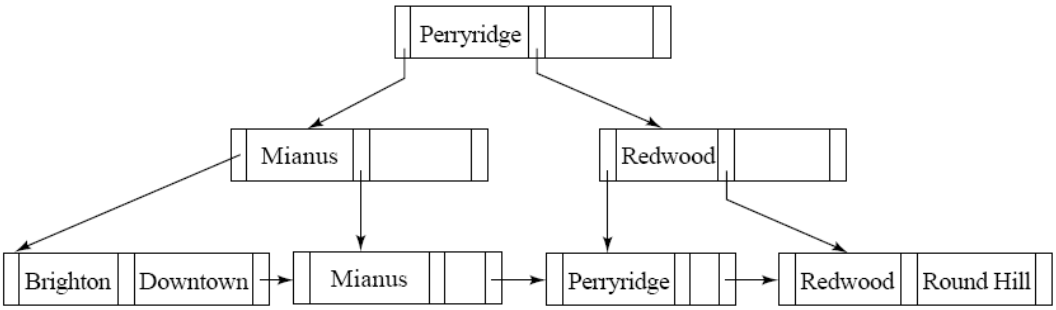
B$^+$-strom - dotazy

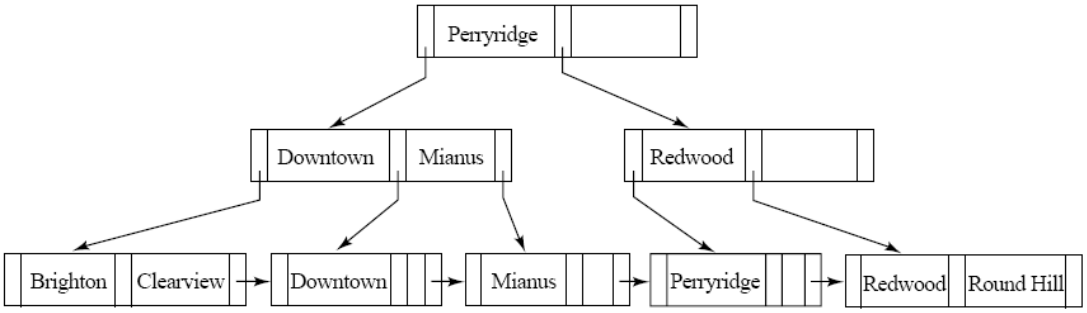
- Najdi všechny záznamy s vyhledávacím klíčem *k*
  - Začni v kořenu stromu
    * V uzlu najdi klíč K_i, který je nejmenší větší než *k*
    * Pokud takové K_i existuje, jdi do potomka P_i
    * Jinak $k \ge K_{m-1}$, jdi do potomka P_m
  - Pokud není aktuální uzel list, pokračuj předchozím bodem

  - Pokud jsme v listu, proveď následující: když $K_i=k$, následuj ukazatel P_i a přečti uložený záznam nebo blok; pokud není žádné K_i rovné k, žádný takový záznam neexistuje.
- Během zpracování dotazu je strom procházen od kořene k nějakému listu
- Pokud je v souboru K vyhledávacích klíčů, potom cesta není delší než $\lceil log_{\lceil n/2 \rceil}(K) \rceil$
- Uzel má obvykle stejnou velikost jako blok na disku, tj. 4 KB, a n bývá kolem 100 (což znamená 40 B na jednu položku indexu)
- Při 1 milionu vyhledávacích klíčů a $n=100$, nejvýše $log_{50}(1000000)=4$ uzlů je přistoupeno během vyhledávání
- Pro porovnání s vyváženým binárním stromem máme asi 20 uzlů přistoupených během hledání ($log_2(1000000) \sim 20$)
  - Tento rozdíl je významný, protože každý přistoupený uzel vyžaduje čtení jednoho bloku z disku, což odpovídá asi 30 milisekundám!

Aktualizace B^+-stromu - vkládání

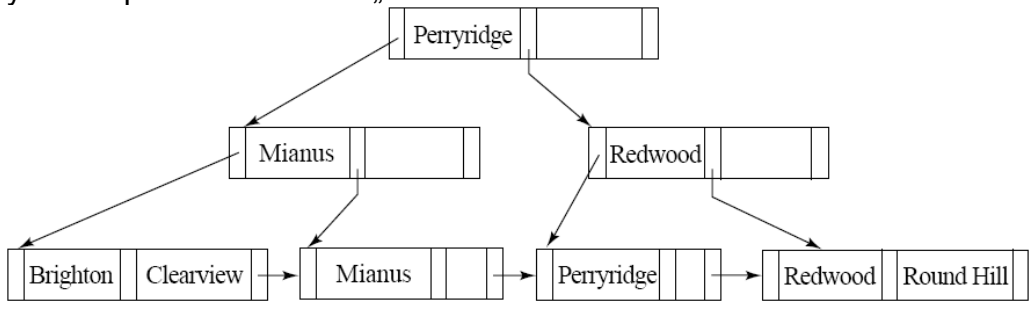
- Najdi list, do kterého má být nový klíč vložen
- Pokud klíč již existuje, nový záznam je přidán do souboru a je vytvořen ukazatel na blok souboru, pokud je třeba.
- Pokud není nový klíč v indexu, opět ulož nový záznam do souboru a potom:
  - Pokud je v listu dostatek místa, vlož nový klíč a ukazatel na záznam na správnou pozici
  - Pokud v listu není volné místo, rozděl uzel a ulož nový klíč a ukazatel na záznam
- Rozdělení uzlu (node split)
  - Vezmi všech n dvojic (klíč, ukazatel na záznam) včetně nového klíče a setřiď je. Prvních $\lceil n/2 \rceil$ dvojic ulož do původního uzlu a zbývající ulož do nového uzlu.
  - Předpokládejme, že nový uzel je p a nechť k je nejmenší klíč v p. Vlož dvojici (k,p) do rodičovského uzlu právě rozděleného uzlu. Pokud je rodičovský uzel přeplněný, rozděl ho a propaguj rozdělení o úroveň výše.
- Rozdělování uzlů postupuje postupně do vyšší a vyšší úrovně stromu, dokud není nalezený nepřeplněný uzel. V nejhorším případě je rozdělený kořen stromu a je vytvořen nový, výsledkem celý strom vyroste o jednu úroveň.



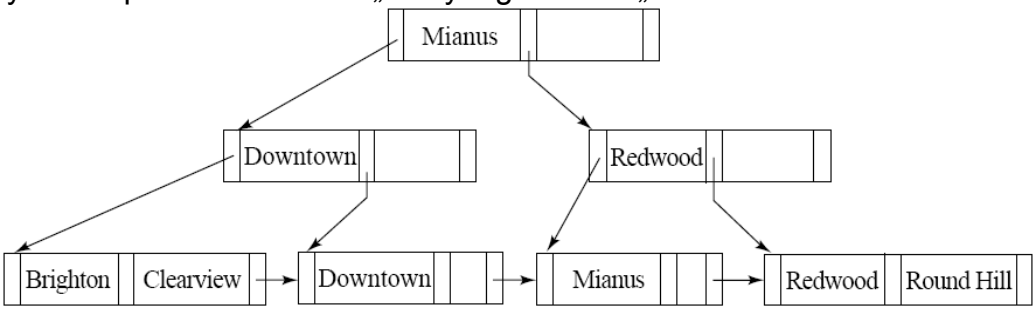


Aktualizace B$^+$-stromu - mazání

- Najdi záznam, který má být smazán, a vymaž ho ze souboru.
- Zruš dvojici (klíč, ukazatel na záznam) z listu, pokud ukazatel není na blok nebo je blok již prázdný.
- Pokud má uzel příliš málo položek kvůli mazání záznamů a pokud se zbývající položky vejdou do sousedního uzlu, potom
  - Vlož všechny klíče z obou uzlů do jednoho (do uzlu vlevo) a smaž prázdný uzel.
  - Z rodičovského uzlu smaž dvojici (K_{i-1},P_i), pokud P_i je ukazatel na právě smazaný uzel. Když je třeba aplikuj rekurzivně na rodiče.
- Pokud má uzel příliš málo položek kvůli mazání záznamů a pokud se zbývající položky nevejdou do sousedního uzlu, potom
  - Rozděl položky z obou uzlů tak, že oba uzly mají alespoň minimální počet položek.
  - Aktualizuj odpovídající položky v rodičovském uzlu
- Mazání uzlu se může kaskádovitě přesunovat do vyšších úrovní, dokud není nalezen uzel alespoň $\lceil n/2 \rceil$dvojicemi. Pokud po mazaní má kořen pouze jednu položku, je zrušen a jeho jediný potomek se stane novým kořenem.
- Výsledek po smazání klíče „Downtown“ z indexu na relaci *účet*



- Smazání klíče „Downtown“ vede ke smazání listového uzlu, které ale nevede k nedostatečnému počtu klíčů v rodičovském uzlu. Tedy kaskádové mazání je ukončeno.
- Výsledek po smazání klíče „Perryridge“ místo „Downtown“



- Po smazání uzlu s klíčem „Perryridge“ dojde ke snížení počtu klíčů v rodičovském uzlu pod minimální hodnotu, ale sousední uzel rodiče (sibling) je již plný a nemůže přijmout ukazatel na další list, proto dojde k rozdělení ukazatelů na listy mezi dva uzly. Všimněme si, že se změnil klíč v kořeni stromu.

Souborová organizace B$^+$-strom

- Problém s degradací indexového souboru (a následné reorganizace) jsou vyřešeny s použitím B$^+$-stromu. Degradace souboru je řešena pomocí souborové organizace B$^+$-strom.
- Listové uzly B$^+$-stromu obsahují přímo celé záznamy místo ukazatelů na ně.
- Protože záznamy jsou větší než samotné ukazatele, sníží se kapacita listu.
- Podmínka na alespoň poloviční naplnění listu je stále platná.
- Vkládání a mazání probíhá totožně jako v indexu B$^+$-strom.
- Dobré využití místa je zde důležitější kvůli větší prostorové náročnosti záznamů. Pro zvýšení využívání volného místa se používá metoda přeskupování záznamů mezi více sousedními uzly než při použití indexu B$^+$-strom, to se děje při rozdělování a slučování uzlů.

Index B-strom

- Podobný s B$^+$-stromy, ale B-stromy neopakují klíče ve vnitřních uzlech, každý klíč se ve stromu vyskytuje nejvýše jednou. To snižuje prostorové nároky indexu.
- Vyhledávací klíče použité ve vnitřních uzle pro navigaci ve stromu se již v listech neopakují, proto je struktura vnitřního uzlu rozšířena o ukazatel na záznam.
- List v B-stromu:

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- Vnitřní uzel v B-stromu: (ukazatelé B_i ukazují buď na záznamy nebo na bloky)

P_1	B_1	K_1	P_2	B_2	K_2	…	P_{m-1}	B_{m-1}	K_{m-1}	P_m

- Výhody indexu B-strom:
  - Má méně uzlů než odpovídající B$^+$-strom
  - Někdy je možné nalézt klíč dříve něž v listu
- Nevýhody B-stromů:
  - Pouze velmi malý počet klíčů je nalezený dříve než v listu

  - Nelistové uzly jsou větší, tedy počet potomků je nižší (protože uzel je uložený v jednom bloku). Důsledkem je vyšší hloubka stromu než v případě B^+-stromů.
  - Vkládání a mazání je složitější než u B^+-stromů
  - Implementace je také obtížnější
- Typicky výhody použití B-stromu nepřevažují nad jeho nevýhodami.

Statické hešování

- **Kyblík** (bucket) je základní úložnou jednotkou obsahující jeden nebo více záznamů (kyblík je obvykle tvořen jedním diskovým blokem). V **hešovací souborové organizaci** získáme kyblík, kde je hledaný záznam uložen, přímo z jeho vyhledávacího klíče pomocí **hešovací funkce**.
- Hešovací funkce *h* je funkce převádějící množinu vyhledávacích klíčů *K* na množinu adres jednotlivých kyblíků *B*.
- Hešovací funkce se používá jak při vyhledávání záznamu, tak při vkládání a mazání.
- Záznamy s rozdílnými klíči mohou být uloženy ve stejném kyblíku, tedy celý obsah kyblíku musí být sekvenčně prohledán.

Hešovací funkce

- Nejhorší hešovací funkce mapuje všechny vyhledávací klíče do jednoho kyblíku, což vede k vyhledávacímu času závislému na počtu klíčů v souboru.
- Ideální hešovací funkce je *rovnoměrná*, tj. každý kyblík obsahuje stejné množství klíčů z množiny všech možných hodnot vyhledávacího klíče.
- Ideální hešovací funkce je *náhodná* a každý kyblík má přeřazen stejný počet záznamů bez ohledu na aktuální rozložení vyhledávacích klíčů v souboru.
- Typicky hešovací funkce pracují s interní binární podobou vyhledávacího klíče. Např. pro řetězcový klíč binární zápis všech znaků v řetězci může být sečten a adresa kyblíku je výsledkem operace modulo celkový počet kyblíků.

Příklad hešovací souborové organizace

bucket 0

bucket 1

bucket 2

bucket 3

Brighton	A-217	750
Round Hill	A-305	350

bucket 4

Redwood	A-222	700

bucket 5

Perryridge	A-102	400
Perryridge	A-201	900
Perryridge	A-218	700

bucket 6

bucket 7

Mianus	A-215	700

bucket 8

Downtown	A-101	500
Downtown	A-110	600

bucket 9

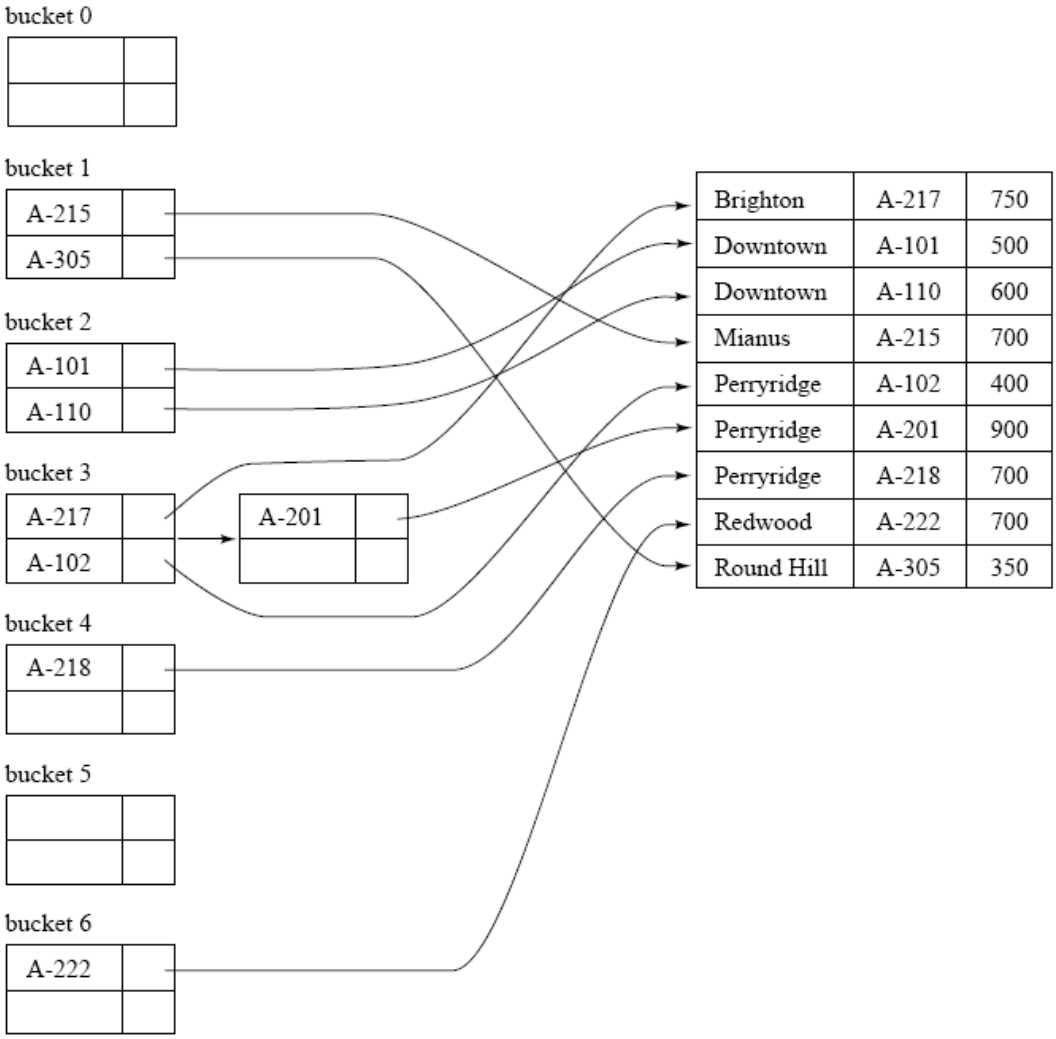
- 10 kyblíků
- binární zápis *i-tého* znaku je chápán jako integer *i* (celé číslo)
- hešovací funkce vrací součet všech integerů modulo 10

Řešení přetečení kyblíku

- K přetečení kyblíku dochází
  - Nedostatečný počet kyblíků
  - Asymetrické rozložení klíčů, které může mít dva důvody:
    * Více záznamů má stejný vyhledávací klíč
    * Zvolená hešovací funkce není rovnoměrná
- Ačkoli pravděpodobnost přetečení kyblíku může být snížena, nelze se jí zbavit úplně a je řešena pomocí **přetokových kyblíků.**
- **Řetězení přetoků** – přetokové kyblíky některého kyblíku jsou řetězeny do jednoho seznamu
- Výše popsané schéma je nazýváno jako **uzavřené hešováni.** Jinou možností je **otevřené hešování**, které není vhodné pro databázové aplikace.

Hešovací indexy

- Hešování může být použito nejenom jako souborová organizace, ale i jako indexová struktura. **Hešovací index** organizuje vyhledávací klíče spolu s ukazateli na záznamy v hešovací souborové organizaci.
- Hešovací indexy jsou vždy používány jako sekundární indexy – pokud i soubor používá hešování, není potřeba vytvářet zvláštní primární index nad stejným vyhledávacím klíčem. Často se používá výraz hešovací index (nebo jen hešování) pro obě techniky sekundární hešovací index i hešovací soubor.



Nedostatky statického hešování

- Ve statickém hešování mapuje hešovací funkce vyhledávací klíče na pevnou množinu adres kyblíků.
  - Ovšem databáze se v čase zvětšují. Pokud je počáteční velikost adresového prostoru příliš malá, výkonnost této techniky je snížena kvůli příliš častému přetečení.
  - Pokud je zvolena nějaká předpokládaná velikost souboru a je vytvořen odpovídající počet kyblíků, dochází ze začátku k významnému plýtvání místem.
  - Je-li velikost databáze snížena, opět se plýtvá místem.

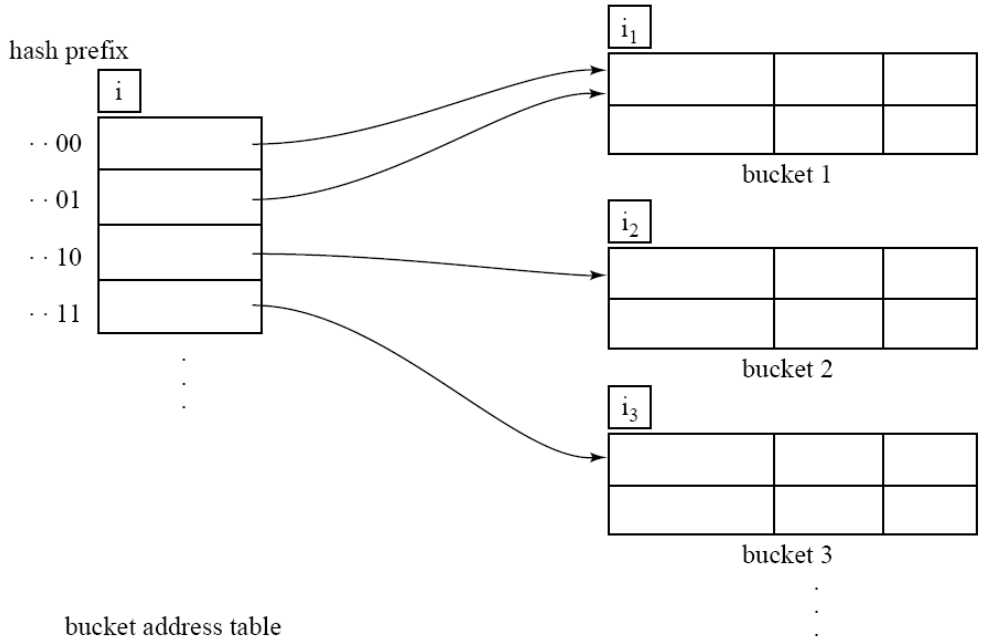
  - Jednou z možností je pravidelná reorganizace souboru s novou hešovací funkcí, ale to je velmi nákladné.
- Těmto problémům se můžeme vyhnout použitím techniky, která umožňuje dynamicky alokovat a rušit kyblíky.

Dynamické hešování

- Vhodné pro databáze, které mění svoji velikost
- Umožňuje modifikovat hešovací funkci
- **Rozšiřitelné hešování** – jednou z forem dynamického hešování
  - Hešovací funkce generuje velká čísla, typicky 32-bitová.
  - Vždy se používá pouze prefix pro získání adresy kyblíku. Délka prefixu je *i* bitů, $0 \leq i \leq 32$
  - Na počátku je $i = 0$
  - Hodnota *i* se zvyšuje nebo snižuje podle velikosti souboru
  - Aktuální počet kyblíků je menší než 2^i, to se také dynamicky mění kvůli slívání a rozdělování kyblíků.

Obecná struktura rozšiřitelného hešování

- Příklad struktury s $i_2 = i_3 = i$, ale $i_1 = i - 1$



Použití rozšiřitelného hešování

- Více adres kyblíků může ukazovat na stejný kyblík. Každý kyblík *j* ukládá hodnotu i_j. Všechny položky ukazující na stejný kyblík musí mít stejnou hodnotu na prvních i_j bitech.
- Pro nalezení kyblíku obsahující klíč K_j:
  1. vypočítej $h(K_j) = X$
  2. použij prvních *i* bitů hodnoty *X* pro vyhledání v tabulce adres kyblíků a následuj ukazatel do příslušného kyblíku

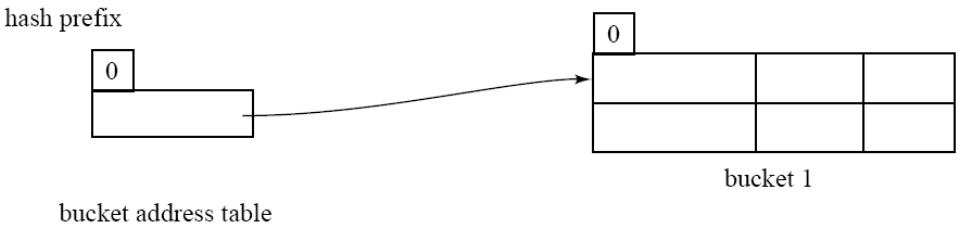
- Při vkládání záznamu s hodnotou klíče K_l proveď kroky uvedené výše a najdi kyblík *j*. Pokud je v kyblíku místo vlož nový záznam. Jinak musí být kyblík rozdělen a vkládání se opakuje.

Aktualizace rozšiřitelného hešování

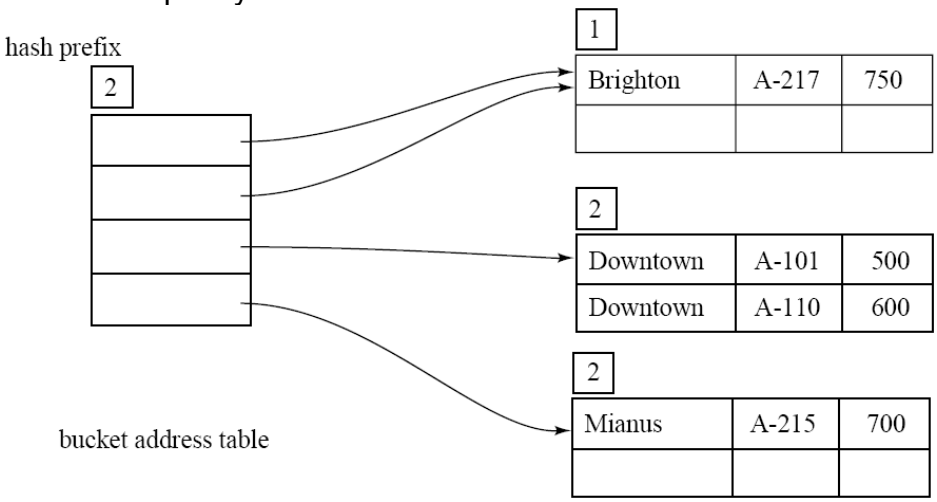
Pro rozdělení kyblíku *j* při vkládání nového záznamu s hodnotou vyhledávacího klíče K_l:

- Pokud $i > i_j$ (více než jeden ukazatel na kyblík *j*)
  - Vytvoř nový kyblík *z* a nastav i_j a i_z na hodnotu $i_j + 1$ (zde i_j je původní hodnota)
  - Změň polovinu ukazatelů ukazujících na *j*, aby ukazovaly na *z*.
  - Smaž a znovu vlož každý záznam z kyblíku *j*
  - Znovu vypočítej adresu kyblíku pro K_l a vlož záznam do kyblíku (další dělení je možné, pokud je kyblík stále plný)
- Pokud $i = i_j$ (pouze jeden ukazatel na kyblík *j*)
  - Zvyš hodnotu *i* a zdvojnásob velikost tabulky adres kyblíků
  - Nahraď každou položku v této tabulce dvěma novými položkami, které ukazují na stejný kyblík
  - Znovu vypočítej adresu kyblíku pro K_l. Nyní je $i > i_j$, tak použij předchozí postup.
- Při vkládání hodnoty může dojít k vícenásobnému dělení kyblíku, pokud se dělí stále stejný a počet dělení překročí nastavenou mez, vytvoř přetokový kyblík místo dalšího dělení.
- Pro smazání hodnoty najdi příslušný kyblík a smaž hodnotu. Samotný kyblík může být uvolněn až je úplně prázdný (ovšem s odpovídající změnou v tabulce adres kyblíků). Slévání kyblíků do jednoho a snižování tabulky adres kyblíků jsou možné.
- Příklad s klíčem *jméno-pobočky (brach-name)* a velikostí kyblíku 2:

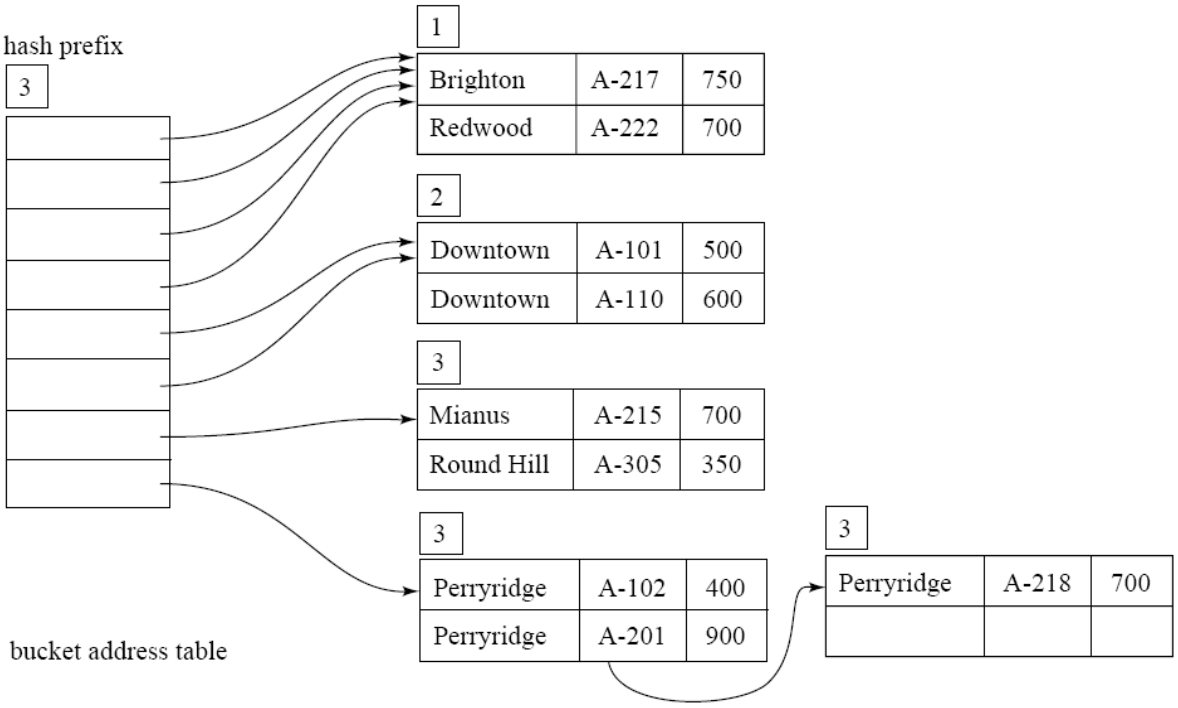
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



- Hešovací index po čtyřech vloženích:



- Výsledek po vložení všech klíčů:



Srovnání řazených indexů a hešování

Zvažované otázky:

- Náklady na pravidelnou reorganizaci
- Četnost vkládání a mazání
- Je žádoucí optimalizovat průměrnou dobu přístup vůči nejhoršímu případu doby přístupu?
- Očekávané typy dotazů:
  - Hešování je obecně lepší při vyhledávání záznamů mající danou hodnotu klíče.
  - Pokud jsou rozsahové dotazy běžné, je lepší použít řazené indexy

Definice indexů v SQL

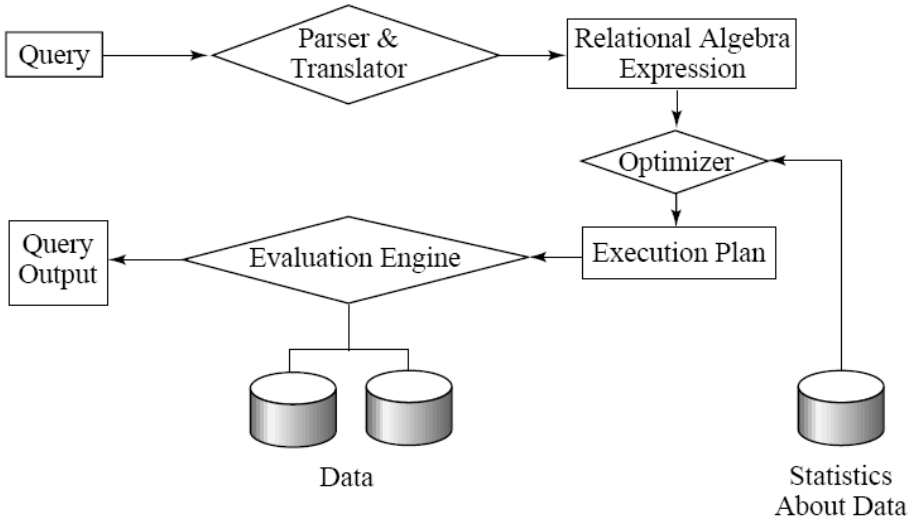
- Vytvoření indexu
  create index <jméno_indexu> **on** <jméno_relace> (<seznam_atributů>)
  Např. **create index** *b-index* **on** *pobočka (jméno-pobočky)*
- Použitím **create unique index** nepřímo zajistíme podmínku, že vyhledávací klíč je kandidátní klíč.
- Pro zrušení indexu
  drop index <jméno_indexu>

Kapitola 12: Zpracování dotazů

- Přehled
- Informace pro odhad nákladů
- Míry pro náklady dotazu
- Operace výběru
- Řazení
- Operace spojení
- Vyhodnocování výrazů
- Transformace relačních výrazů
- Výběr plánu pro vyhodnocení

Základní kroky ve zpracování dotazů

1. analýza dotazu (parsing) a překlad
2. optimalizace
3. vyhodnocení



Analýza dotazu (parsing) a překlad:

- Přelož dotaz do interní reprezentace, která je následně přeložena do relační algebry
- Analyzátor ověří správnou syntaxi a ověří existenci relací

Vyhodnocení:

- **Stroj pro vyhodnocení dotazu** (query-execution/evaluation engine) bere na vstupu plán pro vyhodnocení dotazu, spustí plán a vrátí výsledky dotazu.

Optimalizace – nalezení nejlevnějšího plánu pro vykonání dotazu:

- Mějme výraz v relační algebře, tento výraz může mít několik ekvivalentních (produkují stejný výsledek) výrazů
  Např. $\sigma_{\text{zůstatek<2500}}(\Pi_{\text{zůstatek}}(\textit{účet}))$ je ekvivalentní výrazu:
  $\Pi_{\text{zůstatek}}(\sigma_{\text{zůstatek<2500}}(\textit{účet}))$
- Jakýkoli výraz v relační algebře může být vyhodnocen mnoha způsoby. Komentovaný výraz určující detailně postup vyhodnocení se nazývá *plán pro vyhodnocení.*

Např. má se použít index na atributu *zůstatek* k nalezení účtů se zůstatkem < 2500, nebo se má použít sekvenční průchod celého souboru a vynechat všechny účty s zůstatkem ≥ 2500?

- Mezi všemi možnými výrazy se snažíme najít ten, který má nejlevnější plán pro vyhodnocení. Odhad ceny plánu pro vyhodnocení je založený na *statistických informacích* v databázovém katalogu.

Informace v DB katalogu pro odhad nákladů

- n_r: počet n-tic relaci *r*
- b_r: počet bloků obsahující n-tice z relace *r*
- s_r: velikost n-tice z *r* v bajtech
- f_r: blokovací faktor relace *r* – tj. počet n-tic z *r*, které se vejdou do jednoho bloku
- *V(A,r)*: počet jedinečných hodnot, které se vyskytují v relaci *r* v atributu *A*, což je stejné jako velikost $\Pi_A(r)$
- *SC(A,r)*: (selection cardinality) průměrný počet záznamů, které mají stejnou hodnotu na atributu *A*
- Pokud jsou n-tice relace *r* uloženy společně v jednom souboru, potom:

$$b_r = \lceil n_r / f_r \rceil$$

- f_i: průměrný počet potomků vnitřního uzlu ve stromovém indexu *i* jako např. B^+-strom
- HT_i: počet úrovní stromového indexu *i*, tj. hloubka stromu
  - Pro vyvážený strom na atributu *A* relace *r*: $HT_i = \lceil log_{fi}(V(A,r)) \rceil$
  - Pro hešovací index je $HT_i = 1$
- LB_i: počet bloků na nejnižší úrovni indexu *i*, tj. počet bloků v listech stromu

Míry nákladů dotazů

- Mnoho způsobů jak měřit a odhadovat náklady (cenu), např. počet diskových přístupů, CPU čas nebo dokonce komunikační režie v distribuovaných nebo paralelních systémech.
- Přístupy na disk typicky tvoří převládající náklady a také jsou relativně snadno měřitelné. Tudíž, počet přenosů bloků z disku je používán jako míra pro aktuální cenu vyhodnocení. Předpokládá se, že všechny přístupy mají stejnou cenu.
- Náklady algoritmů závisí na velikosti vyrovnávacích pamětí v operační paměti, protože více paměti snižuje náklady na čtení z disku. Tedy velikost paměti by měl být parametr pro odhad ceny dotazu; často se používá nejhorší odhad.
- Odhad nákladů algoritmu *A* je značen jako E_A. Náklady pro zápis na disk nejsou uvažovány.

Operace výběru

- **Sekvenční průchod souborem** – vyhledávací algoritmus, který hledá a vrací záznamy vyhovující podmínce
- Algoritmus **A1** (lineární hledání) – postupně projdi každý blok v souboru a otestuj všechny záznamy v bloku, jestli splňují podmínku výběru.
  - Odhad nákladů (počet čtení bloku z disku) $E_{A1} = b_r$

  - Jestli je podmínka na atributu, který je vyhledávacím klíčem, pak $E_{A1}=(b_r/2)$ (skonči při nalezení záznamu)
  - Lineární hledání může být aplikováno bez ohledu na
    * Výběrovou podmínku
    * Pořadí záznamů v souboru
    * Dostupnosti indexů
- Algoritmus **A2** (binární hledání) – použitelné, pokud výběrová podmínka je operátor rovnosti na atributu, který je klíčem.
  - Předpokládejme, že bloky souboru jsou uloženy spojitě za sebou
  - Odhad ceny (počet bloků přečtených z disku):

$$E_{A2} = \lceil log_2(b_r) \rceil + \lceil SC(A,r) / f_r \rceil - 1$$

    * $\lceil log_2(b_r) \rceil$ - náklady na nalezení první n-tice binárním hledáním
    * $SC(A,r)$ – počet záznamů splňující podmínku
    * $\lceil SC(A,r) / f_r \rceil$ - počet bloků, které obsahují tyto záznamy
  - podmínka rovnosti na atributu, který je klíč: $SC(A,r) = 1$; odhad je pak $E_{A2}=\lceil log_2(b_r) \rceil$

Statistické informace použité v příkladech

- $f_{účet} = 20$ (20 n-tic relace *účet* v jednom bloku)
- $V(jméno\text{-}pobočky, účet) = 50$ (50 poboček)
- $V(zůstatek, účet) = 500$ (500 různých hodnot *zůstatku*)
- $n_{účet} = 10000$ (10000 účtů)
- nechť existují následující indexy na relaci *účet:*
  - primární: B^+-strom na atributu *jméno-pobočky*
  - sekundární: B^+-strom na atributu *zůstatek*

Příklad dohadu operace výběru

$$\sigma_{jméno\text{-}pobočky=\text{“}Perryridge\text{“}}(účet)$$

- Počet bloků je $b_{účet} = 500$; 10000 n-tic v relaci; každý blok obsahuje 20 záznamů
- Předpokládejme, že relace *účet* je seřazena na atributu *jméno-pobočky*
  - $V(jméno\text{-}pobočky, účet) = 50$
  - 10000/50 = 200 n-tic v relaci *účet*, které splňují podmínku *jméno-pobočky=“Perryridge“*
  - 200/20 = 10 bloků, které obsahují tyto záznamy
  - binárním hledáním najdeme první záznam s $\lceil log_2(500) \rceil=9$ přístupy na disk
- Celkové náklady binárního hledání jsou 9+10-1 = 18 čtení bloku (v porovnání s 500 při lineárním průchodu souboru)

Výběr s použitím indexů

- **Indexové hledání** – vyhledávací algoritmus používající index; podmínka je na vyhledávacím klíči indexu
- **A3** (primární index na kandidátním klíči, rovnost) – vybírá jediný záznam splňující podmínku rovnosti: $E_{A3} = HT_i + 1$
- **A4** (primární index na neklíčovém atributu, rovnost) – vybírá více záznamů, vyhledávací klíč je A: $E_{A4} = HT_i + \lceil SC(A,r) / f_r \rceil$

- **A5** (rovnost na vyhledávacím klíči se sekundárním indexem)
  - Vrací jediný záznam, pokud je vyhledávací klíč kandidátním klíčem $E_{A5}=HT_i + 1$
  - Vrací více záznamů (každý může být v jiném bloku), pokud vyhledávací klíč není klíčem relace: $E_{A5}=HT_i + SC(A,r)$

Implementace složitých výběrů

- **Selektivita** podmínky θ_i je pravděpodobnost, že n-tice v relaci *r* splňuje θ_i. Pokud s_i je počet záznamů splňující tuto podmínku, potom selektivita je s_i / n_r.
- **Konjunkce:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. Odhad počtu n-tic ve výsledku:

$$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

- **Disjunkce:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Odhad počtu n-tic ve výsledku:

$$n_r * \left(1 - \left(1 - \frac{s_1}{n_r}\right) * \left(1 - \frac{s_2}{n_r}\right) * \ldots * \left(1 - \frac{s_n}{n_r}\right)\right)$$

- **Negace:** $\sigma_{\neg\theta}(r)$. Odhad počtu n-tic ve výsledku:

$$n_r - size\left(\sigma_\theta(r)\right)$$

Řazení

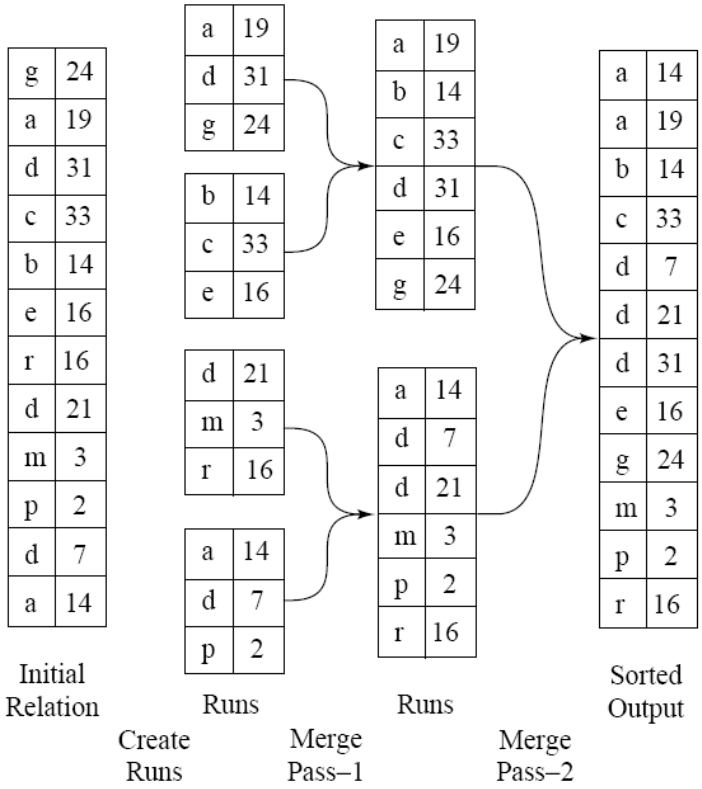
- Můžeme vytvořit index na relaci a poté jej využít ke čtení relace v uspořádaném pořadí. Protože index uspořádává záznamy „logicky“ a ne fyzicky (kde pořadí záznamů může být velmi odlišné), potom vytvoření indexu a jeho procházení může vést k jednomu čtení bloku z disku pro každý záznam.
- Pro relace, které se vejdou do paměti, techniky jako quicksort mohou být použity. Pro relace, které nelze celé načíst do paměti, je vhodnou volbou **externí sort-merge** (seřaď a spoj).

Externí sort-merge

Nechť *M* je velikost paměti ve stránkách (odpovídají blokům):

1. Vytvoř dávky (runs) pro řazení. Nechť $i = 0$. Opakuj, dokud není zpracována celá relace:
   a. Načti *M* bloků relace do paměti
   b. Setřiď načtené bloky
   c. Ulož seřazená data do dávky R_i a zvyš *i*
2. Spoj dávky; pro jednoduchost předpokládejme, že $i < M$. V jednom spojovacím kroku, použij *i* bloků paměti pro načtení vstupních dávek a *1* blok jako výstup. Opakuj následující, dokud nejsou všechny vstupní vyrovnávací paměti prázdné:
   a. Z jednotlivých seřazených bloků v paměti vyber nejmenší záznam
   b. Ulož vybraný záznam do výstupního bloku

c. Smaž vybraný záznam ze vstupního bloku; pokud je blok již prázdný, načti nový blok dávky z disku.



- Je-li $i \geq M$, je nutné provést několik spojovacích kroků.
  - V každém průchodu je *M-1* blízkých skupin dávek spojeno
  - Jeden průchod sníží počet dávek poměrem *M-1* a vytvoří nové dávky delší o stejný poměr
  - Opakujeme dokud nejsou všechny dávky spojeny do jedné
- Analýza nákladů
  - Počáteční vytvoření dávek stejně jako každý průchod slučování vyžaduje $2b_r$ diskových přístupů (kromě posledního průchodu, který nezapisuje výsledek na disk)
  - Celkový počet průchodů je: $\lceil \log_{M-1}(b_r / M) \rceil$

Celkový počet diskový přístupů je: $b_r (2\lceil \log_{M-1}(b_r / M) \rceil + 1)$

Operace spojení

- Několik různých algoritmů pro implementaci spojení (join)
  - Vnořené cykly (nested loops join)
  - Blokované vnořené cykly (block nested loops join)
  - Indexované vnořené cykly
  - Slučované spojení (Merge-join)
  - Hešované spojení
- Volba je založena na odhadu nákladů

Příklad

$$vkladatel \bowtie zákazník$$

Databázový katalog nám poskytuje údaje:

- $n_{zákazník}=10000$
- $f_{zákazník}=25,$ což implikuje, že $b_{zákazník}=10000/25=400$
- $n_{vkladatel}=5000$
- $f_{vkladatel}=50,$ což implikuje, že $b_{vkladatel}=5000/50=100$
- $V(jméno\text{-}zákazníka,vkladatel)=2500,$ což implikuje, že v průměru má každý zákazník dva účty.

Zde předpokládáme, že *jméno-zákazníka* v relaci *vkladatel* je cizí klíč do relace *zákazník.*

Odhady ceny spojení

- Kartézský součin $r \times s$ obsahuje $n_r n_s$ n-tic, každá n-tice je s_r+s_s bajtů dlouhá.
- Pokud $R \cap S = \varnothing$, potom $r \bowtie s$ je stejný jako $r \times s$
- Pokud $R \cap S$ obsahuje klíč schématu R, potom n-tice z relace s je spojena s nejvýše jednou n-ticí relace r; tudíž, počet n-tic ve spojení $r \bowtie s$ není větší než počet n-tic v s.
  Pokud $R \cap S$ v S je cizí klíč odkazující do R, potom počet n-tic ve spojení $r \bowtie s$ je přesně počet n-tic v s.
  Opačný případ, kdy $R \cap S$ je cizí klíč odkazující do S je symetrický.
- V příkladu dotazu *vkladatel* $\bowtie$ *zákazník, jméno-zákazníka* v relaci *vkladatel* je cizí klíč do relace *zákazník,* tedy výsledek obsahuje přesně $n_{vkladatel}=5000$ n-tic
- Pokud $R \cap S = \{A\}$ není klíč ani v R nebo v S.
  Pokud předpokládáme, že každá n-tice v r vytváří n-tice v $r \bowtie s$, počet n-tic v $r \bowtie s$ je odhadován jako:

$$\frac{n_r * n_s}{V(A,s)}$$

  Pokud platí opačný příklad, odhad je následující:

$$\frac{n_r * n_s}{V(A,r)}$$

  Nižší z obou výrazů je pravděpodobně ten přesnější.
- Vypočítej odhady velikostí spojení *vkladatel* $\bowtie$ *zákazník* bez užití informace o cizích klíčích:
  - $V(jméno\text{-}zákazníka,vkladatel)=2500$ a $V(jméno\text{-}zákazníka,zákazník)=10000$
  - Dva odhady jsou $5000*10000/2500=20000$ a $5000*10000/10000=5000.$
  - Zvolíme nižší odhad, který je v tomto případě stejný jako dřívější výsledek pomocí cizích klíčů.

Spojení - vnořené cykly

- Spočítej theta spojení $r \bowtie_\theta s$

```
for each n-tici t_r z r do
    for each n-tici t_s z s do
            testuj dvojici (t_r,t_s), jestli splňuje podmínku θ spojení
            pokud ano, přidej t_r t_s do výsledku
    enddo
```

enddo

- *r* je nazývána **vnější** relace a *s* je **vnitřní** relací spojení.
- Nevyžaduje žádné indexy a může být použit s jakoukoli spojovací podmínkou
- Velmi drahý, protože prochází každou dvojici n-tic z obou relací. Pokud se menší relace vejde do paměti, použij takovou relaci jako vnitřní.
- V nejhorším případě, pokud je dostatek paměti pouze na jeden blok každé relace, odhad ceny spojení je $n_r*b_s + b_r$ diskových přístupů.
- Když se menší relace vejde celá do paměti, použije se jako vnitřní relace a odhad ceny je $b_r + b_s$
- Předpokládejme nejhorší případ, náklady budou *5000*400 + 100=2 000 100* diskových přístupů s vnější relací *vkladatel* a *1000*100 + 400=1 000 400* diskových přístupů s vnější relací *zákazník.*
- Když se menší relace *(vkladatel)* vejde celá do paměti, odhad je *500* diskových přístupů.
- Vhodnější jsou blokované vnořené cykly.

Spojení – blokované vnořené cykly

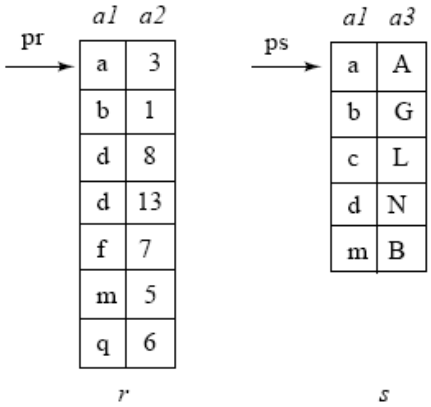
- Varianta vnořených cyklů, která spojuje každý blok vnitřní relace s každým blokem vnější relace.

for each blok B_r z *r* **do**
for each blok B_s z *s* **do**
for each n-tici t_r z B_r **do**
for each n-tici t_s z B_s **do**
testuj dvojici (t_r,t_s), jestli splňuje podmínku θ spojení
pokud ano, přidej t_rt_s do výsledku
enddo
enddo
enddo
enddo

- Nejhorší případ: každý blok vnitřní relace je čten pouze jednou pro každý blok vnější relace, místo jednoho čtení pro každý záznam vnější relace.
- Nejhorší případ odhadujeme: $b_r*b_s + b_r$ diskových přístupů, nejlepší je b_r+b_s, což v našem případě znamená *100*400+100=40 100* čtení bloku z disku.

Slučované spojení

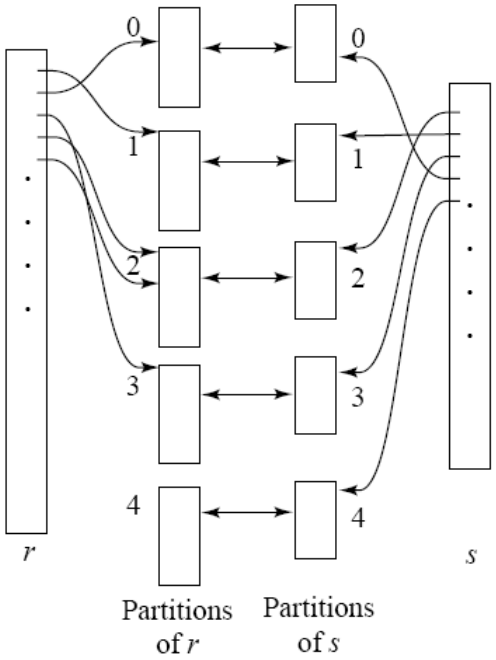
1. Nejprve uspořádej obě relace na jejich atributech pro spojení (pokud již nejsou podle spojovacího atributu uspořádány)
2. Krok spojení je podobný spojovací fázi sort-merge (seřaď a spoj) algoritmu. Hlavní rozdíl je v zacházení s duplikáty v spojovacích atributech – každý pár se stejnou hodnotou na spojovacích atributech musí být vypsán.



- Každá n-tice je čtená pouze jednou, a proto je každý blok také čtený jen jednou. Tedy, počet čtení bloků je b_r+b_s plus náklady na uspořádání relací.

Hešované spojení

- Hešovací funkce *h* je použita pro rozdělení n-tic obou spojovaných relací do množin, které mají stejnou hodnotu hešovací funkce na spojovacích atributech.
  - *h* mapuje spojovací atributy A_s na hodnoty *{0, 1, … max}*, kde A_s označuje společné atributy relací *r* a *s* použité přirozeným spojením.
  - $H_{r0}, H_{r1}, \ldots H_{rmax}$ označují oblasti relace *r* (na počátku prázdné). Každá n-tice $t_r \in r$ je vložena do oblasti H_{ri}, pokud $i = h(t_r[A_s])$.
  - $H_{s0}, H_{s1}, \ldots H_{smax}$ označují oblasti relace *s* (na počátku prázdné). Každá n-tice $t_s \in s$ je vložena do oblasti H_{si}, pokud $i = h(t_s[A_s])$.
- n-tice v H_{ri} jsou porovnávány pouze s n-ticemi v H_{si} – nemusí být porovnávány s jinými oblastmi, protože:
  - n-tice z *r* a n-tice z *s*, které splňují spojovací podmínku, mají stejnou hodnotu na spojovacích atributech.
  - Když je na takovou hodnotu aplikována hešovací funkce, dostaneme hodnotu *i* a n-tice z *r* je uložena do H_{ri} a n-tice z *s* je uložena do H_{si}.



Spojení pomocí hešování je vypočítáno následovně:

1. Pomocí hešovací funkce *h* rozděl relaci *s* na oblasti, jeden blok paměti je použit jako vyrovnávací paměť pro jednu oblast.
2. Podobně rozděl *r.*
3. Pro každé *i*:
   a. Nahraj H_{si} do paměti
   b. Po jedné čti n-tice v H_{ri} z disku a pro každou n-tici t_r najdi odpovídající n-tici t_s v H_{si}. Na výstup dej spojení těchto dvou n-tic.

- Například spojení relací *zákazník* a *vkladatel* s $b_{zákazník}=400$ a $b_{vkladatel}=100$ pomocí hešování vyžaduje *1500* čtení bloků z disku.

Pravidla ekvivalence výrazů

1. Konjunktivní operace výběru (AND) může být převedena na posloupnost jednotlivých operací výběru:
$$\sigma_{\theta 1 \wedge \theta 2}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(E))$$
2. Operace výběru jsou komutativní:
$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) = \sigma_{\theta 2}(\sigma_{\theta 1}(E))$$
3. Pouze poslední projekce z posloupnosti operací projekce je nutná, ostatní mohou být vynechána:
$$\Pi_{L1}(\Pi_{L2}(\ldots(\Pi_{Ln}()))) = \Pi_{L1}(E)$$
4. Operace výběru mohou být kombinovány s operací spojení s podmínkou θ (kartézský součin s podmínkou):
   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$
5. Spojení s podmínkou θ je komutativní:
$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$
6. Operace přirozeného spojení jsou asociativní:
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$
7. a další…

- Příklad
  - Dotaz: najdi všechny účty se zůstatkem větším než 1200 vedené na pobočce v Brně.
$$\sigma_{zůstatek>1200}(účet \bowtie_{pobočka\text{-}jméno=\text{“}Brno\text{“}} pobočka)$$
  - Transformace pomocí pravidla 4b
$$účet \bowtie_{pobočka\text{-}jméno=\text{“}Brno\text{“} \wedge zůstatek>1200} pobočka$$

Kroky v optimalizaci pomocí heuristiky

1. Rozlož konjunktivní operace výběru do posloupnosti jednoduchých výběrů
2. Přesuň výběr co nejblíže k relacím, pro urychlení vyhodnocení dotazu
3. Nejdříve vyhodnoť ty výběry a spojení, které vrací nejmenší výsledek
4. Nahraď kartézské součiny, které jsou následované operací výběru, operací spojení
5. Rozlož seznam atributů projekce na jednotlivé atributy a přesuň je co nejblíže k relacím
6. Najdi místa výrazu, která lze provádět souběžně

Kapitola 13: Transakce

- Koncept transakce
- Stavy transakce
- Implementace atomičnosti a trvanlivosti
- Souběžné spouštění
- Serializovatelnost

Koncept transakce

- Transakce je posloupnost operací (část programu), která přistupuje a aktualizuje (mění) data.
- Transakce pracuje s konzistentní databází.
- Během spouštění transakce může být databáze v nekonzistentním stavu.
- Ve chvíli, kdy je transakce úspěšně ukončena, databáze musí být konzistentní.
- Dva hlavní problémy:
  - Různé výpadky, např. chyba hardware nebo pád systému
  - Souběžné spouštění více transakcí

ACID vlastnosti

Pro zajištění integrity dat musí databázový systém zaručovat:

- **Atomičnost** (Atomicity) – buď všechny operace v transakci jsou provedeny nad databází nebo žádná z nich.
- **Konzistence** (Consistency) – běh jediné (izolované) transakce zachovává konzistenci databáze.
- **Izolovanost** (Isolation) – ačkoli může být více transakcí spouštěno současně, každá transakce nesmí vědět o ostatních současně běžících transakcích. Dočasné mezivýsledky transakce musí být skryté pro ostatní transakce, tj. pro každou dvojici transakcí T_i a T_j platí, že z pohledu transakce T_i je buď T_j dokončena před spuštěním T_i, nebo je T_j spuštěna až po dokončení T_i.
- **Trvanlivost** (Durability) – Po úspěšném dokončení transakce jsou všechny v databázi provedené změny uchovány i při případném výpadku systému.

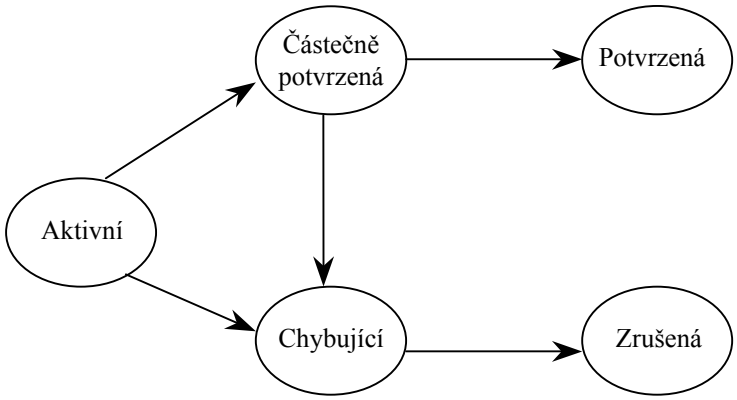
Příklad převodu finančních prostředků

- Transakce převádí \$50 z účtu *A* na účet *B*
  1. **čti(***A***)**
  2. *A := A – 50*
  3. **zapiš(***A***)**
  4. **čti(***B***)**
  5. *B := B + 50*
  6. **zapiš(***B***)**
- Požadavek konzistence – součet zůstatků na účtech *A* a *B* je nezměněn provedením transakce.
- Požadavek atomičnosti – pokud transakce skončí chybou po kroku 3 a před krokem 6, systém by měl zajistit, že provedené změny nebudou uloženy do databáze, jinak by byla v nekonzistentním stavu.

- Požadavek trvanlivosti – pokud byla uživateli již vrácena odpověď o úspěšném provedení transakce, všech provedené změny jsou stálé a musí přežít i výpadek databáze.
- Požadavek izolovanosti – pokud mezi kroky 3 a 6 je dovoleno jiné transakci číst částečně aktualizovanou databázi, bude mít k dispozici nekonzistentní databázi (součet *A+B* bude menší než by měl být).
  Izolovanosti můžeme velmi jednoduše dosáhnout použitím sériového spouštění transakcí, tj. jedna po druhé. Avšak současný běh více transakcí má významné výhody, což uvidíme později.

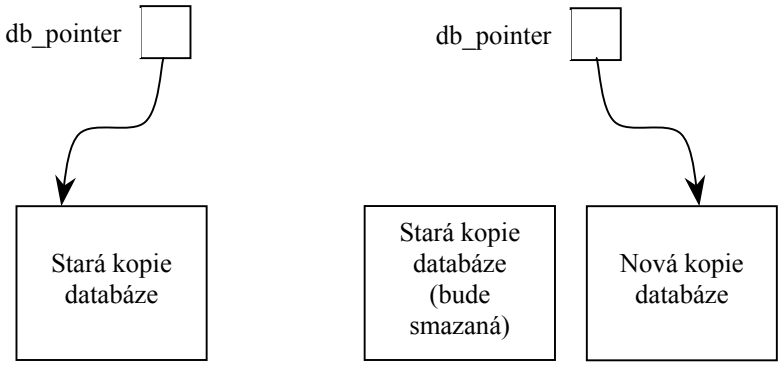
Stavy transakce

- **Aktivní** – počáteční stav; transakce zůstává v tomto stavu, dokud běží
- **Částečně potvrzená** (Partially Committed) – jakmile byla provedena poslední operace transakce
- **Chybující** (Failed) – po zjištění, že normální běh transakce nemůže pokračovat
- **Zrušená** (Aborted) – poté, co byla transakce vrácena (rolled back) a databáze byla vrácena do stavu před spuštěním transakce. Dvě možnosti po zrušení transakce:
  - Znovu spustit transakci – pouze pokud nedošlo k logické chybě
  - Zamítnout transakci
- **Potvrzená** (Committed) – po úspěšném dokončení



Implementace atomičnosti a trvanlivosti

- Databázový podsystém správy obnovy (recovery-management) implementuje podporu pro atomičnost a trvanlivost transakcí.
- Schéma *stínové databáze*:
  - Předpokládejme, že v jednu chvíli je aktivní pouze jedna transakce.
  - Ukazatel **db_pointer** vždy ukazuje na současnou konzistentní kopii databáze.
  - Všechny změny jsou prováděny ve *stínové kopii* databáze a **db_pointer** bude ukazovat na změněnou stínovou kopii, až transakce dosáhne stavu „částečně potvrzená“ a všechny změny byly uloženy na disk.
  - V případě chyby transakce je stará konzistentní kopie (ukazuje na ni **db_pointer**) použita a stínová kopie může být smazána.



(a) před změnou (b) po změně

- Toto schéma předpokládá, že disk nemůže selhat.
- Užitečné pro textové editory, ale velmi neefektivní pro velké databáze: spuštění transakce vyžaduje vytvoření kopie celé databáze.

Souběžné spouštění

- Více transakcí může být spuštěno současně. Výhody jsou:
  - Zvýšené využití procesoru a disku, které vede k vyšší transakční propustnosti: jedna transakce může používat procesor, zatímco jiná disk.
  - Snížená průměrná doba odezvy: krátké transakce nemusí čekat na dokončení dlouhých.
- Schémata pro řízení souběžnosti (Concurrency Control) – mechanizmy pro řízení interakcí (vzájemného působení) souběžných transakcí, které zamezují porušení konzistence databáze.

Plány

- **Plány** jsou posloupnosti, které určují časové pořadí provádění instrukcí souběžných transakcí
  - plán pro množinu transakcí musí obsahovat všechny operace prováděné těmito transakcemi
  - musí zachovávat pořadí instrukcí stejné jako v každé jednotlivé transakci

Příklady plánů

- Nechť T_1 převádí \$50 z účtu *A* na účet *B* a T_2 převádí 10% zůstatku *A* na *B*. Následující plán č.1 je sériové spuštění transakce T_2 po transakci T_1.

T_1	T_2
čti(*A*)	
A := A – 50	
zapiš(*A*)	
čti(*B*)	
B := B + 50	
zapiš(*B*)	
	čti(*A*)
	*temp := A * 0.1*
	A := A – temp
	zapiš(*A*)
	čti(*B*)
	B := B + temp
	zapiš(*B*)

- Nechť T_1 a T_2 jsou stejné transakce. Následující plán č.3 není seriový plán, ale je ekvivalentní plánu č.1.

T_1	T_2
čti(*A*)	
A := A – 50	
zapiš(*A*)	
	čti(*A*)
	*temp := A * 0.1*
	A := A – temp
	zapiš(*A*)
čti(*B*)	
B := B + 50	
zapiš(*B*)	
	čti(*B*)
	B := B + temp
	zapiš(*B*)

- V obou plánech je součet *A+B* zachovaný.
- Následující plán č.4 nezachovává součet *A+B*

T_1	T_2
čti(*A*)	
A := A – 50	
	čti(*A*)
	*temp := A * 0.1*
	A := A – temp
	zapiš(*A*)
	čti(*B*)
zapiš(*A*)	
čti(*B*)	
B := B + 50	
zapiš(*B*)	
	B := B + temp
	zapiš(*B*)

Serializovatelnost

- Základní předpoklady – každá transakce zachovává konzistenci databáze
- Tedy sériový plán zachovává konzistenci databáze
- Plán je serializovatelný, když je ekvivalentní sériovému plánu. Různé formy ekvivalence plánů vedou k následujícím pojmům:
  - Konfliktní serializovatelnost
  - Pohledová serializovatelnost
- Ignorujeme všechny instrukce kromě **čtení** a **zápisu** a předpokládáme, že transakce mohou provádět libovolné výpočty na datech v lokálních vyrovnávacích pamětech mezi čteními a zápisy. Naše zjednodušené plány se skládají pouze z operací **čtení** a **zápisu.**

Konfliktní serializovatelnost

- Instrukce I_i a I_j transakcí T_i a T_j jsou v konfliktu, když existuje nějaké Q, které je přistupované oběma instrukcemi I_i a I_j, a nejméně jedna z nich zapisuje Q.
- 1. I_i = ***čti(Q),*** I_j = ***čti(Q).*** I_i a I_j nejsou v konfliktu.
  2. I_i = ***čti(Q),*** I_j = ***zapiš(Q).*** I_i a I_j jsou v konfliktu.
  3. I_i = ***zapiš(Q),*** I_j = ***čti(Q).*** I_i a I_j jsou v konfliktu.
  4. I_i = ***zapiš(Q),*** I_j = ***zapiš(Q).*** I_i a I_j jsou v konfliktu.
- Intuitivně, konflikt mezi I_i a I_j vynucuje časové pořadí mezi nimi. Pokud I_i a I_j následují za sebou v plánu a nejsou v konfliktu, jejich výsledky zůstanou stejné, ikdyž byly v plánu prohozeny.
- Pokud plán S může být převeden na plán S' pomocí posloupnosti prohození nekonfliktních instrukcí, říkáme, že plány S a S' jsou *ekvivalentní podle konfliktu.*
- Říkáme, že plán S je *serializovatelný podle konfliktu*, jestliže je ekvivalentní podle konfliktu se sériovým plánem.
- Příklad plánu, který není serializovatelný podle konfliktu:

T_3	T_4
čti(Q)	
	zapiš(Q)
zapiš(Q)	

  V uvedeném plánu nejsme schopni vyměnit instrukce tak, že dostaneme sériový plán $<T_3,T_4>$ nebo $<T_4,T_3>$.
- Plán č.3 uvedený níže lze převést na plán č.1 (sériový plán), ve kterém T_2 následuje T_1. Převod je proveden posloupností výměn nekonfliktních instrukcí. Výsledně je plán č.3 serializovatelný podle konfliktu.

T_1	T_2
čti(A)	
zapiš(A)	
	čti(A)
	zapiš(A)
čti(B)	
zapiš(B)	
	čti(B)
	zapiš(B)

Pohledová serializovatelnost

- Nechť *S* a *S'* jsou dva plány pro stejnou množinu transakcí. *S* a *S'* jsou *pohledově ekvivalentní*, jestliže platí následující tři podmínky:
  1. pro každou datovou položku *Q*, když transakce T_i čte počáteční hodnotu *Q* v plánu *S*, potom transakce T_i také musí číst počáteční hodnotu *Q* v plánu *S'.*
  2. pro každou datovou položku *Q*, když transakce T_i provede **čti(**Q**)** v plánu *S* a hodnota je výsledkem transakce T_j, potom transakce T_i musí v plánu *S'* také číst hodnotu *Q*, která je výsledkem transakce T_j.
  3. pro každou datovou položku *Q*, když nějaká transakce provede poslední (závěrečné) **zapiš(**Q**)** v plánu *S*, pak také musí stejná transakce provést poslední **zapiš(**Q**)** v plánu *S'.*
- pohledová ekvivalence je také založena pouze na operacích **čti()** a **zapiš()**.
- Plán *S* je *pohledově serializovatelný*, když je pohledově ekvivalentní sériovému plánu.
- Každý plán, který je serializovatelný podle konfliktu, je také pohledově serializovatelný.
- Následující plán je pohledově serializovatelný, ale ne podle konfliktu:

T_3	T_4	T_6
čti(Q)		
	zapiš(Q)	
zapiš(Q)		
		zapiš(Q)

- Každý plán, který je pohledově serializovatelný, ale není serializovatelný podle konfliktu, používá slepé zápisy (blind writes) – zápis bez předchozího čtení.

Další definice serializovatelnosti

- Následující plán dává stejný výsledek jako sériový plán $<T_1,T_5>$, ale není ekvivalentní pohledově ani podle konfliktu se sériovým plánem.

T_1	T_2
čti(A)	
A := A – 50	
zapiš(A)	
	čti(B)
	B := B – 10
	zapiš(B)
čti(B)	
B := B + 50	
zapiš(B)	
	čti(A)
	A := A + 10
	zapiš(A)

- Rozhodnutí této ekvivalence vyžaduje analýzu výpočetních operací – jiných než čtení a zápis.