DBS

Tvorba datového modelu, ER diagramy, relační databáze, normální formy, dotazovací jazyk SQL, transakce, objektově-relační mapování, replikace, vysoká dostupnost. (DBS)

# Tvorba datového modelu

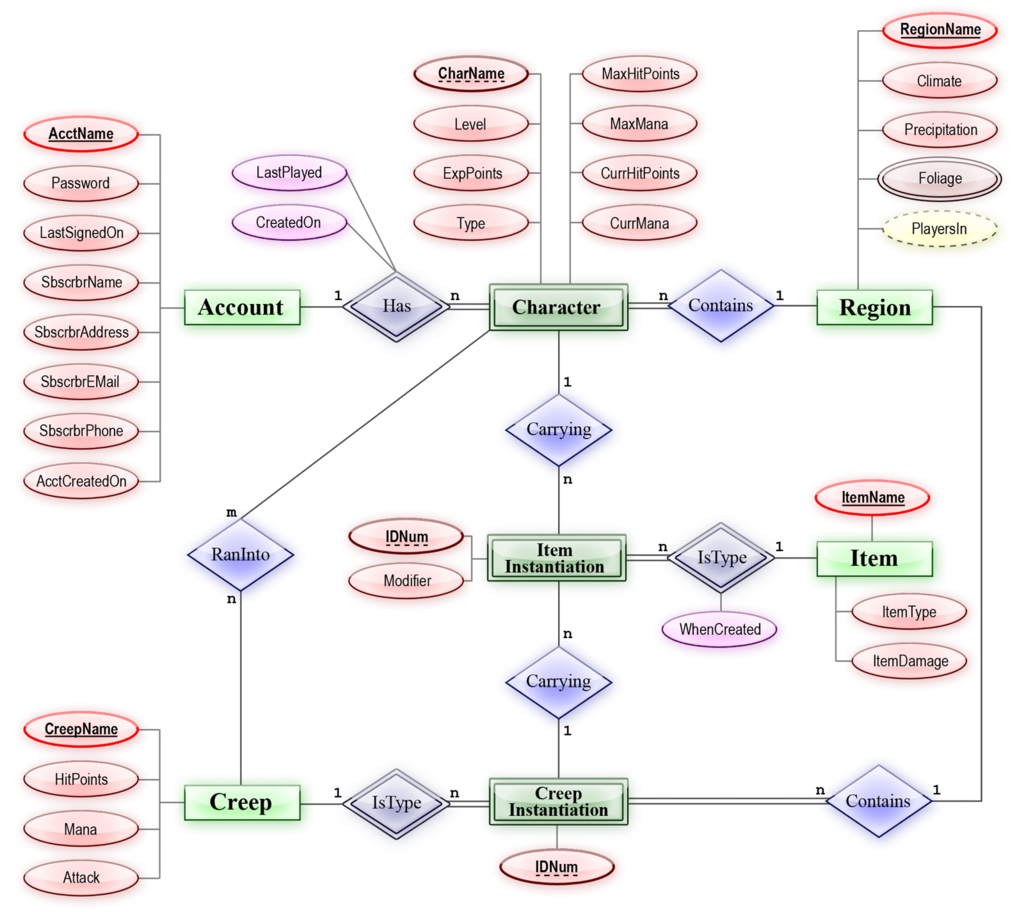
* Proces vytváření konceptu (návrhu) databázového modelu dané aplikace
* Rozděluje se do 3 vrstev:
* **konceptuální –** návrh entit, popisujících reálný problém, a vazeb mezi nimi. ER diagram, UML diagram.
* **Logická –** definice, jak bude samotná databáze vypadat. Objekty a jejich atributy, datové typy, vazby mezi objekty.
* **Fyzická –** specifikace, jak bude databáze implementována, jakou strukturu budou mít uložená data. Tabulky, b-stromy, JSON stromy, …
* Terminologie:
* **Model**= modelovací jazyk
* sada pravidel které se používají k vyjádření vzájemných návazností
* UML model = {třídy, atributy, asociace}
* Relační model = {relační schéma, asociace}
* **Schéma**= vyjadřování významů
* Relační schéma = { Osoba(jméno, věk) }
* **Diagram**= vizualizace schématu
* Proces tvorby konceptuálního modelu :

1. **Analýza požadavků** – identifikování typů entit, jejich vazeb mezi sebou a vlastností
2. **Modelování identifikovaných typů** – výběr jazyka pro modelování, tvorba koncepčního modelu, tvorba datového schématu
3. **Iterativní přizpůsobování datového modelu na základě měnících se požadavků**

* Možnosti zápisu závislostí
* **Unified Modeling Language** (UML) class diagram
* **Entity-Relationship** (ER) diagram
* Object Constraints Language (OCL)
* Object-Role Model (ORM)
* Web Ontology Language (OWL)
* Predicate Logic, Description Logic (DL)

# ER diagramy

* Entity-relationship model (ERM) – používá se pro abstrakci a konceptuální znázornění dat
* Oproti UML nejsou standardizovány, mají různé notace a způsoby vyjádření problematiky
* ER diagramy se skládají z:
  + **Entit** (Account, Creep…)
  + **Atributů** (AcctName, Password…)
  + **Akcí** (Has, Contains, RanInto…)
  + **Vazeb** – kardinalita



# Relační databáze

* Relační databáze využívají jako základu tabulek, které jsou propojeny předem nastavenými vztahy.
* Tabulka obsahuje jednotlivé **sloupce – atributy**a **řádky – záznamy**.
* **Atributy**tabulky **určují vlastnosti objektů**, které se do tabulky budou vkládat Atributy při návrhu tabulky však rozhodně neobsahují samotné hodnoty (!)
* **Atribut**je sloupec v tabulce, vlastnost záznamů v tabulce. U atributu určujeme při návrhu tabulky jeho doménu – rozsah hodnot, kterých může nabývat
* **Atomicita atributu** – atribut je atomický, pokud jej již nelze rozložit na dílčí atributy (jméno na jméno, příjmení a titul například).
* **Primární klíč** – tabulky nemají nijak pevně zafixováno pořadí záznamů (řádků), aby se tedy dalo jednoznačně ukázat na jeden záznam (pro editaci, nebo jeho vybrání), je nutné zvolit jeden atribut tabulky jako unikátní. Takovému atributu říkáme primární klíč – každý záznam v tabulce musí mít hodnotu tohoto atributu unikátní pro celou tabulku (nesmí se vyskytnout dva lidé se stejným RČ). **Každá tabulka musí mít vytvořen primární klíč.**
* **Cizí klíč** – záznamy v tabulkách, které jsou propojeny vztahy musí být nějakým způsobem provázány. Provázání se docílí vytvořením speciálního atributu – tzv. cizího klíče – v jedné z tabulek. Do tohoto atributu se pak bude u jednotlivých záznamů kopírovat hodnota primárního cizího klíče navázaného záznamu z druhé tabulky.
* **vztahy mezi tabulkami -**1:1, n:m, 1:n…

## Integritní omezení:

* zajištění unikátnosti skutečných identifikátorů reálných objektů
* zajišťují dodržování datových typů/domén definovaných sloupcům databázové tabulky.
* zabývají se vztahy dvou tabulek, kde jejich relace je určena vazbou primárního a cizího klíče.
* definují činnosti, které databázový systém provede, pokud jsou porušena některá pravidla.

## Referenční omezení:

* Neodkazujeme se na kraviny (smazané záznamy – neexistující id atd…)

## Indexování

* Záznamy na stránce seřazeny dle “vyhledávacího klíče” (jeden nebo více atributů)
* Stránky jsou udržované v souvislé podobě (data jsou uloženy bez mezer)
* Vyhledávání, rovnosti a (nebo) rozmezí jsou velmi rychlé, protože sekvenční procházení dat je bez problémů.
* Na druhou stranu je ale velmi pomalé vkládání a mazání záznamů, protože data se musí opět urovnat po každé manipulaci.
* v praxi jsou data srovnaná jen na začátku stránky a na konci je volný prostor pro vkládání dalších záznamů. Data se opět seřadí až po určité době)

## SQL

* **Structured Query Language** (Strukturovaný dotazovací jazyk) - je obecný nástroj pro manipulaci, správu a organizování dat uložených v databázi počítače
* SQL **není pouze dotazovací jazyk**, s jeho pomocí je možno také **definovat data**, tedy strukturu tabulky, naplňovat sloupce tabulky (pole záznamů) daty a definovat vztahy a organizaci mezi položkami dat.
* Dále umožňuje řízení přístupu k datům, tedy udělování a odebírání přístupových oprávnění na různých úrovních, čímž chrání data před náhodným nebo úmyslným zničením, neautorizovaným čtením nebo manipulaci s nimi, dále také umožňuje sdílené využívání dat a zajišťuje hladký průběh činností, přistupuje-li k datům více uživatelů současně.
* Pracuje s relačními databázemi, ve kterých se uživatel dívá na data v podobě soustavy provázaných tabulek. Každá tabulka představuje množinu dat, která je uspořádaná v řádcích (záznamech) a sloupcích (položkách). Na hodnotu dat se uživatel odkazuje jako na prvek v matici.
* SQL dále podporuje i jednoduché funkce → https://www.w3schools.com/sql/sql\_ref\_sqlserver.asp
* Dělení příkazů:
  + **DDL – Data Definition**
    - CREATE, ALTER, DROP, RENAME
  + **DQL – Data Query**
    - SELECT
  + **DML – Data Modify**
    - INSERT, UPDATE, DELETE, MERGE
  + **DCL – Data Control**
    - GRANT, REVOKE

### Relační úplnost

* Schopnost jazyka vyjádřit jakoukoliv relaci pomocí jeho vyjadřovacích funkcí
* **Relační algebra je relačně úplná** 
  + Pokud má nějaký jazyk schopnost vyjádřit alespoň to co umí RA, je relačně úplný.
* **SQL je relačně silnější** z hlediska vyjadřování než RA, jelikož má například agregační funkce
  + **Agregační funkce** – AVG, SUM, COUNT, MIN, MAX

### Databázové pohledy – View

View je databázový objekt, který uživateli poskytuje data ve stejné podobě jako tabulka. Na rozdíl od tabulky, kde jsou data přímo uložena, obsahuje pohled pouze předpis, jakým způsobem mají být data získána z tabulek a jiných pohledů. Z toho vyplývají některé základní rozdíly v chování tabulky a pohledu:

* Data tabulky lze přímo modifikovat pomocí příkazů. View se modifikuje obtížně, musí se celé změnit
* Na rozdíl od tabulky pohled nezabírá v paměti téměř žádné místo, protože neobsahuje data, ale pouze předpis pro získání dat (obvykle příkaz SELECT).
* Získání výsledku především u komplikovanějších pohledů může být časově výrazně náročnější než u přímého přístupu k tabulce

Náhledy SQL umožňují vytvořit pro různé uživatele různé pohledy na struktury tabulek a na data. Každý uživatel tak vidí pouze ta data, která má vidět. Přitom data vidí uživatel opět v podobě jednoduché tabulky, i když ve skutečnosti data pocházejí z různých tabulek. Data zobrazovaná v náhledech jsou dynamická, tzn. změní-li se data v tabulkách (databázových souborech), změní se také data, které zobrazuje náhled.

### Cursor

* Funkce umožňující procházet jednotlivé záznamy iterativně.
* Databázové kurzory se nejčastěji používají v uložených procedurách a triggerech
* Doporučené použití: Pokud to jde, vyhnout se použití kurzoru

### Trigger

Uložená procedura spuštěná nějakou událostí

## Vazby

Vztahy (angl. relationships) slouží ke svázání dat, která spolu souvisejí a jsou umístěny v různých databázových tabulkách. Každý vztah je charakterizován třemi základními vlastnostmi:

* *stupněm*
* *kardinalitou*
* volitelností účasti

### Stupeň vztahu

Unární vztah – relace je spojena sama se sebou. Typickým příkladem je vztah zaměstnance a jeho nadřízeného, kdy nadřízený je také jedním ze zaměstnanců, pročež také může mít nadřízeného. Vztah se realizuje vložením primárního klíče relace zaměstnanec ve formě cizího klíče opět do relace zaměstnanec.

* Binární vztah – klasický vztah mezi dvěma relacemi.
* Ternární vztah – vztah mezi třemi relacemi najednou.
* N-ární vztah – vztah mezi n-relacemi zároveň.
* Ternární a n-ární vztahy se nesnadno modelují a v praxi se objevují velice zřídka.

### Kardinalita vztahu

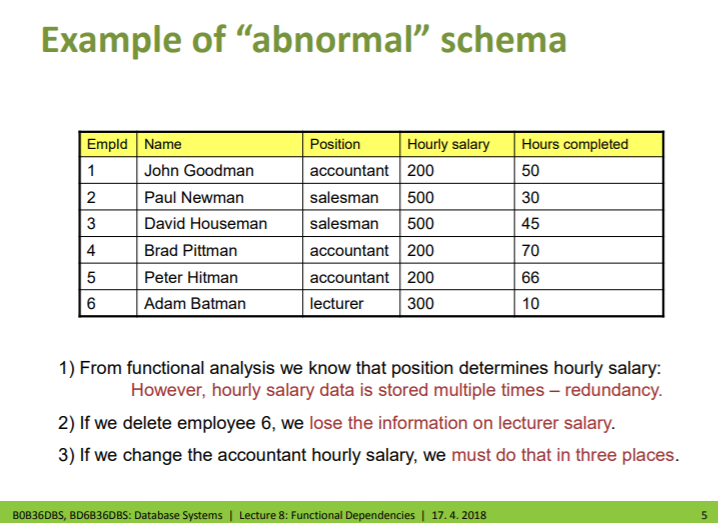
Mezi daty v tabulkách není žádná spojitost, proto nedefinujeme žádný vztah.

* **1:1** používáme, pokud záznamu odpovídá právě jeden záznam v jiné databázové tabulce a naopak. Takovýto vztah je používán pouze ojediněle, protože většinou není pádný důvod, proč takovéto záznamy neumístit do jedné databázové tabulky. Jedno z mála využití je zpřehlednění rozsáhlých tabulek. Jako ilustraci je možné použít vztah řidiče a automobilu – v jednu chvíli (diskrétní časový okamžik) jeden řidič řídí právě jeden automobil a zároveň jeden automobil je řízen právě jedním řidičem.
* **1: N** přiřazuje jednomu záznamu více záznamů z jiné tabulky. Jedná se o nejpoužívanější typ relace, jelikož odpovídá mnoha situacím v reálném životě. Jako reálný příklad může posloužit vztah autobusu a jím cestujícího pasažéra – v jednu chvíli pasažér jede právě jedním autobusem a v jednom autobuse může zároveň cestovat více pasažérů.
* **M: N** umožňuje každému záznamu z jedné tabulky přiřadit libovolný počet záznamů z druhé tabulky, přičemž záznam v druhé tabulce přiřazením k záznamu k první tabulce svou možnost přiřazení „nespotřebuje“, takže jej lze přiřadit k libovolnému počtu záznamů první tabulky. V databázové praxi bývá tento vztah z praktických důvodů nejčastěji realizován kombinací dvou vztahů, a sice 1:N a 1:M, které ukazují do pomocné tabulky, složené z kombinace obou použitých klíčů (jde o třetí, resp. tzv. vazební tabulku). Příkladem z reálného života by mohl být vztah výrobku a jeho vlastnosti – výrobek může mít více vlastností a jednu vlastnost může mít více výrobků

### Volitelnost účasti ve vztahu

Volitelnost účasti ve vztahu vyjadřuje, zda je účast relace ve vztahu povinná nebo volitelná.

# Normální formy

* Proč se tím zabývat normálními formami
  + Redundance dat = Zbytečné ukládání stejné hodnoty vícekrát
  + Duplicita dat znesnadňuje jejich změnu (nutný update více tabulek najednou).
* Řešením je **normalizace** dat

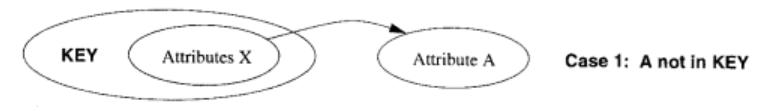
## 1.NF – První normální forma

* Sloupec musí obsahovat pouze atomické hodnoty stejného typy
* Sloupce musí mít různé názvy
* Nesmí záležet na uloženém pořadí dat

## 2.NF – Druhá normální forma

* Musí být v 1. normální formě
* Nesmí obsahovat parciální závislosti

**Parciální závislost**

* + Atribut závisí pouze na části primárního klíče
  + **Tj.:** všechny atributy musí být závislé na celém složeném Pk

**Příklad**

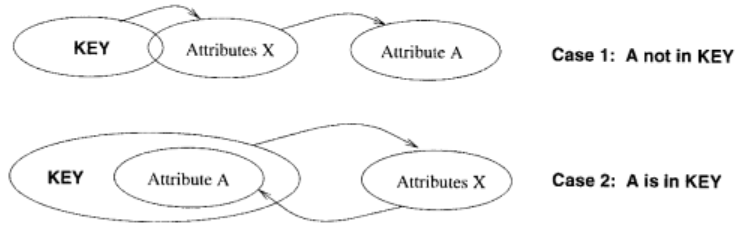
Tabulka závěrečného **hodnocení**: **score\_id**, **student\_id**, **subject\_id**, **score**, **teacher**

Tabulka **předmětu**: **subject\_id**, **subject\_name**

* + Schéma není v 2. NF, jelikož **teacher** závisí pouze na **subject\_id**
  + **Řešení**: Atribut **teacher** se přesune to tabulky **předmět**

## 3.NF – Třetí Normální forma

* Musí být ve druhé normální formě
* Nesmí obsahovat tranzitivní závislosti
* **Tranzitivní závislost**
  + Atribut je závislý na jiném atributu, který je závislý na primárním klíči
  + **Tj.:** všechny atributy musí být závisle na Pk



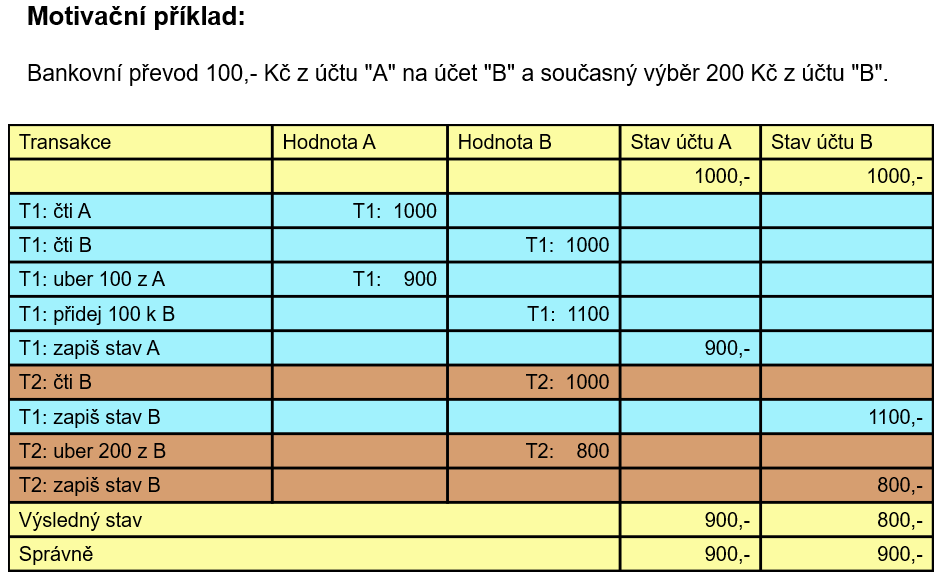
**Příklad**:

Tabulka **knih**: **author**, **book**, **author\_nationality**

* + Schéma není v 3. NF, jelikož **author** => **book** => **author\_nationality**
  + **Řešení**: separátní tabulka autorů.

# Transakce a paralelismus

* posloupnost příkazů, které převedou databázi z jednoho konzistentního stavu do druhého.



Při současném vykonávání transakcí by mohlo dojít k porušení konzistence databáze  
i když každá z transakcí by sama o sobě konzistenci zachovávala

## ACID vlastnosti transakcí

### Atomicita – atomic

* Je dále nedělitelná
* Buď se provede jako celek nebo se neprovede žádná z nich

### Konzistence – consistent

* Při a po provedení transakce není porušeno žádné integritní omezení
* Transakce zachování konzistenci

### Izolovanost – isolated

* Operace v transakci jsou skryty před vnějšími operacemi
* Vracení/Rollback nezasáhne jinou transakci
  + V případě že zasáhne musí být rollback i na zasažené transakci (cascade rollback)

### Trvalost – durable

* Při úspěšném dokončení transakce jsou všechny změny uchovány i v případě výpadku

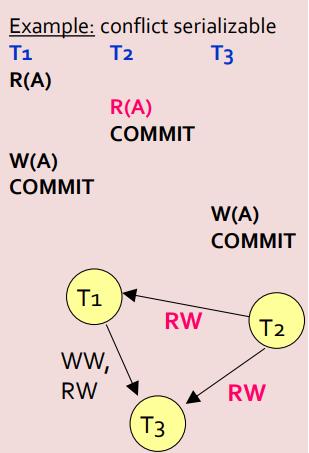
## Serializovatelnost

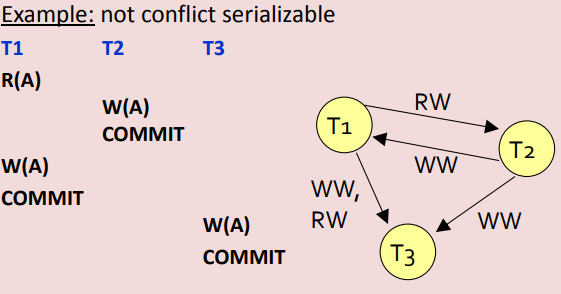
* Plán je serializovatelný, když jeho provedení paralelních transakcí zachová databázi v konzistentním stavu

### Konfliktní operace

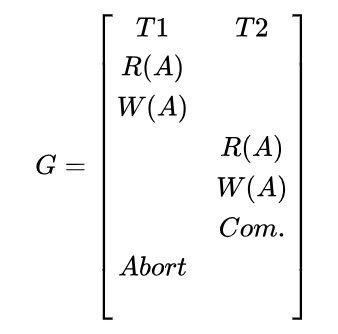
* **RW** (ReadWrite – unrepeatable read) - konflikt, kdy jedna transakce přečte data a poté je přerušena druhou, která je změní. První transakce tedy dále operuje s upravenými daty
* **WR** (WriteRead – reading uncommited data) - konflikt, kdy jedna transace zapíše hodnotu, ale než zápis stihne potvrdit, přečte si hodnotu jiná transakce
* **WW** (WriteWrite – overwrite of uncommited data) - konflikt, kdy jedna transakce zapíše data, ale než je stihne potvrdit, přepíše je jiná transakce.

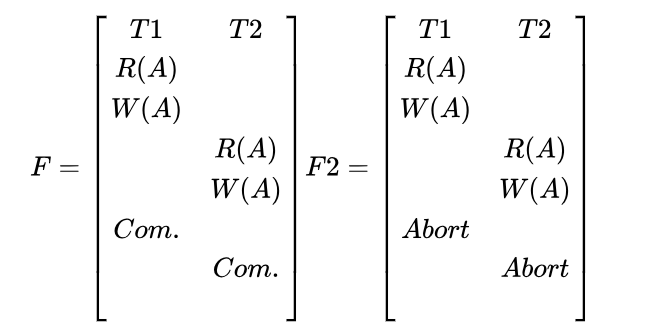
## Conflict serializibility

Conflict-serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

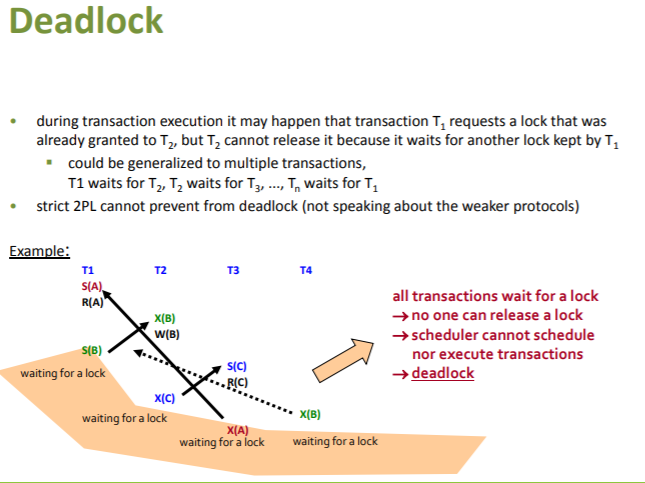
* Detekce konfliktních párů v grafu serializovatelnosti
  + Aby byl konfliktně serializovatelný musí graf být acyklický

## Unrecoverable schedule a kaskádní rušení

* Nastane, pokud transakce T2, která byla závislá na transakci T1 comittovala a T1 abortovala.
* Ochrana proti **kaskádnímu rušení transakcí** je pravidlo: žádná transakce nesmí číst dirty data. To znamená, že musí commitovat až po transakci, na které byla závislá.



## Deadlock



* Odstranění deadlocku
  + napadením jedné z Coffmanových podmínek je naprosto spolehlivé, ale obvykle obtížné a někdy nemožné.
* Deadlocky nejsou tak časté
* Řešení (zamykání (locking) na různé úrovni granularity):
  + zamykání celého systému (=> **sériovost**)
  + jednotlivých tabulek
  + Jednotlivých záznamů (vět)
  + časové značky

### Coffmanovy podmínky / deadlock

K uváznutí dojde jen při splnění všech následujících podmínek:

**Vzájemné vyloučení (Mutual exclusion)**

* Prostředek může v jednom okamžiku používat jenom jeden proces.

**Drž a čekej (Hold & wait)**

* Proces může žádat o další prostředky, i když už má nějaké přiděleny.

**Neodnímatelnost (No preemption)**

* jakmile proces zmíněný prostředek vlastní, nelze mu ho bezpečně odejmout, musí ho sám vrátit.

**Cyklické čekání (Circular wait)**

* Je možné uzavřít cyklus z procesů čekající každý na svého předchůdce – respektive k deadlocku dojde, jakmile je tento cyklus uzavřen.

#### Napadení Coffmanových podmínek

**Drž a čekej**

* Proces musí o všechny prostředky, které potřebuje, zažádat najednou. Buď je všechny dostane, nebo nedostane ani jeden. Takto postupují databáze při použití zámků v jazyku SQL – musí všechny tabulky, které chtějí mít zamčené, zamknout najednou.

**Cyklické čekání**

* Vznik cyklu lze vyloučit například pokud existuje jednoznačné pořadí, v jakém se o prostředky žádá. Na tomto principu je postaveno i hierarchické zamykání, kdy se smí zamykat pouze směrem od kořene stromu dolů.
* Pokud už nastalo: Zrušení některé transakce a její následné restartování deadlock rozhýbe

## 2PL – Two phase locking = dvoufázové zamykání

* Protokol, který se používá pro odstranění konfliktu zápisů/čtení
* Může dojít k deadlocku
* **Zaručuje**: Conflict serializable
* Používá dva druhy zámků

*Sdílený*

* + S(A) – vlastní může **číst** data

*Exkluzivní*

* + X(A) vlastní může **číst** i **zapisovat**
* Dělí se na dvě po sobě jdoucí fáze:

*Expanding phase*

* + Jednotlivé transakce si mohou rezervovat jednotlivé zámky (Sdílený, exklusivní), ale pokud už nějaké mají, nesmí je propustit.

*Shrinking phase*

* + Transakce mohou zámky pouze otevírat a žádný další nesmí požadovat

## Striktní 2PL

* Zpřísňuje 2PL tak, že musí být všechny vyžádané zámky odemknuty najednou když je transakce ukončena (commit/abort)
* Na rozdíl od normálního 2PL zaručuje “**schedule recoverability**” (transakce může být ukončena až kdy jsou ukončeny všechny transakce, které ji ovlivňovali
* Dále se “vyhýbá” **kaskádovým abortům** (aborty, už potvrzených transakcí)

# ORM

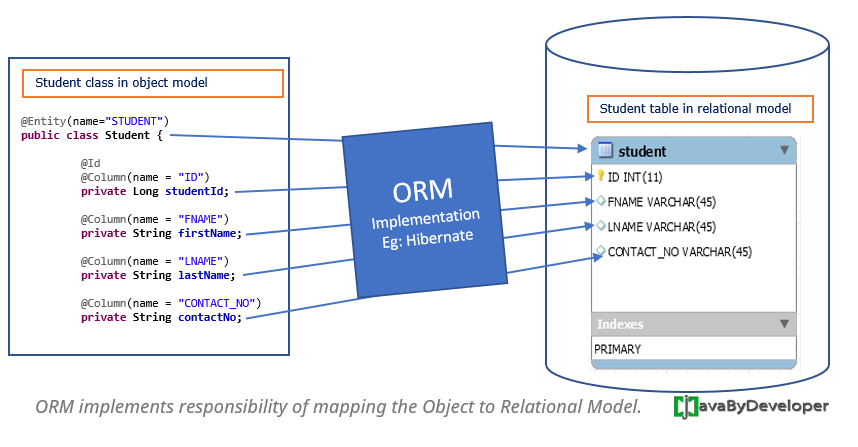
**Objektově relační zobrazení** (**ORM**, **O/RM** nebo **O/R zobrazení**) je [programovací technika](https://cs.wikipedia.org/wiki/Programov%C3%A1n%C3%AD) v [softwarovém inženýrství](https://cs.wikipedia.org/wiki/Softwarov%C3%A9_in%C5%BEen%C3%BDrstv%C3%AD), která zajišťuje automatickou konverzi dat mezi [relační databází](https://cs.wikipedia.org/wiki/Rela%C4%8Dn%C3%AD_datab%C3%A1ze) a [objektově orientovaným programovacím jazykem](https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD).

ORM se dále stará o automatickou konverzi rozdílných datových typů mezi databázovým systémem a programovacím jazykem. Pokročilé techniky ORM také řeší například možnost využití [objektové dědičnosti](https://cs.wikipedia.org/wiki/D%C4%9Bdi%C4%8Dnost_(objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD)), kterou relační databáze nepodporují.

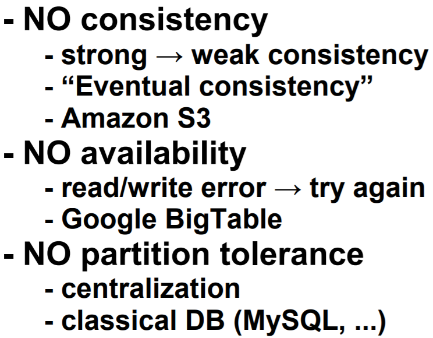
Díky tomu je vývojář při práci s daty, které jsou v aplikaci reprezentovány právě pomocí objektů, do jisté míry odstíněn od nutnosti pracovat s [SQL](https://cs.wikipedia.org/wiki/SQL) dotazy konkrétní relační databáze.

Nicméně **hlavním cílem ORM je synchronizace mezi používanými objekty v aplikaci a jejich reprezentací v databázovém systému tak, aby byla zajištěna persistence dat**. Vývojář potřebuje persistentně uchovávat objekty, ale nepotřebuje se starat, jak se tato persistence provede.

* JPA ORM Hibernate – many to many vazba několika způsoby
  + https://www.baeldung.com/jpa-many-to-many



# Distributed data

**Cap theorem**

states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

1. **Consistency**: Every read receives the most recent write or an error
2. **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
3. **Partition** **tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

No distributed system is safe from network failures; thus, network partitioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability.

When choosing consistency over availability, the system will return an error or a time out if information cannot be guaranteed to be up to date due to network partitioning.

When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

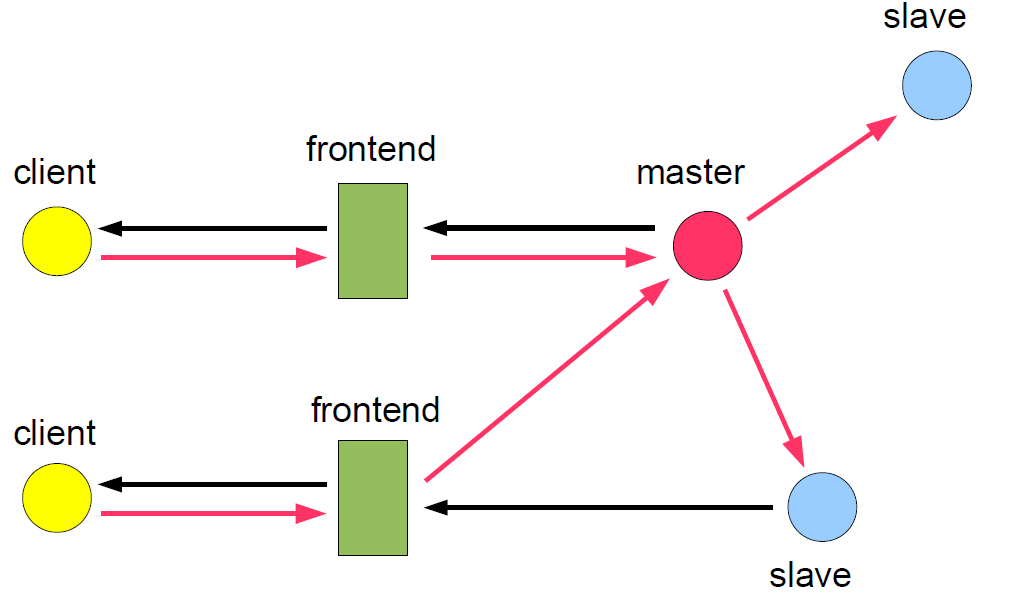
**considerations**

* Replication: number of replicas, none, partial, full
* Mechanism: multi-master, master-slave, quorum based, gossip architecture, lazy replication → weaker consistency

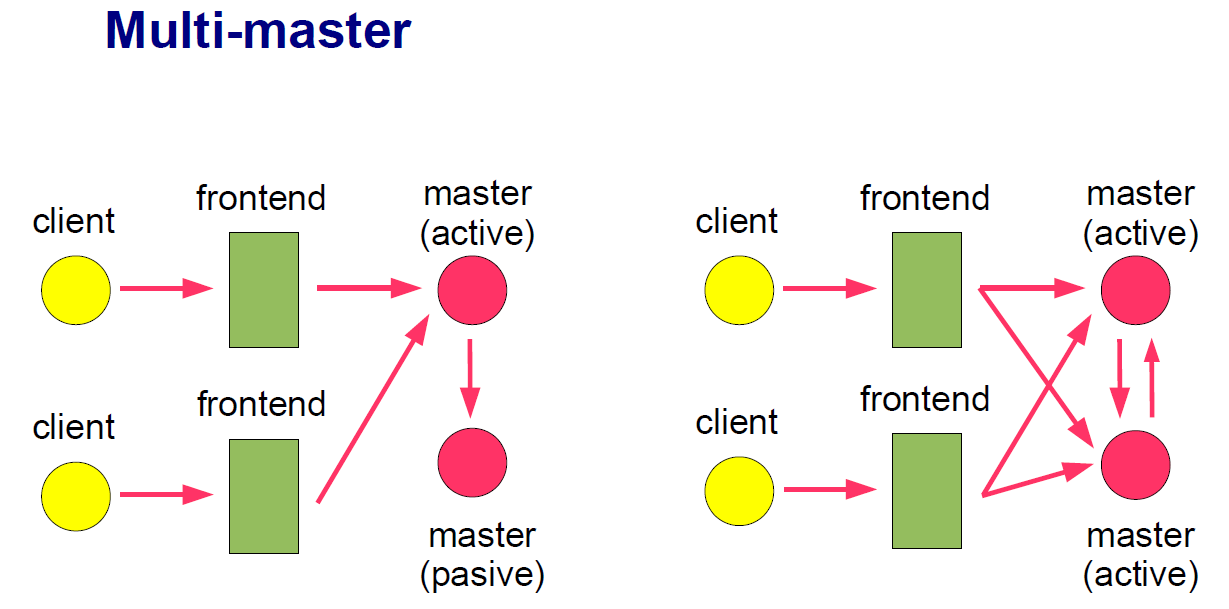
## Data replication

### Master slave

* Master zajišťuje read a write
* Slave zajišťuje pouze read

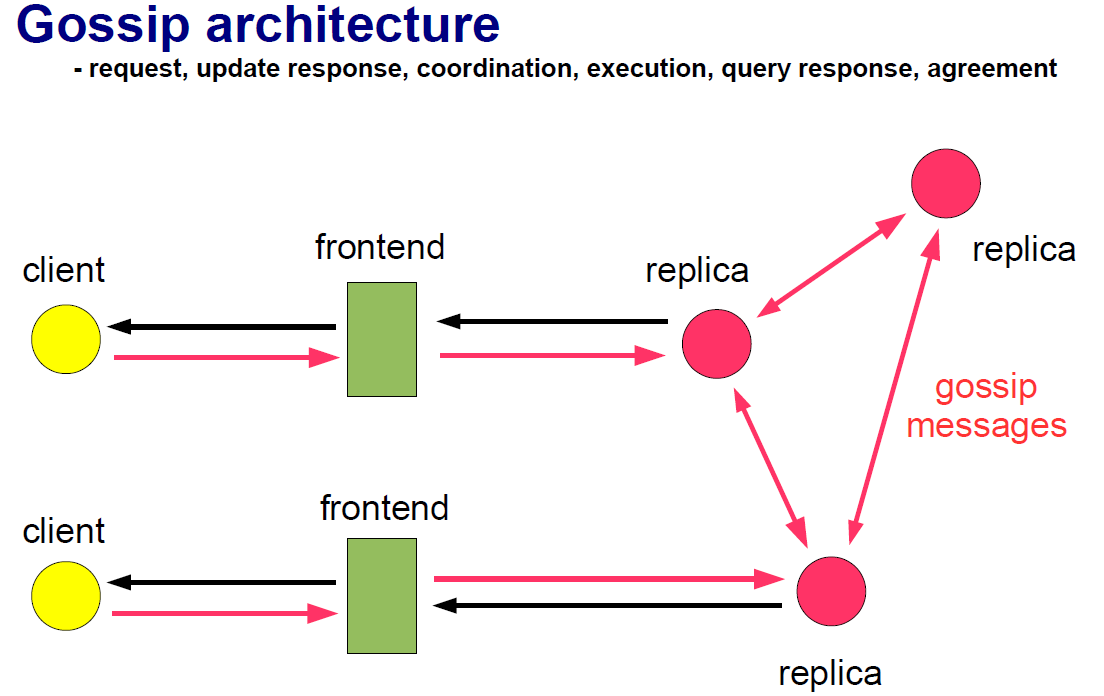


### Multi master / Peer to peer

* Všechny nody jsou mastery, uzel na který se volá write je zodpovědný za distribuci na ostatní nody

### Gossip architecture

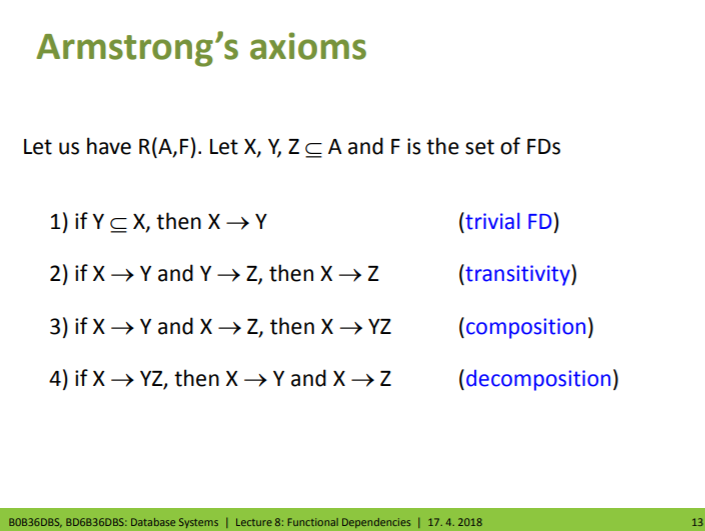
* The gossip architecture is a framework for providing a highly available service which replicates data close to the points where groups of clients request it.
* RMs exchange “gossip” messages periodically in order to convey the updates they have each received from clients.
* A gossip service provides two basic types of operation:
  + **queries** are read-only operations; and
  + **updates** modify but do not read the state.
* A front end can send a query or update to any RM that it chooses (based on availability and response time).
* RM may be temporarily unable to communicate with each other due to failures. Nevertheless, the system guarantees a form of consistency called relaxed consistency
* **Relaxed consistency between replicas**: All RM eventually receive all updates and they apply updates with ordering guarantees that make the replicas sufficiently similar to meet the needs of the application.



# Funkční závislost

* Funkční závislost je v databázi **vztah mezi atributy** takový, že máme-li atribut Y je funkčně závislý na atributu X píšeme X → Y, pak se nemůže stát, aby dva řádky mající stejnou hodnotu atributu X měly různou hodnotu Y.
* Pomocí funkčních závislostí můžeme automaticky navrhnout schéma databáze a předejít problémům jako je **redundance, nekonzistence databáze**, zablokování při vkládání záznamů apod.

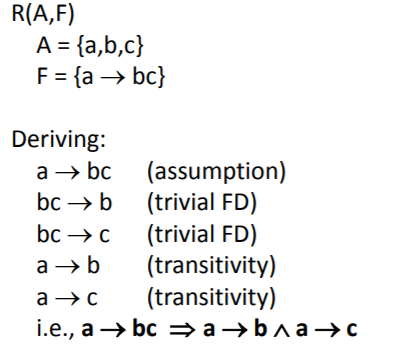
## Armstrongovy axiomy



* Armstrongovy axiomy jsou odvozovací pravidla funkčních závislostí. Pomocí axiomu získáme uzávěr funkčních závislostí spolu s klíči schémat

### Vlastnosti

* **korektní**: Všechny odvozená závislosti odpovídají původnímu schématu.
* **Kompletní**: Jakákoliv závislost leze z nich odvodit
* Axiomy 1, 2, 3 jsou **nezávislé**. *Dekompozice* se dá odvodit pomocí triviality a tranzitivity.



## Algoritmus pro nají všech „nejmenších“ kandidátních klíčů

Mám set atributů **A** a množinu funkčních závislostí **Z**. Máme najít množinu **K** všech klíčů.

* **Klíč/kandidátní klíč** je nejmenší množina atributů, ze které jde pomocí Armstrongových axiomů získat všechny atributy dané relace.

**A = {A, B, C, D, E, F}  
Z =** **{AB → C, C → D, DEF → B, DA → B}**

Algoritmus:

1. Najdeme všechny atributy, které se buď nevyskytují na pravé straně funkčních závislostí v **Z**, nebo se nevyskytují vůbec. Tato skupina atributů musí zákonitě tvořit klíč, jelikož nejdou nijak odvodit.
   * Výsledek **X = {A, F}**
2. Nyní ověříme, jestli množina **X** tvoří klíč, tj. je možné pomocí Armstrongových axiomů odvodit všechny atribut z **A**. Pokud ano, našli jsem klíč a víme, že jakýkoliv další klíč bude nadmnožina **X**, a proto x bude nemenší. Algoritmus končí. Pokud **X** není klíč pokračujeme.
3. Nyní budeme postupně přidávat atributy z **A** a kontrovat, jestli tvoří klíč, tj. je možné pomocí Armstrongových axiomů odvodit všechny atribut z **A**.

* **AFA => AF** Není klíč
* **AFB => ABCDEF** Je klíč
* **AFC => ABCDEF**  Je klíč
* **AFD => ABCDEF** Je klíč
* **AFE => AFE** Není klíč
* **AFF => AF** Není klíč

1. Hledaná množina je: **K = {AFB, AFC, AFD}**

Video: <https://www.youtube.com/watch?v=xM562aZf23M>

Pokud nenajdeme žádné atributy, které jsou pouze na levé straně, pokračujeme bodem dva a postupně zkoušíme všechny možné kombinace, do té doby než najdeme klíč a klíče stejné délky