SQLite
je tzv. relační databáze. Tento pojem označuje databázi založenou na tabulkách. Každá tabulka obsahuje položky jednoho typu. Databázovou tabulku si můžeme představit třeba jako tabulku v Excelu. Položky ukládáme na jednotlivé řádky, sloupce pak označují atributy (vlastnosti, chcete-li), které položky mají.

SQLite databáze není typovaná, to znamená, že nemusíme uvádět datový typ sloupce (celé číslo, reálné číslo, text...), přestože my to dělat budeme, a i když datový typ stanovíme, tak můžeme uložit do tohoto sloupce i hodnoty jiného datového typu.

Pokud chceme s relační databází rozumně pracovat, každý řádek v tabulce by měl být opatřený unikátním identifikátorem. Nejčastěji se používají generované číselné identifikátory.

Slovo *relační* označuje vztah (anglicky relation). Ten je mezi tabulkami nebo mezi entitami v jedné tabulce.

RDBMS
Označení databáze je vlastně nepřesné a v odborné literatuře se setkáme s označením RDBMS (Relation DataBase Management System). Databázový stroj (tedy zde SQLite) není jen úložiště dat. Jedná se o velmi sofistikovaný a odladěný nástroj, který za nás řeší spoustu problémů a zároveň je extrémně jednoduchý k použití. S databází totiž komunikujeme jazykem SQL, kterým jsou v podstatě lidsky srozumitelné věty. Spolu s ukládáním dat je ale třeba dále řešit mnoho dalších věcí. Asi by nás napadlo např. zabezpečení nebo optimalizace výkonu. RDBMS toho ale dělá ještě mnohem více, řeší za nás problém současné editace stejné položky několika uživateli ve stejný okamžik, který by jinak mohl zapříčinit nekonzistenci databáze. RDBMS data v tomto případě zamkne a odemkne až po vykonání zápisu. Dále umožňuje spojovat několik dotazů do transakcí, kdy se série dotazů vykoná vždy celá nebo vůbec. Nestane se, že by se vykonala jen část. Tyto vlastnosti databázového stroje jsou shrnovány zkratkou ACID.

ACID
je akronym slov ***Atomicity*** (nedělitelnost), ***Consistency*** (validita), ***Isolation*** (izolace) a ***Durability*** (trvanlivost). Jednotlivé složky mají následující význam:
• ***Atomicity*** - Operace v transakci se provedou jako jedna atomická (nedělitelná) operace. Tzn. že pokud nějaká část operace selže, vrátí se databáze do původního stavu a žádné části transakce nebudou provedeny. Reálný příklad je např. převod peněz na bankovním účtu. Pokud se nepodaří peníze odečíst z jednoho účtu, nebudou ani připsány na účet druhý. Jinak by byla databáze v nekonzistentním stavu. Pokud bychom si práci s daty řešili sami, mohlo by se nám toto velmi jednoduše stát.
• ***Consistency*** - Stav databáze po dokončení transakce je vždy konzistentní, tedy validní podle všech definovaných pravidel a omezení. Nikdy nenastane situace, že by se databáze nacházela v nekonzistentním stavu.
• ***Isolation*** - Operace jsou izolované a navzájem se neovlivňují. Pokud se sejde v jeden okamžik více dotazů na zápis do stejného řádku, jsou vykonávány postupně, jako ve frontě.
• ***Durability*** - Všechna zapsaná data jsou okamžitě zapsána na trvanlivá úložiště (na pevný disk), v případě výpadku el. energie nebo jiného přerušení provozu RDBMS vše zůstane tak, jak bylo těsně před výpadkem.

Databáze (přesněji databázový stroj) je tedy černá skříňka, se kterou naše aplikace komunikuje a do které ukládá veškerá data. Její použití je velmi jednoduché a je odladěna tak, jak bychom si sami zápis dat v programu asi těžko udělali. Vůbec se nemusíme starat o to, jak jsou data fyzicky uložena, s databází komunikujeme pomocí jednoduchého dotazovacího jazyka SQL.

Proč právě SQLite?
pouze malá knihovna nástrojů, kterou mají již některé jazyky, zvláště ty interpretované, zabudovanou v sobě. Jak už název napovídá, databáze je velmi odlehčená, takže neobsahuje třeba uživatelské oprávnění, konfiguraci (která je možná částečně přes PRAGMA příkazy), či plnou podporu UTF, pokud tedy očekáváte korektní české řazení např. č po c, tak marně, buď si to sami dopíšete přímo v SQLite, nebo to necháte až na aplikaci.

Jazyk SQL
SQL označuje ***Structured Query Language***, tedy strukturovaný dotazovací jazyk. SQL je tzv. jazyk deklarativní. Zatímco u imperativních jazyků počítači vlastně říkáme krok po kroku co má udělat, u jazyků deklarativních pouze říkáme co má být výsledkem a již nás nezajímá, jak tohoto výsledku počítač dosáhne. SQL se původně jmenovalo SEQUEL (Structured English Query Language) a vzniklo v laboratořích společnosti IBM s cílem vytvořit jazyk, kterým by se dalo komunikovat s databází jednoduchou angličtinou.

Do závorky se píší názvy jednotlivých sloupců s jejich datovými typy a případně i dalšími atributy, jako např. zde *PRIMARY KEY* a ***AUTOINCREMENT***, a oddělují se čárkou.

V SQL se většinou píší příkazy velkými písmeny, to proto, že je to lépe odliší od zbytku dotazu nebo od kódu naší aplikace (např. v PHP). Názvy tabulek, sloupců a další identifikátory jsou naopak malými písmeny a podtržítkovou notací. Je dobrým zvykem je vkládat mezi dvojité uvozovky (případně další podporované znaky, jako zpětné uvozovky a podobně), ale nejsou povinné a příkazy proběhnou bez problémů i bez nich.

SQLite - Vytvoření databáze

Pokud bychom chtěli vytvořit databázi bez použití DB Browseru for SQLite, tak stačí spustit v konzoli/terminálu následující příkaz, který vás rovněž přenese do interaktivního SQLite shellu:

```
sqlite3 databaze_pro_web.db
```

Založení tabulky:

```
CREATE TABLE "uzivatele" (
        "uzivatele_id" INTEGER PRIMARY KEY AUTOINCREMENT,
         "jmeno" TEXT,
        "prijmeni" TEXT,
        "datum_narozeni" TEXT,
        "pocet_clanku" INTEGER
);
```

CREATE TABLE, označuje, že chceme vytvořit tabulku. Poté následuje jméno tabulky, které je v SQLite obvyklé dávat do dvojitých uvozovek. Do závorky se píší názvy jednotlivých sloupců s jejich datovými typy a případně i dalšími atributy, jako např. zde ***PRIMARY KEY*** a ***AUTOINCREMENT***, a oddělují se čárkou. Každý příkaz v SQL zakončujeme středníkem, který ani tady nechybí.

Vložení záznamu do tabulky:

```
INSERT INTO "uzivatele" (
        "jmeno",
        "prijmeni",
        "datum_narozeni",
        "pocet_clanku"
)
VALUES (
        'Jan', 'Novák', '1984-11-03', 17
);
```

První řádek je opět jasný, prostě říkáme "Vlož do uživatelů", další řádky jsou sloupce, ve kterých bude mít nová položka nějaké hodnoty. Sloupec s id zde neuvádíme, ten se vyplňuje sám. Následuje slovo *values* a další výčet prvků v závorkách, tentokrát hodnot. Ty jdou v tom pořadí, jaké jsme uvedli u názvů sloupců. Textové hodnoty jsou v uvozovkách nebo apostrofech, všechny hodnoty oddělujeme čárkami.

POZOR! Pokud vkládáme do SQL dotazu text (zde třeba jméno uživatele), nesmí obsahovat uvozovky, apostrofy a pár dalších znaků. Tyto znaky samozřejmě do textu zapsat můžeme, jen se musí ošetřit, aby si databáze nemyslela, že jde o část dotazu. Ještě se k tomu vrátíme.

Vymazání záznamu

Pokud chceme vymazat záznamy z tabulky pomocí SQL, máme k dispozici příkazy ***DELETE FROM*** a ***TRUNCATE TABLE***.

DELETE FROM

V jazyce SQL vypadá odstranění pomocí příkazu DELETE takto:

```
DELETE FROM `uzivatele` WHERE `uzivatele_id` = 2;
```

Příkaz je jednoduchý, voláme "vymaž z uživatelů", kde se hodnota ve sloupci *uzivatele_id* rovná 2. Zaměřme se na klauzuli WHERE, která definuje podmínku. Potkáme ji i v dalších dotazech. Jelikož zde mažeme podle primárního klíče, jsme si jisti, že vždy vymažeme právě jednoho uživatele.

Podmínku samozřejmě můžeme rozvinout, závorkovat a používat operátory AND (a zároveň) a OR (nebo):

```
DELETE FROM "uzivatele" WHERE ("jmeno" = 'Jan' AND "datum_narozeni" >= '1980-1-1') OR ("pocet_clanku" < 3);
```

Příkaz výše vymaže všechny Jany, kteří byli narození po roce 1980 nebo všechny uživatele, kteří napsali méně než 3 články.

POZOR!, nikdy na klauzuli WHERE nezapomeňte, pokud napíšete jen:

```
DELETE FROM "uzivatele";
```

Budou vymazáni všichni uživatelé v tabulce!

TRUNCATE TABLE

Příkaz ***TRUNCATE TABLE*** vymaže všechny záznamy. V SQLite, narozdíl od jiných variant SQL, tento příkaz sice neexistuje, ale je velmi snadné ho nahradit:

```
DELETE FROM `uzivatele`;
DELETE FROM SQLITE_SEQUENCE WHERE name = "uzivatele";
```

Pokud použijeme ***DELETE FROM*** bez podmínky, SQLite automaticky použije ***TRUNCATE*** optimizer, tedy ***TRUNCATE*** optimalizátor. Udělá tedy příkaz TRUNCATE TABLE automaticky. Druhým řádkem vyresetujeme hodnotu primárního klíče, což se v jiných variantách SQL, kde příkaz TRUNCATE TABLE existuje, děje automaticky.

Proč si tedy pamatovat příkaz ***TRUNCATE TABLE*** (pro jiné varianty SQL), když funguje v podstatě stejně jako ***DELETE FROM*** bez použití podmínky? Příkaz **TRUNCATE TABLE** oproti ***DELETE FROM***:

- je rychlejší,
- nevyžaduje oprávnění DELETE pro tabulku,
- nespouští Triggery (což se občas může hodit),
- vyresetuje AUTO INCREMENT hodnotu zpět na počáteční hodnotu (při použití DELETE FROM se pokračuje další hodnotou v pořadí).

Odstranění tabulky (DROP TABLE)

Kdybychom chtěli odstranit tabulku uživatelů pomocí SQL dotazu, tak spustíme následující příkaz:

```
DROP TABLE "uzivatele";
```

Jelikož se databáze ukládají jako normální soubory, kdekoliv, kde si zvolíte, jejich odstranění se provede jako prosté smazání tohoto souboru.

Editace záznamů (UPDATE)

Databáze umožňuje 4 základní operace, které jsou často označovány zkratkou CRUD (Create, Read, Update, Delete). Jsou to tedy vytvoření záznamu, načtení (vyhledání), update (editace) a vymazání záznamu.

K úpravě slouží SQL dotaz UPDATE, úprava nějakého uživatele by vypadala asi takto:

```
UPDATE "uzivatele"
 SET "prijmeni" = 'Dolejší', "pocet_clanku" = "pocet_clanku" + 1
WHERE "uzivatele_id" = 1;
```

Za klíčovým slovem UPDATE následuje název tabulky, poté slovo SET a vždy název sloupce = hodnota. Můžeme měnit hodnoty více sloupců, pouze se oddělí čárkou. Můžeme dokonce použít předchozí hodnotu z databáze a třeba ji zvýšit o 1, jako v ukázce výše.

SQLite - Výběr dat (vyhledávání)

Výběr dat je klíčovou funkcí databází, umožňuje nám totiž pomocí relativně jednoduchých dotazů dělat i složité výběry dat. Od prostého výběru uživatele podle jeho id (např. pro zobrazení detailů v aplikaci) můžeme vyhledávat uživatele splňující určité vlastnosti, výsledky řadit dle různých kritérií nebo dokonce do dotazu zapojit více tabulek, různé funkce a skládat dotazy do sebe (o tom až v dalších dílech).

Dotazování

Základní dotaz pro výběr všech Janů z tabulky by vypadal takto:

```
SELECT * FROM "uzivatele" WHERE "jmeno" = 'Jan';
```

Příkaz je asi srozumitelný, ta hvězdička označuje, že chceme vybrat všechny sloupce. Dotaz tedy česky zní: "Vyber všechny sloupce z tabulky uzivatele, kde je jméno Jan".

Tabulky mají většinou hodně sloupců a většinou nás zajímají jen nějaké. Abychom databázi nezatěžovali přenášením zbytečných dat zpět do naší aplikace, budeme se snažit vždy specifikovat ty sloupce, které chceme. Dejme tomu, že budeme chtít jen příjmení lidí, co se jmenují Jan a ještě počet jejich článků. Dotaz upravíme:

```
SELECT "prijmeni", "pocet_clanku" FROM "uzivatele" WHERE "jmeno" = 'Jan';
```

Pokud nepotřebujete všechny sloupce, vyjmenujte v ***SELECTu*** ty, jejichž hodnoty vás v tu chvíli zajímají. Vždy se snažte podmínku omezit co nejvíce již na úrovni databáze, ne že si vytaháte celou tabulku do aplikace a tam si ji vyfiltrujete. Výčet sloupců, které má dotaz vrátit, nemá nic společného s dalšími sloupci, které v dotazu používáme. Můžeme tedy vyhledávat podle deseti sloupců, ale vrátit jen jeden.

Stejně jako tomu bylo u ***DELETE***, i zde bude fungovat pouze dotaz:

```
SELECT * FROM "uzivatele";
```

Tehdy budou vybráni úplně všichni uživatelé z tabulky.

U podmínkování platí to samé, jako u ***DELETE***, klauzule ***WHERE*** funguje úplně stejně. Zkusme si to. Vyberme všechny uživatele, narozené po roce 1960 a s počtem článků vyšším než 5:

```
SELECT * FROM "uzivatele" WHERE "datum_narozeni" >= '1960-1-1' AND "pocet_clanku" > 5;
```

Operátory

Základní operátory **=, >, <, >=, <=, !=** určitě umíte použít. V SQL máme ale další, řekněme si o ***LIKE***, ***IN*** a ***BETWEEN***.

LIKE

Like umožňuje vyhledávat textové hodnoty jen podle části textu. Funguje podobně, jako operátor "=" (rovná se), pouze můžeme používat 2 zástupné znaky:

- ***%*** (procento) označuje libovolný počet libovolných znaků.
- **_** (podtržítko) označuje jeden libovolný znak.

Příklady použití:

Najděme příjmení lidí začínající na S:

```
SELECT "prijmeni" FROM "uzivatele" WHERE "prijmeni" LIKE 's%';
```

Zadáme normálně text v apostrofech, pouze na některá místa můžeme vložit speciální znaky. Na velikosti písmen nezáleží (hledání je tedy case-insensitive).

Nyní zkusme najít pětipísmenná příjmení, která mají jako 2. znak O:

```
SELECT "prijmeni" FROM "uzivatele" WHERE "prijmeni" LIKE '_o___';
```

IN == IN umožňuje vyhledávat pomocí výčtu prvků. Udělejme si tedy výčet jmen a vyhledejme uživatele s těmito jmény:

```
SELECT "jmeno", "prijmeni" FROM "uzivatele" WHERE "jmeno" IN ('Petr', 'Jan', 'Kateřina');
```

Operátor ***IN*** se používá ještě u tzv. poddotazů.

BETWEEN

BETWEEN (tedy mezi), není ničím jiným, než zkráceným zápisem podmínky ***">= AND <=".*** Již víme, že i datumy můžeme normálně porovnávat, najděme si uživatele, kteří se narodili mezi lety 1980 a 1990:

```
SELECT "jmeno", "prijmeni", "datum_narozeni" FROM "uzivatele" WHERE "datum_narozeni" BETWEEN '1980-1-1' AND '1990-1-1';
```

Export

Export (nebo také "záloha") je soubor s daty, který nám slouží jako záloha databáze, nebo ho potřebujeme pro migraci, či import databáze.
Export můžeme rozdělit na:

- ***kompletní export*** - soubor bude obsahovat jak strukturu tabulek, tak i jejich data
- ***export struktury*** - soubor bude obsahovat pouze strukturu databáze
    - export může obsahovat všechny nebo pouze vybrané tabulky
- ***export dat*** - soubor bude obsahovat pouze data tabulek
    - tabulky můžeme specifikovat

Řazení (ORDER BY)

Doposud jsme nijak neřešili pořadí nalezených položek, které nám dotaz ***SELECT*** vrátil. Ono vlastně žádné ani neexistovalo, databáze uvnitř funguje pomocí určitých sofistikovaných pravidel (které jsou nad rámec tohoto seriálu) a vrátila nám položky tak, jak se jí to zrovna hodilo. Kdybychom v databázi provedli nějakou změnu a zavolali znovu ten samý dotaz, pořadí by pravděpodobně vypadalo úplně jinak. Databáze nám ale navrácený výsledek samozřejmě seřadit dokáže.

Řadit můžeme podle kteréhokoli sloupce. Když budeme řadit podle id, máme položky v pořadí, v jakém byly do databáze vloženy. Dále můžeme řadit podle číselných sloupců, ale i podle těch textových (řadí se podle abecedy). Řadit můžeme i podle datumu a všech dalších datových typů, databáze si s tím vždy nějak poradí. Pojďme si vybrat úplně všechny uživatele a seřaďme je podle příjmení. Slouží k tomu klauzule ORDER BY (řadit podle), která se píše na konec dotazu:

```
SELECT "jmeno", "prijmeni" FROM "uzivatele" ORDER BY "prijmeni";
```

V dotazu by samozřejmě mohlo být i ***WHERE***, pro jednoduchost jsme vybrali všechny uživatele. Určitě si všimnete, že nám tam chybí Marie Černá a další , kteří se nacházejí až na konci. Je to dáno pouze částečnou podporou UTF 8, aby SQLite zůstalo malé. Buď si to můžeme dopsat, nebo necháme řazení až na kódu v aplikaci.

Řadit můžeme podle několika kritérií (sloupců), pojďme si uživatele seřadit podle napsaných článků a ty se stejným počtem řaďme ještě podle abecedy:

```
SELECT "jmeno", "prijmeni", "pocet_clanku" FROM "uzivatele" ORDER BY "pocet_clanku", "prijmeni"
```

Směr řazení (ASC/DESC)

Určit můžeme samozřejmě i směr řazení. Můžeme řadit vzestupně (výchozí směr) klíčovým slovem ASC a sestupně klíčovým slovem DESC. Zkusme si udělat žebříček uživatelů podle počtu článků. Ti první jich tedy mají nejvíce, řadit budeme sestupně. Ty se stejným počtem článků budeme řadit ještě podle abecedy:

```
SELECT "jmeno", "prijmeni", "pocet_clanku" FROM "uzivatele" ORDER BY "pocet_clanku" DESC, "prijmeni";
```

DESC je třeba vždy uvést, vidíte, že řazení podle příjmení je normálně sestupné, protože jsme DESC napsali jen k *pocet_clanku*.

Limit

Zůstaňme ještě u našeho žebříčku uživatelů podle počtu článků. Takto budeme chtít vypsat 10 nejlepších uživatelů. Když jich bude ale milion, asi není dobrý nápad je všechny vybrat a pak jich v aplikaci použít jen 10 a těch 999 990 zahodit. Dáme databázi limit, tedy maximální počet záznamů, které chceme vybrat. Zároveň uvedeme i řazení. Limit píšeme vždy na konec dotazu:

```
SELECT "jmeno", "prijmeni", "pocet_clanku" FROM "uzivatele" ORDER BY "pocet_clanku" DESC, "prijmeni" LIMIT 10;
```

LIMIT a ***ORDER BY*** lze používat i u dalších příkazů, např. u ***DELETE*** nebo ***UPDATE***. Můžeme si tak pojistit, aby byl vymazán nebo editován vždy jen jeden záznam.

Agregační funkce

Databáze nám nabízí spoustu tzv. agregačních funkcí. To jsou funkce, které nějakým způsobem zpracují více hodnot a jako výsledek vrátí hodnotu jednu.

COUNT (Počet)

vrátí počet řádků v tabulce, splňující nějaká kritéria. Spočítejme, kolik z uživatelů napsalo alespoň jeden článek:

```
SELECT COUNT(*) FROM "uzivatele" WHERE "pocet_clanku" > 0;
```

Na COUNT se ptáme pomocí SELECT, není to příkaz, je to funkce, která se vykoná nad řádky a její výsledek je vrácen selectem. Funkce má stejně jako v jiných programovacích jazycích (alespoň ve většině z nich) závorky. Ta hvězdička v nich znamená, že nás zajímají všechny sloupce. Můžeme totiž počítat třeba jen uživatele, kteří mají vyplněné jméno (přesněji kteří ho nemají NULL, ale to nechme na další díly).

Přenos dat z databáze je náročný a může zpomalovat aplikaci. COUNT přenáší jen jedno jediné číslo. Nikdy nepočítejte pomocí přenosu hodnot z databáze a následníém součtu, ale pouze funkcí COUNT!

AVG()

označuje průměr z daných hodnot. Podívejme se, jaký je průměrný počet článků na uživatele:

```
SELECT AVG("pocet_clanku") FROM "uzivatele";
```

SUM()

vrací součet hodnot. Podívejme se, kolik článků napsali dohromady lidé narození po roce 1980:

```
SELECT SUM("pocet_clanku") FROM "uzivatele" WHERE "datum_narozeni" > '1980-1-1';
```

MIN()

vrátí minimum (nejmenší hodnotu). Najděme nejnižší datum narození:

```
SELECT MIN("datum_narozeni") FROM "uzivatele";
```

MAX()

vrátí maximum (největší hodnotu)., najděme maximální počet článků od 1 uživatele:

```
SELECT MAX("pocet_clanku") FROM "uzivatele";
```

Seskupování (Grouping)

Položky v databázi můžeme seskupovat podle určitých kritérii. Seskupování používáme téměř vždy spolu s agregačními funkcemi. Pojďme seskupit uživatele podle jména:

```
SELECT "jmeno" FROM "uzivatele" GROUP BY "jmeno";
```

Každé jméno je zde zastoupeno jen jednou, i když je v databázi vícekrát. Přidejme nyní kromě jména i počet jeho zastoupení v tabulce, uděláme to pomocí agregační funkce ***COUNT(*)***

```
SELECT "jmeno", COUNT(*) FROM "uzivatele" GROUP BY "jmeno";
```

AS

Pro zjednodušení si můžeme v dotazu vytvořit aliasy, tedy přejmenovat třeba nějaký dlouhý sloupec, aby byl dotaz přehlednější. S tímto se ještě setkáme u dotazů přes více tabulek, kde je to velmi užitečné. U tabulek AS používáme ke zjednodušení operací uvnitř dotazu. U sloupců se AS používá k tomu, aby aplikace viděla data pod jiným názvem, než jsou skutečně v databázi. To může být užitečné zejména u agregačních funkcí, protože pro ně v databázi není žádný sloupec a mohlo by se nám s jejich výsledkem špatně pracovat. Upravme si poslední dotaz:

```
SELECT "jmeno", COUNT(*) AS "pocet" FROM "uzivatele" GROUP BY "jmeno";
```

Datové typy

Hned na začátku seriálu jsme se setkali s několika datovými typy. Tehdy jsem vám s nimi nechtěl motat hlavu. Řeč byla o typech INTEGER a TEXT. Databáze (konkrétně zde SQLite) jich má sice ještě několik, ale na rozdíl od jiných SQL databází jich je poměrně málo.

Celé číslo – INTEGER

Dle velikosti samotného čísla, které se ukládá, se použije 1, 2, 3, 4, 5, 6, 7 nebo 8 bajtů.
Pokud hledáte datový typ boolean (hodnoty true/false), tak k ukládání této hodnoty se používá rovněž INTEGER - 0 = false, 1 = true.

Reálné číslo - REAL

K ukládání reálných čísel se používá 64 bitů, přesnost je zhruba 15 až 16 čísel a číslo může nabývat hodnot +-5.0 * 10^−324 až +-1.7 * 10^{308}.

Text

Datový typ TEXT se používá k ukládání řětezců znaků dle použitého kódování databáze (UTF-8, UTF-16BE or UTF-16LE).
Používá se taktéž k ukládání času ve formátu 'rrrr-mm-dd hh:mm:ss.sss', ale jde použít i datový typ INTEGER ve formátu Unix - počet sekund od 1. ledna 1970 0:00, nebo typ REAL.

BLOB

Obecná data v binárním tvaru, použití analogické k typům TEXT. Umožňují do databáze ukládat např. obrázky nebo zvuky.

Hodnota NULL

Datové typy v databázích se malinko odlišují od datových typů, jak je známe v programovacích jazycích. Zatímco třeba v Céčku může mít int hodnoty jen nějakých -32.000 až +32.000 a nic kromě toho, databázový INTEGER může nabývat i hodnoty NULL. NULL nemá vůbec nic společného s nulou (0), označuje to, že hodnota nebyla ještě zadána. Filozofie databází je takto postavena, nezadané hodnoty mají výchozí hodnotu NULL (pokud jim nenastavíme jinou) a každý datový typ má kromě hodnot, které bychom v něm očekávali, navíc možnou hodnotu NULL. Datovému typu tuto hodnotu můžeme i zakázat, více dále.
Pokud např. příkazem INSERT vložíme uživatele a vyplníme jen některé hodnoty, do dalších hodnot se vloží NULL. Zkusme si to:

```
INSERT INTO "uzivatele" ("jmeno", "prijmeni"`) VALUES ('Pan', 'Neuplny');
```

Přínos hodnoty NULL je v tom, že poznáme, jestli byla hodnota zadána. Např. při zadávání čísla neexistuje hodnota, podle které bychom poznali, že číslo není zadáno. Kdybychom si k tomuto určili hodnotu nula (0), nevíme, jestli uživatel číslo nezadal nebo zadal právě nulu. NULL mimo jiné i šetří místo v databázi, kde narozdíl od výchozích hodnot nezabírá místo.

NULL na straně aplikace

Již jsme si řekli o tom, že programovací jazyky hodnotu NULL zpravidla nemají (tedy ty staticky typované). V jazycích jako je třeba dynamické PHP nemusíme datový typ vůbec řešit, i když je dobré vědět, že se na NULL můžeme zeptat, když to budeme potřebovat. V jazycích typovaných, jako je třeba Java nebo C#, musíme použít jiné datové typy. V C# můžeme kterýkoli datový typ označit jako NULLovatelný a on pochopí, že v něm může být i NULL. V Javě budeme používat datové typy s velkými písmeny, tedy např. pro čísla místo int použijeme Integer.

Upřesňující informace k datovým typům

K datovým typům (chcete-li ke sloupcům) můžeme uvést několik upřesňujících informací. Již jsme se setkali s AUTO_INCREMENT. Podívejme se na další.

AUTOINCREMENT:

Při vkládání řádku dejte této položce hodnotu NULL a systém jí automaticky přidělí hodnotu o 1 větší než dal minulému řádku (přírůstek se teoreticky dá změnit, ale tím se teď nebudeme zatěžovat). Výborná věc pro pohodlnou tvorbu unikátních identifikačních klíčů.

UNIQUE

Říká, že nesmí existovat víc řádků, které mají v této položce stejnou hodnotu (s výjimkou hodnoty NULL). Smysl to má pouze u klíčů.

NOT NULL

Tahle hodnota nesmí být prázdná - nepůjde do ní vložit hodnota NULL.

PRIMARY KEY

Tím se určí, že se tenhle sloupec (v každé tabulce max. jeden) bude používat jako klíč. Vhodné pro nějaké relativně krátké identifikační kódy, podle kterých budeme řádky nejčastěji hledat. Primární klíč je vždy NOT NULL a UNIQUE; i když to nenařídíme, dostane tyhle vlastnosti implicitně.

DEFAULT hodnota

Výchozí hodnota, kterou položka dostane, když ji při vkládání řádku neuvedeme. Nefunguje na Texty, Bloby a položky s auto_incrementem.

Nová slova upřesňující datový typ se vkládají za něj, stejně jako tomu bylo u AUTO_INCREMENT. Uveďme si nějaký příklad:

```
CREATE TABLE "uzivatele" (
        "uzivatele_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "jmeno" TEXT NOT NULL,
);
```

Dynamické typování

Naprostá většina SQL databází používá k ukládání statické typování, při vytvoření tabulky zadáte datový typ, který má daný sloupec obsahovat, a datový typ položky se určuje pouze podle toho. SQLite naopak používá dynamické typování, kdy se určuje datový typ podle samotné položky (hodnoty), kterou tam nahráváte.
Až na sloupec s parametry *INTEGER PRIMARY KEY* můžete ukládát co chcete kamkoliv. Možná vás teď napadá, že při vytváření tabulky není třeba zadávat typi pro jednotlivé sloupce. Ano není, je to naprosto validní SQL příkaz v SQLite, avšak doporučuji vám, abyste datové typy definovali a dodržovali, potom víte, v jakém formátu tam ukládat data a v jakém je očekávat třeba ve své aplikaci. Na tento typ se také SQLite pokusí převést data, takže pokud jste sloupec označili jako INTEGER a vložíte do něj '15.0', uloží se číslo 15.

Kompatibilita s ostatními databázemi

V rámci kompatibility s ostatními SQL databázemi mění SQLite podobné datové typy ostatních databází na své vlastní. Naše tabulka uživatelů by se vytvořila v SQL takto:

```
CREATE TABLE `uzivatele` (
        `uzivatele_id` int AUTO_INCREMENT,
         `jmeno` varchar(60),
        `prijmeni` varchar(60),
        `datum_narozeni` date,
        `pocet_clanku` int,
        PRIMARY KEY (`uzivatele_id`)
);
```

Pokud bychom tenhle kód rozběhli na SQLite, tak by byl překvapivě úspěšně proveden, a byl by chápán stejně, jako kdybychom zadali tenhle kód:

```
CREATE TABLE "uzivatele" (
        "uzivatele_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "jmeno" TEXT,
         "prijmeni" TEXT,
        "datum_narozeni" TEXT,
        "pocet_clanku" INTEGER
);
```

Tahle vlastnost vám hodně ulehčí práci, když přecházíte z jiné databáze na SQLite a chcete si přenést i data.

SQLite - Dotazy přes více tabulek (JOIN)

Příprava tabulek a dat

Dnes se zaměříme na dotazy přes více tabulek. Pojďme si nejprve nějaké tabulky vytvořit. Bohatě nám budou stačit uživatelé a články. Protože uživatel bude vypadat trochu jinak, než nám vypadal doteď, založíme si tabulku *uzivatele* znovu:

```
CREATE TABLE "uzivatele" (
        "uzivatele_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "prezdivka" TEXT,
        "email" TEXT,
        "heslo" TEXT
);
```

Do uživatelů si rovnou nějaké vložíme:

```
INSERT INTO "uzivatele" ("prezdivka", "email", "heslo") VALUES
('Míša', 'misaslavikova@gmail.com', 'dGg#@$DetA53d'),
('David', 'capkadavid@seznam.cz', '$#fdfgfHBKBKS'),
('Denny', 'denny@hotmail.com', 'Jmls_aSW2RFss'),
('Ema', 'ema@centrum.cz', 'fw8QT32qmcsld');
```

Články

Článek bude propojen s uživatelem, který ho napsal, tedy s jeho autorem. Tabulky propojíme tak, že do tabulky *clanky* přídáme sloupec s id autora. Tam bude hodnota id uživatele (tedy primární klíč z tabulky *uzivatele*), který článek napsal.
Hovoříme o vazbě 1:N (1 uživatel má N (několik) článků a každý článek patří právě jednomu uživateli). Část (zde článek) má vždy uložené id celku (zde uživatel) kam patří.
Článek bude obsahovat (opět kromě svého id) id autora, krátký popis, url, klíčová slova, titulek, obsah a datum publikace. Založme si tabulku *clanky*:

```
CREATE TABLE "clanky" (
        "clanky_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "autor_id" INTEGER,
        "popis" TEXT,
        "url" TEXT,
        "klicova_slova" TEXT,
        "titulek" TEXT,
        "obsah" TEXT,
        "publikovano" TEXT
);
```

Dále přidáme články a k nim přiřadíme uživatele jako autory. Vzal jsem 4 články zde z ITnetwork, které jsem značně zkrátil a zjednodušil. Dotaz bude následující:

```
INSERT INTO "clanky" ("autor_id", "popis", "url", "klicova_slova", "titulek", "obsah", "publikovano") VALUES
(1, 'Co je to algoritmus? Pokud to nevíte, přečtěte si tento článek.', 'co-je-to-algoritmus', 'algoritmus, co je to, vysvětlení', 'Algoritmus', 'Když se bavíme o algoritmech, pojďme se tedy shodnout na tom, co ten algoritmus vůbec je. Jednoduše řečeno, algoritmus je návod k řešení nějakého problému. Když se na to podíváme z lidského pohledu, algoritmus by mohl být třeba návod, jak ráno vstát. I když to zní jednoduše, je to docela problém. Počítače jsou totiž stroje a ty nemyslí. Musíme tedy dopodrobna popsat všechny kroky algoritmu. Tím se dostáváme k první vlastnosti algoritmu - musí být elementární (skládat se z konečného počtu jednoduchých a snadno srozumitelných kroků, tedy příkazů). "Vstaň z postele" určitě není algoritmus. "Otevři oči, sundej peřinu, posaň se, dej nohy na zem a stoupni si" - to už zní docela podrobně a jednalo by se tedy o pravý algoritmus. My se však budeme pohybovat v IT, takže budeme řešit problémy jako seřaď prvky podle velikosti nebo vyhledej prvek podle jeho
```

obsahu. To jsou totiž 2 základní úlohy, které počítače dělají nejčastěji a které je potřeba dokonale promýšlet a optimalizovat, aby trvaly co nejkratší dobu. Z dalších příkladů algoritmů mě napadá třeba vyřeš kvadratickou rovnici nebo vyřeš sudoku.', '2012-03-21'),

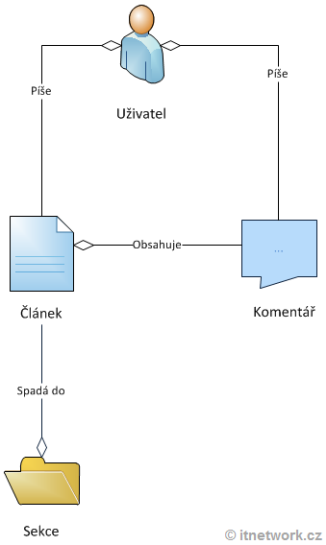
(2, 'Bakterie jsou obdoba buněčného automatu v kombinaci s hrou.', 'bakterie-bunecny-automat', 'bakterie, automat, algoritmus', 'Bakterie', 'Bakterie jsou obdoba buněčného automatu, který vymyslel britský matematik John Horton Conway v roce 1970. Celou tuto hru řídí čtyři jednoduchá pravidla:/n/n1. Živá bakterie s méně, než dvěma živými sousedy umírá./n2. Živá bakterie s více, než třemi živými sousedy umírá na přemnožení./n3. Živá bakterie s dvoumi nebo třemi sousedy přežívá beze změny do další generace./n4. Mrtvá bakterie, s přesně třemi živými sousedy, opět ožívá./nTyto zdánlivě naprosto primitivní pravidla dokáží za správného počátečního rozmístění bakterií vytvořit pochodující skupinky, shluky "vystřelující" pochodující pětice, překvapivě složité souměrné exploze, oscilátory (periodicky kmitající skupinky), či nekonečnou podívanou na to, jak složité a dokonalé obrazce dokáží tyto dvě podmínky vytvořit. Celý program je koncipován jako hra, máte za úkol vytvořit co nejdéle žijící kolonii. <a href="soubory/bakterie.zip" ', '2012-02-14'),

(3, 'Cheese Mouse je oddechová plošinovka.', 'cheese-mouse-oddechova-plosinovka', 'myš, sýr, hra', 'Cheese Mouse', 'Cheese mouse je plošinovka s "horkou ostrovní atmosférou", kde ovládáte myš a musíte se dostat k sýru. V tom vám ale brání nejrůznější nástrahy a nepřatelé jako hadi, krysy, pirane, ale i roboti, mumie a nejrůznější havěť. Hru s několika petrobarevnými světy jsem dělal ještě na základní škole s Veisenem a může se pochlubit 2. místem v Bonusweb game competition, kde vyhrála 5.000 Kč. Vznikala v Game makeru o letních prázdninách, ještě v bezstarostném dětství, což značně ovlivnilo její grafickou stránku. Rád si ji občas zahraji na odreagování a zlepšní nálady. <a href="soubory/cheesemouse.zip" />', '2004-06-22'),

(2, 'Pacman je remake kultovní hry.', 'pacman-remake', 'pacman, remake, pampuch, hra, zdarma', 'Pacman', 'Jedná se o naprosto základní verzi této hry s editorem levelů, takže si můžete vytvořit svá vlastní kola. Postupem času ji hodlám ještě trochu upravit a přidat nějaké nové prvky, fullscreen a lepší grafiku. Engine hry bude také základem mého nového projektu Geckon man, který je zatím ve fázi psaní scénáře. <a href="soubory/pacman.zip" />', '2011-06-03');

Konceptuální model

V následujících dílech si tedy v databázi vytvoříme takový zjednodušený ITnetwork. Pobavme se nejprve o tom, jak to bude vypadat. Dnes stihneme pochopitelně jen malou část. Protože obrázek někdy řekne více, než tisíc slov, začněme právě jím.



Co vidíte je tzv. konceptuální model. Je vytvořený pomocí notace (grafického jazyka) UML a v praxi se takovéto diagramy velmi často tvoří předtím, než začneme psát nějaký kód. Dobře si tak nejprve rozmyslíme, co že to vlastně chceme udělat.

Vidíme, že v systému figuruje uživatel, který může psát komentáře a články. Články spadají do sekcí. Jedná se tedy o databázi takového velmi jednoduchého redakčního systému, který si díky ITnetwork jistě dokážete představit.

Dotazy přes více tabulek

Nyní máme v databázi články a k nim přiřazené uživatele. Pojďme si udělat dotaz přes tyto 2 tabulky, získejme články a k nim připojme přezdívky jejich uživatelů. Slovo připojme jsem nepoužil náhodou, příkaz pro spojení 2 tabulek se totiž jmenuje JOIN. Napišme si dotaz a poté si ho vysvětleme. Dotazy již budeme psát na více řádků, abychom se v tom vyznali:

```
SELECT "titulek", "prezdivka"
FROM "clanky"
JOIN "uzivatele" ON "autor_id" = "uzivatele_id"
ORDER BY "prezdivka";
```

Na prvním řádku příkazu ***SELECT*** pracujeme se sloupci úplně stejně, jako kdyby byly v jedné tabulce, jednoduše vyjmenujeme, co nás zajímá. Jelikož vybíráme články a k nim připojujeme uživatele, budeme vybírat z tabulky *clanky*. Připojení dat z jiné tabulky uděláme pomocí příkazu JOIN, kde uvedeme tabulku, kterou připojujeme, a poté klauzuli ON. Klauzule ON je podobná jako WHERE, jen platí pro připojovanou tabulku a ne pro tu, ze které primárně vybíráme. V podmínce uvedeme, aby se ke každému článku připojil ten uživatel, jehož ***uzivatele_id*** je uvedeno ve sloupci ***autor_id***. Výsledek jsme seřadili podle přezdívky uživatelů. Kdybychom chtěli jen nějaké články, normálně bychom před ***ORDER*** BY uvedli ještě ***WHERE***, jak jsme zvyklí.

INNER JOIN a OUTER JOIN

INNER (vnitřní) a ***OUTER*** (vnější) ***JOIN*** jsou 2 typy příkazu ***JOIN***. Fungují úplně stejně, jediný rozdíl je v tom, co se stane, když položka, na kterou se vazba odkazuje, neexistuje.

INNER JOIN

Pokud uvedeme v SQL dotazu pouze ***JOIN***, pokládá ho SQLite databáze za tzv. ***INNER JOIN***. Pokud by v našem případě neexistoval uživatel s id, které je u článku uvedeno, článek bez uživatele by vůbec nebyl ve výsledcích obsažen. Vazba je nerozdělitelná.

Pojďme si to zkusit, přidejme si článek, který bude odkazovat na id neexistujícího uživatele:

```
INSERT INTO "clanky" ("autor_id", "popis", "url", "klicova_slova", "titulek", "obsah", "publikovano") VALUES
(99, 'Článek s neexistujím uživatelem slouží pro vyzkoušení typů JOINů.', 'clanek-bez-autora', 'clanek, join, autor, chybejici', 'Článek bez autora', 'Tento článek je přiřazen neexistujícímu uživateli s ID 99 a slouží k vyzkoušení různých typů JOINů v SQLite databázi.', '2012-10-21');
```

Vložený článek se odkazuje na uživatele s *uzivatele_id* 99, který v databázi není. Spusťme si nyní znovu náš SQL dotaz s JOINem. Pro přehlednost je lepší uvést, že chceme INNER JOIN.

```
SELECT "titulek", "prezdivka"
FROM "clanky"
INNER JOIN "uzivatele" ON "autor_id" = "uzivatele_id"
```

```
ORDER BY "prezdivka";
```
Výsledek je stále stejný, článek bez autora mezi výsledky není.

LEFT OUTER JOIN

Vnější JOINy umožňují vybírat i ty výsledky, které se nepodařilo spojit z důvodu chybějících položek. SQLite umí pouze ten nejčastěji používaný - ***LEFT JOIN***, který výsledek uzná, pokud existuje levá část vazby (zde článek) a pravá (ta připojovaná, zde uživatel) neexistuje. Do hodnot sloupců z připojované části se vloží NULL.

```
SELECT "titulek", "prezdivka"
FROM "clanky"
LEFT JOIN "uzivatele" ON "autor_id" = "uzivatele_id"
ORDER BY "prezdivka";
```
Článek se vybral, i když se nepodařilo vybrat pravou část (tedy tu připojovanou, uživatele). Před spojováním tabulek je dobré se zamyslet, zda nastane případ, kdy se spojení nepodaří a co v tom případě chceme dělat. U článku by se toto v reálu stát asi nemělo.

Wherování

Teoreticky se můžeme JOINům vyhýbat a používat místo nich jednoduše jen klauzuli ***FROM*** a ***WHERE***. Ve ***FROM*** uvedeme více tabulek oddělených čárkami. Ve ***WHERE*** specifikujeme podmínku spojení tabulek. Databáze si v ideálním případě takovýto dotaz nejprve převede na ***INNER JOIN*** a poté ho zpracuje.

```
SELECT "titulek", "prezdivka"
FROM "clanky", "uzivatele"
WHERE "autor_id" = "uzivatele_id"
ORDER BY "prezdivka";
```
Výsledek je tedy stejný jako při INNER JOINu.

Nevýhoda wherování je, že tak neuděláme všechny JOINy a v určitých případech mohou být dotazy méně optimalizované. Nikdy nevíme, jak dotaz databáze optimalizuje a optimalizace se bude lišit podle typu databáze. Tento způsob berte spíše jako zajímavost a nepoužívejte ho.

SQLite - Další dotazy a vazba M:N

Tabulka komentáře

Pokračujme v našem redakčním systému a vytvořme si tabulku *komentáře*. Komentář se (podobně jako článek) váže na uživatele. Váže se ale také na článek. Máme zde tedy dvě vazby 1:N. Jeden článek má N komentářů, jeden uživatel má N komentářů. Komentář patří vždy pouze jednomu uživateli a jednomu článku.

Jelikož komentář je část a patří do dvou celků (k článku a k uživateli), bude obsahovat 2 sloupce s id článku a id komentáře. Těmto sloupcům s id položky z cizí tabulky říkáme cizí klíče. Již je známe z minula (u článku byl cizí klíč uživatele), jen jsme si neřekli, že se jim tak říká. Kromě nich bude mít komentář text a datum.

```
CREATE TABLE "komentare" (
        "komentare_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "clanek_id" INTEGER,
        "uzivatel_id" INTEGER,
        "obsah" TEXT,
        "datum" TEXT );
```
Vložme si nějaké komentáře:

```
INSERT INTO "komentare" ("uzivatel_id", "obsah", "datum", "clanek_id") VALUES
(4, 'Super článek!', '2012-04-06', 1),
(4, 'Jak je tedy přesně ta podmínka pro vznik bakterie?', '2011-01-28', 2),
(1, 'Zasekla jsem se v této hře, kde najdu klíč do 3. levelu?', '2011-09-30', 3),
(4, 'Jak rozjedu plošinu v 5. levelu?', '2010-08-01', 3),
(1, 'Umřel jsem a nemám hru uloženou, co mám dělat?', '2012-04-14', 4),
(3, 'Dobrá hra!', '2012-04-06', 4), (3, 'Nerozumím tomu!', '2011-04-06', 1),
(2, 'Super článek!', '2012-05-06', 1);
```

Pojďme si zkusit vypsat všechny komentáře spolu s jejich autory a články, ke kterým patří. JOINy už umíme, tento dotaz bude obdobný, jen bude rovnou přes 2 tabulky najednou, čili s dvěma JOINy. JOINů můžeme mít v dotazu samozřejmě kolik chceme, ale měli bychom pamatovat na to, že to nejsou pro databázi úplně jednoduché operace:

```
SELECT "uzivatele"."prezdivka", "komentare"."obsah", "clanky"."titulek"
FROM "komentare"
INNER JOIN "uzivatele" ON "uzivatele"."uzivatele_id" = "komentare"."uzivatel_id"
INNER JOIN "clanky" ON "clanky"."clanky_id" = "komentare"."clanek_id"
ORDER BY "komentare"."datum";
```
Všimněte si, že jsme všechny sloupce předsadili názvem tabulky. Mělo by se to tak dělat vždy, jen minule jsme to pro jednoduchost zanedbali. Může se nám totiž jednoduše stát, že se nám vyskytnou stejné názvy sloupců v různých tabulkách, které zrovna propojujeme. Databáze by nám za to vyhubovala, protože by nevěděla, který sloupec máme na mysli. Zde konkrétně se jmenuje *obsah* obsah komentáře i obsah článku. U složitější struktury databáze se toto stává se sloupci jako datum, id, autor...

U složitějších dotazů přes více tabulek může být výhodné použít aliasy. Aliasy již také umíme, dělají se přes klíčové slovo AS. Použijme je v tomto dotazu:

```
SELECT "u"."prezdivka", "k"."obsah", "c"."titulek"
FROM "komentare" AS "k"
INNER JOIN "uzivatele" AS "u" ON "u"."uzivatele_id" = "k"."uzivatel_id"
INNER JOIN "clanky" AS "c" ON "c"."clanky_id" = "k"."clanek_id"
ORDER BY "k"."datum";
```
Dotaz vypadá mnohem přehledněji, nemusíme opisovat názvy tabulek. Zkrátili jsme si je, zde jen na počáteční písmena.

Sekce

Pokračujme ve struktuře redakčního systému. Články se řadí do sekcí, ty jsou uloženy v tabulce sekce. Je tu však malý háček. Jedna sekce může obsahovat několik článků. Jeden článek však může také patřit do několika sekcí. Pro účely redakčního systému by samozřejmě stačilo, aby článek spadal vždy jen do jedné sekce. Tak bychom se ale nic nenaučili Narážíme na vazbu M:N.

Vazba M:N

Vazbu M:N jsme si již vysvětlili, dalším příkladem by mohli být třeba student a předmět. Každý student chodí na několik předmětů a každý předmět má několik studentů, kteří na něj dochází. Pojďme si založit tabulku sekcí. Bude velmi triviální, protože v ní budou jen 2 sloupce. Jeden s id sekce a druhý s jejím názvem:

```
CREATE TABLE "sekce" (
        "sekce_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "nazev" TEXT
);
```

Naplňme si ji daty:

```
INSERT INTO "sekce" ("nazev") VALUES
('Algoritmy'),
('Hry');
```

Databáze jako taková vazbu M:N neumí. To pro nás ale není překážkou a běžně se to obchází vytvořením tzv. vazební tabulky. Vazební tabulka nenese sama o sobě žádná data a slouží pouze k propojení dvou tabulek. Každý řádek vazební tabulky bude obsahovat id článku a id sekce, tak je spolu propojí. Díky tomu můžeme dotazem zjistit jaké články jsou v sekci nebo do kterých sekcí článek patří. Založme si vazební tabulku, pojmenujeme ji clanek_sekce:

```
CREATE TABLE "clanek_sekce" (
        "clanek_sekce_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "clanek_id" INTEGER,
        "sekce_id" INTEGER
);
```

Nyní ji naplníme daty, která nám články a sekce propojí:

```
INSERT INTO "clanek_sekce" ("clanek_id", "sekce_id") VALUES
(1, 1),
(2, 1),
(2, 2),
(3, 2),
(4, 2);
```

A zkusme si dotaz. Vypišme si články v sekci Algoritmy. Vybereme články, ty propojíme pomocí tabulky *clanek_sekce* se sekcí.

```
SELECT "c"."url", "c"."titulek"
FROM "clanky" AS "c"
INNER JOIN "clanek_sekce" AS "cs" ON "cs"."clanek_id" = "c"."clanky_id"
INNER JOIN "sekce" AS "s" ON "cs"."sekce_id" = "s"."sekce_id"
WHERE "s"."nazev" = 'Algoritmy';
```

Dotaz výše by byl na webu opravdu použit pro vypsání obsahu sekce. Podle vazební tabulky jsme propojili články se sekcí. Vlastně jsme připojili ty řádky vazební tabulky, které spojují daný článek a ten článek potom k jeho sekci.

SQLite – Poddotazy

Poddotazem se myslí dotaz, který je součástí nějakého dalšího dotazu. Správně, dotazy můžeme vkládat do sebe. Výstup nějakého dotazu tedy můžeme použít jako vstup pro další dotaz, protože se jedná v zásadě také o tabulku s daty. V praxi se toto často používá a máme tak možnost dosáhnout na straně databáze výsledků, které bychom jinak museli pomaleji sestavovat až na straně aplikace. Výkonově jsou poddotazy samozřejmě náročné, ale pokud jsou správně optimalizované (viz další díly), měly by být rychlejší, než použití více samostatných dotazů nebo JOINů.

Zkuste se zamyslet nad tím, jak byste vypsali články uživatele s přezdívkou "David". Asi byste vymysleli následující dotaz:

```
SELECT "c"."titulek"  FROM "clanky" AS "c"
INNER JOIN "uzivatele" AS "u" ON "u"."uzivatele_id" = "c"."autor_id"
WHERE "u"."prezdivka" = 'David';
```

Stejného výsledku ale můžeme dosáhnout i pomocí poddotazu:

```
SELECT "titulek"  FROM "clanky"
WHERE "autor_id" = ( SELECT "uzivatele_id" FROM "uzivatele"
        WHERE "prezdivka" = 'David' LIMIT 1 );
```

U složených dotazů se nejprve vykonají vnitřní poddotazy a pak tento výsledek se vloží do vnějšího dotazu.

Zkusme si další příklad pomocí poddotazu. Najděme uživatele s největším počtem článků.

```
SELECT  "u"."prezdivka",
        ( SELECT  COUNT(*) FROM "clanky" AS "c"
        WHERE ("c"."autor_id" = "u"."uzivatele_id")  AS "pocet"
FROM "uzivatele" AS "u"
ORDER BY "pocet" DESC LIMIT 1; );
```

Zde jsme dokonce ve vnitřním dotazu (v poddotazu) použili *u*, které je z vnějšího dotazu. Jelikož ***COUNT(*)*** nám vrátí jednu hodnotu, můžeme ho jednoduše přidat mezi hodnoty, které si přejeme navrátit. Poddotaz zde tedy není v podmínce, ale ve výčtu hodnot.
Pokud poddotaz nějak souvisí s vnějším dotazem (viz minulý příklad), hovoříme o tzv. korelovaném poddotazu. Pokud stojí úplně mimo a spustí se jen jednou, hovoříme o nekorelovaném poddotazu.

Pojďme si ještě zkusit vypsat nezařazené články, tedy ty, které nepatří do žádné sekce:

```
SELECT  "c"."clanky_id",  "c"."titulek"
FROM "clanky" AS "c"
WHERE  (SELECT COUNT(*)
        FROM "clanek_sekce"
        WHERE ("clanek_sekce"."clanek_id" = "c"."clanky_id")) = 0;
```

V našem redakčním systému píší uživatelé komentáře. Bylo by skvělé, kdyby se uživatel mohl podívat na všechny komentáře, které byly napsány od doby, co na web naposledy něco napsal (tedy zjednodušeně od jeho poslední aktivity). Použijeme na datum operátor > (větší) a zjistěme to pro uživatele s přezdívkou 'Denny':

```
SELECT  "obsah",  "datum"  FROM "komentare"
WHERE "datum" > (
        SELECT "datum"  FROM "komentare"
```

```
        INNER JOIN "uzivatele" ON ("uzivatele"."uzivatele_id" = "komentare"."uzivatel_id")
        WHERE ("uzivatele"."prezdivka" = 'Denny')
        ORDER BY "datum" DESC LIMIT 1 )
ORDER BY "datum" DESC;
```

V dotazu vyhledáme komentáře, jejichž datum je pozdější, než všechny komentáře, které daný autor napsal.

Víceřádkové operátory

Až doteď jsme se snažili, aby nám poddotaz vrátil vždy jen 1 výsledek. Bylo to kvůli operátorům, které jsme doposud znali a které uměly pracovat jen s jedním řádkem ***(>, <, =, <=, >=...).*** Databáze (SQLite) má ale další operátory, které s více řádky pracovat umí. S jedním takovým jsme se již setkali, byl to operátor ***IN***.

Operátor IN

IN umožňuje otestovat přítomnost hodnoty ve výčtu hodnot. Předtím jsme si možné hodnoty vypsali ručně, nyní můžeme použít výsledek poddotazu. Pojďme si upravit první příklad tak, aby fungoval i pro více Davidů:

```
SELECT "titulek"
FROM "clanky"
WHERE "autor_id" IN (
        SELECT "uzivatele_id"
        FROM "uzivatele"
        WHERE "prezdivka" = 'David'
);
```

Změna je jednoduchá, místo **=** jsme použili **IN** a odstranili jsme ***LIMIT 1***.
Použití operátoru IN a poddotazu může být intuitivnější a pro databázi i rychlejší, než používání JOINů nebo dokonce wherování.

Dalšími operátory pro více řádků jsou někdy operátory ***ALL*** a ***ANY***.
Tyto operátory se bohužel v SQLite nevyskytují, jdou ale bez problémů opsat pomocí normálních poddotazů.

Operátor EXISTS

Operátor ***EXISTS*** se používá u korelovaných poddotazů. Umožňuje z dalšího poddotazu zjistit, zda vrátil nějakou hodnotu.

Ještě jsme podmínku nenegovali, dělá se to jednoduše pomocí operátoru ***NOT***. Můžeme tak znegovat kteroukoli podmínku v dotazu. Vyberme uživatele, který nemá žádné články. Uděláme to pomocí poddotazu a operátoru EXISTS. Zeptáme se, zda neexistuje článek, jehož autor (uživatel) je ve vnějším dotazu:

```
SELECT "prezdivka"
FROM "uzivatele" AS "u"
WHERE NOT EXISTS (
        SELECT * FROM "clanky" AS "c"
        WHERE "c"."autor_id" = "u"."uzivatele_id"
);
```

SQLite - Změna struktury, transakce a pohledy

Změna struktury databáze

SQLite umožňuje měnit strukturu již existující databáze tak, aby v ní zůstala data. Slouží k tomu SQL příkaz ***ALTER***.

Změna struktury tabulky

U tabulky můžeme modifikovat několik věcí, nejdůležitější pro nás bude přidání, změna a odebrání sloupce. Přidejme si do tabulky *komentare* nový sloupec *palce*, aby návštěvníci mohli komentáře hodnotit:

```
ALTER TABLE "komentare" ADD COLUMN "palce" INTEGER;
```

Pro modifikace a odstranění sloupců je jednodušší používat DB Browser, protože tyto příkazy nejsou v SQLite obsažené. Pokud to přesto chcete udělat pomocí SQL, tak si musíte vytvořit novou tabulku s požadovanou strukturou, přenést dat, odstranit původní tabulky a přejmenovat novou tabulku.
Ukážeme si to na odstranění sloupce palce:

```
CREATE TABLE "komentare_bez_palce" (
        "komentare_id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "clanek_id" INTEGER,
        "uzivatel_id" INTEGER,
        "obsah" TEXT,
        "datum" TEXT
);
INSERT INTO "komentare_bez_palce"
        SELECT "komentare_id","clanek_id","uzivatel_id","obsah","datum"
        FROM "komentare";
DROP TABLE "komentare";
ALTER TABLE "komentare_bez_palce" RENAME TO "komentare";
```

Z příkazů je pro vás nový pouze poslední, který zajišťuje právě přejmenování tabulky:

```
ALTER TABLE "komentare_bez_palce" RENAME TO "komentare";
```

Někdy se nám může stát, že budeme chtít určit jinou počáteční hodnotu primárního klíče, než 1. Řekněme například, že bychom chtěli, aby náš primární klíč začínal hodnotou 1234:

```
UPDATE SQLITE_SEQUENCE SET seq = 1233 WHERE name = 'uzivatele';
```

Hodnota klíče, kterou nastavujeme je opravdu o 1 menší, než ta, kterou chceme.
SQLite je navíc (co se týká cizích klíčů) poměrně chytré. Co se například stane, pokud nastavíme hodnotu nejprve na 1234 a následně na 15? Zprvu nic, ale když hodnota z 15 "doroste" až na hodnotu 1234, reagují na to různé SŘBD různě. SQLite má asi jedno z nejpřívětivějších řešení, protože již vložené klíče automaticky přeskakuje. Místo přepsání hodnoty, či vyhození chyby tak vloží položku pod číslem 1235

Transakce

Již v úvodním dílu našeho seriálu jsme si vysvětlovali zkratku ACID. Jedná se o 4 vlastnosti, které by měla databáze podporovat. Jsou to nedělitelnost, validita, izolace a trvanlivost. Také jsme si již zmiňovali příklad převodu peněz z jednoho účtu na druhý. Tento příklad si s našimi znalostmi můžeme již velmi jednoduše převést do SQL:

```
--odečtení peněz
UPDATE "ucty"
SET "zustatek" = "zustatek" - 100
WHERE "cislo_uctu" = 123456789;
--přičtení peněz
UPDATE "ucty"
SET "zustatek" = "zustatek" + 100
WHERE "cislo_uctu" = 987654321;
```

Z účtu 123456789 jsme převedli 100 korun na účet 987654321. Vše vypadá hezky, ale jedná se o dva oddělené SQL dotazy. Může se jednoduše stát, že se jeden z nich z důvodu nějakého selhání neprovede. Důvodů selhání může být mnoho, např. 2. číslo účtu bude špatně zadáno a databáze vyhodí chybu. 1. příkaz se ovšem provede a aplikace spadne až na příkazu 2. Někomu tedy odečteme 100 Kč a ty zmizí. Dostáváme se do nekonzistentního stavu.

Transakce je soubor několika dotazů, které databáze chápe jako jeden dotaz. Buď se provedou všechny dotazy v transakci nebo se neprovede žádný. Transakci také můžeme do poslední chvíle odvolat. Pokud 2 výše zmíněné bankovní operace vložíme do transakce, můžeme si být jisti, že výsledek bude vždy podle očekávání.

Dalším příkladem by mohlo být uložení uživatele a jeho adresy, kdy se uživatel vloží do tabulky *uzivatele* a jeho adresa do tabulky *adresy*. Jeden záznam bez druhého nemá smysl, proto se musí vložit oba nebo žádný. Jinak by v databázi byl nekonzistentní uživatel (bez adresy) nebo adresa bez uživatele.

Transakci započneme příkazem ***BEGIN TRANSACTION*** nebo příkazem ***BEGIN*** (ten se používá i na jiných platformách). Některé databáze započnou transakci úplně samy (např. Oracle). Následuje série příkazů, které jsou součástí transakce. Celá transakce je potvrzena příkazem ***COMMIT*** nebo příkazem ***END TRANSACTION***;. Naše jednoduchá bankovní transakce by vypadala takto:

```
BEGIN TRANSACTION;
--odečtení peněz
UPDATE "ucty"
SET "zustatek" = "zustatek" - 100
WHERE "cislo_uctu" = 123456789;
--přičtení peněz
UPDATE "ucty"
SET "zustatek" = "zustatek" + 100
WHERE "cislo_uctu" = 987654321;
COMMIT;
```

Implicitní transakce

Každý příkaz, který mění databázi, což je prakticky vše kromě SELECTu, si automaticky vytvoří transakci, pokud ji nevytvoříme my. Pokud zadáme třeba toto:

```
UPDATE "uzivatele" SET "email" = 'ema@centrum.com' WHERE "prezdivka" = 'Ema';
```

Databáze ve skutečnosti spustí toto:

```
BEGIN TRANSACTION;
UPDATE "uzivatele" SET "email" = 'ema@centrum.com' WHERE "prezdivka" = 'Ema';
COMMIT;
```

Tato vlastnost se ruší pouze uvnitř explicitních transakcí. Nyní se všechny změny v databázi budou dít jen virtuálně a až po zavolání ***COMMIT*** se fyzicky uloží. V kterékoli části takovéto transakce můžeme zavolat příkaz ***ROLLBACK***, který celou transakci odvolá. Tento příkaz volá implicitně databáze v případě nějaké chyby za běhu transakce.

Pokud si budete chtít hrát s transackemi přímo DB Browseru, tak se vám to nepodaří, protože ten uzavírá do transakcí sám, proto musíte sáhnout po primitivnějších nástrojích třeba po klasickém sqlite shellu.

Pohledy

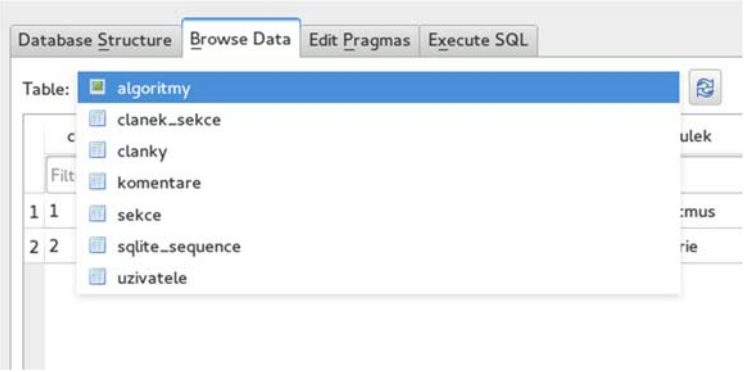
Databázové pohledy slouží k ukládání často používaných nebo složitých SQL dotazů. Chovají se jako virtuální tabulky, které můžeme navenek používat jako by existovaly fyzicky v databázi. Ve skutečnosti se ale jedná pouze o dynamické dotazy, které tabulky v paměti sestaví až při spuštění. Jednoduše řečeno se jedná o uložený příkaz ***SELECT***.

Přínosem je zapouzdření nějakého složitého dotazu přes několik tabulek do jediného pohledu, který navenek vystupuje jako jedna tabulka. Tu poté používáme v dalších dotazech, které jsou o to jednodušší. Pohledy také umožňují regulovat práva pro přístup k tabulce.

Vytvořme si pohled a uložme si do něj nějaký SQL dotaz. Naše databáze je velmi malá, tak mě napadlo jen uložit si články s určitou kategorií. Tyto články bychom třeba poté zobrazovali v nějakém widgetu a dále s nimi pracovali i na různých místech, proto by nám zjednodušilo práci mít je v pohledu:

```
CREATE VIEW "algoritmy" AS
SELECT *
FROM "clanky"
WHERE "clanky_id" IN
(
        SELECT "clanek_id"
        FROM "clanek_sekce"
        WHERE "sekce_id" =
        (
                SELECT "sekce_id"
                FROM "sekce"
                WHERE "nazev" = 'Algoritmy'
        )
);
```

Pohled je vytvořen. V DB Browseru se nám ukázal mezi tabulkami, jen s jinou ikonou:



Nyní s ním můžeme pracovat jako s tabulkou, kdykoli budeme chtít něco z algoritmů, stačí použít tuto tabulku. Např.:

```
SELECT "titulek" FROM "algoritmy";
```

SQLite - Optimalizace výkonu databáze

Volba databáze

Prvním faktorem je samozřejmě zvolená databáze. SQLite na tom není s výkonem vůbec špatně, ale není ideální pro velké projekty. V rychlosti často předčí PostgreSQL nebo MySQL. V korporátní sféře se používají hlavně databáze na platformě Oracle, které jsou opravdu robustní a stavěné pro vysoký výkon a snadnou škálovatelnost. Najdeme je u aplikací např. ve státní správě, finančnictví a podobně. Bohužel se jedná o velmi drahou záležitost. SQLite nám jistě bude stačit ještě dlouho a změna databáze kvůli výkonu se dělá opravdu až v případě, že již nejsou žádné jiné možnosti.

Volba dotazu

Dalším faktorem jsou samotné dotazy, tedy jak jsou položeny. Vlastností jazyka SQL je mimo jiné i to, že ten samý dotaz můžeme napsat několika způsoby. Můžeme použít několik dotazů za sebou, můžeme použít JOINy, dokonce různé JOINy, můžeme použít poddotazy, nějaké vestavěné funkce v SQLite atd.

Databázové indexy

Asi největší vliv mají na výkon databáze tzv. databázové indexy. O primárním klíči toho již víme hodně. Primární klíč (neboli také primární index) označuje sloupec, ve kterém jsou hodnoty unikátní. Neříkali jsme si ale o tom, co takový index pro databázi vnitřně znamená. Data jsou v databázích reprezentována v podobě tzv. binárních stromů. Tyto stromy umožňují rychlé vyhledávání dat (přesněji v čase log n). Vyhledávání dat probíhá právě pomocí indexů, pokud hledáme v tabulce podle sloupce s indexem, hledání je velmi rychlé. Tento dotaz je tedy opravdu rychlý:

```
SELECT "titulek" FROM "clanky" WHERE "clanky_id" = 2;
```

Je to proto, že v podmínce WHERE máme pouze sloupec, na kterém je databázový index. Mnohem častěji ale budeme vyhledávat podle url článku (např. když ho zobrazujeme uživateli podle adresy):

```
SELECT "titulek" FROM "clanky" WHERE "url" = 'bakterie-bunecny-automat';
```

Na sloupci *url* ovšem žádný index nemáme, databázový stroj bude muset projet všechny řádky v tabulce a podívat se, zda odpovídají podmínce. To je velmi pomalé. Problém můžeme vyřešit přidáním tzv. pomocného indexu. Těch můžeme mít v tabulce několik a nemusí pro ně platit unikátnost, jako je tomu u indexu primárního. Index je vlastností tabulky a přidáme ho tak, že mu přiřadíme jméno, v tomto případě *url_index* a vybereme sloupec (případně sloupce):

```
CREATE INDEX "url_index" ON "clanky" ("url");
```

Co se teď vlastně stalo? Tabulka *clanky* nyní obsahuje 2 indexy. Jeden primární a další pomocný. Čtení dat přes sloupec s primárním indexem nebo přes tento s pomocným indexem je velmi rychlé. Nepatrně se zpomalil zápis, protože je třeba udržovat 2 indexy místo jednoho. U webových aplikací však značně převládá čtení, článek přibude jen jednou za čas, ale jsou návštěvníky zobrazovány (čteny z databáze) neustále. Na ITnetwork jsme přidáním pouhých 2 indexů zrychlili časy u jednoho skriptu z 1.2 sekundy na 0.1 sekundy. Indexy se zavádí velmi jednoduše a jedná se o mocný nástroj z hlediska výkonu.

Kdy použít indexy?

Tato otázka souvisí s tím, kdy vůbec optimalizovat.
Zpočátku by se nemělo moc optimalizovat. Nemůžeme totiž dopředu vědět, jak dlouho databázi různé dotazy zaberou a špatně navržená optimalizace by mohla databázi naopak zpomalit. Rychlost samozřejmě záleží na množství dat v tabulkách. S nějakými zkušenostmi můžeme určité jednoduché optimalizace udělat, např. s tím url článku je docela jasné, když se podle toho článek vždy vybírá. Optimalizujeme jen ty dotazy, které se provádí často.

Další tzv. úzká hrdla aplikace (tedy místa, kde je pomalá) odhalí až čas nebo dostatečně velká testovací data. Neměli bychom hned zběsile navrhovat všemožné indexy, ale počkat, jak to bude s výkonem vypadat na produkčním prostředí. Odezva běžných činností na webu nebo v aplikaci (tedy odezva databáze + odezva aplikační logiky) je v pořádku asi do 500ms (v korporátní sféře), při vyšších hodnotách bychom se měli zamyslet nad tím, co by šlo udělat lépe. Pro běžný web by měla být odezva ještě nižší.

Indexy přes více sloupců

Pokud často hledáme podle více kritérií a tento dotaz je pomalý, můžeme udělat index rovnou přes více sloupců. Např. pokud často hledáme uživatele podle jména a příjmení (teď obecně, v našem RS vypadá uživatel jinak):

```
SELECT "jmeno", "prijmeni", "adresa"
FROM "uzivatele"
WHERE "jmeno" = 'Jan' AND "prijmeni" = 'Novak';
```

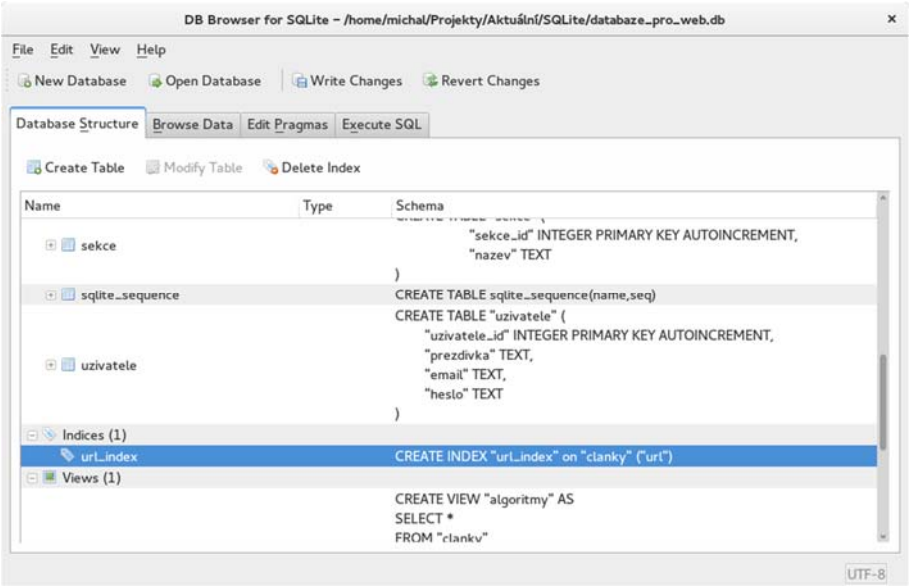
Pokud bychom tento dotaz prováděli velmi často, např. by ho prováděla úřednice na úřadě pro každého zákazníka, kterých je v systému milion, určitě by bylo vhodné ho optimalizovat. Vytvoření dvou indexů na sloupci jméno a příjmení by však nebylo tak efektivní. Proto vytvoříme jeden index

rovnou přes 2 sloupce:

CREATE INDEX "jmeno_prijmeni_index" ON "uzivatele" ("jmeno", "prijmeni");

Výsledkem dotazu tedy nejsou 2 indexy, ale jediný index pro dva sloupce. Opět opakuji, že toto se vyplatí jen v případě, když se dotaz provádí často a jsou s ním výkonnostní problémy, určitě nemůžeme vytvořit skupiny indexů na všechny dotazy v aplikaci. Indexy si můžeme přidělit další vlastnosti, které pro nás však zde nejsou důležité.

Indexy lze vidět i spravovat v DB Browseru, v záložce Database Structure:



Normalizace a denormalizace

Data v naší databázi by měla být normalizovaná. To znamená, že například pro entitu zákazníka bychom měli mít tabulku *zákazníci* a pro jeho adresu tabulku *adresy*. Adresa by neměla být vtěsnána do jednoho pole v tabulce *zákazníci*, ale rozdělena na ulici, číslo popisné, město a psč do sloupců v samostatné tabulce *adresy*. Hovoříme o atomických hodnotách ve sloupcích. Pokud budeme databáze takto navrhovat, vyhýbáme se problémům při dalším rozšiřování naší aplikace. Je definováno několik tzv. normálních forem, ale tím se nebudeme zatěžovat.

V budoucnu bychom třeba zjistili, že někdy zákazník potřebuje 2 adresy (dodací a fakturační) a tabulka *zákazníci* by už nevypadala úplně dobře. Problém je samozřejmě také s tím, když se rozhodneme nějak pracovat třeba se samotnou ulicí. Tu každý uživatel může do adresy zapsat po svém. Data v databázi by měla být také co nejvíce surová. Např. smajlíky v komentářích budeme nahrazovat za tag <img> až v aplikaci a nebudeme ukládat do databáze již takto naformátované HTML. To proto, že kdybychom chtěli změnit názvy obrázků, byl by to problém, byly by rozházené všude v textech.

Oproti normalizaci se někdy provádí i tzv. denormalizace. Jedná se o porušení dobrých praktik, které jsem vyjmenoval výše, za cílem zvýšení výkonu aplikace. Více tabulek spojíme do jedné, abychom se vyhnuli jejich JOINování. Hodnoty již nejsou tak atomické, jak by měly být. Toto řešení je velmi ošklivé a měli bychom se k němu uchylovat až ve chvíli, kdy opravdu není jiná možnost jak zvýšit výkon, než tím, že si zkazíme návrh aplikace a budeme riskovat její kvalitu.

Další možnosti optimalizace

Když je nějaký dotaz opravdu komplikovaný, měli bychom se sami sebe zeptat, zda danou část aplikace můžeme navrhnout tak, aby dotaz vypadal jinak. Mnohdy zjistíme, že se daný problém dá vyřešit elegantněji. Mám na mysli, že když je zadání příliš komplikované, je dobré se zeptat i na to, zda je to takto opravdu nutné. Mnohdy budeme překvapeni. Ta data, která se příliš často nemění, můžeme i nějak zachovat na straně aplikace, aby se zbytečně stále nenačítala z databáze. To se ale již týká optimalizací na straně aplikační vrstvy a ne té databázové. Možností je mnoho a jednoduchou úpravou lze udělat z minutových dotazů vteřinové.

Klauzule HAVING

Klauzule ***HAVING*** se používá:

- v příkazu ***SELECT***, pro filtrování seskupených záznamů
- pokud chceme v podmínce použít agregační funkce (***COUNT(), MIN(), MAX()*** a další)

Klauzule ***HAVING*** se nejčastěji používá společně s ***GROUP BY*** k třídění skupin záznamů podle určité podmínky. Pokud ***GROUP BY*** vynecháme, tak se ***HAVING*** chová stejně jako ***WHERE***.

Následující kód ukazuje správnou syntaxi klauzule ***HAVING***:

```
SELECT vybrane_sloupce
FROM nazev_tabulky
WHERE vyhledavaci_podminka
GROUP BY seskupeni_podle_sloupce
HAVING podminka_seskupeni;
```

V této syntaxi určíme podmínku v klauzuli ***HAVING***.
Pokud řádek, který je generován klauzulí ***GROUP BY***, vyhovuje podmínce podminka_seskupeni, dotaz jej zahrne do sady výsledků.
Všimněme si, že ***HAVING*** aplikuje podmínku filtrování na každou skupinu řádků. WHERE aplikuje podmínku filtrování na každý individuální řádek.

Rozdíl mezi HAVING a WHERE

Hlavní rozdíl mezi klauzulemi ***HAVING*** a ***WHERE*** je viditelný, když je použijeme společně s ***GROUP BY***. V tomto případě se ***WHERE*** použije před seskupením a ***HAVING*** se použije k filtrování záznamů až po seskupení. Klauzule, použité v příkazu ***SELECT***, mají dané pořadí zpracování. Pořadí zpracování klauzulí v příkazu ***SELECT*** znázorňuje následující diagram:

Další důležitý rozdíl je, že ***WHERE*** nesmí obsahovat agregační funkce jako např.: ***COUNT(), MIN(), MAX()*** a další. V případě, že chceme v podmínce použít agregační funkce, použijeme klauzuli ***HAVING***. V případě, že lze filtrování záznamů z tabulky provést bez použití agregačních funkcí, měli bychom to provést v klauzuli ***WHERE***. Značně to zvyšuje výkon dotazu (snižuje čas potřebný na zpracování dotazu), jelikož se řazení a třídění bude provádět na značně menší sadě dat.

Příklady použití HAVING

Ukážeme si několik případů správného použití klauzule ***HAVING***. V následujících případech budeme využívat tabulku detail_objednavky, která bude na rozdíl od předešlých lekcí úplně nová. Můžete si pro to tedy udělat novou databázi či ji přidat do existující databáze. Tabulka uchovává informace o detailu objednávky, konkrétně informaci o zakoupeném zboží, ceně za zboží a počtu kusů zakoupeného zboží:

detail_objednavky
id_objednavky
* id_zbozi
* pocet_kusu
* cena_za_kus

Vytvoříme si ji pomocí tohoto příkazu:

```
CREATE TABLE IF NOT EXISTS detail_objednavky(
        id_objednavky int not null,
        id_zbozi int not null,
        pocet_kusu int not null,
        cena_za_kus int not null,
        primary key (id_objednavky, id_zbozi)
);
```

Poté ji můžeme naplnit těmito daty:

```
INSERT INTO detail_objednavky VALUES (1,1,15,159),(1,8,2,199),(2,5,3,1959),(2,1,4,2499),(3,24,1,99);
```

Nyní můžeme spustit následující dotaz, který vypíše jednotlivé objednávky, počet zakoupených kusů zboží a celkovou cenu objednávky:

```
SELECT id_objednavky,
        SUM(pocet_kusu) AS kusy,
        SUM(pocet_kusu*cena_za_kus) AS cena_celkem
FROM detail_objednavky
GROUP BY id_objednavky;
```

Jako výstup dostaneme toto:

id_objednavky	kusy	cena_celkem
1	17	2783
2	7	15873
3	1	99

Nyní můžeme použít klauzuli HAVING k vypsání pouze těch objednávek, které mají celkovou cenu vyšší než 100 Kč:

```
SELECT id_objednavky,
        SUM(pocet_kusu) AS kusy,
        SUM(pocet_kusu*cena_za_kus) AS cena_celkem
FROM detail_objednavky
GROUP BY id_objednavky
HAVING cena_celkem > 100;
```

Výsledek je menší o jeden řádek než minule, protože objednávka s id_objednavky = 3 měla celkovou cenu 99 Kč, proto tedy neprošla podmínkou v klauzuli HAVING:

id_objednavky	kusy	cena_celkem
1	17	2783
2	7	15873

Můžeme používat také komplexnější podmínky. Následující kód vypíše objednávky, které mají celkovou cenu vyšší než 100 Kč a zároveň celkový počet objednaných kusů je větší než 10 ks:

```
SELECT id_objednavky,
        SUM(pocet_kusu) AS kusy,
        SUM(pocet_kusu*cena_za_kus) AS cena_celkem
FROM detail_objednavky
GROUP BY id_objednavky
HAVING cena_celkem > 100 AND kusy > 10;
```

Výsledek je opět o řádek kratší a vrátil nám pouze objednávku s číslem 1, protože tato jediná objednávka prošla naší podmínkou v klauzuli HAVING:

id_objednavky	kusy	cena_celkem
1	17	2783

Použití HAVING a WHERE v jednom dotazu

Nyní si ukážeme příklad dotazu, ve kterém jsme použili obě klauzule. V následujícím příkladu budeme používat tabulku zakaznik, která vypadá následovně:

zakaznik
id_zakaznika
* jmeno
* prijmeni
* vek
* mesto

Pro její vytvoření můžeme použít tento dotaz:

```
CREATE TABLE IF NOT EXISTS zakaznik(
        id_zakaznika integer not null,
        jmeno varchar(50) not null,
        prijmeni varchar(50) not null, vek int not null,
        mesto varchar(50) not null,
        primary key(id_zakaznika autoincrement)
);
```

K naplnění dat pak tento dotaz:

```
INSERT INTO zakaznik VALUES (null,'Matěj','Eliáš',20,'Ostrava'),(null,'Karel','Svoboda',21,'Ostrava'),
(null,'Jiří','Novák',17,'Ostrava'),(null,'Petr','Novotný',45,'Praha'),
(null,'Jan','Horák',14,'Praha'),(null,'Prokop','Buben',34,'Brno');
```

Obsah tabulky zakaznik vypadá takto:

id_zakaznika	jmeno	prijmeni	vek	mesto
Filtr	Filtr	Filtr	Filtr	Filtr
1	Matěj	Eliáš	20	Ostrava
2	Karel	Svoboda	21	Ostrava
3	Jiří	Novák	17	Ostrava
4	Petr	Novotný	45	Praha
5	Jan	Horák	14	Praha
6	Prokop	Buben	34	Brno

Řekněme, že chceme vypsat, kolik zákazníků žije v jednotlivých městech. To provedeme následujícím dotazem za pomocí klauzule GROUP BY a použitím agregační funkce COUNT():

```
SELECT mesto, COUNT(id_zakaznika)
FROM zakaznik
GROUP BY mesto;
```

Dotaz nám vrátí tento výstup:

mesto	COUNT(id_zakaznika)
Brno	1
Ostrava	3
Praha	2

Vidíme, že z Brna je jeden zákazník, z Prahy jsou dva zákazníci a z Ostravy jsou tři zákazníci. Nyní dotaz doplníme o podmínku, která nám z výsledku vyloučí města, ve kterých žije pouze jeden zákazník. Protože počet zákazníků zjišťujeme pomocí agregační funkce COUNT(), tak musíme tuto podmínku vložit do klauzule HAVING:

```
SELECT mesto, COUNT(id_zakaznika) AS pocet
FROM zakaznik
GROUP BY mesto
HAVING pocet > 1;
```

Z výsledku nám vypadlo Brno, ve kterém žil pouze jeden zákazník:

mesto	pocet
Ostrava	3
Praha	2

Nyní předchozí dotaz doplníme o podmínku, která nám vypíše pouze počet dospělých zákazníků. Jelikož je věk sloupec, můžeme podmínku vložit pomocí klauzule WHERE:

```
SELECT mesto, COUNT(id_zakaznika) AS pocet
FROM zakaznik
WHERE vek > 17
GROUP BY mesto
HAVING pocet > 1;
```

Výsledek vypadá následovně:

mesto	pocet
Ostrava	2

Všimněme si, že počet zákazníků v Ostravě klesl o jednoho zákazníka, protože zákazník Jiří Novák má pouze 17 let, není tedy dospělý. Dále zmizela také Praha, protože v Praze je jeden zákazník dospělý a druhý není. Po vyloučení neplnoletých zákazníků v Praze zbyl pouze jeden dospělý zákazník. Ten však neprošel naší podmínkou, která zobrazuje pouze města, ve kterých je dva a více zákazníků. Tímto jsme si také dokázali, že klauzule WHERE se provedla dříve, než se záznamy rozdělily do skupin. Následně se sada dat, která již byla zbavena zákazníků, kteří nejsou plnoletí, setřídila do skupin podle města. A nakonec se testovala podmínka, zda jsou ve městě alespoň dva zákazníci, která je zapsaná v klauzuli HAVING.

SQLite – Triggery

Trigger je uložený program, který se spustí automaticky jako reakce na určitou akci s přidruženou tabulkou. Například můžeme vytvořit trigger, který se spustí, když odstraníme z nějaké tabulky řádek, nebo když nějaký řádek naopak přidáme.

V SQLite triggery reagují na tři druhy událostí:

- UPDATE
- INSERT
- DELETE

SQL standard má dva typy triggerů:

• Prvním je row-level trigger. Ten se spouští zvlášť pro každý řádek tabulky. Pokud tedy do tabulky vložíme pět řádků, tak se row-level trigger spustí pětkrát.

• Druhým je statement-level trigger, který se spouští pro každou transakci zvlášť. Tedy pokud v jednom dotazu vložíme pět řádků, tak se statement-level trigger provede pouze jednou.

Jazyk SQLite podporuje pouze row-level triggery

Tvorba triggerů

Jak jsme si již řekli, triggery se spouští jako reakce na jednu ze tří událostí. Můžeme si také vybrat, jestli se spustí před touto událostí, anebo až po této události. Tento výběr specifikujeme klíčovými slovy BEFORE a AFTER. Syntaxe při tvorbě triggerů je následující:

```
CREATE TRIGGER nazev_triggeru {BEFORE,AFTER} {UPDATE,INSERT,DELETE}
ON nazev_pridruzene_tabulky FOR EACH ROW
BEGIN
        telo_triggeru;
END;
```

Začneme klíčovými slovy ***CREATE TRIGGER***, za kterými následuje název námi vytvořeného triggeru. Následně se specifikuje událost, při které se trigger spustí a jestli se spustí před touto událostí, nebo po této události.
Pak přiřadíme tabulku, na kterou bude trigger reagovat pomocí klíčového slova ***ON*** následovanou ***FOR EACH ROW***.
Tělo triggeru pak obalují klíčová slova ***BEGIN*** a ***END***.
V těle triggeru máme dostupné dvě nová klíčová slova a to ***OLD*** a ***NEW***. Těmito klíčovými slovy určujeme, zda odkazujeme na hodnotu novou, nebo na hodnotu předchozí. Když se například spustí trigger reagující na aktualizaci dat v tabulce, tak klíčovým slovem ***OLD*** budeme odkazovat na původní hodnotu před aktualizací, naopak klíčovým slovem ***NEW*** na novou hodnotu po aktualizaci.

Smazání triggeru

K mazání triggerů používáme příkaz DROP TRIGGER následovaný nepovinnými klíčovými slovy IF EXISTS. Pak jen zadáme název triggeru, který chceme smazat. Syntaxe pro smazání triggeru je následující:

```
DROP TRIGGER [IF EXISTS] nazev_triggeru;
```

Příklady využití triggerů

BEFORE INSERT trigger

Tento trigger se spustí automaticky před vložením nového záznamu do tabulky. Vytvoříme si jednoduchou tabulku pobocky:

```
CREATE TABLE IF NOT EXISTS pobocky(
        id_pobocky INTEGER NOT NULL,
        mesto TEXT NOT NULL,
        nazev TEXT NOT NULL,
        pocet_pracovniku INTEGER NOT NULL,
        primary key(id_pobocky AUTOINCREMENT)
);
```

Tato tabulka uchovává informace o jednotlivých pobočkách naší smyšlené firmy.

Dále si vytvoříme malou tabulku statistika_pobocek obsahující pouze jeden záznam, který budeme postupně aktualizovat. Tabulka bude uchovávat informaci o aktuálním počtu všech zaměstnanců ve všech pobočkách dohromady:

```
CREATE TABLE IF NOT EXISTS statistika_pobocek(
        pocet_pracovniku_celkem INTEGER NOT NULL
);
INSERT INTO statistika_pobocek VALUES(0);
```

Nyní přichází čas na náš trigger. Účelem triggeru bude aktualizace aktuálního počtu všech zaměstnanců v tabulce statistika_pobocek pokaždé, než vložíme novou pobočku do tabulky pobocky:

```
CREATE TRIGGER before_insert_pobocky BEFORE INSERT
```

```
ON pobocky FOR EACH ROW
BEGIN
        UPDATE statistika_pobocek
        SET pocet_pracovniku_celkem = pocet_pracovniku_celkem + NEW.pocet_pracovniku;     END;
```

V těle triggeru jsme využili příkaz UPDATE, kterým jsme aktualizovali celkový počet pracovníků v tabulce statistika_pobocek.

Funkčnost triggeru otestujeme přidáním nových záznamů do tabulky pobocky:

```
INSERT INTO pobocky VALUES(null,'Ostrava','ITnetwork',50),
(null,'Brno','ITnetwork',60),(null,'Praha','ITnetwork',70);
```

Vypíšeme si obsah tabulky statistika_pobocek:

```
SELECT * FROM statistika_pobocek;
```

Výsledek vypadá takto:

pocet_pracovniku_celkem
180

Přidáním tří nových záznamů do tabulky pobocky se aktualizoval celkový počet zaměstnanců v tabulce statistika_pobocek.

BEFORE UPDATE trigger

Tento trigger se bude automaticky spouštět, před jakoukoli aktualizací dat v přiřazené tabulce.

Začněme vytvořením tabulky historie_pobocek. Tato tabulka bude ukládat historii změn v tabulce pobocky. Tedy pokud dojde k nějaké změně v tabulce pobocky, tak se záznam o této změně automaticky uloží do tabulky historie_pobocek:

```
CREATE TABLE IF NOT EXISTS historie_pobocek(
        id INTEGER NOT NULL,
        id_pobocky INTEGER NOT NULL,
        mesto TEXT NOT NULL,
        nazev TEXT NOT NULL,
        pocet_pracovniku INTEGER NOT NULL,
        cas_zmeny TEXT NOT NULL,
        akce TEXT NOT NULL,
        PRIMARY KEY(id AUTOINCREMENT)
);
```

Nyní si vytvoříme trigger:

```
CREATE TRIGGER before_update_pobocky BEFORE UPDATE
ON pobocky FOR EACH ROW
BEGIN
        INSERT INTO historie_pobocek
        VALUES (null,OLD.id_pobocky,OLD.mesto,OLD.nazev,OLD.pocet_pracovniku, DATE('now'),'Update');
        UPDATE statistika_pobocek
        SET pocet_pracovniku_celkem = pocet_pracovniku_celkem - OLD.pocet_pracovniku +
NEW.pocet_pracovniku;
        END;
```

V těle triggeru využíváme dva příkazy. Příkaz INSERT, kterým vložíme původní data před aktualizací do tabulky historie_pobocek a příkaz UPDATE, kterým aktualizujeme aktuální stav celkového počtu pracovníků v tabulce statistika_pobocek.

Funkčnost vyzkoušíme aktualizací hodnot v tabulce pobocky:

```
UPDATE pobocky SET nazev = 'ITnetwork.cz', pocet_pracovniku = 100 WHERE id_pobocky = 1;
```

Příkazem jsme změnili název pobočky v Ostravě a změnili jsme také její počet pracovníků. Nyní si vypíšeme obsah tabulky historie_pobocek, abychom viděli, zda se naše změna uložila:

```
SELECT * FROM historie_pobocek;
```

Jak můžeme vidět, tak se nám původní hodnoty uložily:

id	id_pobocky	mesto	nazev	pocet_pracovniku	cas_zmeny	akce
1	1	Ostrava	ITnetwork	50	2021-07-18	Update

Můžeme si také vypsat obsah tabulky statistika_pobocek, abychom viděli, zda se nám aktualizoval celkový počet pracovníků:

```
SELECT * FROM statistika_pobocek;
```

Celkový počet pracovníků se úspěšně aktualizoval:

pocet_pracovniku_celkem
230

AFTER DELETE trigger

Na závěr si ukážeme AFTER DELETE trigger. Tento trigger se spustí automaticky po smazání řádku z přiřazené tabulky:

```
CREATE TRIGGER after_delete_pobocky AFTER DELETE
ON pobocky FOR EACH ROW
BEGIN
        INSERT INTO historie_pobocek
        VALUES (null,OLD.id_pobocky,OLD.mesto,OLD.nazev,OLD.pocet_pracovniku, DATE('now'),'Delete');
        UPDATE statistika_pobocek
        SET pocet_pracovniku_celkem = pocet_pracovniku_celkem - OLD.pocet_pracovniku;      END;
```

V těle triggeru využíváme dva příkazy. Příkaz INSERT uloží záznam o smazání společně se smazanými hodnotami do tabulky historie_pobocek. Příkaz UPDATE aktualizuje aktuální počet všech pracovníků, tedy odečte počet pracovníků ze smazané pobočky.

Trigger vyzkoušíme smazáním pobočky v Praze:

```
DELETE FROM pobocky WHERE id_pobocky = 3;
```

Vypíšeme si historii poboček následujícím příkazem:

```
SELECT * FROM historie_pobocek;
```

Historie poboček vypadá následovně:

id	id_pobocky	mesto	nazev	pocet_pracovniku	cas_zmeny	akce
1	1	Ostrava	ITnetwork	50	2021-07-18	Update
2	3	Praha	ITnetwork	70	2021-07-18	Delete

Poslední řádek je záznam o smazání pobočky v Praze.

Vypíšeme si obsah tabulky statistika_pobocek:

```
SELECT * FROM statistika_pobocek;
```

Výsledek vypadá takto:

pocet_pracovniku_celkem
160

Počet pracovníků tedy koresponduje s údaji v tabulce pobočky.

Klauzule CASE-WHEN-THEN

Může se nám stát, že v některých případech chceme, aby trigger za nějaké podmínky reagoval jinak. Narozdíl od jiných variant SQL sice nemůžeme kvůli omezenosti SQLite použít klauzuli IF-THEN-ELSE, ale celkem slušně nám poslouží klauzule ***CASE-WHEN-THEN***. Řekněme například, že budeme vyžadovat, aby se k poli akce v tabulce historie_pobocek připsal dodatek, popisující změnu počtu zaměstnanců.

Pokud bude nový počet zaměstnanců:

- menší, připíšeme " - zmenšena",
- větší, připíšeme " - zvětšena",
- stejná, připíšeme " - nezměněna".

Nejdříve smažeme původní trigger before_update_pobocky:

```
DROP TRIGGER `before_update_pobocky`;
```

Nyní vytvoříme trigger znovu, ale tentokrát s naší podmínkou:

```
CREATE TRIGGER before_update_pobocky BEFORE UPDATE
ON pobocky FOR EACH ROW
BEGIN
        INSERT INTO historie_pobocek
                VALUES (null,OLD.id_pobocky,OLD.mesto,OLD.nazev,OLD.pocet_pracovniku, DATE('now'),
                CASE    WHEN NEW.pocet_pracovniku > OLD.pocet_pracovniku THEN 'Update - zvětšena'
                        WHEN NEW.pocet_pracovniku < OLD.pocet_pracovniku THEN 'Update - zmenšena'
                        WHEN NEW.pocet_pracovniku = OLD.pocet_pracovniku THEN 'Update - nezměněna'
                END);
        UPDATE statistika_pobocek
        SET pocet_pracovniku_celkem = pocet_pracovniku_celkem - OLD.pocet_pracovniku + NEW.pocet_pracovniku;
END;
```

Pokud zvětšíme počet pracovníků pobočky v Ostravě:

```
UPDATE pobocky SET pocet_pracovniku = 200 WHERE id_pobocky = 1;
```

V tabulce historie_pobocek se objeví nový záznam, který nás informuje, že počet pracovníků se zvětšil:

```
1   1   Ostrava   ITnetwork      50   2020-09-09 13:19:00   Update
2   3   Praha     ITnetwork      70   2020-09-09 13:39:55   Delete
3   1   Ostrava   ITnetwork.cz   100  2022-07-08 07:38:40   Update - zvětšena
```

Pokud zmenšíme počet pracovníků pobočky v Brně:

```
UPDATE pobocky SET pocet_pracovniku = 50 WHERE id_pobocky = 2;
```

V tabulce historie_pobocek se objeví nový záznam, který nás informuje, že počet pracovníků se zmenšil:

```
1   1   Ostrava   ITnetwork      50   2020-09-09 13:19:00   Update
2   3   Praha     ITnetwork      70   2020-09-09 13:39:55   Delete
3   1   Ostrava   ITnetwork.cz   100  2022-07-08 07:38:40   Update - zvětšena
4   2   Brno      ITnetwork      60   2022-07-08 07:46:05   Update - zmenšena
```

A pokud počet pracovníků zůstane stejný:

```
UPDATE pobocky SET nazev = "ITnetwork - Brno" WHERE id_pobocky = 2;
```

V tabulce historie_pobocek správně přibude informace, že se počet zaměstnanců nezměnil:

```
1   1   Ostrava   ITnetwork      50   2020-09-09 13:19:00   Update
2   3   Praha     ITnetwork      70   2020-09-09 13:39:55   Delete
3   1   Ostrava   ITnetwork.cz   100  2022-07-08 07:38:40   Update - zvětšena
4   2   Brno      ITnetwork      60   2022-07-08 07:46:05   Update - zmenšena
5   2   Brno      ITnetwork      50   2022-07-08 07:48:16   Update - nezměněna
```

SQLite - Fulltextové vyhledávání

Vyhledávání výrazů v SQLite (LIKE)

V jazyce SQLite máme mnoho způsobů, jak nalézt nějaký konkrétní výraz. Jeden z nejpoužívanějších způsobů je klauzule ***LIKE***, pomocí které můžeme porovnávat řetězce proti zadanému pravidlu. Např.:

```
... WHERE jmeno_zakaznika LIKE 'Karel'
```

V tomto pravidlu můžeme použít také znak procento **%,** který je zástupným znakem pro "cokoli", nebo taky "libovolný počet znaků". Kdybychom tedy chtěli zjistit emaily, které mají doménu např. itnetwork.cz, tak bychom to mohli udělat následovně:

```
... WHERE email LIKE '%@itnetwork.cz'
```

Dále lze použít znak podtržítka **_,** který nahrazuje jeden libovolný znak:

```
... WHERE jmeno LIKE 'Kare_'
```

Možná jste někdy uvažovali, že si vytvoříte vlastní blog. Jedna z hlavních funkcí, kterou by měl dobrý blog mít, je vyhledávání, ve kterém můžeme najít články podle jejich obsahu. Prohledávání dlouhých článků použitím výše zmíněných metod je ale velmi pomalé, neefektivní a nespolehlivé.
Proto existuje v jazyce SQLite alternativa - Fulltextové vyhledávání.

Fulltextové vyhledávání

Pokud jste někdy používali internetové vyhledávače, jako třeba Google nebo Seznam, tak jste používali fulltextové vyhledávání (Full-text search, zkratka FTS). Vyhledávače si ukládají obsahy stránek do databáze a umožňují vyhledávat obsah na základě klíčových slov.

Fulltextové vyhledávání je technika pro vyhledávání dokumentů, které přesně neodpovídají kritériím vyhledávání.

Tyto dokumenty jsou pak databázové entity, které obsahují textová data. Textová data mohou být příspěvky na blogu, popisy produktů v e-shopech atd.

Například budeme vyhledávat výraz *"Kalhoty a tričko"*. FTS nám může vrátit dokumenty, které obsahují jedno z těchto slov. Také třeba dokumenty, které obsahují obě tato slova nebo klidně dokumenty, kde jsou tyto slova v opačném pořadí. Toto jsou funkce, které nejsme schopni zajistit pouhým použitím klauzule ***LIKE***. Při použití klauzule ***LIKE*** navíc neumíme určit, v jakém pořadí výsledky vyhledávání seřadit tak, aby byly co nejrelevantnější.

Při použití fulltextového vyhledávání vytváří SQLite ***FULLTEXT*** index z textových dat (např. článku) a při hledání použije sofistikovaný algoritmus k určení, které dokumenty odpovídají vyhledávání a seřadí je podle relevantnosti.

Další velikou výhodou fulltextového vyhledávání je, že při aktualizaci obsahu dokumentu si SQLite automaticky aktualizuje také obsah indexu, aby databáze vždy vracela aktuální výsledky. Dále je důležité zmínit, že velikost indexu ***FULLTEXT*** je relativně malá, tedy použití fulltextové vyhledávání je datově nenáročné. Nejdůležitější je ale rychlost vyhledávání. Ta je oproti použití klauzule ***LIKE*** mnohonásobně vyšší.

Vytvoření testovací tabulky

Ještě, než se ponoříme hlouběji do FTS v SQLite, vytvoříme si testovací tabulku prispevky, na které si ukážeme příklad FTS:

```
CREATE TABLE prispevky(
        prispevek_id INTEGER NOT NULL,
        nazev TEXT NOT NULL,
        obsah TEXT NOT NULL,
        PRIMARY KEY(prispevek_id AUTOINCREMENT)
);
```

a vložíme si do ní nějaká testovací data:

```
INSERT INTO prispevky(nazev, obsah) VALUES
        ("Java", "Java je objektově orientovaný jazyk."),
        ("PHP", "PHP je serverový jazyk."),
        ("C++", "C++ se často používá pro vývoj AAA herních titulů."),
        ("Python", "Python je díky jednoduché syntaxi dobrý pro testování nových algoritmů."),
        ("Maďarština", "Maďarština je jazyk, kterým se mluví v Maďarsku.");
```

Vytvoření virtuální tabulky

Na rozdíl od fulltextového vyhledávání v MySQL, kde vytvoříme normální tabulku a pouze do ní přidáme ***FULLTEXT index***, musíme v SQLite vytvořit takzvanou virtuální tabulku.

Virtuální tabulka se na první pohled tváří jako běžná tabulka, ovšem rozdíl je, jak SQLite přistupuje k datům. Místo toho, aby četl a měnil data přímo v souboru na úložišti, volá metody, které se aplikují na virtuální objekt tabulky. To pro nás není ani tak moc důležité, pouze je potřeba vědět, že budeme pracovat s virtuální tabulkou. Co je ale dobré vědět je, že při vytváření virtuální tabulky nezadáváme:

- který z atributů je primární klíč
- datové typy atributů

Vytvoření virtuální tabulky s FTS

Vytvoření virtuální tabulky provedeme následovně:

```
CREATE VIRTUAL TABLE nazev_tabulky USING FTS5(sloupec1, sloupec2);
```

Jak vidíme, za názvem tabulky nepíšeme rovnou názvy sloupců, ale nejdříve musíme definovat, že budeme používat FTS. To se dělá pomocí USING FTS5. Vedle FTS5 existují ještě FTS4 a FTS3. Ty jsou oproti FTS5 starší, ale mohli bychom je bez problémů použít.

Nyní si konkrétně vytvoříme tabulku prispevky_virtual, ve které budeme indexovat sloupec obsah a nazev:

```
CREATE VIRTUAL TABLE prispevky_virtual USING FTS5(nazev, obsah);
```

SQLite vytváří fulltext index ze všech zadaných sloupců, nikoli pouze z těch, které si vybereme.

Přidání FULLTEXT INDEXu do virtuální tabulky

V případě, že chceme přidat ***FULLTEXT INDEX*** do již vytvořené virtuální tabulky, můžeme to udělat pomocí dvou příkazů.

Nejdříve si vytvoříme virtuální tabulku, která má shodný počet sloupců jako stávající:

```
CREATE VIRTUAL TABLE IF NOT EXISTS nazev_virtualni_tabulky USING FTS5(sloupec1, sloupec2);
```

A následně do ní data z původní tabulky překopírujeme:

```
INSERT INTO nazev_virtualni_tabulky SELECT * FROM nazev_puvodni_tabulky;
```

Pro naši testovací tabulku:

```
CREATE VIRTUAL TABLE IF NOT EXISTS prispevky_virtual USING FTS5(nazev, obsah);
INSERT INTO prispevky_virtual SELECT nazev, obsah FROM prispevky;
```

Kopírujeme sloupce kromě primárního klíče. Můžeme si pro něj ve virtuální tabulce připravit sloupeček také, ale je to zbytečné, protože se na něj pohlíží pouze jako na další záznam, nikoliv jako na jednoznačný identifikátor.

Odebrání FULLTEXT INDEXu z tabulky

Smazat pouze fulltext index jako v MySQL bohužel logicky nejde. Musíme si opět poradit menší vychytávkou, jako při přidávání ***FULLTEXT INDEXu***.

Nejdříve si vytvoříme tabulku, která má shodný počet sloupců jako stávající (můžeme přidat i primární klíč):

```
CREATE TABLE nazev_tabulky(
        id INTEGER NOT NULL,
        nazev TEXT NOT NULL,
        obsah TEXT NOT NULL,
        PRIMARY KEY(id AUTOINCREMENT)
);
```

Pozor na datový typ sloupců.

A potom data opět překopírujeme:

```
INSERT INTO nazev_tabulky(nazev, obsah) SELECT * FROM nazev_virtualni_tabulky;
```

Vyhledávání

Samotné vyhledávání pak probíhá pomocí klauzule ***WHERE*** a nové funkce ***MATCH()***:

```
SELECT sloupce FROM nazev_tabulky
WHERE
        sloupec_hledani
        MATCH('hledany_vyraz')
        ORDER BY rank;
```

Po slovu ***WHERE*** následuje název sloupce, ve kterém hledáme shodu. Funkcí ***MATCH()*** určíme, jaký hledaný výraz hledáme. Pomocí ***ORDER BY*** rank řekneme, aby se výsledky řadily podle relevantnosti.

V našem případě bychom hledali článek ve fulltextovém vyhledávání například takto:

```
SELECT nazev,obsah FROM prispevky_virtual
WHERE
        obsah
        MATCH('jazyk')
        ORDER BY rank;
```

SQLite nám pak vrátí výsledky seřazené podle relevantnosti.

Praktický příklad vyhledávání:

Zkusme teď vyhledat v naší tabulce prispevky_virtual několik výrazů pomocí ***LIKE*** a následně pomocí FTS. Dejme tomu, že nevíme název jazyka, který je objektově orientovaný.

Pomocí ***LIKE*** bychom hledali například takto:

```
SELECT nazev FROM prispevky_virtual
WHERE
        obsah
        LIKE "%objektově orientovaný%";
```

Pomocí ***FTS*** pak takto:

```
SELECT nazev FROM prispevky_virtual
WHERE
        obsah
        MATCH("objektově orientovaný")
        ORDER BY rank;
```

Výsledek je v obou případech stejný, tedy Java. Co kdybychom ale neznali přesnou skladbu věty a hledali pomocí klíčových slov orientovaný objektově:

Pomocí ***LIKE*** bychom hledali takto:

```
SELECT nazev FROM prispevky_virtual
WHERE
        obsah
        LIKE "%orientovaný objektově%";
```

Pomocí ***FTS*** pak takto:

```
SELECT nazev FROM prispevky_virtual
WHERE
        obsah
        MATCH("orientovaný objektově")
        ORDER BY rank;
```

Zdánlivě stejné zadání, nicméně nyní ***dostaneme výsledek pouze, hledáme-li pomocí FTS***. Stejně tak, pokud bychom znali pořadí slov, ale ne jejich přesnou posloupnost. Chceme např. vyhledat jazyk pro vytváření AAA (herních) titulů:

Pomocí ***LIKE*** bychom hledali takto:

```
SELECT nazev FROM prispevky_virtual
WHERE
        obsah
        LIKE "%AAA titulů%";
```

Pomocí ***FTS*** pak takto:

```
SELECT nazev FROM prispevky_virtual
WHERE
        obsah
        MATCH("AAA titulů")
        ORDER BY rank;
```

Výsledek opět dostaneme pouze při hledání pomocí FTS.

Další možnosti FTS

Do vyhledávaného výrazu nemusíme zadávat pouze slova, která hledáme, ale můžeme například i říci, která se ve výsledku objevit nesmí, nebo mohou, ale nemusí. K tomu slouží takzvané operátory:

- **(bez operátoru)** - Výraz musí být přítomen
- **NOT** - Výraz nesmí být přítomen
- **AND** - musí obsahovat všechny spojené fráze
- **OR** - může frázi obsahovat, ale nemusí

Další značky a jejich konkrétní význam jsou uvedeny v oficiální dokumentaci SQLite.

Příklad použití značky:

```
SELECT nazev,obsah FROM prispevky_virtual
WHERE
        obsah
        MATCH('jazyk NOT java')
        ORDER BY rank;
```

V tomto případě bude SQLite prohledávat sloupec obsah tabulky prispevky_virtual a bude hledat konkrétní články, kde se nevyskytuje slovo java, ale mají slovo jazyk.

SQLite krok za krokem - Cizí klíče

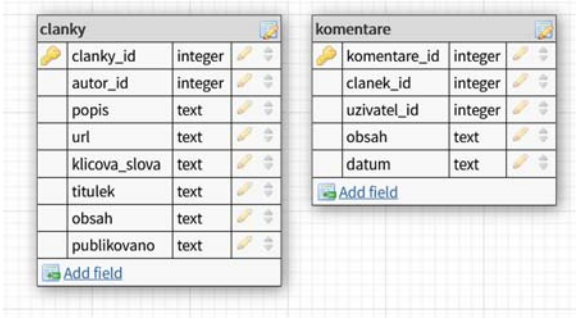
Cizí klíče jsou nepostradatelnou součástí korektní a normalizované databáze.

Cizí klíč je sloupec (nebo skupina sloupců) v tabulce, který je přímo spojený se sloupcem (nebo skupinou sloupců) v jiné tabulce. Tomuto jevu se také někdy říká reference, případně odkaz. Přidáním cizího klíče vytváříme tzv. integritní omezení, které do tabulky umožní vložit pouze povolené hodnoty. Také určuje, co se děje se souvisejícími daty, když je spojený záznam smazán nebo upraven. Například můžeme zabránit smazání záznamu v tabulce, ke kterému jsou přímo připojeny jiné záznamy v jiných tabulkách. Případně můžeme související záznamy nastavit na neutrální hodnotu nebo je smazat také v jediném dotazu.

Pojďme se podívat na tabulky clanky a komentare z databáze databaze_pro_web, kterou jsme vytvářeli v předchozích lekcích. Buď si stáhneme přiloženou databázi pod touto lekcí nebo si přes SQL query vložme tyto dva dotazy na vytvoření dvou tabulek do nové databáze:

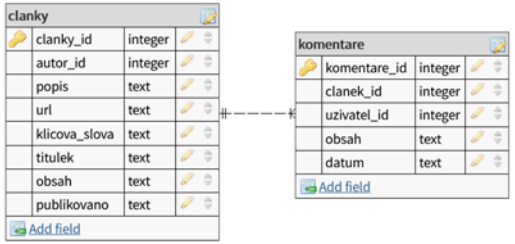
```
CREATE TABLE IF NOT EXISTS `clanky` (
        `clanky_id` INTEGER NOT NULL,
        `autor_id` INTEGER,
        `popis` TEXT,
        `url` TEXT,
        `klicova_slova` TEXT,
        `titulek` TEXT,
        `obsah` TEXT,
        `publikovano` TEXT,
        PRIMARY KEY (`clanky_id` AUTOINCREMENT)
);
CREATE TABLE IF NOT EXISTS `komentare` (
        `komentare_id` INTEGER,
        `clanek_id` INTEGER,
        `uzivatel_id` INTEGER,
        `obsah` TEXT,
        `datum` TEXT,
        PRIMARY KEY (`komentare_id` AUTOINCREMENT)
);
```

Grafické znázornění těchto tabulek:



Jak můžeme vidět, žádnou vytvořenou relaci mezi tabulkami nemáme, ale můžeme v tabulce komentare vidět odkaz na *článek* a *uživatele*, tedy sloupce clanek_id (primární klíč tabulky clanky) a uzivatel_id (primární klíč tabulky uzivatel, tu však zatím nepotřebujeme).

Vypadá to tedy, že se jedná o logickou relaci, která říká, že každý článek může mít jeden nebo více komentářů. Zároveň taky říká, že každý komentář je připojený k právě jednomu článku. Pojďme si tedy ukázat, jak by taková relace vypadala graficky:



Nyní již databázový server ví, že tato data spolu souvisí a existuje mezi nimi definovaný vztah.

Pokud jsme výše zakládali tabulky přes SQL query do nové databáze, vložme si cvičná data do obou tabulek:

```
INSERT INTO "clanky" VALUES
    (1,1,'Co je to algoritmus? Pokud to nevíte, přečtěte si tento článek.','co-je-to-algoritmus','algoritmus, co je to, vysvětlení','Algoritmus','Když se bavíme o algoritmech, pojďme se tedy shodnout na tom, co ten algoritmus vůbec je. Jednoduše řečeno, algoritmus je návod k řešení nějakého problému. Když se na to podíváme z lidského pohledu, algoritmus by mohl být třeba návod, jak ráno vstát. I když to zní jednoduše, je to docela problém. Počítače jsou totiž stroje a ty nemyslí. Musíme tedy dopodrobna popsat všechny kroky algoritmu. Tím se dostáváme k první vlastnosti algoritmu - musí být elementární (skládat se z konečného počtu jednoduchých a snadno srozumitelných kroků, tedy příkazů). "Vstaň z postele" určitě není algoritmus. "Otevři oči, sundej peřinu, posaň se, dej nohy na zem a stoupni si" - to už zní docela podrobně a jednalo by se tedy o pravý algoritmus. My se však budeme pohybovat v IT, takže budeme řešit problémy jako seřaď prvky podle velikosti nebo vyhledej prvek podle jeho obsahu. To jsou totiž 2 základní úlohy, které počítače dělají nejčastěji a které je potřeba dokonale promýšlet a optimalizovat, aby trvaly co nejkratší dobu. Z dalších příkladů algoritmů mě napadá třeba vyřeš kvadratickou rovnici nebo vyřeš sudoku.','2012-03-21 00:00:00'),
    (2,2,'Bakterie jsou obdoba buněčného automatu v kombinaci s hrou.','bakterie-bunecny-automat','bakterie, automat, algoritmus','Bakterie','Bakterie jsou obdoba buněčného automatu, který vymyslel britský matematik John Horton Conway v roce 1970. Celou tuto hru řídí čtyři jednoduchá pravidla:/n/n 1. Živá bakterie s méně, než dvěma živými sousedy umírá./n 2. Živá bakterie s více, než třemi živými sousedy umírá na přemnožení./n 3. Živá bakterie s dvoumi nebo třemi sousedy přežívá beze změny do další generace./n 4. Mrtvá bakterie, s přesně třemi živými sousedy, opět ožívá./n Tyto zdánlivě naprosto primitivní pravidla dokáží za správného počátečního rozmístění bakterií vytvořit pochodující skupinky, shluky "vystřelující" pochodující pětice, překvapivě složité souměrné exploze, oscilátory (periodicky kmitající skupinky), či nekonečnou podívanou na to, jak složité a dokonalé obrazce dokáží tyto dvě podmínky vytvořit. Celý program je koncipován jako hra, máte za úkol vytvořit co nejdéle žijící kolonii. &lt;a href=&quot;soubory/bakterie.zip&quot;','2012-02-14 00:00:00'),
    (3,3,'Cheese Mouse je oddechová plošinovka.','cheese-mouse-oddechova-plosinovka','myš, sýr, hra','Cheese Mouse','Cheese mouse je plošinovka s "horkou ostrovní atmosférou", kde ovládáte myš a musíte se dostat k sýru. V tom vám ale brání nejrůznější nástrahy a nepřatelé jako hadi, krysy, pirane, ale i roboti, mumie a nejrůznější havěť. Hru s několika petrobarevnými světy jsem dělal ještě na základní škole s Veisenem a může se pochlubit 2. místem v Bonusweb game competition, kde vyhrála 5.000 Kč. Vznikala v Game makeru o letních prázdninách, ještě v bezstarostném dětství, což značně ovlivnilo její grafickou stránku. Rád si ji občas zahraji na odreagování a zlepšní nálady. &lt;a href=&quot;soubory/cheesemouse.zip&quot; /&gt;','2004-06-22 00:00:00'),
    (4,2,'Pacman je remake kultovní hry.','pacman-remake','pacman, remake, pampuch, hra, zdarma','Pacman','Jedná se o naprosto základní verzi této hry s editorem levelů, takže si můžete vytvořit svá vlastní kola. Postupem času ji hodlám ještě trochu upravit a přidat nějaké nové prvky, fullscreen a lepší grafiku. Engine hry bude také základem mého nového projektu Geckon man, který je zatím ve fázi psaní scénáře. &lt;a href=&quot;soubory/pacman.zip&quot; /&gt;','2011-06-03 00:00:00');
INSERT INTO "komentare" VALUES
    (1,1,4,'Super článek!','2012-04-06 00:00:00'),
    (2,2,4,'Jak je tedy přesně ta podmínka pro vznik bakterie?','2011-01-28 00:00:00'),
    (3,3,1,'Zasekla jsem se v této hře, kde najdu klíč do 3. levelu?','2011-09-30 00:00:00'),
    (4,3,4,'Jak rozjedu plošinu v 5. levelu?','2010-08-01 00:00:00'),
    (5,4,1,'Umřel jsem a nemám hru uloženou, co mám dělat?','2012-04-14 00:00:00'),
    (6,4,3,'Dobrá hra!','2012-04-06 00:00:00'),
    (7,1,3,'Nerozumím tomu!','2011-04-06 00:00:00'),
    (8,1,2,'Super článek!','2012-05-06 00:00:00');
```

Přidání cizího klíče

Než se začneme zabývat přidáním cizího klíče, je důležité zmínit, že relace mezi tabulkami je možné vytvářet pouze pokud má naše databáze zapnuté cizí klíče. To můžeme zjistit jednoduchým příkazem:

```
PRAGMA foreign_keys;
```

Ten vrátí hodnotu 1, pokud databáze má cizí klíče zapnuté. V opačném případě vrátí 0.

Jelikož databáze z přechozích lekcí má hodnotu nastavenou na OFF, je potřeba to změnit:

```
PRAGMA foreign_keys = ON;
```

V takovýchto relacích je vždy jedna tabulka nadřízená a jedna tabulka podřízená. Nadřízené tabulce se také někdy říká rodičovská tabulka. Vztah (referenci) mezi tabulkami vytváříme vždy z podřízené tabulky k nadřízené tabulce.

Obecný kód pro přidání vztahu je takovýto:

```
FOREIGN KEY (nazev_sloupce) # sloupec podřízené tabulky odkazující na klíč nadřízené tabulky
REFERENCES nadrizena_tabulka(nazev_sloupce)
ON UPDATE ... # co se děje při aktualizaci záznamu
ON DELETE ... # co se děje při smazaní záznamu
```

Tento kód použijeme již při vytváření tabulky takto:

```
CREATE TABLE nazev(
        ...,
        sloupec INTEGER NOT NULL,
        FOREIGN KEY (sloupec)
        REFERENCES nadrizena_tabulka(nazev_sloupce)
        ON UPDATE ...
        ON DELETE ...
);
```

To, že je SQLite odlehčená verze SQL se nám bohužel potvrdí i zde. Na rozdíl od cizích klíčů v MySQL zde nemůžeme přidat cizí klíč do již existující tabulky. Jak tento problém obejít si ukážeme za chvíli.

Pojďme si tedy vytvořit výše uvedený vztah mezi tabulkami clanky a komentare. Pro pořádek si zopakujme, jakého vztahu chceme docílit. Chceme, aby každý článek mohl mít jeden nebo více komentářů a každý komentář mohl být připojený k právě jednomu článku.

Dále je důležité si rozmyslet, co se stane s komentáři, pokud dojde ke smazání článku, ke kterému byly tyto komentáře napsány.

Pokud bychom měli nerelační databázi, tak po smazání článku by nám v databázi zbyly komentáře, které odkazují na neexistující článek.

Mohlo by se také stát, že vytvoříme nový článek s původním identifikátorem (id) článku a naimportovali by se tak původní komentáře. Toto id již mohl převzít jiný článek a díky tomu nám vznikl v databázi zmatek.

Tomu by se dalo předejít tím, že bychom společně se smazáním článku museli projít všechny komentáře a sledovat sloupec clanek_id. Pokud by nějaký komentář měl clanek_id shodné s id článku, museli bychom tuto hodnotu nastavit na NULL.

To je spousta zbytečné práce, které se dá předejít vytvořením jednoduché relace. Pro tento konkrétní příklad budeme chtít, aby došlo ke smazání všech souvisejících komentářů, když odstraníme článek. Také budeme chtít aktualizovat id článku ve všech souvisejících komentářích, pokud by se toto id náhodou změnilo (ale měnit by se určitě nemělo).

Jelikož tabulku již máme vytvořenou, musíme si poradit malou vychytávkou:
Nejprve vytvoříme novou tabulku komentare2, do které přidáme požadovaný vztah:

```
CREATE TABLE IF NOT EXISTS `komentare2` (
        `komentare_id` INTEGER,
        `clanek_id` INTEGER NOT NULL,
        `uzivatel_id` INTEGER,
        `obsah` TEXT,
        `datum` TEXT,
        PRIMARY KEY (`komentare_id` AUTOINCREMENT),
        FOREIGN KEY (`clanek_id`)
        REFERENCES clanky(`clanky_id`)
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
```

Potom spustíme kód níže. Ten překopíruje data z původní tabulky komentare do nové komentare2. Dále smaže starou tabulku komentare a nakonec přejmenuje komentare2 na komentare:

```
INSERT INTO `komentare2` SELECT * FROM `komentare`;
DROP TABLE `komentare`;
ALTER TABLE `komentare2` RENAME TO `komentare`;
```

Tím jsme si zajistili i to, že nebude moci být vytvořený komentář, který nepatří k žádnému článku. Tedy nebude mít clanek_id = ***NULL***. Konkrétně tím, že jsme při vytváření tabulky komentare2 natavili sloupečku clanek_id hodnotu ***NOT NULL***.

Pojďme si nyní projít krok po kroku tyto příkazy. Do závorek za ***FOREIGN KEY*** jsme uvedli název sloupce v podřízené tabulce komentare, který odkazuje na primární klíč v nadřízené tabulce clanky. Za klíčové slovo jsme uvedli tabulku a v závorce konkrétní sloupec na který tento cizí klíč odkazuje. Dále přichází na řadu dva nejdůležitější údaje, které určují, co se stane se souvisejícími záznamy, pokud bude rodičovský záznam upraven nebo smazán. V našem případě jsme hodnotu nastavili na ***CASCADE*** v obou případech. To znamená, že pokud bude rodičovský záznam smazán (tedy náš článek), tak databázový server, společně s ním, kaskádově smaže i související komentáře. To stejné platí i s updatem. Pokud bude rodičovský záznam aktualizován (konkrétně jeho id), bude aktualizace také kaskádově přenesena na související komentáře, díky tomu nám nemůže dojít k nekonzistenci dat.

Pojďme si to není vyzkoušet. Jako první si vypíšeme všechny články:

```
SELECT clanky_id, popis, titulek FROM clanky;
```

Dostaneme tento výsledek:

clanky_id	popis	titulek
1	Co je to algoritmus? Pokud to nevíte, přečtěte si tento článek.	Algoritmus
2	Bakterie jsou obdoba buněčného automatu v kombinaci s hrou.	Bakterie
3	Cheese Mouse je oddechová plošinovka.	Cheese Mouse
4	Pacman je remake kultovní hry.	Pacman

Vypišme si také komentáře k prvnímu článku:

```
SELECT komentare_id, clanek_id, datum FROM komentare WHERE clanek_id = 1;
```

Dostaneme tento výsledek, kde můžeme vidět, že jsou 3:

komentare_id	clanek_id	datum
1	1	2012-04-06 00:00:00
7	1	2011-04-06 00:00:00
8	1	2012-05-06 00:00:00

Nyní pojďme změnit id článku z 1 na 10:

```
UPDATE clanky SET clanky_id = 10 WHERE clanky_id = 1;
```

Když si znovu vypíšeme komentáře, které odkazují na článek s id = 1, tak dostaneme prázdný výsledek, jelikož tyto komentáře byly kaskádově upraveny společně s článkem. Můžeme si to ověřit výpisem komentářů ke článku s id = 10:

```
SELECT komentare_id, clanek_id, datum FROM komentare WHERE clanek_id = 10;
```

Dostaneme tento výsledek:

komentare_id	clanek_id	datum
1	10	2012-04-06 00:00:00
7	10	2011-04-06 00:00:00
8	10	2012-05-06 00:00:00

Jak můžeme vidět, komentáře byly skutečně kaskádově upraveny společně s článkem. Upozorňuji, že původně články byly jen 4 s id 1-4. Nyní pojďme tento článek smazat:

```
DELETE FROM clanky WHERE clanky_id = 10;
```

Když si nyní vypíšeme všechny komentáře ke článku s id = 10, tak dostaneme prázdný výsledek, jelikož tyto komentáře byly kaskádově smazány společně s článkem.

SQLite krok za krokem - Cizí klíče 2

Jak jsme se již naučili v minulé lekci, součástí příkazu, který nám vytvoří vztah mezi tabulkami jsou také dva důležité údaje - ON UPDATE a ON DELETE. Příklad dotazu:

```
FOREIGN KEY nazev_sloupce # sloupec podřízené tabulky odkazující a klíč nadřízené tabulky
REFERENCES nadrizena_tabulka(nazev_sloupce)
ON UPDATE ...
ON DELETE ...
```

Za výraz ***ON UPDATE*** uvádíme, co se bude dít s připojenými entitami, které jsou součástí vztahu, když bude rodičovský záznam (záznam, na který odkazují) aktualizován.
Za výraz ***ON DELETE*** zase uvádíme, co se bude dít s připojenými entitami, když bude rodičovský záznam smazán.

Výrazy ***ON DELETE*** a ***ON UPDATE*** nabývají pěti hodnot:

- **NO ACTION** - SQLite neprovádí se záznamy ve vztahu s upravovaným sloupečkem nic
- **CASCADE** - SQLite kaskádově (postupně) odstraní/upraví všechny připojené záznamy
- **SET NULL** - SQLite nastaví hodnotu cizího klíče, který odkazuje na odstraněnou/upravenou entitu na
- **NULL RESTRICT** - SQLite zabrání úpravě/odstranění záznamů s tímto vztahem
- **SET DEFAULT** - Podobně jako při SET NULL se změní hodnota, tentokrát ale ne na NULL, ale na hodnotu nastavenou jako defaultní (výchozí)

Smazání existujících vztahů

Než budeme v čemkoli pokračovat, je důležité námi vytvořené vztahy také smazat. Pokud si již nepamatujeme, jaké vztahy jsme vytvářeli, můžeme v kartě Databázová struktura vybrat možnost Změnit tabulku:

V dolní části okna se nám potom objeví tento výsledek:

```
CREATE TABLE "komentare" (
        "komentare_id" INTEGER,
        "clanek_id" INTEGER NOT NULL,
        "uzivatel_id" INTEGER,
        "obsah" TEXT,
        "datum" TEXT,
        FOREIGN KEY("clanek_id") REFERENCES "clanky"("clanky_id") ON UPDATE CASCADE ON DELETE
CASCADE,
        PRIMARY KEY("komentare_id" AUTOINCREMENT)
);
```

Tímto jsme získali kód, který SQLite databáze použila pro vytvoření tabulky komentare. Z toho nás zajímá tato část:

```
FOREIGN KEY("clanek_id")
REFERENCES "clanky"("clanky_id")
ON UPDATE CASCADE
ON DELETE CASCADE
```

V této části můžeme vidět, že vztah existuje mezi tabulkami komentare a clanky.

Nyní budeme vztah odstraňovat. Jak na to? Opět to musíme kvůli omezenosti SQLite udělat malou oklikou. Nejprve vytvoříme novou tabulku bez vztahů, potom do ní data z té původní překopírujeme a následně ji přejmenujeme a původní tabulku smažeme:

```
CREATE TABLE IF NOT EXISTS `komentare2` (
        `komentare_id` INTEGER,
        `clanek_id` INTEGER,
        `uzivatel_id` INTEGER,
        `obsah` TEXT,
        `datum` TEXT,
        PRIMARY KEY (`komentare_id` AUTOINCREMENT)
);
INSERT INTO `komentare2` SELECT * FROM `komentare`;
DROP TABLE `komentare`;
ALTER TABLE `komentare2` RENAME TO `komentare`;
```

SET NULL

V minulé lekci jsme si ukázali hodnotu ***CASCADE***, proto budeme pokračovat s hodnotou ***SET NULL***. Jak již víme, pokud dojde k úpravě, nebo smazání rodičovského záznamu, nastaví cizí klíč související záznamy na hodnotu ***NULL***. V první řadě musíme nastavit sloupec clanek_id v tabulce komentare tak, aby mohl být ***NULL***. Dále pak budeme muset přidat samotný vztah. Opět použijeme naši vychytávku, ale na konec dotazu přidáme ještě příkaz k vytvoření vztahu:

```
FOREIGN KEY (clanek_id)
REFERENCES clanky(clanky_id)
ON UPDATE SET NULL
ON DELETE SET NULL;
```

Výsledný dotaz tedy bude vypadat takto:

```
CREATE TABLE IF NOT EXISTS `komentare2` (
        `komentare_id` INTEGER,
        `clanek_id` INTEGER,
        `uzivatel_id` INTEGER,
        `obsah` TEXT,
        `datum` TEXT,
        PRIMARY KEY (`komentare_id` AUTOINCREMENT),
        FOREIGN KEY (clanek_id)
        REFERENCES clanky(clanky_id)
        ON UPDATE SET NULL
        ON DELETE SET NULL
);
INSERT INTO `komentare2` SELECT * FROM `komentare`;
DROP TABLE `komentare`;
ALTER TABLE `komentare2` RENAME TO `komentare`;
```

Pojďme si vypsat obsah tabulky clanky:

```
SELECT clanky_id, titulek FROM clanky;
```

Vidíme, že v databázi existují 3 články (na konci předchozí lekce jsme první článek a k němu

přidružené komentáře vymazali):

clanky_id	titulek
2	Bakterie
3	Mouse
4	Pacman

Nyní si vypíšeme obsah tabulky komentare:

```
SELECT komentare_id, clanek_id, obsah FROM komentare;
```

Dostaneme tento výsledek:

k_id	c_id	obsah
2	2	Jak je tedy přesně ta podmínka pro vznik bakterie?
3	3	Zasekla jsem se v této hře, kde najdu klíč do 3. levelu?
4	3	Jak rozjedu plošinu v 5. levelu?
5	4	Umřel jsem a nemám hru uloženou, co mám dělat?
6	4	Dobrá hra!

Jak můžeme vidět, tak ke článku s id = 4 existují 2 komentáře.

Pojďme si vyzkoušet vztah mezi tabulkami. Odstraňme článek s id = 4:

```
DELETE FROM clanky WHERE clanky_id = 4;
```

Opět vypíšeme obsah tabulky komentare:

```
SELECT komentare_id, clanek_id, obsah FROM komentare;
```

Dostaneme tento výsledek:

k_id	c_id	obsah
2	2	Jak je tedy přesně ta podmínka pro vznik bakterie?
3	3	Zasekla jsem se v této hře, kde najdu klíč do 3. levelu?
4	3	Jak rozjedu plošinu v 5. levelu?
5	NULL	Umřel jsem a nemám hru uloženou, co mám dělat?
6	NULL	Dobrá hra!

Jak můžeme vidět, tak cizí klíč, který odkazoval na smazaný článek, je nyní na hodnotě NULL.

RESTRICT

Nyní si pojďme vyzkoušet hodnotu ***RESTRICT***. Toto nastavení zabrání smazaní článku, pokud k němu existují komentáře. Takové nastavení by v tomto případě nedávalo smysl a více by se hodilo předchozí nastavení, ale pojďme si to přece jen vyzkoušet.

To opět uděláme zopakováním našeho oblíbeného SQLite kolečka pro smazání předchozího vztahu a následné nastavení hodnoty ***ON UPDATE*** a ***ON DELETE*** na ***RESTRICT***:

```
CREATE TABLE IF NOT EXISTS `komentare2` (
        `komentare_id` INTEGER,
        `clanek_id` INTEGER,
        `uzivatel_id` INTEGER,
        `obsah` TEXT,
        `datum` TEXT,
        PRIMARY KEY (`komentare_id` AUTOINCREMENT),
        FOREIGN KEY (clanek_id)
        REFERENCES clanky(clanky_id)
        ON UPDATE SET NULL
        ON DELETE SET NULL
);
INSERT INTO `komentare2` SELECT * FROM `komentare`;
DROP TABLE `komentare`;
ALTER TABLE `komentare2` RENAME TO `komentare`;
```

Ke článku s id = 3 existují dva komentáře, proto pojďme vyzkoušet smazat tento článek:

```
DELETE FROM clanky WHERE clanky_id = 3;
```

SQLite databáze zabránila smazaní článku s následující chybovou hláškou:

```
Execution finished with errors.
Result: FOREIGN KEY constraint failed
At line 1:
DELETE FROM clanky WHERE clanky_id = 3;
```

Tato hláška říká, že nelze odstranit nebo aktualizovat rodičovský záznam, protože existuje vztah, který tomu brání. Musíme tedy před smazáním článku smazat všechny komentáře s ním související a poté smazat článek.

SQL injection

SQL injection je termín, označující narušení databázového dotazu škodlivým kódem od uživatele. Představme si, že naše tabulka s uživateli je součástí databáze nějaké aplikace. A také, že umožníme uživateli (naší aplikace) mazat uživatele podle příjmení. Do dotazu vložíme tedy nějakou proměnnou, která pochází od uživatele:

```
DELETE FROM "uzivatele" WHERE "prijmeni" = '$prijmeni';
```

$prijmeni je proměnná, obsahující třeba tento text:

```
Novák
```

Dotaz se tedy sestaví takto:

```
DELETE FROM "uzivatele" WHERE "prijmeni" = 'Novák';
```

Dotaz se provede a vymaže všechny Nováky. To zní jako to, co jsme chtěli. Teď si ale představte, co se stane, když někdo do proměnné zadá toto:

```
' OR 1 --
```

Výsledný dotaz bude vypadat takto:

```
DELETE FROM "uzivatele" WHERE "prijmeni" = '' OR 1 --';
```

Protože 1 je z logického hlediska vždy pravda a v podmínce je, že buď musí mít uživatel prázdné příjmení nebo musí platit pravda (což platí), vymaže dotaz všechny uživatele v tabulce. Poslední uvozovky se útočník zbavil komentářem (dvě pomlčky), který v dotazu zruší vše do konce řádku. Šikovnější útočníci dokáží udělat injekci v kterémkoli SQL příkazu, nejen v DELETE.

Řešení

Nebojte, řešení je velmi jednoduché. Problém dělá několik speciálních znaků v proměnné, jako jsou uvozovky a několik dalších. Pokud tyto znaky potřebujeme, musíme je tzv. odescapovat, tedy předsadit zpětným lomítkem. V aplikaci to za nás nějakým způsobem řeší ovladač databáze, buď to dělá úplně sám nebo data musíme pomocí něj před vložením do dotazu nejprve odescapovat. Určitě si to zjistěte, než začnete s databází pracovat. Pokud budete používat zdejší návody, bude to v nich vždy uvedeno.

Odescapovaný dotaz by vypadal takto:

```
DELETE FROM "uzivatele" WHERE "prijmeni" = '\' OR 1 --';
```

Takový dotaz je neškodný, protože část vložená uživatelem je považována jako text. V textu se nevyhodnotí uvozovka a tím pádem ani komentář. Další variantou, jak aplikaci zabezpečit proti injekci, je obsah proměnné do dotazu vůbec nezadávat. V dotazu jsou poté uvedeny pouze zástupné znaky (otazníky):

```
DELETE FROM "uzivatele" WHERE "prijmeni" = ?;
```

A proměnné se pošlou databázi potom zvlášť a najednou. Ona si je tam sama navkládá tak, aby nevzniklo žádné nebezpečí. To je však teorie okolo konkrétního ovladače databáze a jak bylo řečeno, naleznete ji u jazyka, ze kterého budete s databází komunikovat (např. v sekci PHP, Python).

Testovací data

```
INSERT INTO "uzivatele" (
    "jmeno",
    "prijmeni",
    "datum_narozeni",
    "pocet_clanku"
)
VALUES
('Jan',  'Novák',  '1984-11-03', 17),
('Tomáš', 'Marný', '1942-10-17', 12),
('Josef', 'Nový', '1958-7-10', 5),
('Alfons', 'Svoboda', '1935-5-15', 6),
('Ludmila', 'Dvořáková', '1967-4-17', 2),
('Petr', 'Černý', '1995-2-20', 1),
('Vladimír', 'Pokorný', '1984-4-18', 1),
('Ondřej', 'Bohatý', '1973-5-14', 3),
('Vítezslav', 'Churý', '1969-6-2', 7),
('Pavel', 'Procházka', '1962-7-3', 8),
('Matěj', 'Horák', '1974-9-10', 0),
('Jana', 'Veselá', '1976-10-2', 1),
('Miroslav', 'Kučera', '1948-11-3', 1),
('František', 'Veselý', '1947-5-9', 1),
('Michal', 'Krejčí', '1956-3-7', 0),
('Lenka', 'Němcová', '1954-2-11', 5),
('Věra', 'Marková', '1978-1-21', 3),
('Eva', 'Kučerová', '1949-7-26', 12),
('Lucie', 'Novotná', '1973-7-28', 4),
('Jaroslav', 'Novotný', '1980-8-11', 8),
('Petr', 'Dvořák', '1982-9-30', 18),
('Jiří', 'Veselý', '1961-1-15', 2),
('Martina', 'Krejčí', '1950-8-29', 4),
('Marie', 'Černá', '1974-2-26', 5),
('Věra', 'Svobodová', '1983-3-2', 2),
('Pavel', 'Dušín', '1991-5-1', 9),
('Otakar', 'Kovář', '1992-12-17', 9),
('Kateřina', 'Koubová', '1956-11-15', 4),
('Václav', 'Blažek', '1953-10-20', 6),
('Jan', 'Spáčil', '1967-5-6', 3),
('Zdeněk', 'Malačka', '1946-3-10', 6);
```

DB Browser for SQLite

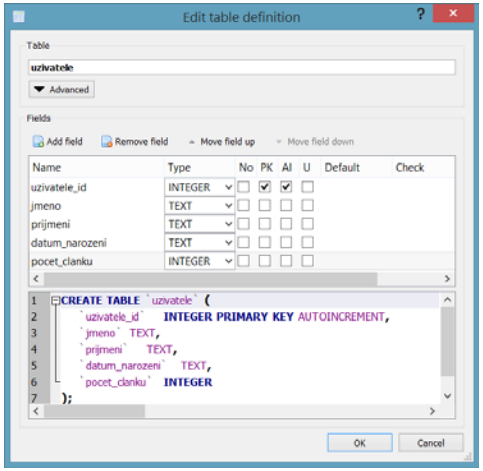
DB Browser for SQLite je velmi oblíbená, jednoduchá aplikace pro prohlížení, úpravu i vytváření SQLite databází v příjemném grafickém kabátu. Není všemocná, složitější věci si budeme muset psát pomocí SQL příkazů, ale pro základní úpravy bez znalostí SQL plně dostačuje a složitější věci se v průběhu seriálu naučíme.

Vytvoření databáze a tabulky v DB brouseru

(Obvykle nám pro jeden projekt (web) postačí jedna databáze.)
Klikněme v DB Browser for SQLite na *New Database*. Vybereme složku, kde chceme databázi uložit a vyplníme název databáze s příponou .db. V databázích je zvykem pojmenovávat položky bez diakritiky, malými písmeny a s podtržítkovou notací. Potvrdíme a vyskočí na nás okno pro vytvoření první tabulky.

První buňka po nás chce jméno tabulky. Každá tabulka by měla mít sloupec, jehož hodnota je pro každou položku unikátní. Tímto sloupcem začneme a klikneme na tlačítko *Add Field*, v tabulce níže se nám objeví řádek, jehož první údaj — jméno — přepíšeme na *uzivatele_id*, datový typ necháme na INTEGER - celém čísle a dále zaškrtneme PK a AI. PK značí PRIMARY KEY, což znamená, že tento sloupec slouží ke identifikaci řádku v tabulce a jeho hodnota musí být unikátní. AI je zkratka AUTOINCREMENT, tedy, že se bude hodnota uzivatele_id automaticky navyšovat a uživatelé se budou postupně číslovat. Id se mi osvědčilo pojmenovávat s prefixem tabulky, ale není to nutné

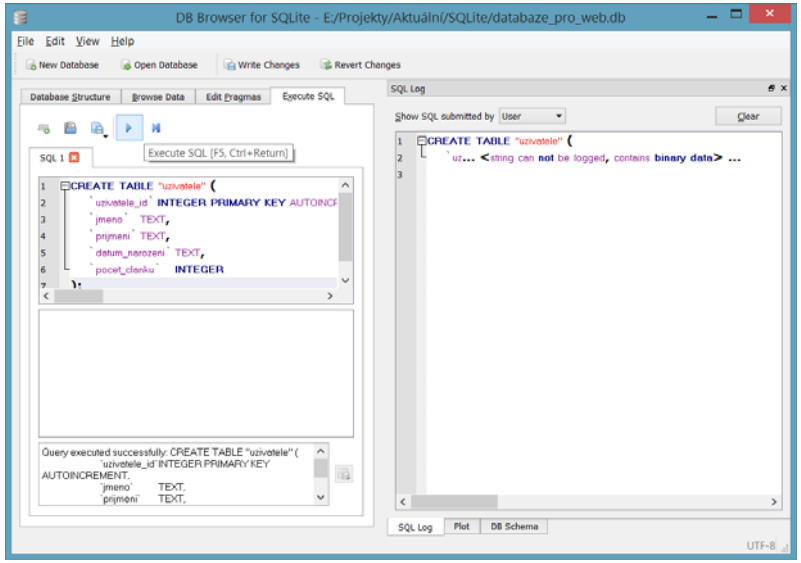
Teď přidáme další sloupce, klikneme znovu na *Add Field* a tentokrát zadáme jmeno. V druhém sloupci nastavíme hodnotu TEXT. Toto provedeme obdobně i pro sloupce prijmeni a datum_narozeni. Poslednímu sloupci se jménem pocet_clanku ponecháme datový typ INTEGER. Všimneme si, že níže se nám vygeneroval kód v jazyce SQL



Po kliknutí na OK uvidíme na obou panelech strukturu naší databáze, kde by se měla nacházet pouze naše tabulka *uzivatele* a vedle ní kód SQL pro její vytvoření, vygenerovaný DB Browserem. Pokud chceme v DB Browseru uložit jakékoliv úpravy, aby se projevily v databázi, musíme nahoře kliknout na *Write Changes*, což uděláme právě teď.

Vytvoření tabulky v DB wievru za použití SQL dotazu

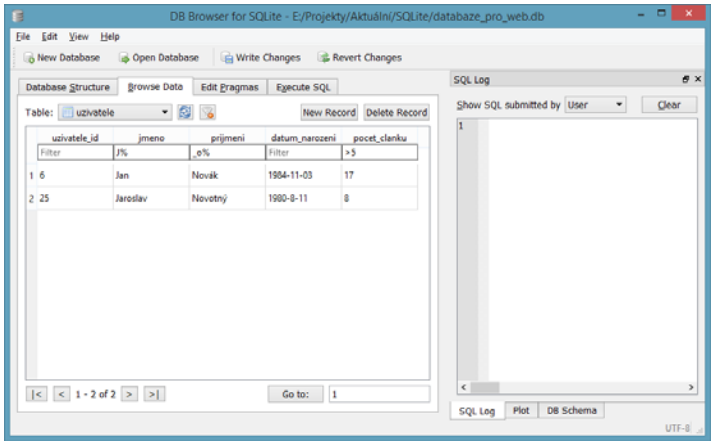
Přejdeme na panel *Execute SQL* a a do textového pole nahoře napíšeme SQL dotaz pro vytvoření tabulky (CREATE TABLE). Po kliknutí na šipku by se nám níže měla zobrazit hláška, že dotaz proběhl úspěšně, to si můžeme zkontrolovat i v panelu *Database Structure*, kde uvidíme úplně to samé, jako po vytvoření tabulky naklikáním.



Zadávání SQL dotazů v DB Browseru má tu výhodu, že jednak vám klíčová slova napovídá, ale také, že můžete psát více SQL dotazů pod sebou a spouštět je jednotlivě pomocí symbolu dvojité šipky, nebo klávesovou zkratkou Ctrl+E.

Dotazování

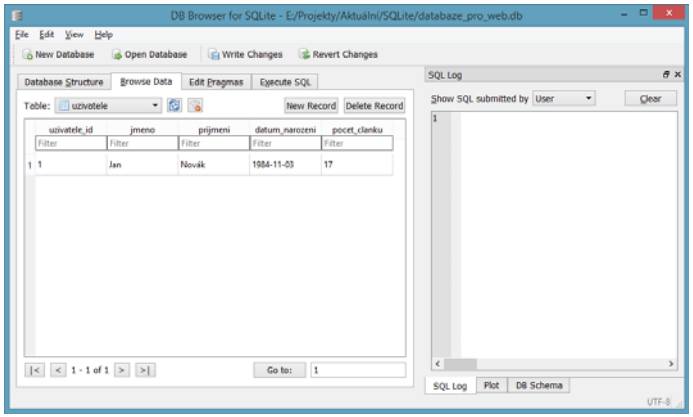
Dotaz na data, tedy jejich vyhledání/výběr naleznete v prostředí DB Browseru na panelu *Browse Data*. Můžete si to zkusit, stačí zadat nějakou hodnotu do nějakého pole *Filter*. Operátory, které ve vyhledávání hrají velkou roli, si vysvětlíme dále.



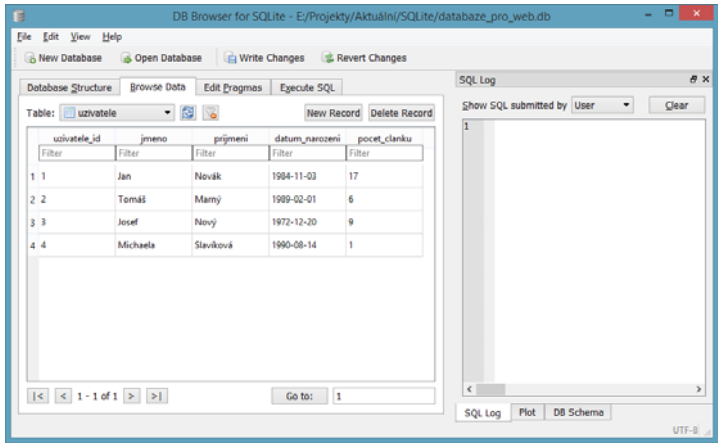
DB Browser byl pro nás zpočátku takovou berličkou, ale nyní pro nás již přestává být zajímavý. Budeme ho používat hlavně pro spouštění dotazů a ukazování jejich výsledků.

Vložení záznamu do tabulky

Vložení nového uživatele si ukážeme opět nejprve přes DB Browser for SQLite. Prvně si otevřeme soubor s databází. Poté přejdeme na panel *Browse Data*, ujistíme se, že níže máme vybranou správnou tabulku *uzivatele* a klikneme na *New Record*. Do tabulky se vložil nový záznam, jehož údaje jsou zatím, kromě *uzivatele_id*, který se vyplní díky AUTOINCREMENT samo, NULL. Po kliknutí na údaj jej můžeme změnit dle libosti.



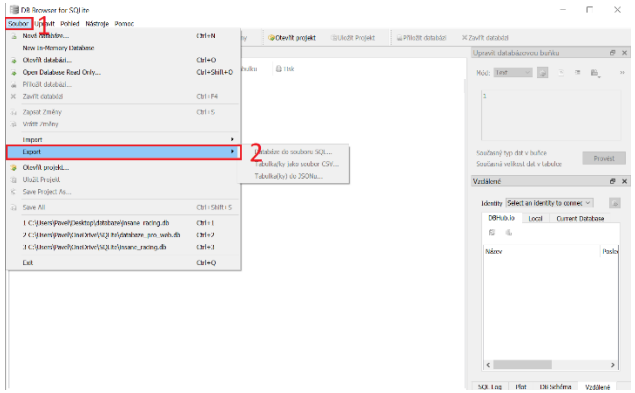
Znovu připomínám, že pokud chceme doopravdy zapsat změny do databáze, musíme kliknout na *Write Changes*, což můžeme preventivně udělat právě teď. A po naklikání pár záznamů bude stránka vypadat nějak takto:



Vymazání záznamu se dělá tím tlačítkem *Delete Record.*

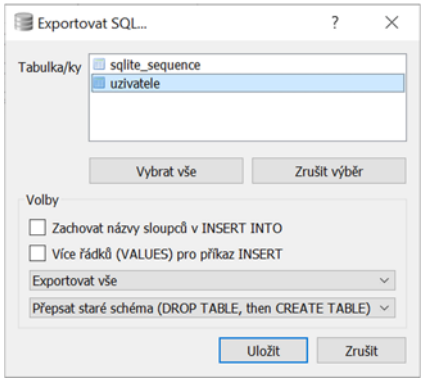
Jak exportovat?

Exportovat databázi nebo tabulku je velmi jednoduché. Stačí kliknout na Soubor -> Export. Dále si můžeme vybrat ze tří možností popsaných níže.



1) Databáze do souboru SQL...

Jak název operace napovídá, tato akce se nám bude hodit, pokud budeme chtít exportovat databázi do souboru .sql.



Výběr tabulek

Nejprve musíme vybrat tabulky, které chceme exportovat. Je možné (stejně jako na obrázku), že bude v nabídce i tabulka sqlite_sequence. Tu si vytváří SQLite samo, pokud existuje tabulka se sloupcem označeným jako AUTOINCREMENT. V tabulce sqlite_sequence je uloženo poslední použité id. Můžeme, ale nemusíme ji vybrat, neboť výsledný soubor to nijak neovlivní.

Možnosti INSERT

Dále máme možnost Zachovat názvy sloupců v INSERT INTO. Soubor sql je skupina příkazů pro RDBMS, které se spouští, a tím se vytváří tabulky a plní se daty. Rozdíl v příkazu bude následující:

Při zaškrtnutí možnosti:

INSERT INTO "uzivatele" ("uzivatele_id","jmeno","prijmeni","pocet_clanku") VALUES (1,'Jan','Novák','1984-11-03','17');

Při NEzaškrtnutí možnosti:

INSERT INTO "uzivatele" VALUES (1,'Jan','Novák','1984-11-03','17');

Více řádků pro INSERT

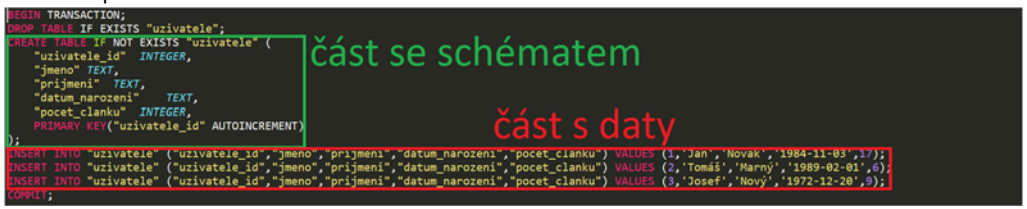
Volba Více řádků (VALUES) pro příkaz INSERT buď použije jeden INSERT, který bude vkládat všechny záznamy, nebo (při nezaškrtnutí) se každý jeden záznam bude vkládat vlastním příkazem INSERT. Stejně jako v předchozím případě je ale výsledek následného importu našeho budoucího souboru stejný. Jedná se pouze o změnu syntaxe.

Obsah exportu

Další nabídka je rozbalovací s možnostmi, co chceme exportovat:

- Exportovat vše
- Exportovat pouze schéma
- Exportovat pouze data

Soubor exportu se může dělit na schéma a data:



Výběrem jedné z možností volíme obsah následného souboru. Soubor může obsahovat:

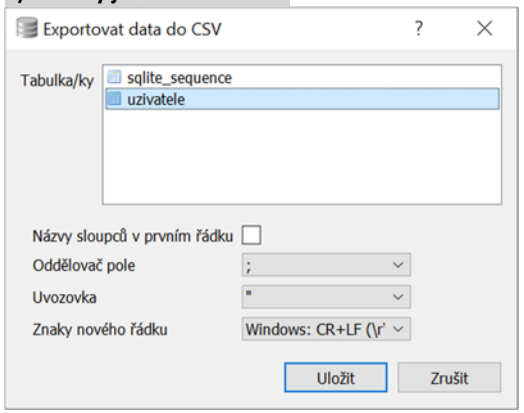
- schéma i data (jako na obrázku výše)
- schéma, které vytvoří prázdnou tabulku
- data, která se vloží do již vytvořené tabulky

Existující tabulka

Poslední rozbalovací položkou volíme mezi možnostmi:

- Keep old schema (CREATE TABLE IF NOT EXISTS) - data se buď připíší do existující tabulky, nebo se vytvoří nová tabulka, do které se vloží exportovaná data
- Přepsat staré schéma (DROP TABLE then CREATE TABLE) - existující tabulka se odstraní a vytvoří se nová tabulka, do které se vloží exportovaná data.

2) Tabulky jako soubor CSV...



***Soubor** CSV* - Comma Seperated Values je soubor, ve kterém jsou data oddělena separátorem (např. čárkou). Je nejen lidsky čitelný, ale umožnuje uložená data importovat například do Excelu. Nejprve si vybereme tabulky, které chceme exportovat. Soubor CSV je ale jednoduchý, jak tedy pozná, jaká data patří jaké tabulce? Odpověď je: nepozná. Proto se pro každou tabulku vytvoří její vlastní CSV soubor.

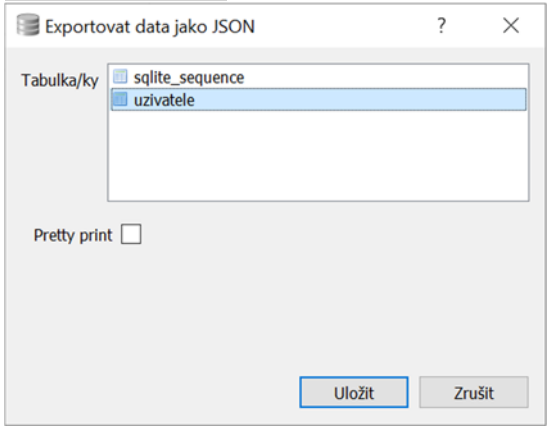
Názvy sloupců v prvním řádku
Zaškrtnutím tohoto checkboxu se exportují i názvy sloupců, která se zapíšou do CSV souboru stejně jako data.

Oddělovač pole
Zde si vybereme znak, kterým budou jednotlivá data v rámci jednoho záznamu (řádku) od sebe oddělena. Můžeme vybrat předem dané znaky, nebo zvolit svůj vlastní pomocí volby Ostatní. Je zde i možnost Karta. Takový soubor potom nemá sloupce viditelně oddělené žádným znakem, nicméně programy typu Excel ho bez problému dokáží i tak správně přečíst.

Uvozovka
Pokud existují data, která obsahují stejný znak, jakým je oddělovač, pak celé slovo je zabaleno do uvozovek. Proto existuje možnost Uvozovka, kde volíme podobu uvozovky.

Znaky nového řádku
Operační systémy používají různé znaky pro označení nového řádku. Zde mámne možnost vybrat, jak budou od sebe jednotlivé řádky (záznamy) odděleny.

3) Tabulky do JSONu...



JSON se často využívá v JavaScriptu a API webových aplikací. Opět máme možnost vybrat, které tabulky se vyexportují.

Pretty print
Zaškrtávací pole Pretty print je označení pro úhledné formátování textu za účelem zvýšení jeho čitelnosti.
S nevybranou volbou Pretty print se všechny údaje vypíší do jednoho řádku:

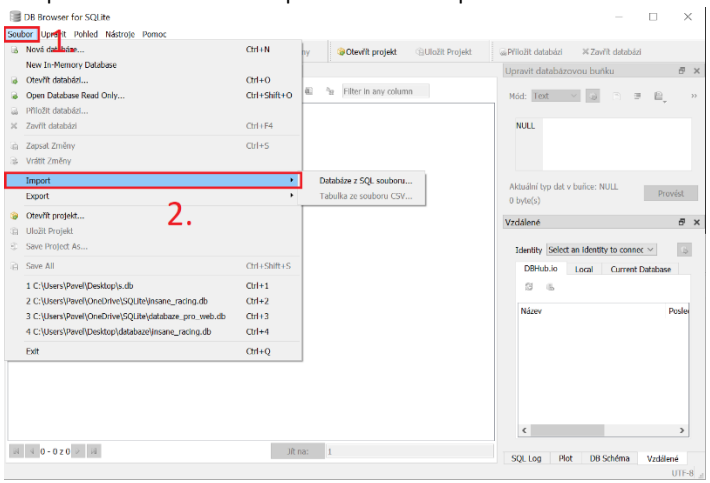

```
[{"datum_narozeni":"1984-11-03","jmeno":"Jan","pocet_clanku":17,"prijmeni":"Novák","uzivatele_id":1},{"datum_narozeni":"1989-02-01
```


S vybranou volbou Pretty print:


```
[
    {
        "datum_narozeni": "1984-11-03",
        "jmeno": "Jan",
        "pocet_clanku": 17,
        "prijmeni": "Novák",
        "uzivatele_id": 1
    },
```

Import

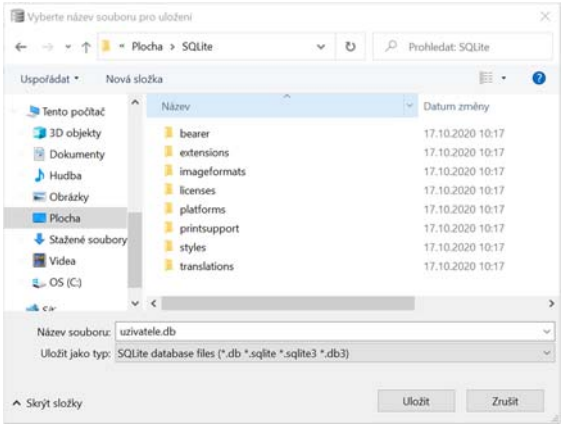
K importu dat se dostaneme přes Soubor -> Import:



A dále můžeme vybrat jednu ze dvou možností: Databáze z SQL souboru... a Tabulka ze souboru CSV.... K importu tabulky musíme mít již připravenou databázi (i kdyby prázdnou), ovšem k importu databáze nemusíme mít nic. Proto začneme importem databáze.

1) Databáze z SQL souboru...

Zvolíme možnost Databáze z SQL souboru.... Poté vybereme náš stáhnutý soubor uzivatele.sql a klikneme na Otevřít. Poté se nám otevře následující okno:

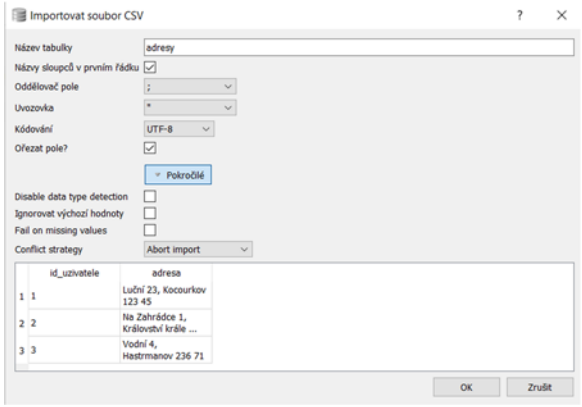


Zde vybereme, kam se uloží samotná databáze, se kterou bude SQLite pracovat. Dojde tedy v podstatě k překladu a překopírování .sql souboru do souboru, se kterým umí pracovat SQLite (v našem případě soubor .db). Tento soubor potom budeme při úpravě obsahu editovat, mezitím, co soubor .sql zůstane nezměněn. Do Název souboru: vložíme uzivatele.db. Poté klikneme na Uložit. Tím se vytvoří soubor .db a databáze se otevře. Import je tím dokončen. Databázi si nechme otevřenou.

Můžeme importovat i soubory s koncovkou .txt, pokud je jejich obsah stejný jako obsah souboru .sql.

2) Tabulka ze souboru CSV...

Teď, když máme naimportovanou databázi, můžeme naimportovat tabulku z našeho stáhnutého souboru adresy.csv. Tentokrát zvolíme tedy druhou možnost Tabulka ze souboru CSV.... Poté vybereme náš stáhnutý soubor adresy.csv a klikneme na Otevřít. Zobrazí se nám okno s možnostmi importu:



Název tabulky

Zde máme možnost ponechat nabídnutý název, nebo si ho změnit. Jedná se o název tabulky, pod kterým se tabulka importuje. My ponecháme nabídnutý název adresy.

Názvy sloupců v prvním řádku

Otevřeme si importovaný soubor adresy.csv v aplikaci Poznámkový blok. V prvním řádku vidíme názvy sloupců id_uzivatele a adresa. Máme tedy exportovanou tabulku včetně názvů sloupců. Můžeme tedy volbu Názvy sloupců v prvním řádku zaškrtnout a importovat hodnoty z prvního řádku jako názvy sloupců tabulky.

Oddělovač pole

Vybereme znak, kterým jsou jednotlivá data v rámci jednoho záznamu (řádku) od sebe oddělena. Zase se podíváme do souboru adresy.csv v aplikaci Poznámkový blok. Vidíme použitý oddělovač ;. Jako Oddělovač pole tedy vybereme ;.

Pokud nenalezneme ve výběru použitý oddělovač z importovaného souboru, můžeme použitý oddělovač vložit pomocí volby Other.
Možnost Karta je pro soubory viditelně neoddělené žádným znakem.

Uvozovka
Pokud v importovaném souboru existují data, která obsahují stejný znak, jakým je oddělovač, pak celé slovo je zabaleno do uvozovek. Proto existuje možnost Uvozovka, kde volíme podobu uvozovky. V našem souboru adresy.csv takový znak nemáme, proto necháme defaultní možnost ".

Kódování
Tato nabídka nám umožní zvolit, jaké kódování používá náš CSV soubor. Aby se nám správně zobrazovala česká diakritika, ponecháme defaultní možnost UTF-8.

Ořezat pole?
Při zaškrtnutí této možnosti se z polí odstraní přebytečné mezery na začátku a na konci hodnot. Např. ze záznamu _____Miroslav____ tak dostaneme pouze Miroslav. (_ v ukázce představuje mezeru). Volbu necháme zaškrtnutou.

Pokročilé nastavení
Po rozkliknutí této volby se nám otevřou další možnosti:

Disable data type detection
Zaškrtnutím vypínáme detekci datových typů - data se naimportují bez uvedení datového typu. Necháme nezaškrtnuto.

Ignorovat výchozí hodnoty
Tato volba se řeší v případě importu do již existující tabulky a importovaný soubor obsahuje prázdná pole. Pokud má tabulka pro tyto pole defaultní hodnotu (takovou, která se vyplní při prázdném poli), bude tato defaultní hodnota doplněna do pole. Pokud však vybereme při importu tuto možnost, defaultní hodnoty se nevloží a pole zůstane prázdné. Necháme nezaškrtnuto.

Fail on missing values
Při zaškrtnutí této možnosti se prověří, zda pro nějaké pole v importovaném souboru platí tyto podmínky:

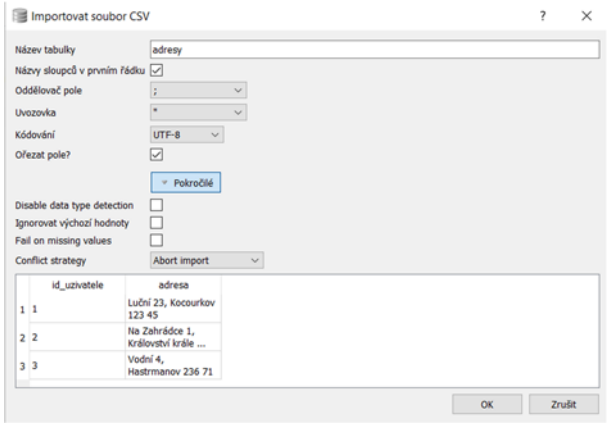
- pole je označené jako NOT NULL tj. musí být vyplněné
- pole není vyplněné
- pro pole není zadána žádná výchozí hodnota

V takovémto případě se import ukončí s chybovou hláškou. Necháme nezaškrtnuto.

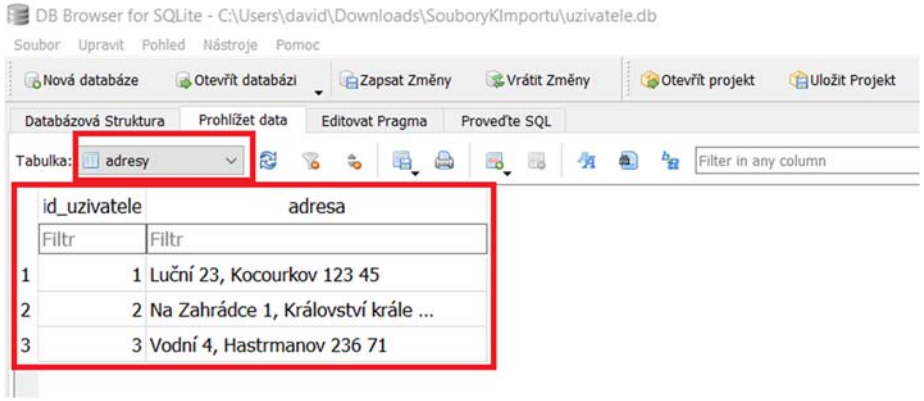
Conflict stratégy
Zde máme možnost vybrat řešení konfliktní situace mezi dvěma hodnotami v poli, které je označené jako unikátní. Tato situace může nastat třeba pro hodnoty pole označené jako primární klíč. Můžeme vybrat tyto možnosti:

- Abort import - Import se ukončí.
- Ignore row - Řádek, který vyvolává konflikt bude při importu přeskočen.
- Replace existing row - Již existující řádek bude nahrazen novými daty z importovaného souboru.

Ponecháme defaultní první možnost Abort import. Nakonec stiskneme OK:



Kontrola dat
Otevřeme si tabulku adresy v SQLite:



Vidíme, že import byl úspěšný.