

SQL III.

Window functions, funkcje a procedury, triggers,
views

Obsah prednášky

- Organizácia
- Windows functions
- Funkcie a procedúry
- Triggers
- View

Organizácia

- Pridané možné riešenia pre cvičenia 2,3
- Pridané príklady pre cvičenie 4
- Otázky k zadaniu 2 na konci prednášky

Window functions

Window functions - motivácia

- Štandardné využitie agregáčnych funkcií nám vyráta výsledok pre určitú skupinu dát (množinu záznamov) a výsledok je v podobe jedného záznamu
 - Strácame informácie o jednotlivých záznamoch
 - Postačuje pre veľké množstvo scenárov – v niektorých nepostačuje alebo menej efektívne

Window functions

- Podobá sa agregáčným funkciám, ale výsledok nie je zoskupený do jedného výstupu
 - Je pridany k aktuálnemu záznamu
 - Dosiahnutie takéhoto výsledku je možné aj bez windows functions
 - Menej čitateľné, môže byť menej alebo tiež viac efektívne

Window functions - syntax

```
SELECT .... nazov_funkcie(....) OVER (.....)  
FROM tabulka
```

Definuje ako prebieha rozdelenie
alebo zoradenie dát.
Tiež sa označuje ako WINDOW

Agregračná funkcia alebo
window funkcie

Špeciálne funkcie

- Predstavujú špeciálne funkcie, ktoré sa viažu na využitie v rámci **window functions**
 - ROW_NUMBER()
 - RANK()
 - DENSE_RANK()
 - PERCENT_RANK()
 - LAG()
 - LEAD()
 - NTILE(n)
 - CUME_DIST
- <https://www.postgresql.org/docs/13/functions-window.html>

Window Functions - Order BY

- V rámci OVER je možné využiť aj ORDER BY pre zoradenie záznamov
 - podľa zoradenia sú následne priradované hodnoty v prípade funkcií pre ranking (slúžia pre ohodnotenie poradia záznamu)

RANKING (II)

- Prirad'ovanie poradia pre jednotlivé záznamy podľa zoskupovacieho výrazu v rámci **OVER()**
- Funkcie, ktoré sem zarad'ujeme
 - RANK() – vytvára poradie pre jednotlivé záznamy. V prípade, že dva záznamy majú rovnakú hodnotu, tak dostávajú rovnaké poradie a ďalší záznam je posunutý o počet výskytov danej hodnoty.
 - napr. dva záznamy dostanu poradie 1, tak ďalší záznam v poradí dostane poradie 3.
 - DENSE_RANK() – nevytvára rozdiel v prípade rovnosti záznamov.
 - napr. dva záznamy dostanu poradie 1, tak ďalší záznam v poradí dostane poradie 2.

RANKING (2)

- Funkcie, ktoré sem zaradujeme

- PERCENT_RANK() – výsledok poradia dáva ako zlomok, ktorý je $(r-1)/(n-1)$, kde **r** je poradie záznamu a **n** je počet záznamov. V prípade že $n = 1$, tak výsledok je **null**
- CUME_DIST – cumulative distribution pre záznam je p/n , kde **p** je počet záznamov, ktoré predchádzajú alebo sú rovné s poradím daného záznamu a **n** je počet záznamov
- ROW_NUMBER – usporiada záznamy a každému záznamu pridá unikátnu hodnotu, ktorá korešponduje s poradím usporiadania. Rovnaké hodnoty v rámci usporiadania majú rozdielnú hodnotu.
- NTILE(n) – pre konštantu **n**, funkcia vytvorí taký počet sektorov, do ktorého rozdelí záznamy podľa spôsobu zoradenia a priradí im hodnotu daného sektora. Možnosť využitia v histogramoch alebo napr. rozdelenie na kvartáli.

Window Functions

- V rámci RANK funkcií vieme definovať, kde sa majú zobrazíť záznamy, ktoré obsahujú null hodnotu
 - **null first**
 - **null last**
- Možnosť použitia viacerých funkcií v rámci jedného SELECT dopytu
 - napr. získanie celkového poradia a poradia v rámci určitej skupiny

Window Functions - PARTITION BY

- V rámci klauzuly **OVER** je možné špecifikovať podľa akého atributu zoskupovať záznamy
 - slúži na to príkaz **PARTITION BY**
 - Podobne ako **GROUP BY**

```
SELECT id_student,  
        id_predmetu  
        ROW_NUMBER() OVER (PARTITION BY id_predmetu)  
FROM zapisane  
ORDER BY id_predmetu
```

WINDOWING (II)

- možnosť definovania fixného okna
 - záznam patrí len do jednej skupiny záznamov napr. rok
 - štandardne ako funguju agregáčné funkcie a tiež s čím sme pracovali doteraz
- je možné definovať aj pohyblivé okno
 - záznam môže patriť do viacerých skupín záznamov napr. rok 2015 môže byť rataný spolu s 2014 a v rámci druhej skupiny spolu s 2016

WINDOWING (2)

- Možnosť definovania ako ma vyzeráť dané okno, ktoré záznamy sa budú brať do danej skupiny
- Možnosť definovania **ROWS**
 - **PRECEDING** – predchádzajúce záznamy
 - ROWS 3 PRECEDING
 - **FOLLOWING** – nasledujúce záznamy
 - **BETWEEN** – definovanie rozsahu napr. 2 pred a 3 nasledujúce
 - ROWS BETWEEN 3 PRECEDING AND 2 FOLLOWING
 - **CURRENT ROW** – aktuálny záznam
 - ROWS BETWEEN 3 PRECEDING AND CURRENT ROW
 - **UNBOUNDED** – neobmedzené pred alebo podľa toho, či je použité FOLLOWING alebo PRECEDING
 - ROWS UNBOUNDED PRECEDING

WINDOWING (3)

```
SELECT firstname,lastname,  
        AVG(goals + assists)  
        OVER (ORDER BY year  
        ROWS BETWEEN 3 PRECEDING AND  
        2 FOLLOWING) AS points  
FROM seasons;
```


Funkcie a procedúry

SQL funkcie a procedúry

- Okrem štandardných funkcií v SQL (napr. operácie s reťazcami znakov, agregačné funkcie atď.) je možné vytvoriť aj vlastné funkcie
- Okrem definovania vlastných funkcií je možné definovať aj vlastné procedúry
- SQL štandard definuje syntax pre vytváranie funkcií a procedúr jednotlivé DBMS však využívajú vlastné verzie syntaxe
 - princíp je rovnaký s SQL štandardom
 - potrebné je naučiť sa zakaždým konkrétnu syntax pre daný DBMS
 - Oracle (PL/SQL), Microsoft SQL server (TransactSQL), PostgreSQL (PL/pgSQL)

Výhody

- Presunutie biznis logiky bližšie k DB – nie je nutné ju realizovať v rámci aplikácie
 - výhoda v prípade, keď rôzne aplikácie používajú rovnakú funkcionality (funkciu) je možnosť zmeniť len v DB a nie je nutné prepisovať každú aplikáciu zvlášť

Funkcia vs Procedúra

Procedúra	Funkcia
Môže vrátiť nula, jednu alebo viacero hodnôt	Musí vrátiť hodnotu (skalár, tabuľku)
Možnosť použitia transakcie v rámci procedúry	Nie je možné použiť transakciu v rámci funkcie
Môže mať input a output parametre	Iba input parametre
Možnosť volania funkcií z procedúry	Nie je možné volať procedúru z funkcie
Nie je možné použiť v rámci SELECT/WHERE/HAVING	Možnosť použiť v rámci dopytu (SELECT/WHERE/HAVING)
Možnosť modifikovania objektov v DB	Nie je možné modifikovať objekty v DB

Definovanie a použitie funkcie

```
CREATE FUNCTION dept_count (dept_name VARCHAR(20))  
  RETURN INTEGER  
BEGIN  
  DECLARE d_count INTEGER;  
  SELECT COUNT(*) INTO d_count  
  FROM instructor  
  WHERE instructor.dept_name =dept_name  
  RETURN d_count;  
END
```

```
SELECT dept_name, budget  
FROM department  
WHERE dept_count(dept_name) > 12
```

Nami vytvorená
funkcia

Definovanie a použitie procedúry

```
CREATE PROCEDURE dept_count_proc ( IN dept_name VARCHAR(20), OUT d_count INTEGER)
BEGIN
    SELECT COUNT(*) INTO d_count
    FROM instructor
    WHERE instructor.dept_name = dept_count_proc.dept_name
RETURN d_count;
END
```

```
DECLARE d_count INTEGER;
CALL dept_count_proc('physics',d_count);
```

Nami vytvorená
procedúra

Performance

- Závisí aké operácie vykonávajú jednotlivé funkcie/procedúry
- Pozor na performance:
 - v prípade komplexných funkcií, ktoré vykonávajú operácie nad veľkým množstvom záznamov
 - potreba vždy zvážiť použitie funkcie

Procedúry/Funkcie a SQL

- SQL podporuje takmer všetky konštrukcie, ktoré sa využívajú v ostatných programovacích jazykoch
 - sú definované v štandarde SQL v rámci časti **Persistent Storage Module (PSM)**
- V rámci SQL vieme uskutočniť
 - while, repeat
 - for
 - If-then-else
 - case
 - Exceptions a handlers
 - viac v popise štandardu SQL časť PSM
- SQL tiež umožňuje aby jednotlivé procedúry obsahovali rovnaké meno s rôznym počtom argumentov prípadne rovnaký počet argumentov s tým, že aspoň jeden argument je iného typu
 - názov a počet argumentov identifikuje procedúru

Procedúry/Funkcie externé programovacie jazyky

- Samotné SQL umožňuje volanie procedúr/funkcií realizovaných pomocou externých programovacích jazykov
- Externé programovacie jazyky napr. C sa musia vyrovnáť s hodnotou null, ktorá je špecifiká v rámci SQL
- Riziko vykonávania funkcií a procedúr mimo SQL je, že v prípade chyby v rámci danej funkcie je možné spôsobenie chyby v rámci internej štruktúry databázy a tiež sa znižuje bezpečnosť systému
 - externý program získava čiastočnú kontrolu nad DBMS

Triggers

Trigger

- predstavuje príkaz, ktorý databáza vykoná automatický ako vedľajší efekt modifikácie databázy
- Pre **trigger** je potrebné definovať nasledujúce
 - Kedy má byť Trigger vykonaný
 - **Udalosť** - ktorá spustí kontrolu daného Triggera
 - **Podmienka** - ktorá musí byť splnená aby Trigger mohol pokračovať ďalej vo vykonávaní
 - Akciu - ktorá sú vykonané v prípade, že boli splnené podmienky pre spustenie daného **trigger-a**

Využitie

- Pre implementovanie integritných obmedzení, ktoré nie je možné dosiahnuť mechanizmami obmedzení v rámci SQL
- Pre možnosti upozornení v prípade splnenia určitých podmienok
- Aktualizovanie inej hodnoty v tabuľke

Trigger - SQL

- Rovnako ako v prípade funkcií a procedúr, SQL štandard definuje syntax pre používanie Trigger.
 - Väčšina implementácií DBMS však používa vlastnú neštandardnú syntax
 - využívajú však podobné princípy

Trigger - ukážka

- Ukážka Triggera zabezpečujúceho referenčnú integritu atribútu ***time_slot_id*** v tabuľke ***section***

```
CREATE TRIGGER timeslot_check1 AFTER INSERT ON section
REFERENCING NEW ROW AS nrow
FOR EACH ROW
WHEN (nrow.time_slot_id NOT IN (
        SELECT time_slot_id
        FROM time_slot))
BEGIN
        ROLLBACK
END;
```

Trigger - ukážka

- Ukážka Triggera zabezpečujúceho referenčnú integritu atribútu *time_slot_id* v tabuľke *section*

```
CREATE TRIGGER timeslot_check1 AFTER INSERT ON section
REFERENCING NEW ROW AS nrow
FOR EACH ROW
WHEN (nrow.time_slot_id NOT IN (
    SELECT time_slot_id
    FROM timeslot))
BEGIN
    ROLLBACK;
END;
```

Definuje, že trigger je iniciovaný po každom pridaní (Insert operácií)

Vytvorí premennú s názvom nrow (nazývana aj **Transition variable**), ktorá obsahuje hodnotu vloženého riadka/záznamu

INSERT môže vkladať viacero záznamov a preto sa TRIGGER vykoná pre každý riadok samostatne

WHEN špecifikuje podmienku

Definuje akcie, ktoré sa vykonajú v prípade splnenia podmienky

Trigger - ukážka 2

- Zabezpečuje referenčnú integritu pri operácii delete nad tabuľkou ***time_slot*** a atribútom ***time_slot_id***
- Trigger skontroluje, či je daný časový slot stále prítomný v rámci tabuľky timeslot a či existuje daký záznam v tabuľke section, ktorý odkazuje na daný časový slot
 - ak existuje, tak urobí rollback

```
CREATE TRIGGER timeslot_check2 AFTER DELETE ON time_slot
REFERENCING OLD ROW AS orow
FOR EACH ROW
WHEN (orow.time_slot_id NOT IN (
    SELECT time_slot_id
    FROM time_slot)
AND orow.time_slot_id IN (
    SELECT time_slot_id
    FROM section))
BEGIN
    ROLLBACK
END;
```


Trigger - ukáž

Definuje, že trigger je iniciovaný po každom odobratí záznamu (DELETE operácií)

- Zabezpečuje referenčnú integritu pri operácii delete nad tabuľkou **time_slot** a atribútom **time_slot_id**
- Trigger skontroluje, či je daný časový slot stále prítomný v rámci tabuľky timeslot a či existuje daký záznam v tabuľke section, ktorý odkazuje na daný časový slot
 - ak existuje, tak urobí rollback

```
CREATE TRIGGER timeslot_check2 AFTER DELETE ON time_slot
REFERENCING OLD ROW AS orow
FOR EACH ROW
WHEN (orow.time_slot_id NOT IN (
    SELECT time_slot_id
    FROM time_slot)
AND orow.time_slot_id IN (
    SELECT time_slot_id
    FROM section))
BEGIN
    ROLLBACK;
END;
```

Uloženie vymazaného riadku do premennej

Trigger

- Pre zabezpečenie referenčenej integrity by bolo však potrebné vytvoriť Trigger pre operáciu **UPDATE**
 - v rámci Trigger by sme použili operáciu **AFTER UPDATE OF ... ON ...**
 - **OF** určuje tabuľku **ON** určuje atribút
 - **ON** nemusí byť prítomný –
 - Použité keď chceme špecifikovať v akom prípade sa ma uskutočniť daný TRIGGER
- REFERENCING **NEW** ROW AS
 - vieme použiť pri operáciach INSERT, UPDATE
- REFERENCING **OLD** ROW AS
 - vieme použiť pri operáciach UPDATE, DELETE

Trigger

- môže byť definovaný aj pred vykonaním samotnej udalosti (INSERT, UPDATE, DELETE)
 - **BEFORE UPDATE OF ... ON**
 - výhoda je, že predtým ako sa zmena uskutoční a vyvolá error, vieme ju opraviť na oprávnenú hodnotu
 - napr. prázdnu hodnotu zmeníme na null
- Okrem FOR EACH ROW je možné robiť aj TRIGGER nad celým príkazom **FOR EACH STATEMENT**
- Okrem REFERENCING NEW/OLD ROW AS existuje aj **REFERENCING NEW/OLD TABLE AS**
 - je možné použiť iba v prípade **AFTER** eventov

Trigger - vypnutie a vymazanie

- Trigger môže byť vymazaný pomocou DROP
 - vtedy ho nie je možné znova použiť a je ho potrebné znova vytvoriť
- Je ho možné aj DISABLE, kedy sa daný Trigger nevykonáva a je ho možné neskôr opäť aktivovať
- PostgreSQL
 - V prípade nanovo vytvorenej funkcie je potrebné znova vytvoriť Trigger. Pôvodný trigger obsahuje referenciu na starú funkciu viac postgresql dokumentácia

Trigger

- nie je vždy vhodné používať Trigger
 - v prípade, že existuje vstavaná funkcionálna DBMS, ktorá vykonáva zamýšľanú funkcionálnu
 - napr. ON DELETE CASCADE , udržiavanie materialized VIEW ...
 - keď nahrávame dáta do DB z backup súboru – v takom prípade už triggere boli spustené nad danými dátami – je vhodné vypnúť Triggere aby nedochádzalo k opätovnému spusteniu
 - niektoré DBMS umožňujú nastaviť Trigger ako **NOT FOR REPLICATION**
- Zlyhanie Trigger spôsobí zlyhanie celého výrazu
- Trigger môže spustiť ďalší trigger
 - môžu vzniknúť cykly
 - niektoré DBMS limitujú počet Trigger, ktoré je možné spustiť za sebou

View

Motivácia

- Je možné ukryť určité dáta pred niektorými používateľmi
 - nie je vždy nutné aby každý používateľ videl všetko napr. platy zamestnancov
- Umožňujú zjednodušenie dopytov
 - neznamená však že sú rýchlejšie
- flexibilnejší prístup k dátam – niečo ako interface – ak sa zmení aj štruktúra pod pohľadom, tak aplikácia sa nemení - máte rovnakú štruktúru
 - závisí však od prípadu ako je vytvorený view

Prečo View?

- Je možné výsledok SELECT-u uložiť do tabuľky a následne ich sprístupniť používateľom
- Takýto prístup má nevýhodu v tom, že v prípade modifikovania tabuliek, z ktorých vznikla nová tabuľka, tak táto nová tabuľka neobsahuje aktualizované dáta
 - sú uložené na disku, čo môže mať v určitých prípadoch opodstatnenie

View

- View nie je uložené na disku
 - predstavuje "virtuálnu" tabuľku
 - jeho hodnoty sa vytvárajú vždy, keď niektorý dopyt pracuje s daným View
 - v prípade, že by dochádzalo k ukladaniu na disk hrozí neaktuálnosť údajov
- Zvyčajná implementácia View v rámci DBMS
 - vždy keď dopyt využíva View, tak View je nahradený výrazom, ktorý je vykonávaný pre definované View

syntax:

```
CREATE VIEW name AS query_expression;
```

```
CREATE VIEW faculty AS
```

```
    SELECT id, name, dept_name
```

```
    FROM instructor;
```

View - použitie

- definovaný View je možné použiť kdekoľvek v dopyte, kde je možné vkladať tabuľku (relation)
- možnosť definovania mien stĺpcov pre jednotlivé View

```
CREATE VIEW department_total_salary(dept_name, total_salary) AS  
    SELECT dept_name, SUM(salary)  
    FROM instructor  
    GROUP BY dept_name;
```

- Definované View môže byť použité pre definovanie ďalšieho View

Pridávanie a aktualizácia dát

- problém s pridávaním, modifikáciou a mazaním dát v rámci View
 - je potrebné ich pretransformovať na modifikáciu tabuliek s ktorými pracuje View

```
CREATE VIEW faculty AS  
    SELECT id, name, dept_name  
    FROM instructor;
```

```
INSERT INTO faculty  
VALUES ('30765', 'Green', 'Music');
```

Pridanie záznamu - príklad I

- Máme

```
CREATE VIEW faculty AS  
    SELECT id, name, dept_name  
    FROM instructor;
```

```
INSERT INTO faculty  
VALUES ('30765', 'Green', 'Music');
```

- Tabuľka *instructor* obsahuje atribúty: *ID*, *name*, *dept_name*, *salary*
- Pri použití *INSERT INTO* do View je potrebné aktualizovať práve tabuľku *instructor*, kde však máme aj parameter *salary*
- Dve možnosti riešenia:
 - Odmietneme pridanie záznamu a vrátime error používateľovi
 - Pridáme záznam ('30765', 'Green', 'Music', *null*) do tabuľky *instructor*

Pridanie záznamu - príklad 2

- Máme View

```
CREATE VIEW instructor_info AS  
    SELECT id, name, building  
    FROM instructor, department  
    WHERE instructor.dept_name = department.dept_name;
```

- Ideme vložiť nasledujúci záznam do View

instructor

ID	name	dept_name	salary
12458	Kamensky	Sport	10000

department

dept_name	building	budget
Sport	PepsiCenter	50000

```
INSERT INTO instructor_info  
VALUES ('69987', 'White', 'Taylor');
```

Pridanie záznamu - príklad 2

- Tabuľky DB

instructor

ID	name	dept_name	salary
12458	Kamensky	Sport	10000

department

dept_name	building	budget
Sport	PepsiCenter	50000

- Problémy spojené s príkladom 2
 - predpokladáme, že neexistuje inštruktor s daným ID
 - jediný spôsob ako urobiť pridanie záznamu do View je pridanie záznamov do dvoch tabuliek
 - do tabuľky instructor ('69987', 'White', null, null)
 - do tabuľky department (null, 'Taylor', null)
 - dané inserty nemajú však vplyv na zobrazenie záznamu vo View (sú vyberané podmienkou instructor.dept_name = department.dept_name kde sa porovnávajú hodnoty **null**)
 - môže nastať problém s obmedzeniami v tabuľkách ako je napr. pridanie null hodnoty ako primárneho kľúča

View - pridanie a aktualizácia dát

- V dôsledku vznikajúcich problémov s modifikáciou nad view je takáto modifikácia zakázaná a je dovolená iba v určitých prípadoch
 - je to závisle od použitia DBMS
- Všeobecné podmienky modifikácie v rámci View sú:
 - View obsahuje v rámci klauzuly FROM iba jednu tabuľku
 - SELECT obsahuje iba mena atributov a žiadne expresné výrazy, agregácie alebo DISTINCT
 - Atribúty, ktoré nie su uvedené v SELECT klauzule môžu byť nastavené na NULL a teda nevyvolaju podmienku obmedzenia a rovnako nie sú súčasťou primárneho kľúča
 - SELECT neobsahuje GROUP BY a HAVING
- Splnenie všeobecných podmienok modifikácie ešte nerieši problém, že v prípade pridania záznamu sa daný záznam neobjaví v rámci View
 - možnosť obmedzenia cez **WITH CHECK OPTION**

Materialized View

- určité DBMS umožňujú uloženie View na disk a v prípade, že dôjde k zmene údajov, tak View je udržiavaný mechanizmami aktuálny
- výhoda je rýchlosť získania výsledku – nie je nutné znova robiť dopyt pre získanie View
 - pre aplikácie, ktoré využívajú často View
 - pre dopyty, ktoré vykonávajú agregáciu nad veľkým množstvom záznamov
- Aktualizácia dát v rámci DBMS pre materialized View môže prebiehať
 - okamžite po modifikácii dát
 - po požiadavke na prístup daného View
 - periodický – hrozí neaktuálnosť dát vo View – nie je vhodné použiť ak aplikácia vyžaduje aktuálnosť dát
- Niektoré DBMS umožňujú vybrať si spôsob ako bude materialized View udržiavaný
- Treba brať do úvahy overhead pri aktualizácii dát a tiež dodatočné požiadavky na úložisko

Zhodnotenie

- Windows functions
 - Možnosť definovania pohyblivého okna a aplikovanie funkcií pre jednotlivé záznamy
- Triggers
 - Presunutie logiky bližšie k DB
- View
 - Možnosť skrytia určitých dát pred používateľmi

Čo ďalej?

- Ako vytvoriť návrh databázy