SQL

Stored procedures

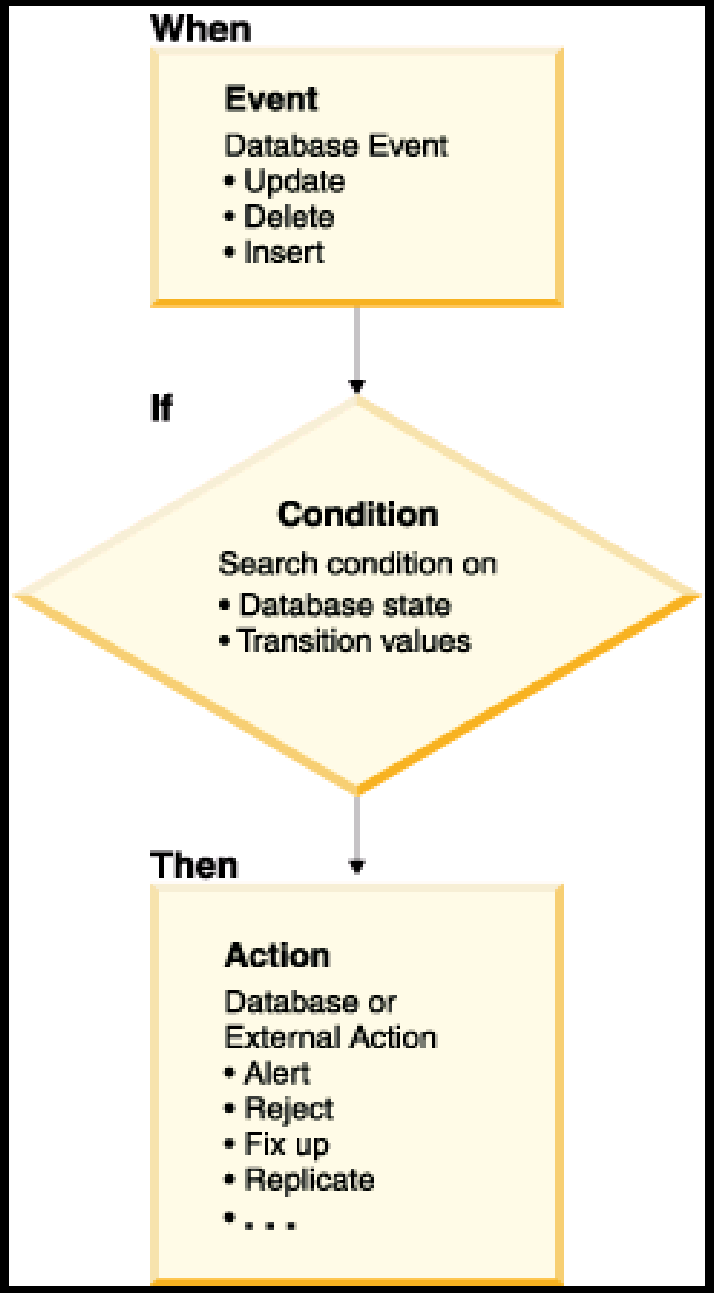
Triggers

Triggers

- Trigger (spoušť) je procedura která je automaticky spuštěna DBMS jako reakce na specifikovanou akci v databázi
- Aktivní databáze:
  - DB, která obsahuje definici spouští

Popis spouště

- Event (událost) - změna v DB, která vyvolá spuštění
- Condition (podmínka) – dotaz nebo test, který je proveden pokud je spoušť aktivována
- Action (akce) – procedura, která je provedena při spuštění a pokud je splněna podmínka


When
Event
Database Event
• Update
• Delete
• Insert
If
Condition
Search condition on
• Database state
• Transition values
Then
Action
Database or
External Action
• Alert
• Reject
• Fix up
• Replicate
• . . .

Event (akce)

- INSERT
- UPDATE
- DELETE

- Nezáleží na to kdo (který uživatel) provede akci
- Uživatel si není vědom aktivity spouští v DB

Příklad použití

- Mějme DB studentů (STUDENT)
- Mějme dále tabulku obsahující informace o počtu studentů mladších 18 let
- Při vložení nového záznamu do tabulky STUDENT je aktivována spoušť, která zvýší atribut v tabulce obsahující počet studentů mladších 18 let

After x before

- Podle funkce je možné tuto změnu provést před nebo po provedení změn nad tabulkou STUDENT

Triggers in SQL

```
CREATE TRIGGER init_count
BEFORE INSERT  ON Students/* event */
DECLARE
 count INTEGER;
BEGIN
 count:=0;                    /* action */
END
```

```
CREATE TRIGGER inc_count
AFTER INSERT  ON Students /* event */
WHEN (new.age<18)      /* condition */
FOR EACH ROW
BEGIN                          /* action */
 count:=count+1;
END
```

Jak zjistím, kterou hodnotou je v DB pracováno?

- New
  - Označuje nově vkládanou/upravovanou n-tici (řádek) do relace v DB
- Old
  - Pokud byla hodnota n-tice v DB měněna pak se lze pomocí *new* odkázat na nově vložená data
  - *Old* data před změnou

Trigery x Omezení

- Spouště slouží v DB k zajištění konzistence dat v DB
- IO slouží ke stejnému účelu, zachování konzistenci dat vždy
- Spouště jsou aktivovány pouze na určité stimuly

Příklad, kdy je lépe použít Trigger

- Mějme relaci
  - Objednavky (oid,počet,idzakaznika, jednotkovacena)
- Při založení objednávky jsou první tři atributy vyplněny zákazníkem (případně prodavačem na pokladně)
- Atribut *jednotkovacena* je vyplněn podle jiné relace *produkty*

- Atribut *jednotkovacena* je však nutné vyplnit, aby byla objednávka kompletní!
- Pokud by nebyla uvedena, pak by při pozdější změně (např. sleva), byla změněna i ve všech objednávkách již uzavřených

- Napíšeme trigger, který nám tuto hodnotu z tabulku produkty zjistí a vloží do tabulky objednávky
- Je to vhodné vzhledem k ušetření práce prodavače a zároveň to minimalizuje možnost vzniku chyby při vkládání dat a tím pádem i vzniku nekonzistence v DB

- Mějme dále za úkol provést ještě další kontroly
- Například při platbě chtějme kontrolovat, zda-li celková cena není větší než, zůstatek na účtu zákazníka
- To se dá provést pomocí triggeru, ale i CHECK

- Nicméně použití triggeru nám umožní implementovat i složitější kontrolní mechanizmy
- Např. můžeme dovolit zákazníkovi překročit jeho limit účtu o ne více než 10%, pokud má zákazník u naší společnosti záznam déle než 1 rok.
- Dále pak můžeme požadovat přidání zákazníka do tabulky pro zvýšení limitu platby

Condition (podmínka)

- true/false statement (věk>18)
- Dotaz
  - Pokud dotaz vrátí neprázdnou množinu jako výsledek, pak odpovídá TRUE
  - Jinak FALSE

SQL Oracle syntax

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
{BEFORE|AFTER} {INSERT|DELETE|UPDATE}
ON <table_name>
[REFERENCING [NEW AS <new_row_name>]
[OLD AS <old_row_name>]]
[FOR EACH ROW [WHEN (<trigger_condition>)]]
<trigger_body>
```

Trigger granularita

- FOR EACH ROW
  - Provede se tolikrát, kolik je řádek v množině ovlivněných záznamů.
  - Pokud je třeba odkazovat/používat konkrétní řádku/řádky z množiny ovlivněných záznamů, pak použít FOR EACH ROW.
  - Příklad: pokud chceme porovnávat hodnoty nově vložených a starých záznamů v případě AFTER UPDATE triggeru.
- FOR EACH STATEMENT
  - Provede se jednou pro celý trigger/událost.
- Pokud je množina ovlivněných záznamů prázdná (např. pokud není splněná podmínka pro UPDATE, DELELE, či INSERT), pak
  - FOR EACH ROW trigger se neprovede vůbec
  - FOR EACH STATEMENT trigger se spustí/provede.

Příklad-
udržování počtu zaměstnanců pomocí FOR EACH ROW

```
CREATE TRIGGER NEW_HIRED
AFTER UPDATE ON EMPLOYEE
FOR EACH ROW
 UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

To samé s FOR EACH STATEMENT

```
CREATE TRIGGER NEW_HIRED
AFTER UPDATE ON EMPLOYEE
REFERENCING NEW_TABLE AS NEWEMPS
FOR EACH STATEMENT
  UPDATE COMPANY_STATS
  SET NBEMP = NBEMP + (SELECT COUNT(*) FROM NEWEMPS)
```

Aktivace Triggeru

- BEFORE, AFTER aktivaci/provedení spouště.
- Například, aktivace následují spouště je po (AFTER) provedení operace INSERT nad tabulkou employee

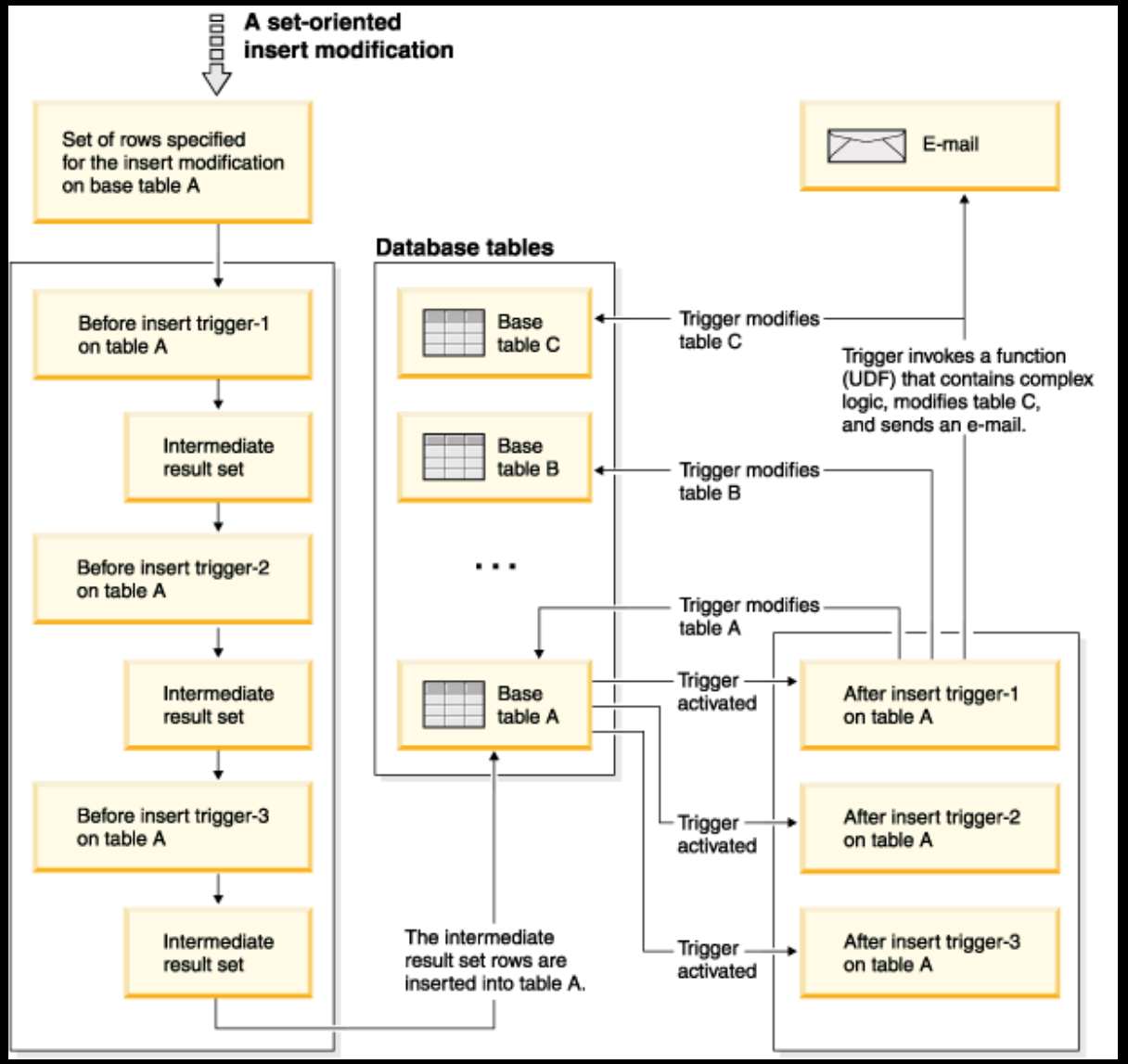
```
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
 UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

Before

- Pokud je jako čas aktivace zvoleno BEFORE, akce spouště jsou aktivovány pro každý řádek ovlivněných záznamů před provedení vlastního dotazu nad databází.
- Z toho vyplývá, že tabulka, nad kterou se provádí dotaz, bude modifikována až po provedení všech operací BEFORE spouště.
- Pozn. BEFORE spoušť musí mít granularitu FOR EACH ROW.

After

- Akce spouště jsou aktivovány for each row v množině ovlivněných záznamů nebo pro daný příkaz (záleží na granularitě spouště).
- Spoušť je aktivována až po provedení všech kontrol integritních omezení, které může spoušť (akce ve spoušti) ovlivnit.
- Pozn. AFTER spouště mohou mít granularitu
  - FOR EACH ROW
  - FOR EACH STATEMENT.


A set-oriented
insert modification
Set of rows specified for the insert modification on base table A
Before insert trigger-1 on table A
Intermediate result set
Before insert trigger-2 on table A
Intermediate result set
Before insert trigger-3 on table A
Intermediate result set
Database tables
Base table C
Base table B
Base table A
E-mail
Trigger modifies table C
Trigger invokes a function (UDF) that contains complex logic, modifies table C, and sends an e-mail.
Trigger modifies table B
Trigger modifies table A
Trigger activated
Trigger activated
Trigger activated
After insert trigger-1 on table A
After insert trigger-2 on table A
After insert trigger-3 on table A
The intermediate result set rows are inserted into table A.

- Pokud jsou nad tabulkou definovány jak before tak i after spouště, pak se jako první provedou všechny before spouště
  - První spoušť, která je spuštěna vezme jako vstup množinu záznamů, které budou ovlivněný dotazem (UPDATE, INSERT, DELETE) a provede veškeré změny definované v rámci spouště.
  - Výstup první before spouště je pak vstupem následující before spouště.
  - Jakmile jsou aktivovány a dokončeny všechny before spouště, jsou všechny změny provedeny na databázových objektech (včetně vlastního dotazu co spoušť aktivoval)
- Následně jsou aktivovány všechny after spouště asociované s danou akcí.
  - After spouště pak mohou modifikovat stejnou/jinou tabulku, mohou také spouštět externí akce (poslat email)

Použití Before

- BEFORE spouště jsou jakýmsi rozšířením systém integritních omezení.
- Používají se pro:
  - Provedení validace vstupních dat,
  - Automatické generován hodnot pro nově vkládané/modifikované záznamy
  - Čtení záznamů z odkazovaných tabulek pro ověření referencí.
- BEFORE spouště nejsou používány pro další modifikace databázových objektů, protože jsou aktivovány před provedení změn vlastním dotazem a tedy jsou aktivovány před kontrolou integritních omezení

Použití After

- AFTER trigery mohou být chápány jako modul aplikační logiky který je proveden jako odezva na určitou událost v DB.
- AFTER trigery vždy pracují s db, která je v konzistetním stavu.
- Jsou spouštěny až po kontrole IO.
- Např:
  - Spouštění operací jako odezvu na upravující operace v db
  - Operace mimo databázi, např. spouštění alarmů, externích programů atd.
  - Akce mimo db nejsou pod kontrolou db mechanismů pro rollback

Omezení Before

- Before trigery nemohou obsahovat následující operace v SQL příkazech svého těla:
- UPDATE
- DELETE
- INSERT

Příklad

- CREATE TABLE T4 (a INTEGER, b CHAR(10));
  CREATE TABLE T5 (c CHAR(10), d INTEGER);
- Vytvoříme trigger, který vloží záznam do tabulky T5 pokud je vložen záznam do T4. Triger zkontroluje, zda-li nově vložený záznam má první složku 10 nebo méně a pokud ano, tak vloží reverzní záznam do T5:

```
CREATE TRIGGER trig1
AFTER INSERT ON T4
REFERENCING NEW AS newRow
FOR EACH ROW
WHEN (newRow.a <= 10)
BEGIN
INSERT INTO T5 VALUES(newRow.b, newRow.a);
END trig1;
```

MySQL

```
mysql> delimiter //
mysql> CREATE TRIGGER upd_check
       BEFORE UPDATE ON account
       FOR EACH ROW
       BEGIN
            IF NEW.amount < 0 THEN
                 SET NEW.amount = 0;
            ELSEIF NEW.amount > 100 THEN
                 SET NEW.amount = 100;
            END IF;
       END;//
mysql> delimiter ;
```

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY
  KEY);
CREATE TABLE test4( a4 INT NOT NULL AUTO_INCREMENT PRIMARY
  KEY, b4 INT DEFAULT 0 );
DELIMITER |
CREATE TRIGGER testref
  AFTER INSERT ON test1
  FOR EACH ROW
  BEGIN
       INSERT INTO test2 (a2) VALUES NEW.a1;
       DELETE FROM test3 WHERE a3 = NEW.a1;
       UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END; |
DELIMITER ;
INSERT INTO test3 (a3) VALUES (NULL), (NULL), (NULL),
  (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL);
INSERT INTO test4 (a4) VALUES (0), (0), (0), (0), (0),
  (0), (0), (0), (0), (0);
```

```
CREATE TRIGGER testref
  AFTER INSERT ON test1
  FOR EACH ROW
  BEGIN
  INSERT INTO test2 (a2) VALUES NEW.a1;
  DELETE FROM test3 WHERE a3 = NEW.a1;
  UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;


INSERT INTO test1 VALUES (1), (3), (1), (7), (1),
  (8), (4), (4);
```

SELECT * FROM test1;

```
+------+
| a1 |
+------+
| 1 |
| 3 |
| 1 |
| 7 |
| 1 |
| 8 |
| 4 |
| 4 |
+------+
```

```
CREATE TRIGGER testref
  AFTER INSERT ON test1
  FOR EACH ROW
  BEGIN
  INSERT INTO test2 (a2) VALUES NEW.a1;
  DELETE FROM test3 WHERE a3 = NEW.a1;
  UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;

INSERT INTO test1 VALUES (1), (3), (1), (7), (1),
  (8), (4), (4);
```

```
SELECT * FROM
  test2;
+------+
| a2 |
+------+
  | 1 |
  | 3 |
  | 1 |
  | 7 |
  | 1 |
  | 8 |
  | 4 |
  | 4 |
+------+
```

```
CREATE TRIGGER testref
  AFTER INSERT ON test1
  FOR EACH ROW
  BEGIN
       INSERT INTO test2 (a2) VALUES NEW.a1;
       DELETE FROM test3 WHERE a3 = NEW.a1;
       UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
```

```
INSERT INTO test1 VALUES (1), (3), (1),
 (7), (1), (8), (4), (4);
```

```
SELECT *
  FROM
  test3;
+----+
| a3 |
+----+
  | 2 |
  | 5 |
  | 6 |
  | 9 |
  | 10 |
+----+
```

```
CREATE TRIGGER testref
  AFTER INSERT ON test1
  FOR EACH ROW
  BEGIN
        INSERT INTO test2 (a2) VALUES NEW.a1;
        DELETE FROM test3 WHERE a3 = NEW.a1;
        UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
```

```
INSERT INTO test1 VALUES (1), (3), (1), (7), (1),
  (8), (4), (4);
```

SELECT * FROM test4;

```
+----+------+
| a4 | b4 |
+----+------+
 | 1 | 3 |
 | 2 | 0 |
 | 3 | 1 |
 | 4 | 2 |
 | 5 | 0 |
 | 6 | 0 |
 | 7 | 1 |
 | 8 | 1 |
 | 9 | 0 |
 | 10 | 0 |
+----+------+
```

Uložené procedury

- CREATE PROCEDURE
- Můžeme definovat dva typy uložených procedur.
  - Externí. Tělo procedury je napsaná v programovacím jazyce. Procedura pak během svého běhu volá externí programy, rutiny.
  - SQL. Tělo procedury je napsané v SQL a je definované pouze pro prostředí SQL serveru.

MySQL

```
CREATE
  [DEFINER = { user | CURRENT_USER }]
  PROCEDURE sp_name ([proc_parameter[,...]]) [characteristic ...]
  routine_body
CREATE [DEFINER = { user | CURRENT_USER }]
  FUNCTION sp_name ([func_parameter[,...]])
  RETURNS type
  [characteristic ...] routine_body


proc_parameter: [ IN | OUT | INOUT ] param_name type
func_parameter: param_name type
type: Any valid MySQL data type
characteristic: LANGUAGE SQL | [NOT] DETERMINISTIC | { CONTAINS
  SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } | SQL
  SECURITY { DEFINER | INVOKER } | COMMENT 'string' routine_body:
  Valid SQL procedure statement
```

Parametry

- Parametry jsou defaultně jako IN.
- Pokud chceme specifikovat parametr jinak musíme použít klíčová slova OUT nebo INOUT před jménem parametru.

Return

- Return umožňuje definovat návratový typ procedury.

routine_body

- *routine_body* (tělo procedury) obsahuje validní SQL.
- Můžeme použít jak jednoduchý SELECT nebo INSERT, tak i složený příkaz uzavřený do BEGIN a END.

Příklady PROCEDURE

```
mysql> delimiter //
mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
        BEGIN
                SELECT COUNT(*) INTO param1 FROM t;
        END;
        //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> CALL simpleproc(@a);
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @a;
+------+
| @a |
+------+
| 3 |
 +------+
1 row in set (0.00 sec)
```

Příklady FUNCTION

```
mysql>
CREATE FUNCTION hello (s CHAR(20)) RETURNS
  CHAR(50)
  RETURN CONCAT('Hello, ',s,'!');

Query OK, 0 rows affected (0.00 sec)
mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world! |
+----------------+
1 row in set (0.00 sec)
```

Change of characteristics of a stored procedure or function.

```
ALTER {PROCEDURE | FUNCTION} sp_name
  [characteristic ...]

characteristic: { CONTAINS SQL | NO SQL | READS
 SQL DATA | MODIFIES SQL DATA } | SQL SECURITY {
 DEFINER | INVOKER } | COMMENT 'string'
```

DROP

```
DROP {PROCEDURE | FUNCTION}
[IF EXISTS] sp_name
```

CALL

```
CALL sp_name([parameter[,...]])
```

- CALL spustí uloženou proceduru definovanou pomocí CREATE PROCEDURE.

DECLARE

```
DECLARE var_name[,...] type [DEFAULT value]
```

- Pomocí declare definujeme lokální proměnné.
- Může definovat výchozí hodnotu proměnné pomocí klauzule DEFAULT.
- Hodnota může být samozřejmě výraz, tedy ne konstantní hodnota.
- Pokud neuvedeme DEFAULT, pak je výchozí hodnotu NULL.

SELECT INTO

```
CREATE PROCEDURE sp1 (x VARCHAR(5))
BEGIN
DECLARE xname VARCHAR(5) DEFAULT 'bob';
DECLARE newname VARCHAR(5);
DECLARE xid INT;
SELECT xname,id INTO newname,xid FROM
 table1 WHERE xname = xname;
SELECT newname;
END;
```