

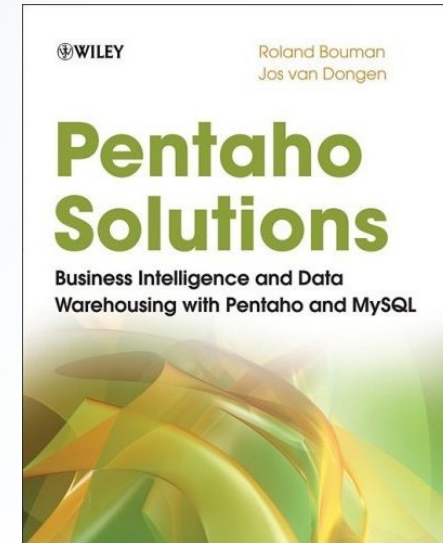
**MySQL**  
**User Conference and Expo 2010**

# **Optimizing Stored Routines**

# Welcome, thanks for attending!



- Roland Bouman; Leiden, Netherlands
- Ex MySQL AB, Sun Microsystems
- Web and BI Developer
- Co-author of “Pentaho Solutions”
- Blog: <http://rpbouman.blogspot.com/>
- Twitter: @rolandbouman



# Program

- Stored routine issues
- Variables and assignments
- Flow of control
- Cursor handling
- Summary

# Program

- Stored routine issues?
- Variables and assignments
- Flow of control
- Cursor handling
- Summary

# Stored Routines: Definition

- Stored routines:
  - stored functions (SQL functions)
  - stored procedures
  - triggers
  - events

# Performance Issues

- SQL inside stored routines is still SQL, ...but...
  - invocation overhead
  - suboptimal computational performance
- Benchmarking method
  - `BENCHMARK(1000000, expression)`
  - Appropriate for computation speed
  - 1 million times
- MySQL 5.1.36, Windows

# Invocation overhead

- Plain expression (10 mln)

```
mysql> SELECT BENCHMARK(10000000, 1);
+-----+
| benchmark(10000000, 1) |
+-----+
|                        0 |
+-----+
1 row in set (0.19 sec)
```

- Equivalent function (10 mln)

```
mysql> CREATE FUNCTION f_one() RETURNS INT RETURN 1;

mysql> SELECT BENCHMARK(10000000, f_one());
+-----+
| benchmark(10000000, f_one) |
+-----+
|                        0 |
+-----+
1 row in set (24.59 sec)
```

- Slowdown 130 times

# Computation inefficiency

- Plain addition

```
mysql> SELECT BENCHMARK(10000000, 1+1);
+-----+
| benchmark(10000000, 1+1) |
+-----+
|                          0 |
+-----+
1 row in set (0.30 sec)
```

- Equivalent function

```
mysql> CREATE FUNCTION f_one_plus_one() RETURNS INT RETURN 1+1;

mysql> SELECT BENCHMARK(10000000, f_one_plus_one());
+-----+
| benchmark(10000000, f_one_plus_one()) |
+-----+
|                                          0 |
+-----+
1 row in set (28.73 sec)
```

# Computation inefficiency

- Raw measurements

	plain expression	function	ratio
1 f_one()	0.19	24.59	0.0077
1+1 f_one_plus_one()	0.29	28.73	0.0101

- Correction for invocation overhead

	plain expression	function	ratio
1 f_one()	0.00	00.00	
1+1 f_one_plus_one()	0.10	4.14	0.0242

- Slowdown about 40 times

- after correction for invocation overhead

# Program

- Stored routine issues
- Variables and assignments
- Flow of control
- Cursor handling
- Summary

# Types of Variables

- User-defined variables
  - session scope
  - runtime type

```
SET @user_defined_variable := 'some value';
```

- Local variables
  - block scope
  - declared type

```
BEGIN  
  DECLARE v_local_variable VARCHAR(50);  
  SET v_local_variable := 'some value';  
  ...  
END;
```

# User-defined variable Benchmark

- Baseline

```
CREATE FUNCTION f_variable_baseline()  
RETURNS INT  
BEGIN  
    DECLARE a INT DEFAULT 1;  
    RETURN a;  
END;
```

- Local variable

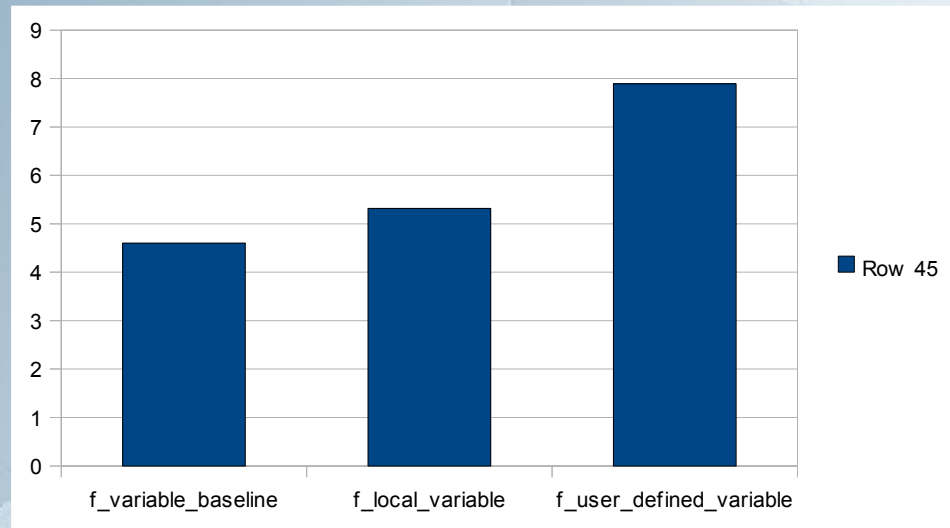
```
CREATE FUNCTION f_variable_baseline()  
RETURNS INT  
BEGIN  
    DECLARE a INT DEFAULT 1;  
    SET a := 1;  
    RETURN a;  
END;
```

- User-defined variable

```
CREATE FUNCTION f_variable_baseline()  
RETURNS INT  
BEGIN  
    DECLARE a INT DEFAULT 1;  
    SET @a := 1;  
    RETURN a;  
END;
```

# User-defined variables

- User-defined variables about 5x slower



baseline	local variable	User-defined variable
4.6	5.32	7.89
0.0	0.72	3.29
		$0.72/3.29 = 0,22$

# Assignments

- SET statement

```
SET v_variable := 'some value';
```

- SELECT statement

```
SELECT 'some value' INTO v_variable;
```

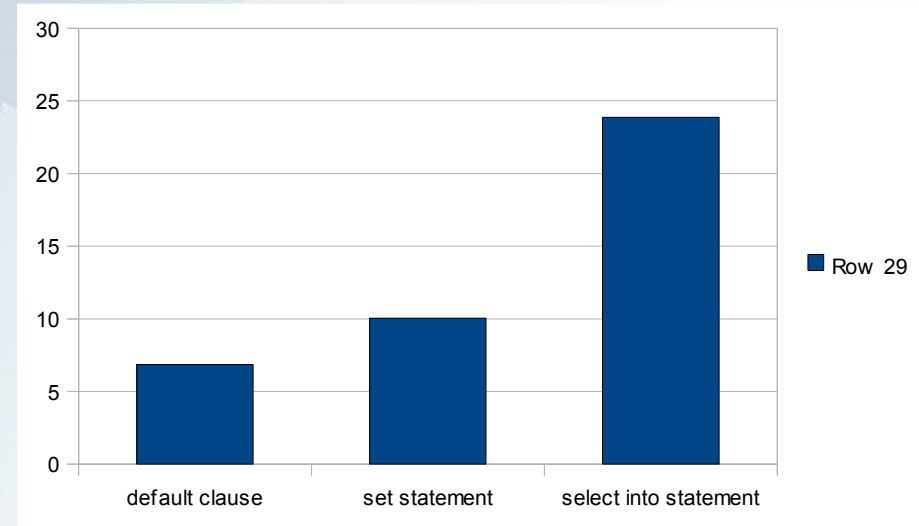
- DEFAULT clause

```
BEGIN  
    DECLARE v_local_variable VARCHAR(50)  
            DEFAULT 'some value';  
    ...  
END;
```

# Assignment Benchmarks

- SELECT INTO about 60% slower than SET
- SET about 40% slower than DEFAULT

baseline	DEFAULT	SET	SELECT
8.2	15.06	18.25	32.08
0	6.86	10.05	23.88
		100%	42.09%
	100%	68.26%	



# More about SELECT INTO

- Assigning from a SELECT...INTO statement:
  - ok if you're assigning from a real query
  - not so much if you're assigning literals

```
SELECT 1  
,      'some value'  
INTO   v_number  
,     v_string
```

```
SELECT COUNT (*)  
,      user_id  
INTO   v_count  
,     v_user_id  
FROM   t_users
```

# Sample function: Sakila rental count

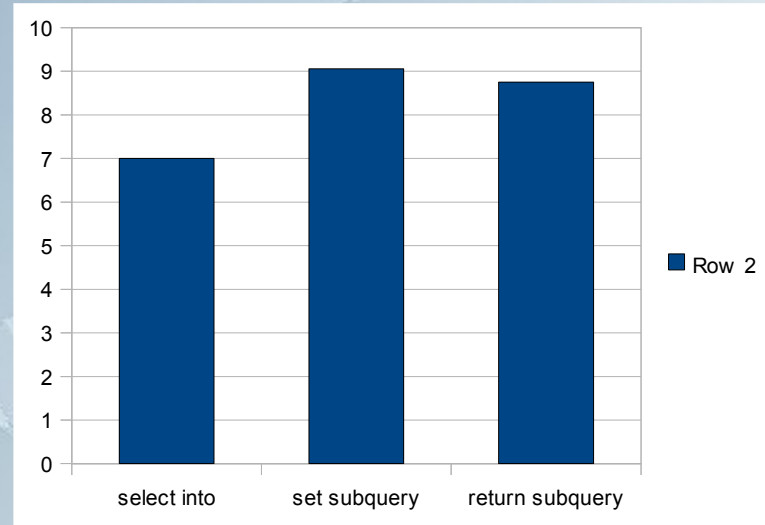
```
CREATE FUNCTION f_assign_select_into(p_customer_id INT) RETURNS INT
BEGIN
    DECLARE c INT;
    SELECT SQL_NO_CACHE, COUNT(*)
    INTO c
    FROM sakila.rental
    WHERE customer_id = p_customer_id;
    RETURN c;
END;
```

```
CREATE FUNCTION f_assign_select_set(p_customer_id INT) RETURNS INT
BEGIN
    DECLARE c INT;
    SET c := ( SELECT SQL_NO_CACHE, COUNT(*)
              FROM sakila.rental
              WHERE customer_id = p_customer_id);
    RETURN c;
END;
```

```
CREATE FUNCTION f_noassign_select(p_customer_id INT) RETURNS INT
BEGIN
    RETURN ( SELECT SQL_NO_CACHE, COUNT(*)
            FROM sakila.rental
            WHERE customer_id = p_customer_id);
END;
```

# Sakila Rental count benchmark

- SET about 25% slower than SELECT INTO



N	select into	set subquery	return subquery
100000	7.00	9.06	8.75

# More on variables and assignments

- Match expression and variable data types
  - example: calculating easter

```
CREATE FUNCTION f_easter_int_nodiv(  
  p_year INT  
) RETURNS DATE  
BEGIN  
  DECLARE a    SMALLINT DEFAULT p_year % 19;  
  DECLARE b    SMALLINT DEFAULT FLOOR(p_year / 100);  
  DECLARE c    SMALLINT DEFAULT p_year % 100;  
  DECLARE d    SMALLINT DEFAULT FLOOR(b / 4);  
  DECLARE e    SMALLINT DEFAULT b % 4;  
  DECLARE f    SMALLINT DEFAULT FLOOR((b + 8) / 25);  
  DECLARE g    SMALLINT DEFAULT FLOOR((b - f + 1) / 3);  
  DECLARE h    SMALLINT DEFAULT (19*a + b - d - g + 15) % 30;  
  DECLARE i    SMALLINT DEFAULT FLOOR(c / 4);  
  DECLARE k    SMALLINT DEFAULT c % 4;  
  DECLARE L    SMALLINT DEFAULT (32 + 2*e + 2*i - h - k) % 7;  
  DECLARE m    SMALLINT DEFAULT FLOOR((a + 11*h + 22*L) / 451);  
  DECLARE v100 SMALLINT DEFAULT h + L - 7*m + 114;  
  
  RETURN STR_TO_DATE(  
    CONCAT(p_year, '-', v100 DIV 31, '-', (v100 % 31) + 1)  
    , '%Y-%c-%e'  
  );  
END;
```

# Matching expression and variable data types

- Multiple expression of this form:

```
DECLARE b SMALLINT DEFAULT FLOOR(p_year / 100);
```

- Divide and round to next lowest integer
  - Alternative: using integer division (DIV)

```
DECLARE b SMALLINT DEFAULT p_year DIV 100;
```

- 13x performance increase!
  - ...but: beware for negative values

# Improved easter function:

```
CREATE FUNCTION f_easter_int_nodiv(  
    p_year INT  
) RETURNS DATE  
BEGIN  
    DECLARE a SMALLINT DEFAULT p_year % 19;  
    DECLARE b SMALLINT DEFAULT p_year DIV 100;  
    DECLARE c SMALLINT DEFAULT p_year % 100;  
    DECLARE d SMALLINT DEFAULT b DIV 4;  
    DECLARE e SMALLINT DEFAULT b % 4;  
    DECLARE f SMALLINT DEFAULT (b + 8) DIV 25;  
    DECLARE g SMALLINT DEFAULT (b - f + 1) DIV 3;  
    DECLARE h SMALLINT DEFAULT (19*a + b - d - g + 15) % 30;  
    DECLARE i SMALLINT DEFAULT c DIV 4;  
    DECLARE k SMALLINT DEFAULT c % 4;  
    DECLARE L SMALLINT DEFAULT (32 + 2*e + 2*i - h - k) % 7;  
    DECLARE m SMALLINT DEFAULT (a + 11*h + 22*L) DIV 451;  
    DECLARE v100 SMALLINT DEFAULT h + L - 7*m + 114;  
  
    RETURN STR_TO_DATE(  
        CONCAT(p_year, '-', v100 DIV 31, '-', (v100 % 31) + 1)  
        , '%Y-%c-%e'  
    );  
END;
```

- 30% faster than using **FLOOR** and /
- Also applicable to regular SQL

# Variable and assignment Summary

- Don't use user-defined variables
  - Use local variables instead
- If possible, use DEFAULT
  - If you don't, time is wasted
- Beware of SELECT INTO
  - Only use it for assigning values from queries
  - Use SET instead for assigning literals
- Match expression and variable data type

# Program

- Stored routine Issues?
- Variables and assignments
- **Flow of control**
- Cursor handling
- Summary

# Flow of Control

- Decisions, alternate code paths
- Plain SQL operators and functions:
  - `IF ()`, `CASE...END`
  - `IFNULL ()`, `NULLIF ()`, `COALESCE ()`
  - `ELT ()`, `FIELD ()`, `FIND_IN_SET ()`
- Stored routine statements:
  - `IF...END IF`
  - `CASE...END CASE`

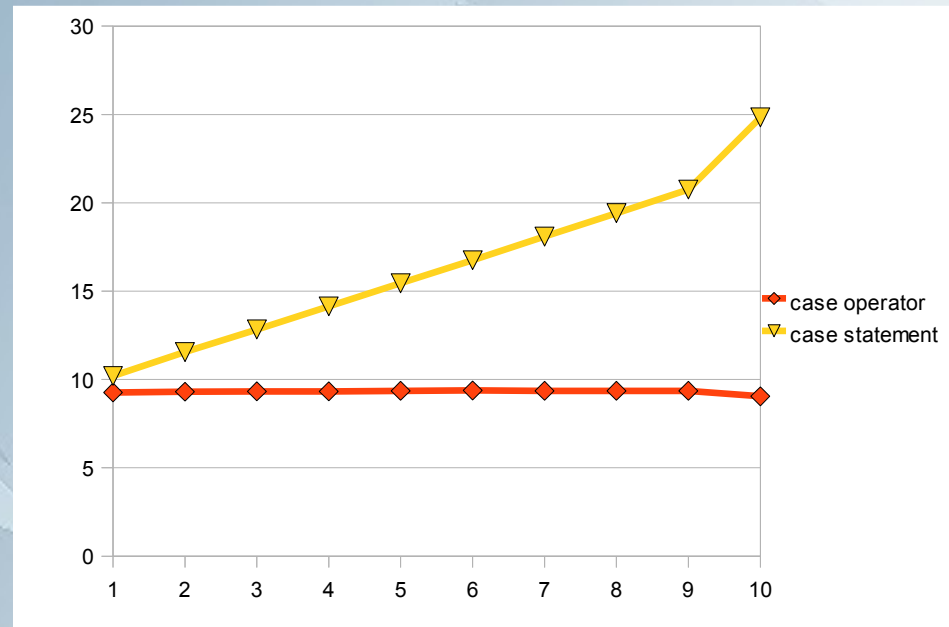
# Case operator vs Case statement

```
CREATE FUNCTION
f_case_operator(
    p_arg INT
)
RETURNS INT
BEGIN
    DECLARE a CHAR(1);
    SET a := CASE p_arg
        WHEN 1 THEN 'a'
        WHEN 2 THEN 'b'
        WHEN 3 THEN 'c'
        WHEN 4 THEN 'd'
        WHEN 5 THEN 'e'
        WHEN 6 THEN 'f'
        WHEN 7 THEN 'g'
        WHEN 8 THEN 'h'
        WHEN 9 THEN 'i'
        ELSE NULL
    END;
    RETURN NULL;
END;
```

```
CREATE FUNCTION
f_case_statement(
    p_arg INT
)
RETURNS INT
BEGIN
    DECLARE a CHAR(1);
    CASE p_arg
        WHEN 1 THEN SET a := 'a';
        WHEN 2 THEN SET a := 'b';
        WHEN 3 THEN SET a := 'c';
        WHEN 4 THEN SET a := 'd';
        WHEN 5 THEN SET a := 'e';
        WHEN 6 THEN SET a := 'f';
        WHEN 7 THEN SET a := 'g';
        WHEN 8 THEN SET a := 'h';
        WHEN 9 THEN SET a := 'i';
        ELSE NULL
    END;
    RETURN NULL;
END;
```

# Case operator vs Case statement

- linear slowdown of the CASE statement



argument	1	2	3	4	5	6	7	8	9	10
case operator	9,27	9,31	9,33	9,33	9,36	9,38	9,36	9,36	9,36	9,05
case statement	10,2	11,55	12,83	14,14	15,45	16,75	18,09	19,41	20,75	24,83

# Flow of control summary

- Use conditional expressions if possible

# Program

- Stored routine Issues?
- Variables and assignments
- Flow of control
- **Cursor handling**
- Summary

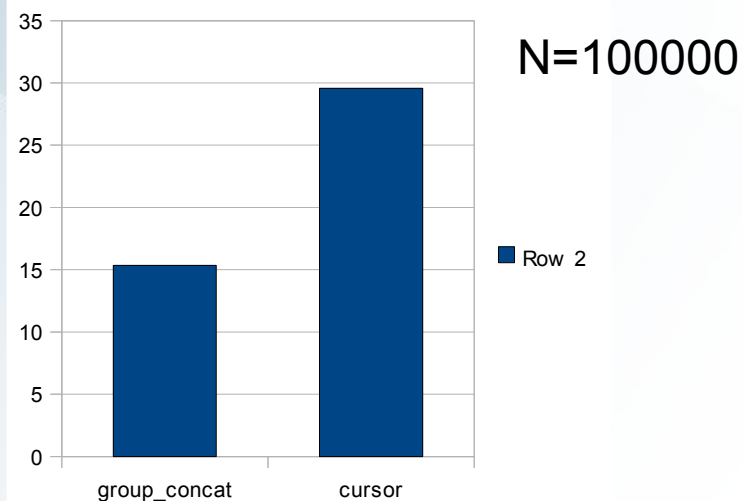
# Cursor Handling

- Why do you need that cursor anyway?
- Only very few cases justify cursors
  - Data driven stored procedure calls
  - Data driven dynamic SQL

# You need a cursor to do what?!

```
CREATE FUNCTION f_film_categories(p_film_id INT)
RETURNS VARCHAR(2048)
BEGIN
    DECLARE v_done BOOL DEFAULT FALSE;
    DECLARE v_category VARCHAR(25);
    DECLARE v_categories VARCHAR(2048);
    DECLARE film_categories CURSOR FOR
        SELECT      c.name
        FROM        sakila.film_category fc
        INNER JOIN  sakila.category      c
        ON         fc.category_id = c.category_id
        WHERE      fc.film_id      = p_film_id;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET v_done := TRUE;
    OPEN film_categories;
categories_loop: LOOP
    FETCH film_categories INTO v_category;
    IF v_done THEN
        CLOSE film_categories;
        LEAVE categories_loop;
    END IF;
    SET v_categories := CONCAT_WS(
        ',', v_categories, v_category
    );
END LOOP;
RETURN v_categories;
END;
```

```
SELECT      fc.film_id
,          GROUP_CONCAT(c.name)
FROM      film_category fc
LEFT JOIN category c
ON fc.category_id = c.category_id
GROUP BY  fc.film_id
```



group_concat	cursor
15,34	29,57

# Cursor Looping

## REPEAT, WHILE, LOOP

- Loop control
- What's inside the loop?
  - Treat nested cursor loops as suspicious
  - Be very weary of SQL statements inside the loop.

# Why to avoid cursor loops with REPEAT

- Always runs at least once
  - So what if the set is empty?
- Iteration before checking the loop condition
  - Always requires an additional explicit check inside the loop
- Loop control scattered:
  - Both in top and bottom of the loop

# Why to avoid cursor loops with REPEAT

```
BEGIN
  DECLARE v_done BOOL DEFAULT FALSE;
  DECLARE csr FOR SELECT * FROM tab;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET v_done := TRUE;

  OPEN csr;
  REPEAT
    FETCH csr INTO var1, ..., varN;
    IF NOT v_done THEN
      -- ... do stuff...
    END IF;
  UNTIL
    v_done
  END REPEAT;
  CLOSE csr;
END;
```

Loop is entered, without checking if the resultset is empty

1 positive and one negative check to see if the resultset is exhausted;

# Why to avoid cursor loops with **WHILE**

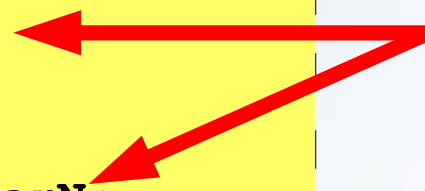
- Slightly better than REPEAT
  - Only one check at the top of the loop
- Requires code duplication
  - One FETCH needed outside the loop
- Loop control still scattered
  - condition is checked at the top of the loop
  - FETCH required at the bottom

# Why to avoid cursor loops with WHILE

```
BEGIN
  DECLARE v_has_rows BOOL DEFAULT TRUE;
  DECLARE csr FOR SELECT * FROM tab;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET v_has_rows := FALSE;

  OPEN csr;
  FETCH csr INTO var1,...,varN;
  WHILE v_has_rows DO
    -- ... do stuff...
    FETCH csr INTO var1,...,varN;
  END WHILE;
  CLOSE csr;
END;
```

Fetch required both  
outside (just once) and  
inside the loop



# Why to write cursor loops with LOOP

- No double checking (like in REPEAT)
- No code duplication (like in WHILE)
- All loop control code in one place
  - All at top of loop

# Why you should write cursor loops with LOOP

```
BEGIN
  DECLARE v_done BOOL DEFAULT FALSE;
  DECLARE csr FOR SELECT * FROM tab;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET v_done := TRUE;

  OPEN csr;
my_loop: LOOP
  FETCH csr INTO var1,...,varN;
  IF v_done THEN
    CLOSE csr;
    LEAVE my_loop;
  END IF;
  -- ... do stuff...
END LOOP;
END;
```

# Cursor summary

- Avoid cursors if you can
  - Use GROUP\_CONCAT for lists
  - Use joins, not nested cursors
  - Only for data driven dynamic SQL and stored procedure calls
- Use LOOP instead of REPEAT and WHILE
  - REPEAT requires double condition checking
  - WHILE requires code duplication
  - LOOP allows you to keep all loop control together

# Program

- Stored routine Issues?
- Variables and assignments
- Flow of control
- Cursor handling
- **Summary**

# Summary

- Variables
  - Use local rather than user-defined variables
- Assignments
  - Use DEFAULT and SET for simple values
  - Use SELECT INTO for queries
- Flow of Control
  - Use functions and operators rather than statements
- Cursors
  - Avoid if possible
  - Use LOOP, not REPEAT and WHILE