SQL Puzzlers

Prague PostgreSQL Developers Day 2011

Tomáš Vondra (tv@fuzzy.cz)

SQL Puzzlers

Co je to puzzler?

- Traps, pitfalls and corner cases.
- Příklad který se chová jinak než byste čekali.
- Puzzlery nejsou bugy!

Proč puzzlery?

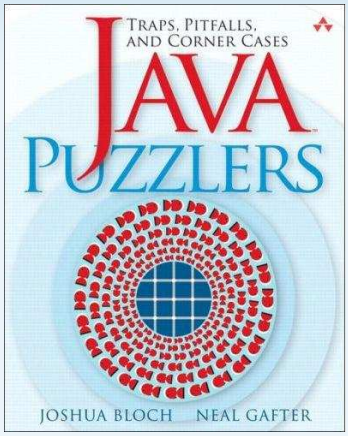
- Je to zábava!
- Ukazují místa kterým nerozumíte (a myslíte si že ano).
- Nejzajímavější jsou důkazy neplatnosti (Karl Popper)

- puzzlery jsou příklady které se chovají neočekávaně
- puzzlery pracují s chytáky, problematickými místy a mezními případy
- puzzlery nejsou bugy – jsou to problematické vlastnosti, založené na málo známých detailech, perverzních požadavcích standardů apod.
- prostě napíšete kus SQL s naprosto jasným výsledkem, a ono se to ze záhadných důvodů chová úplně jinak
- Karl Popper
  - filozof vědy, propagující „kritický racionalismus“
  - negativní příklady (v rozporu s hypotézou) jsou zajímavější
- všechny puzzlery fungují na aktuální stable verzi (9.0.3)
- Nic mi nevěřte! Budu mít záměrně zavádějící komentáře.

Java Puzzlers

- Traps, pitfalls and corner cases.
- Joshua Bloch and Neal Gafter
- Addison-Wesley Professional, 2005
- http://www.javapuzzlers.com

- ale Java je imperativní jazyk



- velmi populární kniha ve světě Javy
- není úplně nejnovější, ale to co v ní je pořád platí
- sada příkladů na problémy s API, knihovnami, nedomyšlenosti apod.
- rozhodně stojí za těch £18 (Amazon)
- ale Java je imperativní jazyk, což SQL není – puzzlery prezentované dále tak jsou trochu odlišné
- typická témata
  - perverzní požadavky SQL standardu
  - zámky
  - implementační detaily (málo známá implementační omezení)
  - RULES
  - chování některých datových typů (CHAR, INTERVAL, FLOAT, …)

1. Pořadí záznamů

V jakém pořadí budou vypsány řádky ve výsledku dotazu:

```
SELECT * FROM moje_tabulka
```

a proč?

(a) dle primárního klíče tabulky
(b) dle fyzického uložení na disku (bloky 0 … N)
(c) náhodně
(d) v jiném pořadí (jakém a proč?)

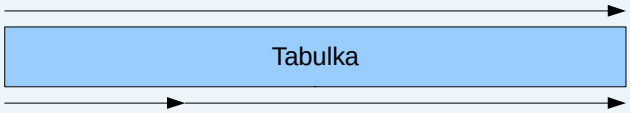
- tradiční jednoduchý sekvenční sken tabulky, nic složitého ;-)

1. Pořadí záznamů

(d) jiné pořadí – dle fyzického uložení na disku, ale nemusí začínat blokem 0

synchronized sequential scans

- od verze 8.3
- optimalizuje situaci kdy probíhá několik skenů naráz
- druhý sken se "napojí" tam kde právě čte druhý sken
- související patch omezující „cache pollution“



- dává smysl aby sekvenční sken vracel data v tom pořadí jak jsou na disku
- až do verze 8.2 byly řádky vraceny ve počínaje blokem 0 až do N
- to je ale neoptimální pokud probíhá několik sekvenčních skenů najednou, protože každý čte tabulku „nezávisle“
- ve verzi 8.3 byl doplněn patch „synchronized sequential scans“ díky kterému se druhý sekvenční sken napojí tam kde je zrovna první
- navíc byl přidán patch na „cache pollution“ - sekvenční skeny velkých tabulek vytlačovaly z cache ostatní data, takže se používá jen malý cirkulární buffer

2. SUBSELECT … FOR UPDATE

```
CREATE TABLE tab (sloupec INT);
INSERT INTO tab VALUES (0);
```

BEGIN;	BEGIN;
UPDATE tab SET sloupec=1;	
	SELECT (SELECT sloupec FROM tab) FROM tab FOR UPDATE; /* čeká ? */
COMMIT;	
	COMMIT;

Čeká update a jakou hodnotu vidí?

(a) čeká a vidí 0

(b) čeká a vidí 1

(c) nečeká a vidí 0

(d) nečeká a vidí 1

- puzzler na zamykání
- pozor na formulaci dotazu v druhé session

2. SUBSELECT … FOR UPDATE

(d) čeká ale vidí 0

- Aktuální hodnotu vidí jen přímé odkazy (ne "scalar subquery").
- Pokud spustíte „přirozeně formulovaný“ dotaz

  SELECT sloupec FROM tab FOR UPDATE;

  tak to získáte očekávanou hodnotu „1“
- Poměrně častý problém (FOR UPDATE + „scalar subquery“).
- Pokud lze, tak přeformulovat na JOIN (netrpí tímto neduhem).

- očekávaná hodnota je rozhodně „1“ - konec konců se čeká na commit první session (s UPDATE), takže druhá session by měla vidět výsledek
- to ale platí jen pro přímé odkazy, nikoliv pro skalární subqueries
- pokud můžete nepoužívejte skalární subqueries a FOR UPDATE (stejně se to chová i s FOR SHARE)
- možná reformulace na JOIN, ale nelze vždy (např. pokud subquery obsahuje LIMIT 1)
- v Oracle se chová naprosto stejně (neočekávaně)

3. CHARACTER

Tabulka s CHAR(2) sloupcem a dva inserty:

```
CREATE TABLE my_table (my_val CHAR(2));

INSERT INTO my_table VALUES ('xxxxxxxxxxxxxxxxxxxx');
INSERT INTO my_table VALUES ('x                  ');
```

Co se stane?

(a) oba příkazy selžou (exception „value too long“)

(b) první selže, druhý projde

(c) první projde, druhý selže

(d) oba příkazy projdou

- celkem primitivní příklad na datový typ CHAR

3. CHARACTER

(b) první selže, druhý projde

```
INSERT INTO my_table VALUES ('xxxxxxxxxxxxxxxxxxxx');
INSERT INTO my_table VALUES ('x                   ');
```

http://www.postgresql.org/docs/9.0/interactive/datatype-character.html

> *… An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) …*
>
> *… Pokus uložit do sloupce s některým z těchto typů delší řetězec skončí chybou, s výjimkou situace kdy všechny přebývající znaky jsou mezery, v kterémžto případě bude řetězec oříznut na maximální délku. (Tato poněkud bizarní výjimka je vyžadována SQL standardem.) …*

- ne vše funguje přirozeně – to by asi měly spadnout oba příkazy
- je to jeden z bizarních požadavků SQL Standardu
- pokud jsou přebývající znaky jenom mezery, tak jsou oříznuty na maximální délku daného sloupce
- Oracle to má špatně (logicky ale proti standardu), přesně opačně než když prázdný řetězec považuje za NULL

4. FLOAT

Máme tabulku

```
CREATE TABLE my_table (
   my_val  FLOAT(4)
);

INSERT INTO my_table VALUES (1.2345678);
```

Co se stane?

(a) dojde k „ERROR: numeric field overflow“

(b) příkaz projde, ale hodnota se zaokrouhlí na 1.2346

(c) projde, hodnota bude zachována

- všichni známe datový typ NUMERIC resp. DECIMAL
  - „precision“ (celkový počet číslic) a „scale“ (počet číslic za deset. čárkou)
  - automatické zaokrouhlení při překročení „scale“ (desetinná místa)
  - chyba při překročení „precision – scale“ (místa před čárkou)
- ale jak se to chová u FLOAT?

4. FLOAT

(c) projde, hodnota bude zachována (dle IEEE aritmetiky)

FLOAT(p)

- přesnost „p“ určuje počet binárních číslic (v mantise)
- dochází k převodu na standardní IEEE typy, tj.
  - 1 - 24 => REAL
  - 25 – 53 => DOUBLE PRECISION

- http://www.postgresql.org/docs/9.0/interactive/datatype-numeric.html

- no, chová se to obdobně ale ne úplně přesně
- přesnost je dána v počtu binárních číslic, nikoliv decimálních
- u DECIMAL jsou hodnoty pevně daná maxima, nelze je překročit, zatímco u FLOAT jen dochází k výběru mezi REAL a DOUBLE PRECISION

5. INTERVAL

Co vrátí tato jednoduchá operace s datumem a intervaly?

```
SELECT '2011-01-31'::date
    + '1 month'::interval
    - '1 month'::interval;
```

(a) 2011-01-31

(b) 2011-02-01

(c) 2011-01-28

(d) ani jedno (např. spadne)

- intervaly a intervalová aritmetika

5. INTERVAL

(c) 2011-01-28

Postup je zhruba následující

- vezmi 2011-01-31
- přičti měsíc => 2011-02-31
- to je neplatné datum, vezmi poslední den =>**2011-02-28**
- odečti měsíc => 2011-01-28
- **výsledek je 2011-01-28**

není „aditivně inversní“

- intervalová aritmetika má určitá temná zákoutí která jsou ale důsledkem nedomyšlenosti našeho kalendáře (různě dlouhé měsíce, letní a zimní čas apod.)
- v tomto případě narážíme na problém s kratším únorem
- Oracle neprovádí „chytré posuny“ ale spadne

```
select to_date('2011-01-31', 'YYYY-MM-DD')
+ interval '0-1' year to month
- interval '0-1' year to month
from dual

ERROR at line 1:
ORA-01839: date not valid for month specified
```

6. VIEWS + FOR UPDATE

```
CREATE TABLE tab (id SERIAL, sloupec INT);
INSERT INTO tab VALUES (DEFAULT, 0);

CREATE VIEW tab_view AS SELECT * FROM tab;

CREATE RULE view_update_rule AS
  ON UPDATE TO tab_view
  DO INSTEAD UPDATE tab SET sloupec = NEW.sloupec
             WHERE id = NEW.id;

CREATE RULE view_delete_rule AS
  ON DELETE TO tab_view
  DO INSTEAD DELETE FROM tab WHERE id = OLD.id;
```

- příklad na „updatable views“ tj. views nad kterými lze provádět DML operace (INSERT, UPDATE, DELETE)
- využívá se RULES, tj. interních přepisovacích pravidel – zadaný příkaz je možno přepsat na něco jiného

6. VIEWS + FOR UPDATE

BEGIN	BEGIN
UPDATE tab_view SET sloupec=(sloupec+1);	
	UPDATE tab_view SET sloupec=(sloupec+1); /* waits */
COMMIT;	
	COMMIT;

Co je ve sloupci za hodnotu?

(a) sloupec=1

(b) sloupec=2

(c) něco jiného nebo dojde k chybě

- velmi jednoduchý scénář
  - dvě paralelní sessions
  - obě spouští stejný dotaz „sloupec=(sloupec+1)“

6. VIEWS + FOR UPDATE

(a) sloupec = 1

Jeden z těch UPDATE příkazů se vlastně „ztratí.“

Pokud spustíte přímo na tabulce, dostanete sloupec=2.

Pozor na RULES!

- důsledkem je že jeden z UPDATE příkazů se jakoby „ztratí“
- pokud to spustíte přímo na tabulce, dostanete očekávanou hodnotu „2“
- výsledný query plan je také velmi komplikovaný (ve srovnání s query planem přímo nad tabulkou)
- obecně RULES jsou velmi zrádné, jejich použití se příliš nedoporučuje pokud to není bezpodmínečně nutné

7. CHARACTER II.

Co vrátí tento dotaz?

```
SELECT ('x  '::char(3)) = ('x '::char(2));
```

(a) TRUE

(b) FALSE

(c) NULL

- jednoduchý příklad na porovnávání sloupců typu „CHAR“

7. CHARACTER II.

(a) TRUE

```
SELECT ('x  '::char(3)) = ('x '::char(2));
```

http://www.postgresql.org/docs/9.0/interactive/datatype-character.html

> *… the padding spaces are treated as semantically insignificant. Trailing spaces are disregarded when comparing two values of type character, and they will be removed when converting a character value to one of the other string types.*

> *… mezery na konci řetězce jsou považovány za sémanticky bezvýznamné. Koncové mezery jsou pomíjeny při porovnávání dvou znakových hodnot, a při přetypování na některý z dalších znakových typů budou odstraněny.*

- další chyták skrytý v požadavcích SQL standardu – při porovnání se ignorují mezery, protože jsou sémanticky bezvýznamné
- podobně lze demonstrovat na příkazu

  ```
  SELECT 'x '::char(2) || 'x'
  ```

  který vrací „xx“ namísto očekávaného „x x“
- Oracle se chová stejně

8. INTERVAL II.

```
SELECT '2011-02-28'::date + '1 day'::interval
                    + '1 month'::interval;

SELECT '2011-02-28'::date + '1 month'::interval
                    + '1 day'::interval;
```

Jaké jsou výsledky?

(a) oba vrací 2011-04-01

(b) první vrací 2011-04-01, druhý 2011-03-29

(c) něco jiného

- a další příklad na intervaly a operace s nimi

8. INTERVAL II.

(b) první vrací 2011-04-01, druhý 2011-03-29

První

- vezmi 2011-02-28
- přičti den => 2011-03-01
- přičti měsíc => 2011-04-01

Druhý

- vezmi 2011-02-28
- přičti měsíc => 2011-03-28
- přičti den => 2011-03-29

není komutativní

- opět využívá problémů s kratším měsícem
- znamená to že operace s intervaly nejsou komutativní, tj. výsledek záleží na tom v jakém pořadí je provedete
- Oracle spadne – nedokáže zpracovat první příklad (chybné datum)

9. NOT IN

Vytvořme tabulku a naplňme ji daty:

```
CREATE TABLE test_table (id INT);
INSERT INTO test_table VALUES (1),(2),(NULL);
```

Co je výsledkem dotazu

```
SELECT i FROM generate_series(1,5) s(i)
      WHERE i NOT IN (SELECT id FROM test_table);
```

(a) nic (0 řádek)
(b) řádky s hodnotami 1, 2, 3, 4, 5
(c) řádky s hodnotami 3, 4, 5

- pokud neznáte funkci „generate_series“ tak ta generuje tabulku s jedním celočíselným sloupcem, pro každou hodnotu z intervalu jeden řádek
- ve výše uvedeném příkladě má tabulka 5 řádků (1 až 5)
- s(i) je jenom alias

9. NOT IN

(a) nic

```
SELECT i FROM generate_series(1,5) s(i)
    WHERE i NOT IN (1, 2, NULL);
```

- Dle třístavové logiky (true, false, null) je výsledek porovnání s NULL vždy NULL (tj. ani true, ani false).
- Často projde testy (nikoho nenapadne vložit NULL), a pak spadne na produkci po vložení dat (často přes subselecty).
- Vtipné u Oracle, kde prázdný řetězec je ekvivalentní hodnotě NULL (konflikt se SQL standardem).

- v důsledku třístatové logiky to nevrátí nic
- pro opačný operátor „IN“ to takto nefunguje, protože stačí alespoň jeden výskyt (zatím co zde to nelze rozhodnout)
- zvláště problematické u Oracle, kde prázný řetězec je NULL

10. SELECT … FOR SHARE

```
CREATE TABLE tab (val BOOLEAN);
INSERT INTO tab VALUES (TRUE);
INSERT INTO tab VALUES (FALSE);
```

BEGIN;	BEGIN;
UPDATE tab SET val = (NOT val);	
	SELECT * FROM tab WHERE val FOR SHARE;
	/* čeká */
COMMIT;	
	COMMIT;

Jaký je výsledek druhé session?

(a) vidí oba řádky

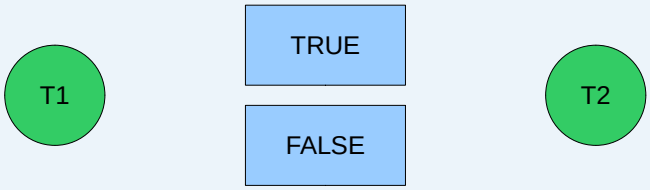
(b) nevidí nic

(c) vidí jen jeden řádek (který?)

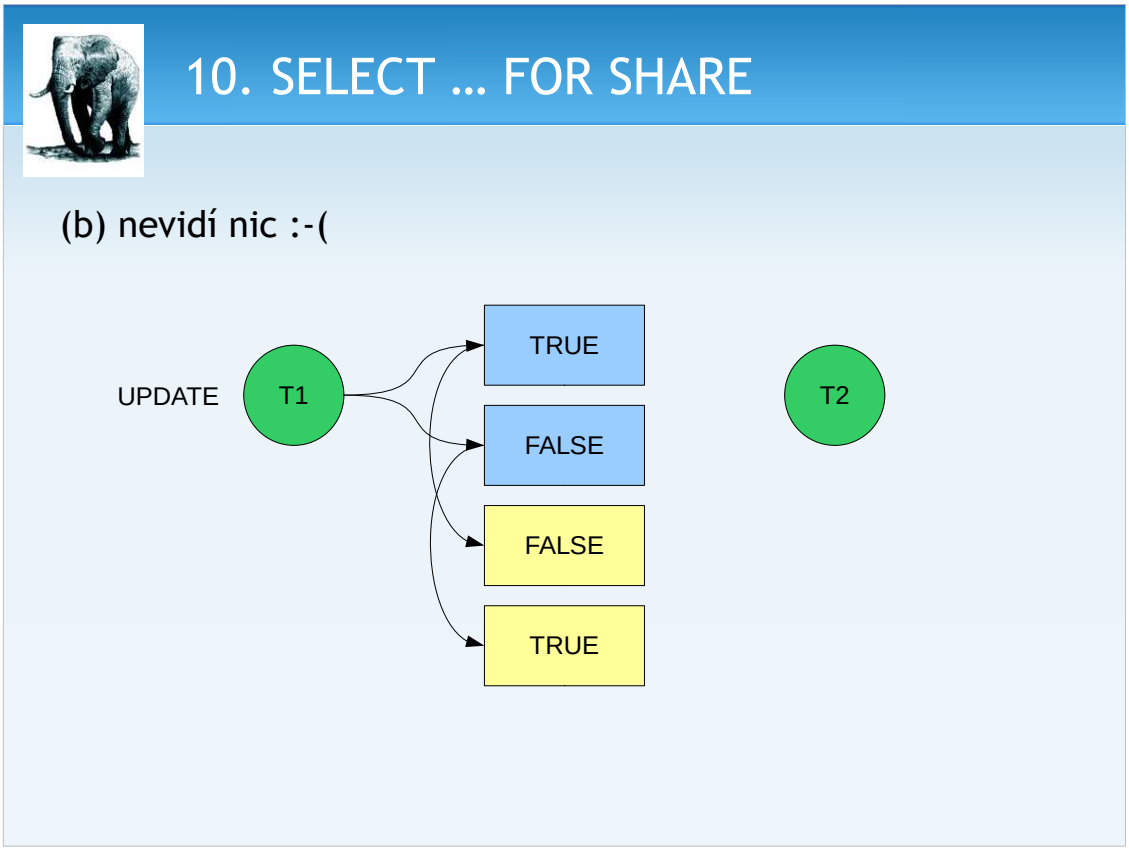
- jedna session updatuje, druhá načítá (FOR SHARE)
- FOR SHARE – zamyká tak aby ostatní sessions nemohly provádět modifikaci (UPDATE, DELETE, SELECT FOR UPDATE)

10. SELECT … FOR SHARE

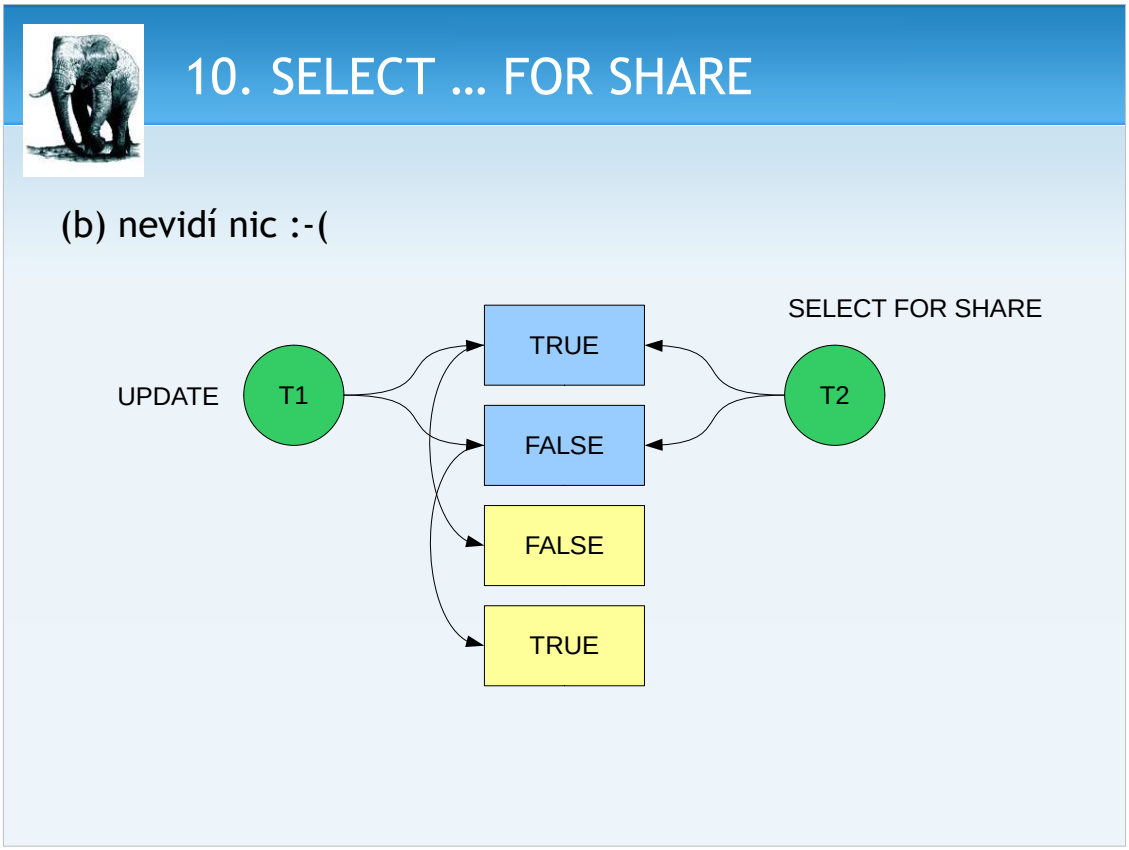
(b) nevidí nic :-(



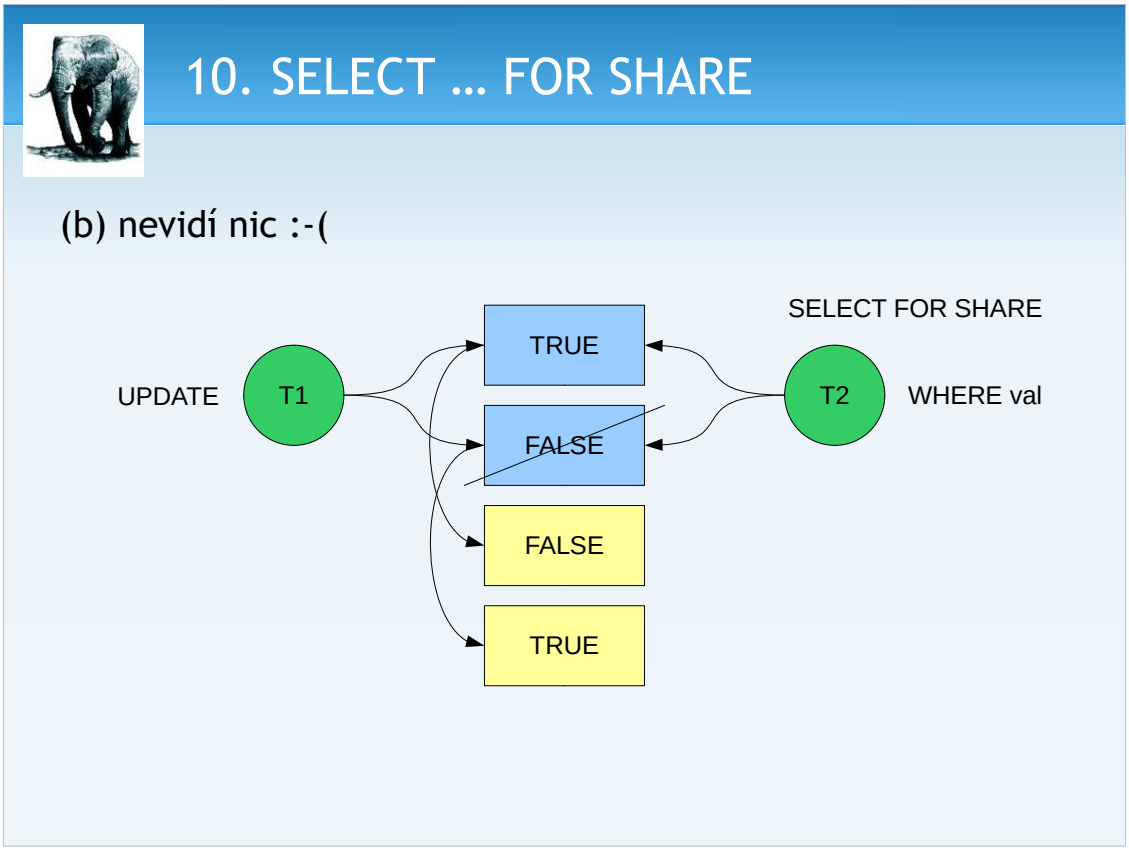
- počáteční stav



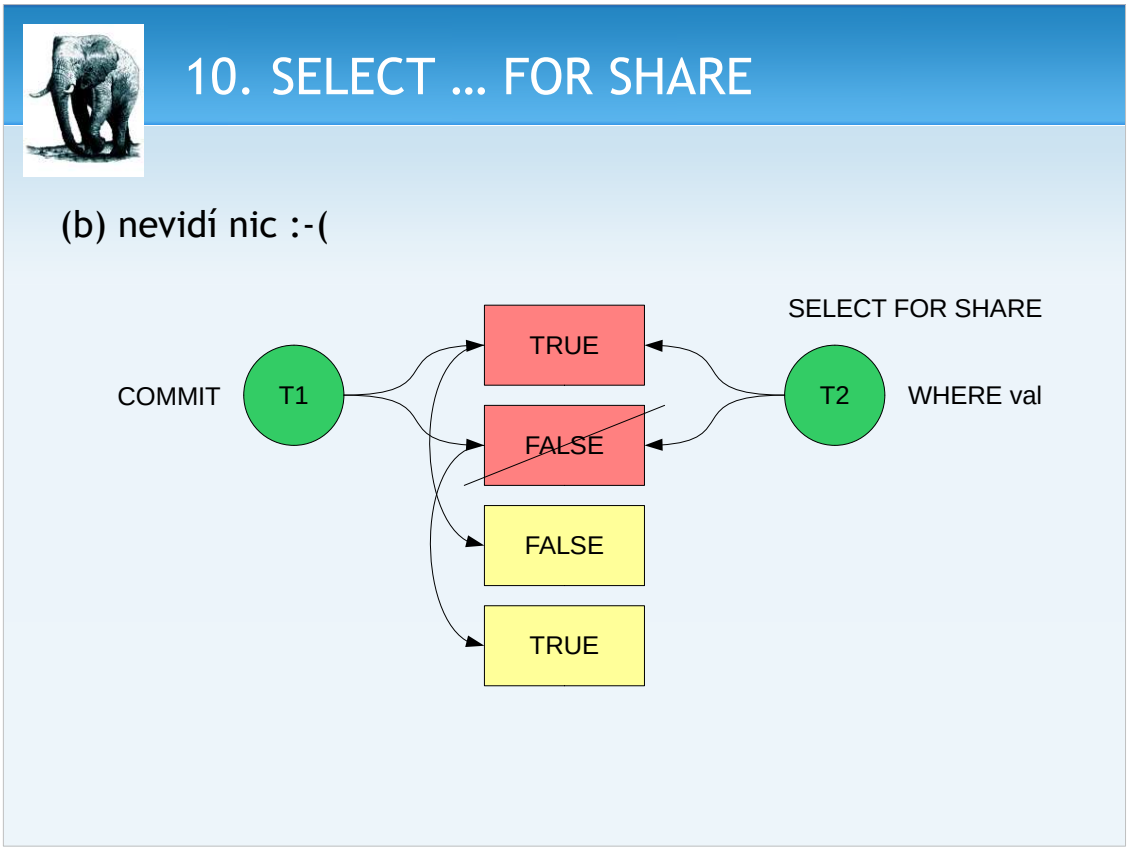
- první session provádí UPDATE, tj. pro každý řádek se vytvoří řádek s opačnou hodnotou (NOT val)



- druhá session spouští SELECT FOR SHARE – čeká na zamčených řádcích (resp. na jednom řádku vyhovujícím WHERE podmínce)



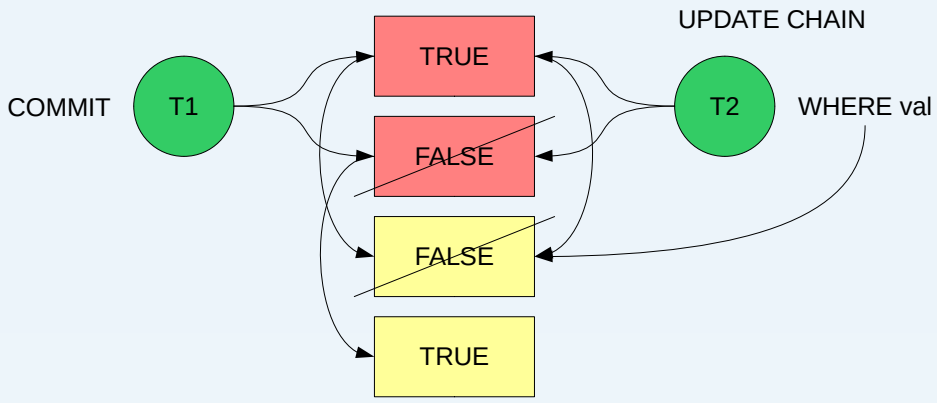
- druhá session spouští SELECT FOR SHARE – čeká na zamčených řádcích (resp. na jednom řádku vyhovujícím WHERE podmínce)



- první session provedla COMMIT

10. SELECT … FOR SHARE

(b) nevidí nic :-(



- to samé pro „FOR UPDATE“

- druhá session pokračuje přes update chain na novou verzi řádku, ale jen na tom řádku který původně vyhovoval
- ten ale teď již nevyhovuje :-(
- výsledkem je tedy prázdná množina
- implementační omezení, Oracle správně vrací jeden řádek

11. INTERVAL III.

```
SELECT '2011-01-31'::date + '1 month'::interval
                          + '1 month'::interval;

SELECT '2011-01-31'::date + '2 months'::interval;
```

Jaké jsou výsledky?

(a) oba vrací 2011-03-31

(b) první vrací 2011-03-28, druhý 2011-03-31

(c) něco jiného

- další operace s intervaly, už je to trochu monotónní
- slibuju že už je to poslední puzzler na intervaly

11. INTERVAL III.

(c) první vrací 2011-03-28, druhý 2011-03-31

První

- vezmi 2011-01-31
- přičti den => 2011-02-31 => oprava na 2011-02-28
- přičti měsíc => **2011-03-28**

Druhý

- vezmi 2011-01-31
- přičti 2 měsíce => **2011-03-31**

není asociativní

- takže kdomě toho že operace s intervaly nejsou komutativní ani additivně inverzní, nejsou ani asociativní
- opatrně s intervaly

12. CHARACTER III.

Co vrátí tento dotaz:

```
SELECT 'xxxxxxxxxx'::char(1);
```

(a) chybu (value too long)

(b) 'x'

(c) 'xxxxxxxxxx'

(d) něco jiného

- a další případ na manipulaci s datovým typem CHAR

12. CHARACTER III.

(b) 'x'

Při přetypování se nekontroluje délka – poněkud zvláštní požadavek SQL standardu.

http://www.postgresql.org/docs/9.0/interactive/datatype-character.html

> *If one explicitly casts a value to character varying(n) or character(n), then an over-length value will be truncated to n characters without raising an error. (This too is required by the SQL standard.)*
>
> *Pokud explicitně přetypováváte hodnotu na typ character varying(n) nebo character(n), potom příliš dlouhé hodnoty budou oříznuty na n znaků bez vyhození chyby. (Toto je také vyžadováno SQL standardem.)*

- opět poněkud zvláštní chování – prostě se zapomene kus řetězce
- a opět se jedná o poněkud perverzní požadavek SQL standardu
- daleko logičtější by bylo vyhodit výjimku

13. VIEWS + DELETE

```
CREATE TABLE tab (id SERIAL, sloupec INT);
INSERT INTO tab VALUES (DEFAULT, 0);

CREATE VIEW tab_view AS SELECT * FROM tab;

CREATE RULE view_update_rule AS
  ON UPDATE TO tab_view
  DO INSTEAD UPDATE tab SET sloupec = NEW.sloupec
             WHERE id = NEW.id;

CREATE RULE view_delete_rule AS
  ON DELETE TO tab_view
  DO INSTEAD DELETE FROM tab WHERE id = OLD.id;
```

- a další příklad na pohledy – použijeme úplně stejnou strukturu jako v předchozím puzzleru na pohledu
- tabulka a nad ní pohled 1:1

13. VIEWS + DELETE

BEGIN	BEGIN
UPDATE tab_view SET sloupec=1;	
	DELETE FROM tab_view WHERE sloupec=0; /* waits */
COMMIT;	
	COMMIT;

Co je výsledkem?

(a) řádek se smaže

(b) nic se nesmaže, zůstane tam sloupec = 1

(c) dojde k chybě

- první session mění hodnotu sloupce z 0 na 1
- druhá session maže řádek s hodnotou 0

13. VIEWS + FOR UPDATE

(a) řádek se smaže

Opět jiné chování než s tabulkami (tam se řádek nesmaže)!

Pozor na RULES!

- opět příklad na nepředvídatelné chování RULEs (updatable views)
- pokud nemusíte, RULES nepoužívejte (i triggery jsou lepší)

14. .NAME

```
CREATE TABLE my_table (
   val_a  INT,
   val_b  INT
);

INSERT INTO my_table VALUES (1, 2);
INSERT INTO my_table VALUES (3, 4);

SELECT my_table.name FROM my_table;
```

Co je výsledkem?

(a) chyba „neexistující sloupec“
(b) dva řádky s hodnotou „my_table“ (název tabulky)
(c) dva řádky, jeden „(1,2)“ a druhý „(3,4)“
(d) něco jiného

14. .NAME

(c) dva řádky:

```
db=# SELECT my_table.name FROM my_table;
 name
-------
 (1,2)
 (3,4)
(2 rows)
```

Zápis „my_table.name“ totiž znamená přetypování řádku na „NAME“ (lze ještě přetypovat na „TEXT“).

Případně lze použít zápis „name(my_table)“ resp. „text(my_table)“.

Ve vývojové větvi (9.1devel) už nefunguje.

- .NAME znamená „přetypuj“ řádek na hodnotu typu NAME
- NAME je speciální typ používaný pro názvy objektů v systémových tabulkách
- velmi matoucí, zejména pokud tabulka má sloupec s názvem „name“
- přidáním sloupce s názvem „name“ se to rozbije
- to jsou přesně ty důvody proč toto bylo z 9.1devel odstraněno

15. TO FAIL OR NOT TO FAIL

```
CREATE TABLE tab_a (id INT PRIMARY KEY, value_a TEXT);
CREATE TABLE tab_b (id INT PRIMARY KEY, value_b TEXT);

INSERT INTO tab_a VALUES (1, 'row a');
INSERT INTO tab_b VALUES (1, '2010-01-01');

INSERT INTO tab_a VALUES (2, 'row b');
INSERT INTO tab_b VALUES (2, 'not a date');
```

ID	VALUE_A
1	row a
2	row b
...	...

ID	VALUE_B
1	2010-01-01
2	not a date
...	...

- X

15. TO FAIL OR NOT TO FAIL

```
SELECT tab_a.id
  FROM tab_a JOIN tab_b USING (id)
 WHERE value_a = 'row a'
   AND to_date(value_b, 'YYYY-MM-DD') = to_date('2010-01-01', 'YYYY-MM-DD');
```

Co se stane?
(a) spadne ('invalid date' exception)
(b) projde, vrátí „1“
(c) něco jiného

ID	VALUE_A
1	row a
2	row b
...	...

ID	VALUE_B
1	2010-01-01
2	not a date
...	...

- x

15. TO FAIL OR NOT TO FAIL

chyták ;-)

- nelze předem říct, závisí na zvoleném query planu
  - enable_seqscan = 0 => spadne (invalid date)
  - enable_seqscan = 1 => projde
- query plany jsou někdy překvapivě nestabilní
  - bulk loady dat (ETL)
  - sloupce s více interpretacemi (někdy číslo, někdy datum)
  - dotazy z ORM
  - ...

- cílem je ukázat že to jestli dotaz projde nebo spadne není jen otázka SQL a dat, ale také exekučního plánu
- některé dotazy mají exekuční plán překvapivě nestabilní

16. LOCK + SAVEPOINT

BEGIN	BEGIN
SELECT * FROM tab FOR UPDATE;	
SAVEPOINT my_savepoint;	
UPDATE tab SET val = 999;	
ROLLBACK TO s;	
	UPDATE tab SET val = 111; /* čeká? */
COMMIT;	COMMIT;

Co se stane?

(a) čeká

(b) nečeká

(c) něco jiného

- další puzzler na zámky

16. LOCK + SAVEPOINT

(b) nečeká

- Avoid locking a row and then modifying it within a later savepoint or PL/pgSQL exception block. A subsequent rollback would cause the lock to be lost.
- If a row locked in the current transaction is updated or deleted, or if a shared lock is upgraded to exclusive: in all these cases, the former lock state is forgotten.
- This is an implementation deficiency which will be addressed in a future release of PostgreSQL.
- http://www.postgresql.org/docs/9.0/static/sql-select.html

- puzzler ilustruje následující posloupnost příkazů
  - vytvoření zámku (FOR UPDATE)
  - vytvoření savepointu
  - opakované získání zámku nebo jeho upgrade (na restriktivnější)
  - rollback na savepoint
- tato posloupnost má za následek ztrátu zámku
- je to problém aktuální implementace :-(
- Oracle správně zámky podrží

17. ORDER BY + FOR UPDATE

```
CREATE TABLE tab (val INT);
INSERT INTO tab VALUES (1), (2), (3);
```

BEGIN	BEGIN
SELECT * FROM tab WHERE val=1 FOR UPDATE;	
	SELECT * FROM tab ORDER BY val FOR UPDATE;
UPDATE tab SET val=4 WHERE val=1; COMMIT;	
	COMMIT;

Jaké pořadí bude ve výsledku?

(a) 1, 2, 3

(b) 2, 3, 4

(c) jiné pořadí

- poslední puzzler na zamykání ;-)
- první session si zamkne „první“ řádek (val=1)
- druhá session načítá řádky v určitém pořadí

17. ORDER BY + FOR UPDATE

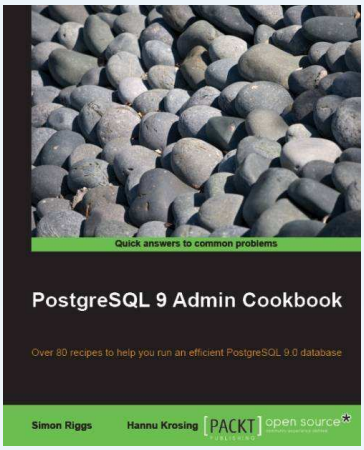
(c) 4, 2, 3

- It is possible for a SELECT command using ORDER BY and FOR UPDATE/SHARE to return rows out of order. This is because ORDER BY is applied first. The command sorts the result, but might then block trying to obtain a lock on one or more of the rows.
- http://www.postgresql.org/docs/9.0/static/sql-select.html

- text říká že dotazy s klauzulí FOR UPDATE / FOR SHARE mohou i v kombinaci s ORDER BY vracet záznamy mimo pořadí
- problém je v tom že záznamy se nejdříve setřídí a až poté se zamykají, a následně se následuje update chain (ale už se výsledek nepřetřiďuje)
- takže druhá session záznamy setřídí, při zamykání řádku s val=1 čeká na druhou session která hodnotu změní na 4 (ale pořadí už se nemění)
- implementační omezení, Oracle vrací správně

Zlatý hřeb večera ;-)

Odměnou za první správnou odpověď na následující puzzler je kniha „PostgreSQL 9 Admin Cookbook“



- jinými slovy nemyslím že by to někdo uhodl

18. PARTITIONING + RULES

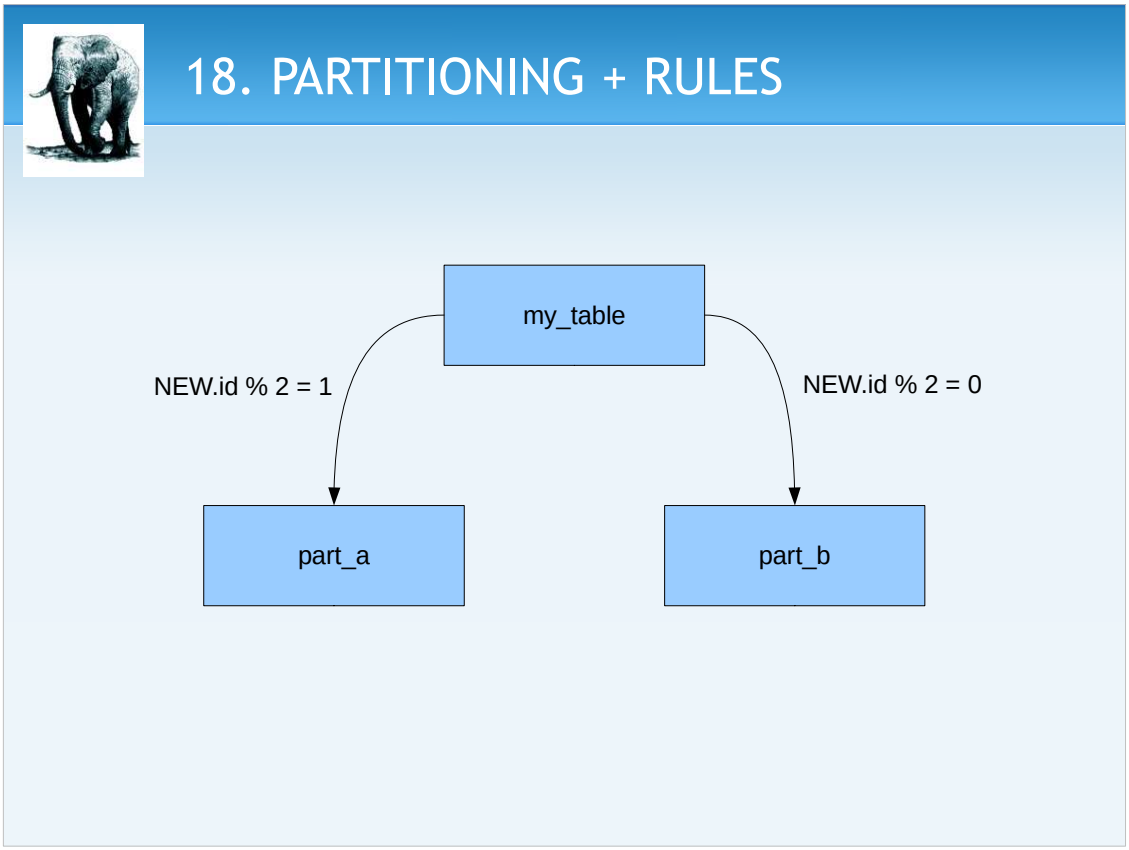
```
CREATE TABLE my_table (
    id      SERIAL,
    val     INT
);

CREATE TABLE part_a () INHERITS (my_table);
CREATE TABLE part_b () INHERITS (my_table);

CREATE RULE part_a AS
   ON INSERT TO my_table
   WHERE (NEW.id % 2 = 1)    /* liché hodnoty do part_a */
   DO INSTEAD INSERT INTO part_a VALUES (NEW.*);

CREATE RULE part_b AS
   ON INSERT TO my_table
   WHERE (NEW.id % 2 = 0)    /* sudé hodnoty do part_b */
   DO INSTEAD INSERT INTO part_b VALUES (NEW.*);
```

- jedna tabulka, dvě subpartitions
- do první subpartition (part_a) přesměrováváme liché hodnoty
- do druhé subpartition (part_b) přesměrováváme sudé hodnoty
- opět používáme RULES ;-) - kdepak asi bude zakopaný pes



- ilustrace předchozího příkladu – hlavní tabulka a dvě partitions

18. PARTITIONING + RULES

Spustíme dva dotazy:

```
INSERT INTO my_table(val) VALUES (1);
INSERT INTO my_table(val) VALUES (2);
```

Co je výsledkem?

(a) „1“ se vloží do part_a, „2“ do part_b

(b) „1“ se vloží do part_b, „2“ do part_a

(c) obě hodnoty se vloží do part_a

(d) něco jiného

- vkládáme dva řádky – lichý a sudý

18. PARTITIONING + RULES

(d) něco jiného

```
db=# select * from my_table;
 id | val
----+-----
  6 |   2
  8 |   2
(2 rows)
```

- „1“ - nevloží se nikam
- „2“ - vloží se do „part_a“ i do „my_table“

Navíc „id“ neskáče po jedné, ale po více hodnotách. Proč?

- Jak je to sakra možné? Proč se 1 ztratila a proč se 2 vložila dvakrát, a ještě ke všemu s tak divnými Idčky?
- Když chci vědět jak něco funguje, použiju … EXPLAIN.

18. PARTITIONING + RULES

```
                                  QUERY PLAN
------------------------------------------------------------------------------------------
 Insert  (cost=0.02..0.03 rows=1 width=0)
   ->  Result  (cost=0.02..0.03 rows=1 width=0)
         One-Time Filter: (((((nextval('my_table_id_seq'::regclass))::integer % 2) = 1) IS NOT TRUE)
                        AND ((((nextval('my_table_id_seq'::regclass))::integer % 2) = 0) IS NOT TRUE))

 Insert  (cost=0.01..0.03 rows=1 width=0)
   ->  Result  (cost=0.01..0.03 rows=1 width=0)
         One-Time Filter: (((nextval('my_table_id_seq'::regclass))::integer % 2) = 1)

 Insert  (cost=0.01..0.03 rows=1 width=0)
   ->  Result  (cost=0.01..0.03 rows=1 width=0)
         One-Time Filter: (((nextval('my_table_id_seq'::regclass))::integer % 2) = 0)
(11 rows)
```

- no a je to jasné – exekuční plán má tři různé větve, a v podmínce každé se vyhodnocuje DEFAULT klauzule sloupce „id“ což není nic jiného než „nextval“ na sekvenci
- dvě větve odpovídají RULES pro partitions
- třetí větev odpovídá „neplatí ani jedno RULE pro partitions“

Odkazy

- Marko Tikkaja (PgDay Europe 2010)
  - http://wiki.postgresql.org/images/9/97/Concurrency.pdf
- Jeff Davis (pgDay San Jose 2009)
  - http://wiki.postgresql.org/images/2/23/Postgresql_pitfalls.pdf
  - aritmetika s intervaly, select for share, …
- Plamen Ratchev, Ten Common SQL Programming Mistakes
  - http://www.simple-talk.com/sql/t-sql-programming/ten-common-sql-programming-mistakes/
- Hubert Lubaczewski (Depesz)
  - http://www.depesz.com/index.php/2008/05/10/prepared-statements-gotcha/
- Simon On Software (FOUND vs. ROW_COUNT)
  - http://simononsoftware.com/postgresql-found-problem/
- Archivy mailing listů (hledejte slova jako „gotcha“ apod.)
- Dokumentace (hledejte „Caution“ a „SQL Standard“)

- zdrojů je spousta, stačí hledat „gotchas“ apod.