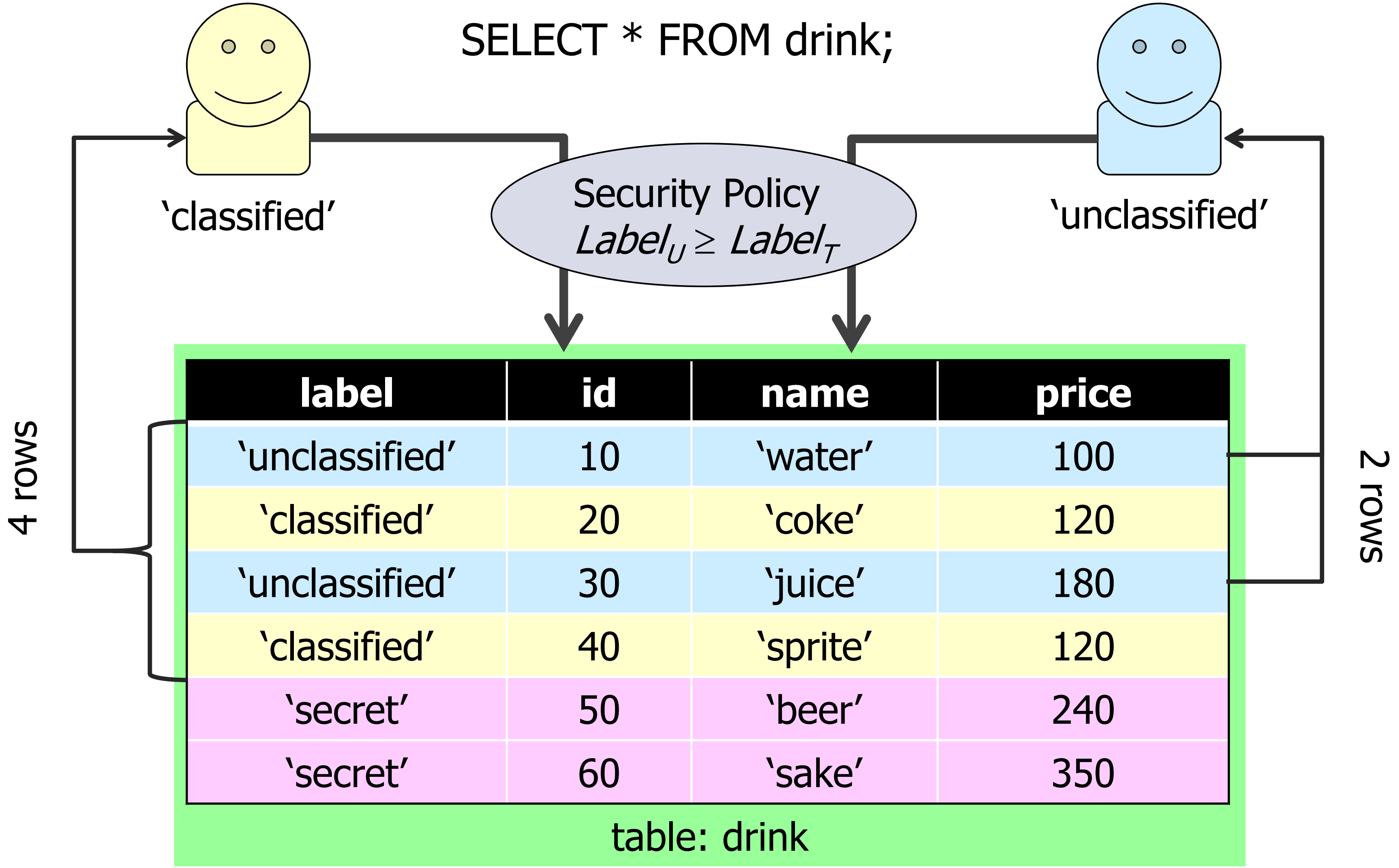
Row Level Security

KaiGai Kohei <kaigai@kaigai.gr.jp>
tw: @kkaigai

How RLS should work (1/2)



SELECT * FROM drink;

label	id	name	price
'unclassified'	10	'water'	100
'classified'	20	'coke'	120
'unclassified'	30	'juice'	180
'classified'	40	'sprite'	120
'secret'	50	'beer'	240
'secret'	60	'sake'	350

table: drink

How RLS should work (2/2)

shop_id = 100

SELECT * FROM drink NATURAL JOIN drink_order

id	name	price	shop_id	quantum	data
10	‘water’	100	100	8	2013-02-16
30	‘juice’	180	100	10	2013-02-18

id	name	price
10	‘water’	100
20	‘coke’	120
30	‘juice’	180
40	‘sprite’	120
50	‘beer’	240
60	‘sake’	350

table: drink

Security Policy
$\mathit{shop\_id}_U = \mathrm{shop\_id}_T$

id	shop_id	quantum	date
10	100	8	2013-02-16
20	200	5	2013-02-17
10	200	6	2013-02-18
30	100	10	2013-02-18

table: drink_order

WHERE is simple solution?

```
postgres=> CREATE VIEW soft_drink AS
     SELECT * FROM drink WHERE price < 200;
CREATE VIEW
postgres=> GRANT SELECT ON soft_drink TO public;
GRANT
postgres=> SET SESSION AUTHORIZATION bob;
SET
postgres=> SELECT * FROM soft_drink;
 id |  name  | price
----+--------+-------
 10 | water  |   100
 20 | coke   |   120
 30 | juice  |   180
 40 | sprite |   120
(4 rows)

postgres=> SELECT * FROM drink;
ERROR:  permission denied for relation drink
```

Nightmare of Leaky View (1/3)

```
postgres=> SELECT * FROM soft_drink WHERE f_leak(name);
NOTICE:  f_leak => water
NOTICE:  f_leak => coke
NOTICE:  f_leak => juice
NOTICE:  f_leak => sprite
NOTICE:  f_leak => beer
NOTICE:  f_leak => sake
 id |  name  | price
----+--------+-------
 10 | water  |   100
 20 | coke   |   120
 30 | juice  |   180
 40 | sprite |   120
(4 rows)
```

Nightmare of Leaky View (2/3)

```
postgres=> CREATE OR REPLACE FUNCTION f_leak (text)
  RETURNS bool COST 0.000001 AS
  $$
  BEGIN
  RAISE NOTICE 'f_leak => %', $1;
  RETURN true;
  END
  $$ LANGUAGE plpgsql;
CREATE FUNCTION

postgres=> EXPLAIN(costs off)
  SELECT * FROM soft_drink WHERE f_leak(name);
                 QUERY PLAN
--------------------------------------------
 Seq Scan on drink
   Filter: (f_leak(name) AND (price < 200))
(2 rows)
```

f_leak() のコストは `<` 演算子より小さい

Nightmare of Leaky View (3/3)

```
postgres=> CREATE VIEW v_both AS
   SELECT * FROM t_left JOIN t_right ON a = x
   WHERE b like '%hoge%';
CREATE VIEW

postgres=> EXPLAIN (COSTS OFF)
   SELECT * FROM v_both WHERE f_leak(y);
                 QUERY PLAN
---------------------------------------------
 Hash Join
   Hash Cond: (t_left.x = t_right.a)
   ->  Seq Scan on t_left
         Filter: f_leak(y)
   ->  Hash
         ->  Seq Scan on t_right
               Filter: (b ~~ '%hoge%'::text)
(7 rows)
```

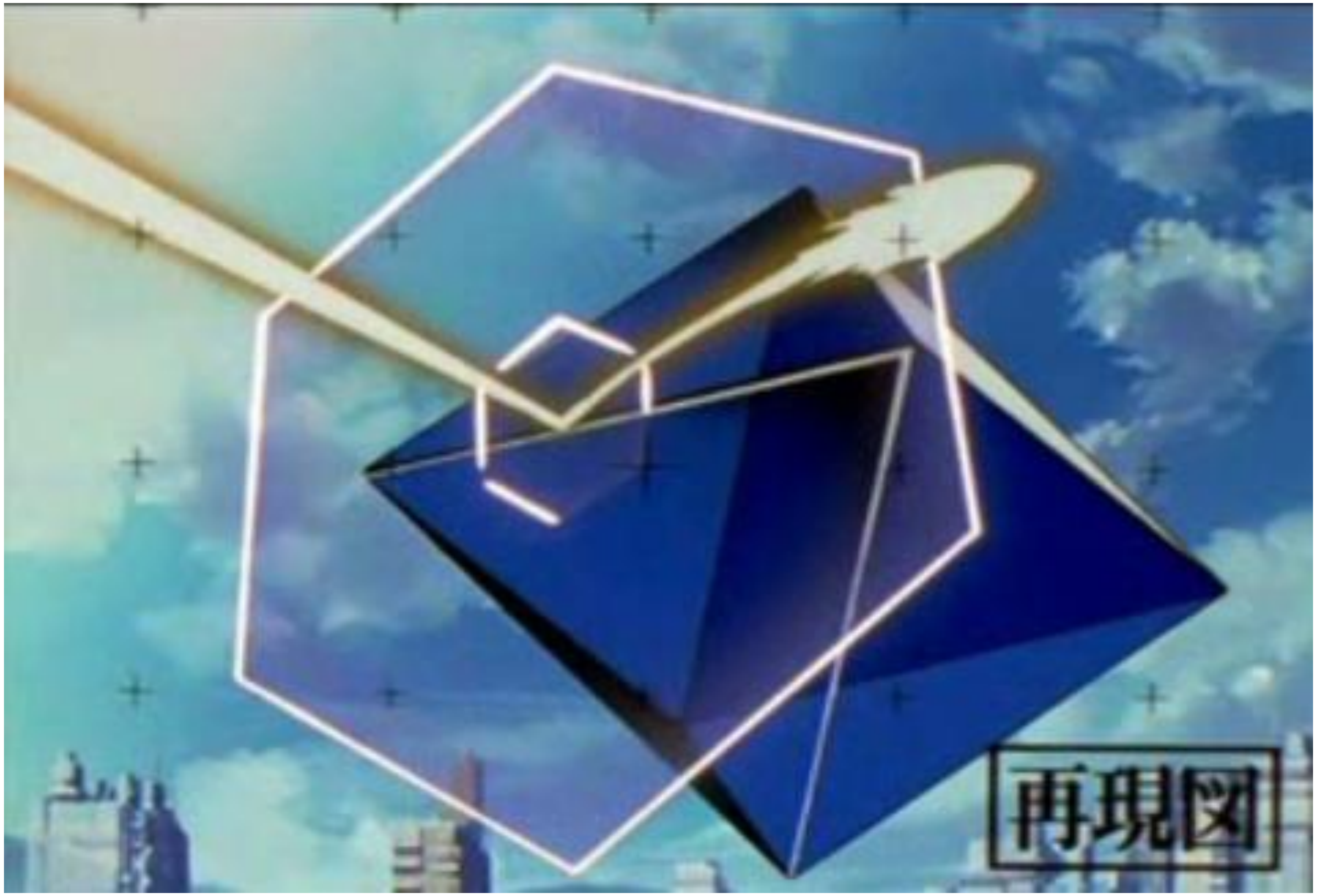
f_leak()の実行は、
t_left だけに依存

Problem to be tackled

```
SELECT * FROM v_both WHERE f_leak(y);
          ↓

SELECT * FROM (
   SELECT * FROM t_left JOIN t_right ON a = x
       WHERE b like '%hoge%’
) AS v_both WHERE f_leak(y)
```

- **問題の本質は、クエリ最適化が条件句を評価する順序を入れ替えてしまう事。**
- **少なくともセキュリティ目的のビューなら、サブクエリの境界を跨いで条件句を移動させてはならない。**

Security Barrier (1/3)



Security Barrier (2/3)

```
postgres=> CREATE OR REPLACE VIEW soft_drink
             WITH (security_barrier)
             AS SELECT * FROM drink WHERE price < 200;
CREATE VIEW
postgres=> SET SESSION AUTHORIZATION bob;
SET
postgres=> SELECT * FROM soft_drink WHERE f_leak(name);
NOTICE:  f_leak => water
NOTICE:  f_leak => coke
NOTICE:  f_leak => juice
NOTICE:  f_leak => sprite
 id |  name  | price
----+--------+-------
 10 | water  |   100
 20 | coke   |   120
 30 | juice  |   180
 40 | sprite |   120
(4 rows)
```

Security Barrier (3/3)

```
postgres=> EXPLAIN (costs off)
           SELECT * FROM soft_drink WHERE f_leak(name);
            QUERY PLAN
-----------------------------------
 Subquery Scan on soft_drink
   Filter: f_leak(soft_drink.name)
   ->  Seq Scan on drink
         Filter: (price < 200)
(4 rows)
```

- CREATE VIEW … WITH (security_barrier) AS …
- security_barrier属性を持つVIEWの内側へは、条件句を push-down させない。
  - 利用者が意図した通りの順序で条件句が評価される
  - 代償として、最適な実行性能は得られなくなる

Security-performance trade off

```
postgres=> CREATE VIEW my_team WITH (security_barrier)
    AS SELECT * FROM employee WHERE boss = current_user;
CREATE VIEW
postgres=> EXPLAIN (costs off)
           SELECT * FROM my_team WHERE id = 100;
                QUERY PLAN
-------------------------------------------
 Subquery Scan on my_team
   Filter: (my_team.id = 100)
   ->  Seq Scan on employee
         Filter: (boss = "current_user"())
(4 rows)
```

- id = 100 を使ってインデックススキャンができるはず
- だが、security_barrier属性のために、先にemployeeを全件スキャン ➔ その後で id = 100 を評価

Leakproof Function (1/3)

Leakproof Function (2/3)

- 副作用の無い (= 間違いなく安全な) 関数なら、security_barrierビューの内側に押し込んでも大丈夫
  - ➔ それを示すのが、LEAKPROOF 属性

```
postgres=# CREATE FUNCTION nabeatsu(integer)
           RETURNS bool LEAKPROOF AS
$$
BEGIN
  IF ($1 % 3 = 0) THEN RETURN true; END IF;
  WHILE $1 > 0 LOOP
    IF ($1 % 10 = 3) THEN RETURN true; END IF;
    $1 = $1 / 10;
  END LOOP;
RETURN false;
END
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Leakproof Function (3/3)

```
postgres=> EXPLAIN (costs off)
           SELECT * FROM my_team WHERE nabeatsu(id);
                       QUERY PLAN
--------------------------------------------------------
 Seq Scan on employee
   Filter: ((boss = "current_user"()) AND nabeatsu(id))
(2 rows)
```

- デフォルトLEAKPROOFなビルトイン関数もある
  - integer対integer の 等価演算子など

```
postgres=> EXPLAIN (costs off)
           SELECT * FROM my_team WHERE id = 300;
                 QUERY PLAN
--------------------------------------------
 Index Scan using employee_pkey on employee
   Index Cond: (id = 300)
   Filter: (boss = "current_user"())
(3 rows)
```

In case of Oracle

```
----------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes |
----------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    81 |
|*  1 |  VIEW               | V    |     3 |    81 |
|*  2 |   HASH JOIN         |      |     3 |   120 |
|*  3 |    TABLE ACCESS FULL| B    |     3 |    60 |
|   4 |    TABLE ACCESS FULL| A    |     4 |    80 |
----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------


   1 - filter("F_LEAK"("X")=1)  <== This is correct
   2 - access("A"."ID"="B"."ID")
   3 - filter("B"."Y"<>'bbb')
```

Toward v9.3, Beyond v9.3

- Feature list in v9.2
  - VIEWのsecurity_barrier属性
  - FUNCTIONのleakproof属性
- Feature list in v9.3(?)
  - ALTER TABLE ... SET ROW SECURITY (...)
- Feature list in v9.4(???)
  - Writer side checks
  - Security label column
  - Label based mandatory row-level access control

Syntax of Row-level Security (1/2)

```
ALTER <table_name>
    SET ROW SECURITY FOR <cmd>
    TO (<expression>);
<cmd> := ALL | SELECT | INSERT | UPDATE | DELETE
```

- <cmd> で指定したDML構文の実行時に、<expression>で指定した条件句（セキュリティポリシー）が付加される。
- クエリの条件句よりも、セキュリティポリシーが先に評価される。

```
postgres=> ALTER TABLE my_table
           SET ROW SECURITY FOR ALL TO (a % 2 = 0);
ALTER TABLE
postgres=> ALTER TABLE my_table
           SET ROW SECURITY FOR ALL TO
               (a = ANY(SELECT x FROM sub_tbl));
ALTER TABLE
```

Syntax of Row-level Security (2/2)

```
ALTER t SET ROW SECURITY FOR ALL
        TO (owner = current_user);
```

```
SELECT * FROM t WHERE f_leak(x);

          ↓

SELECT * FROM (
    SELECT * FROM t WHERE owner = current_user
) t WHERE f_leak(x)
```

security_barrier
付きサブクエリ

- テーブルへの参照 ➔ 条件句付きテーブルスキャンを含むサブクエリ (with security_barrier) に置き換える。
- superuser の場合には適用されない。
  - ビルトインRLSの場合。extensionが勝手に付けるのは自由。

How does RLS work? (1/2)

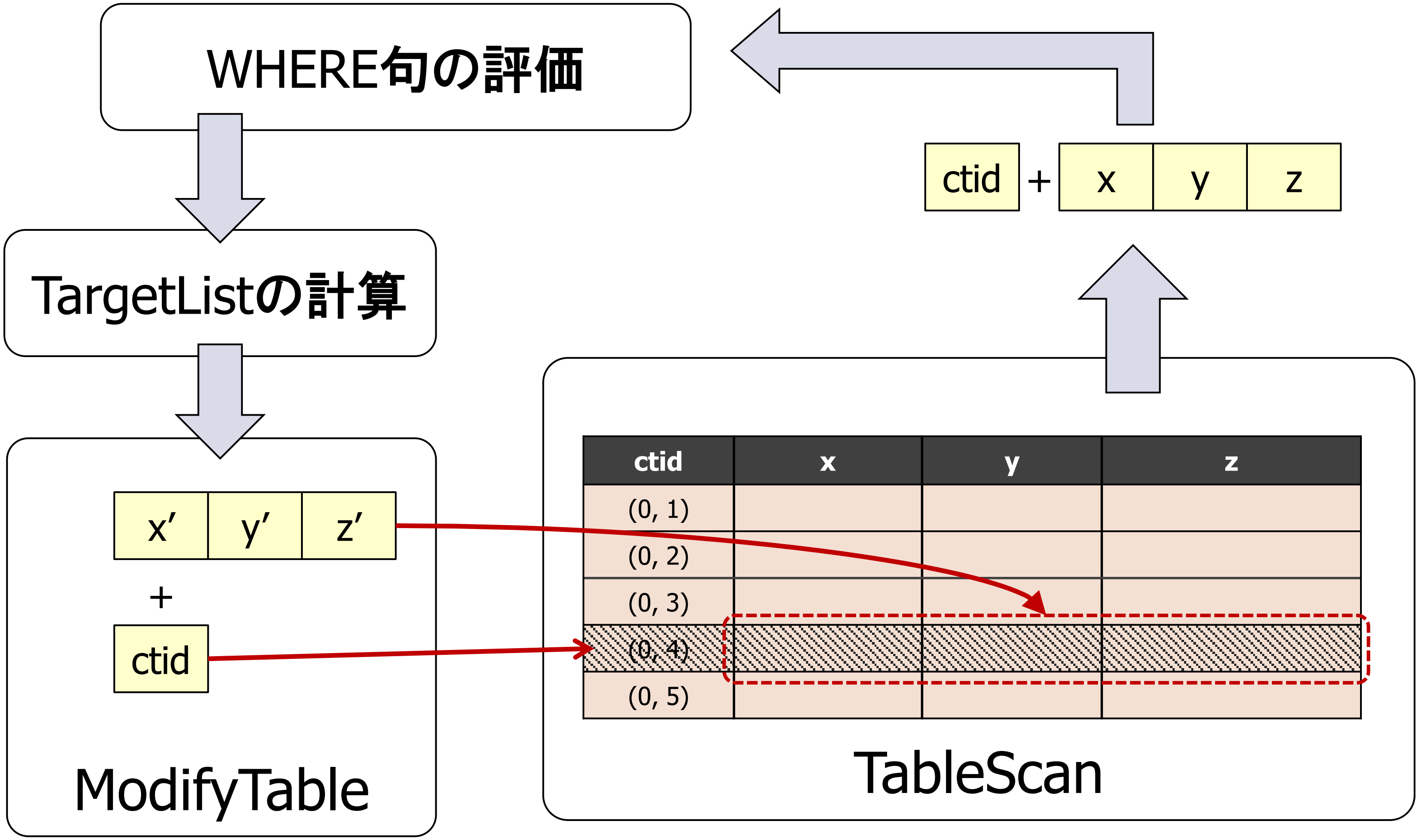
```
postgres=> ALTER TABLE t
        SET ROW SECURITY FOR ALL TO (owner = current_user);
ALTER TABLE

postgres=> EXPLAIN (costs off)
           SELECT * FROM t WHERE f_leak(b) AND a > 0;
                   QUERY PLAN
------------------------------------------------
 Subquery Scan on t
   Filter: f_leak(t.b)
   ->  Index Scan using my_table_pkey on t t_1
         Index Cond: (owner = "current_user"())
         Filter: (a > 0)
(5 rows)
```

How does RLS work? (2/2)

```
postgres=> EXPLAIN (costs off)
           UPDATE t SET b = b WHERE f_leak(b);
                      QUERY PLAN
------------------------------------------------------
 Update on t
   ->  Subquery Scan on t_1
         Filter: f_leak(t_1.b)
         ->  Index Scan using my_table_pkey on t t_2
               Index Cond: (owner = "current_user"())
```

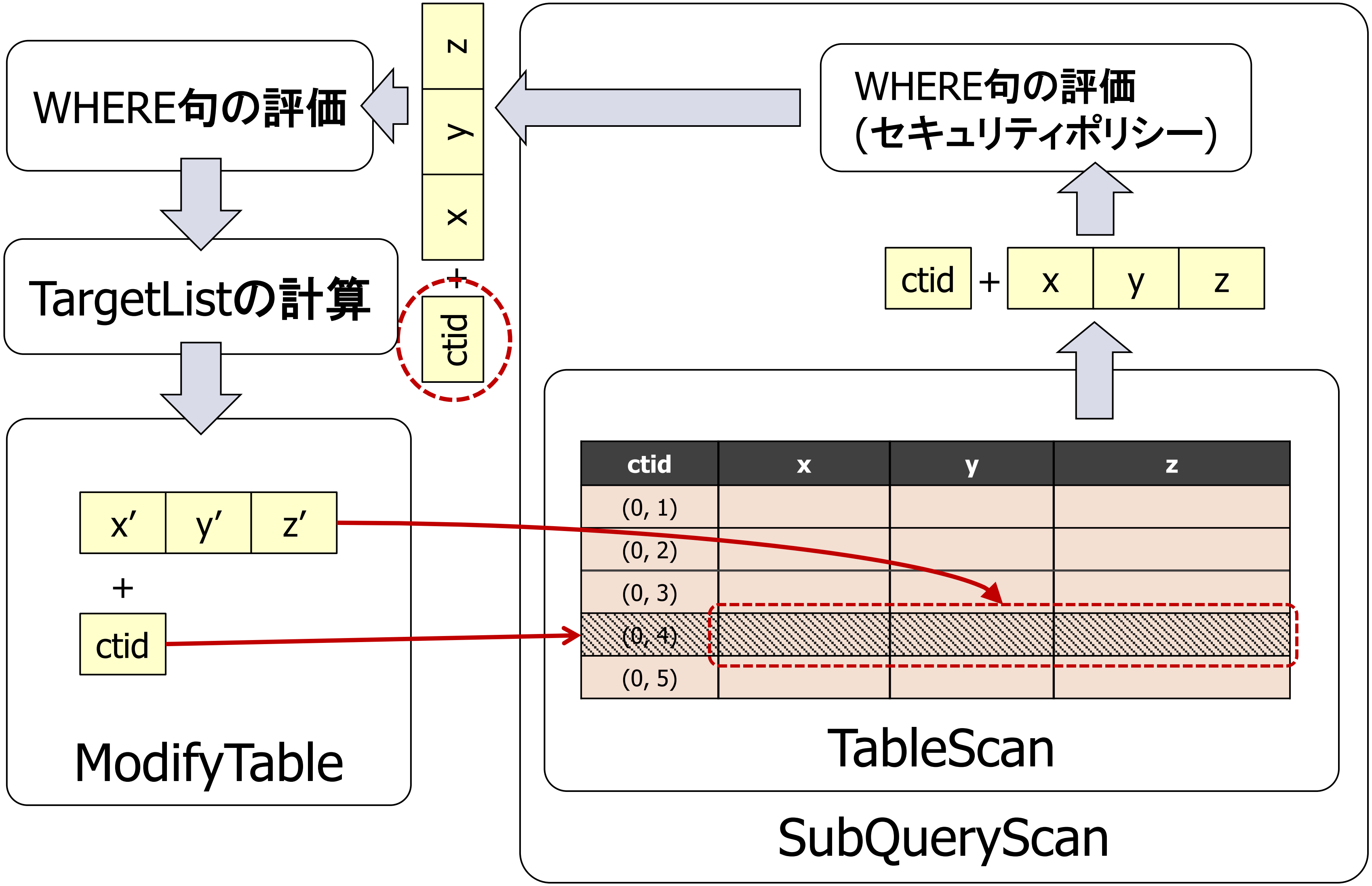
```
postgres=> EXPLAIN(costs off)
           DELETE FROM t WHERE f_leak(b);
                      QUERY PLAN
------------------------------------------------------
 Delete on t
   ->  Subquery Scan on t_1
         Filter: f_leak(t_1.b)
         ->  Index Scan using my_table_pkey on t t_2
               Index Cond: (owner = "current_user"())
```

Table Update and RLS (1/2)



ctid	x	y	z
(0, 1)			
(0, 2)			
(0, 3)			
(0, 4)			
(0, 5)			

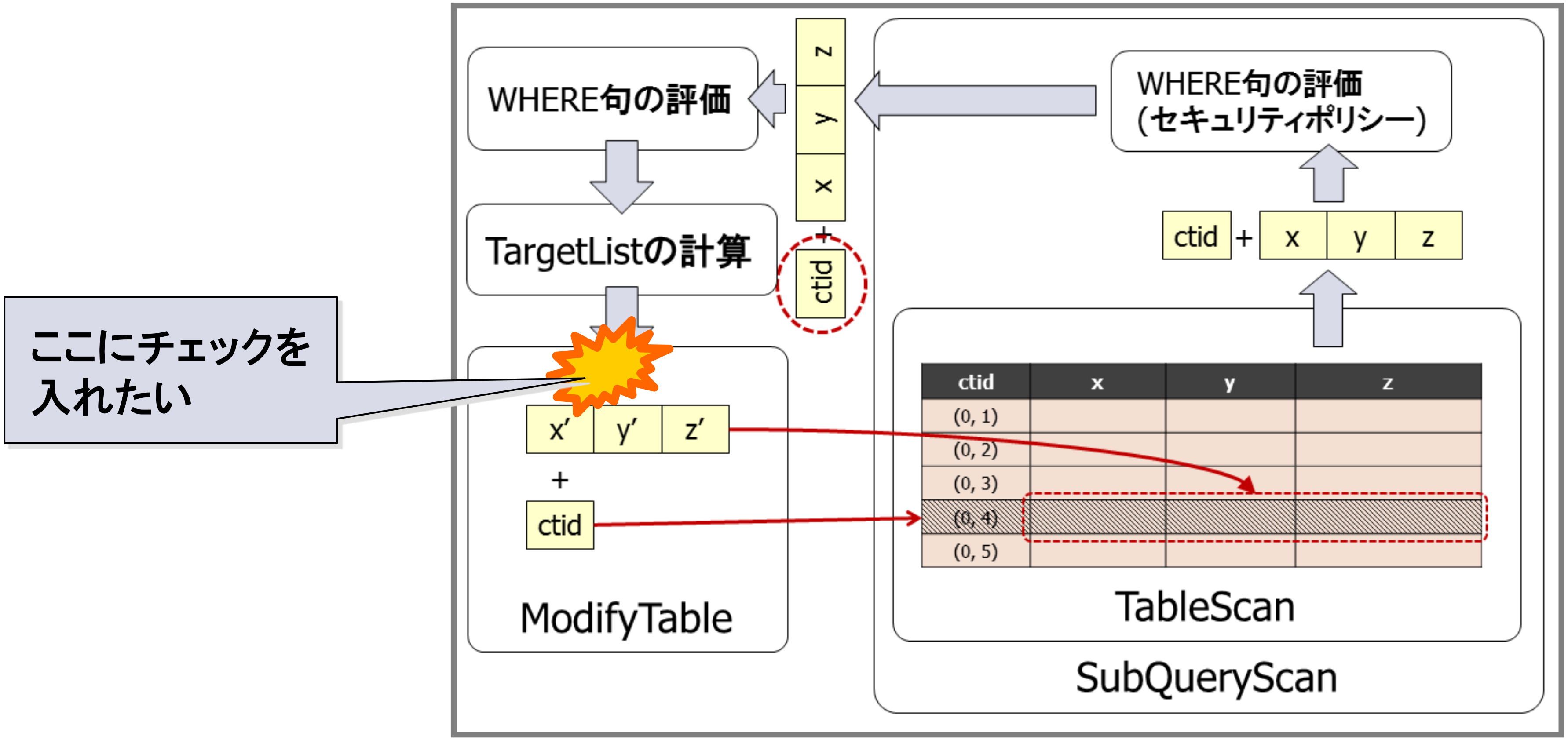
Table Update and RLS (2/2)



ctid	x	y	z
(0, 1)			
(0, 2)			
(0, 3)			
(0, 4)			
(0, 5)			

Future development (1/2)

- SQLコマンド毎に異なるセキュリティポリシーの設定
- INSERT/UPDATEの直前にもチェックを入れる
  - TargetListを計算した結果、それはPolicy Violationな値かもしれない!!



ここにチェックを入れたい

ctid	x	y	z
(0, 1)			
(0, 2)			
(0, 3)			
(0, 4)	▨	▨	▨
(0, 5)			

Future development (2/2)

- Labal-based Row-level Security
  - 要は、SE-PostgreSQLの行レベルアクセス制御対応
- 実現に向けたプロセス
  - Row-level Security基本機能
  - INSERT/UPDATE直前のチェック機能
  - Security Label用の列を自動的に追加する機能
  - 要素の動的追加が可能な Enum 型
  - ↑ これらを全部活用して contrib/sepgsql の機能拡張☺

Any Questions?