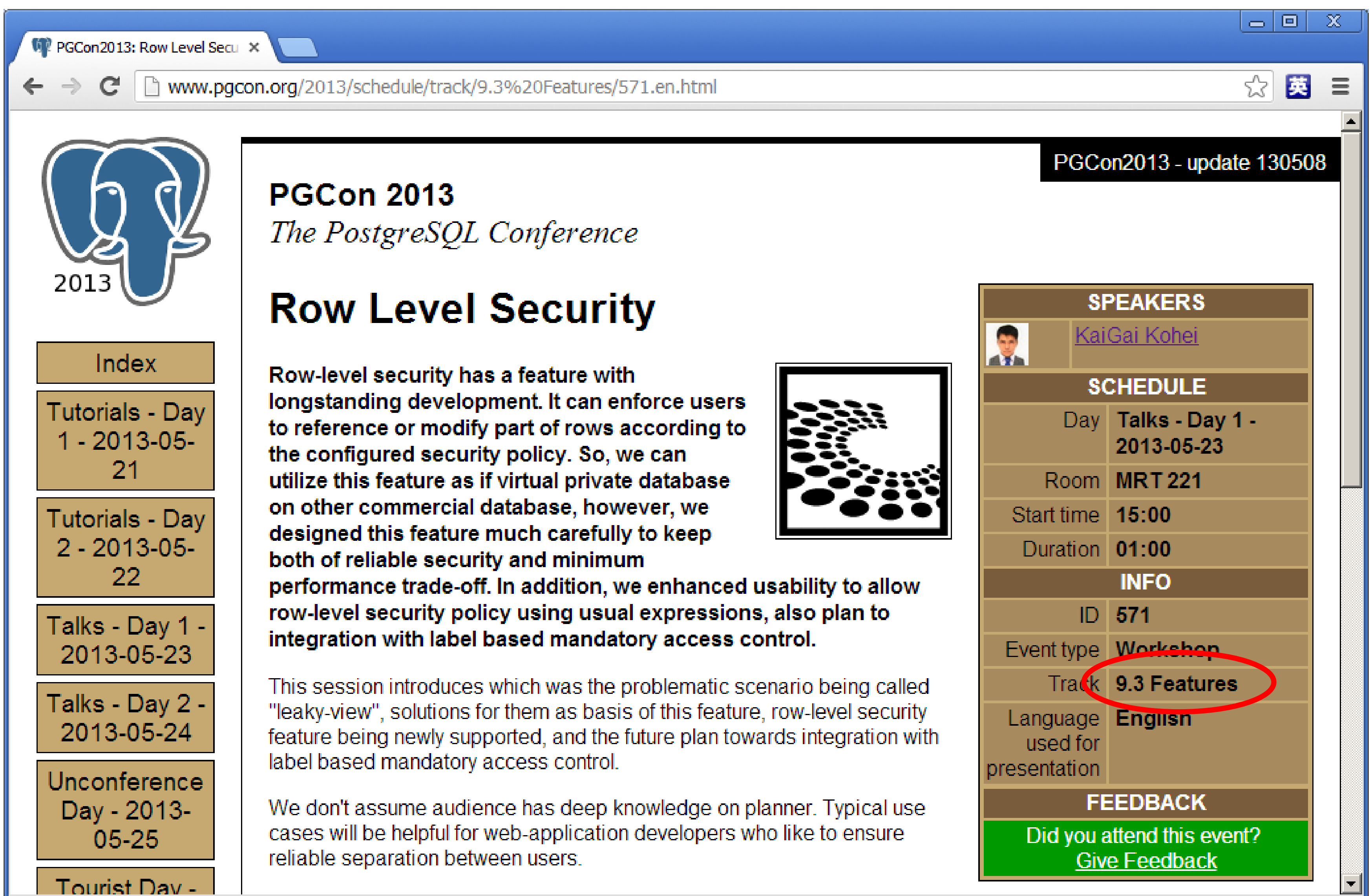
Row Level Security

NEC Europe, ltd
SAP Global Competence Center
KaiGai Kohei <kohei.kaigai@emea.nec.com>

Row Level Security being targeted towards v9.4

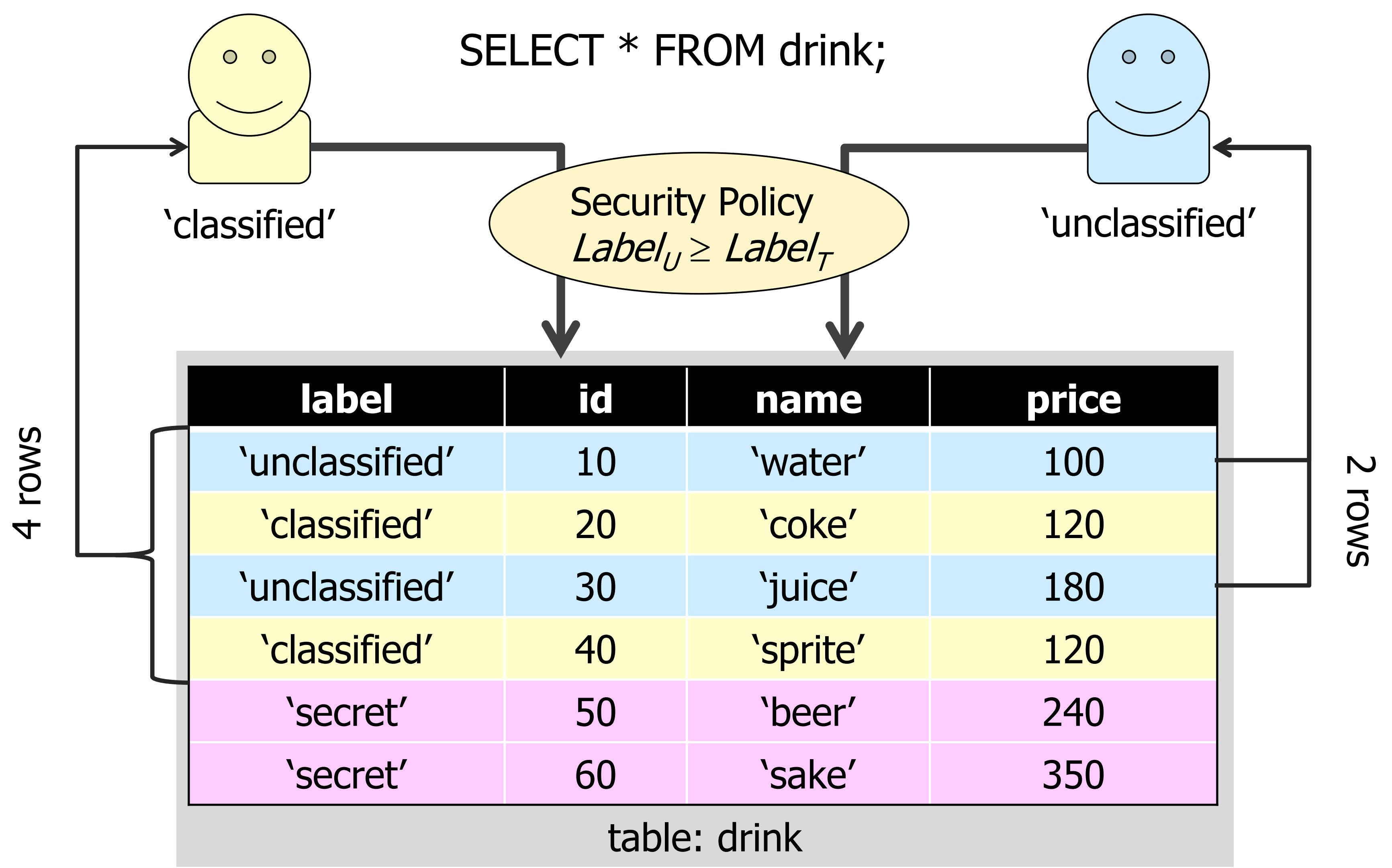


Empowered by Innovation NEC

Agenda

- Our motivation
- Background Story
  - Leaky-view Problem
  - Security Barrier
  - Leakproof Function
- Row Level Security

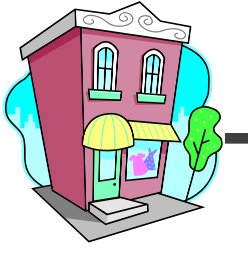
How RLS should work (1/2)



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

Empowered by Innovation NEC

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

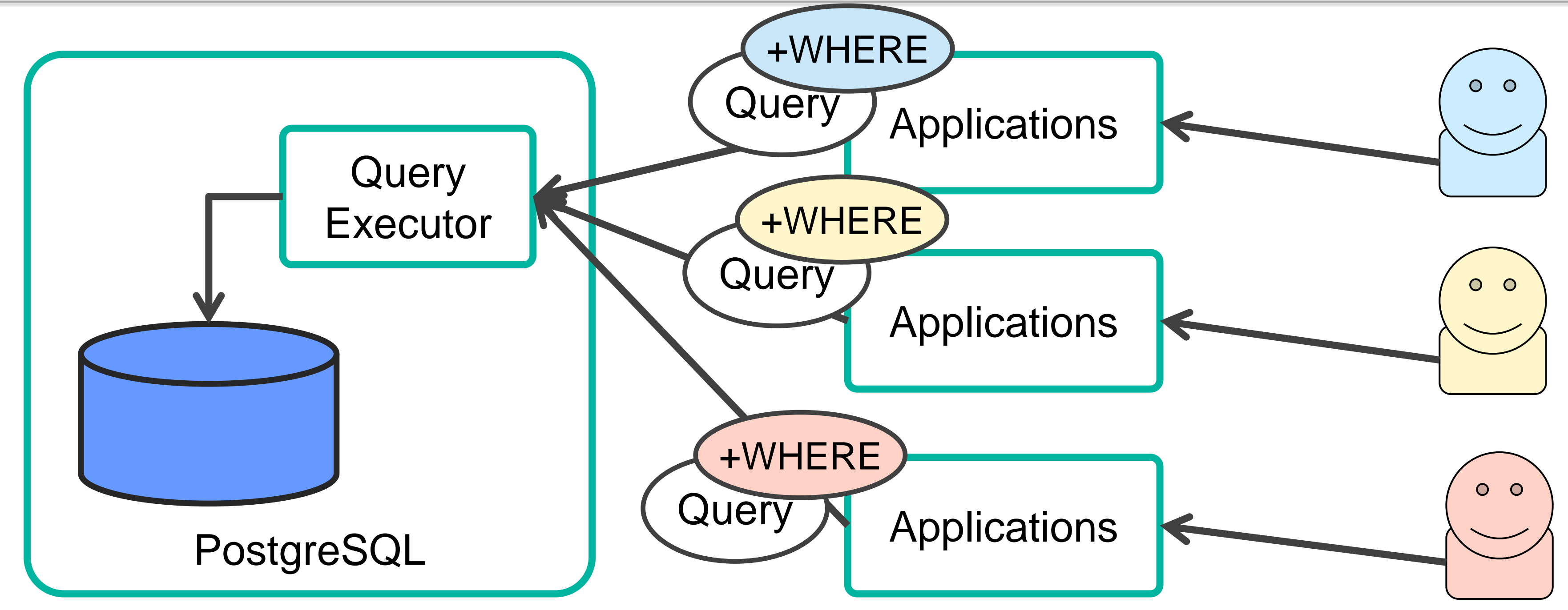
Security Policy
$shop_id_U = \text{shop_id}_T$

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

Empowered by Innovation NEC

Motivation (1/2) – Responsibility of access control



- A case when a shared table is accessed by multiple users
- Not easy to ensure applications are bug/vulnerability free
- Move the responsibility of correct access control from Application to RDBMS ➔ contralization & consistency

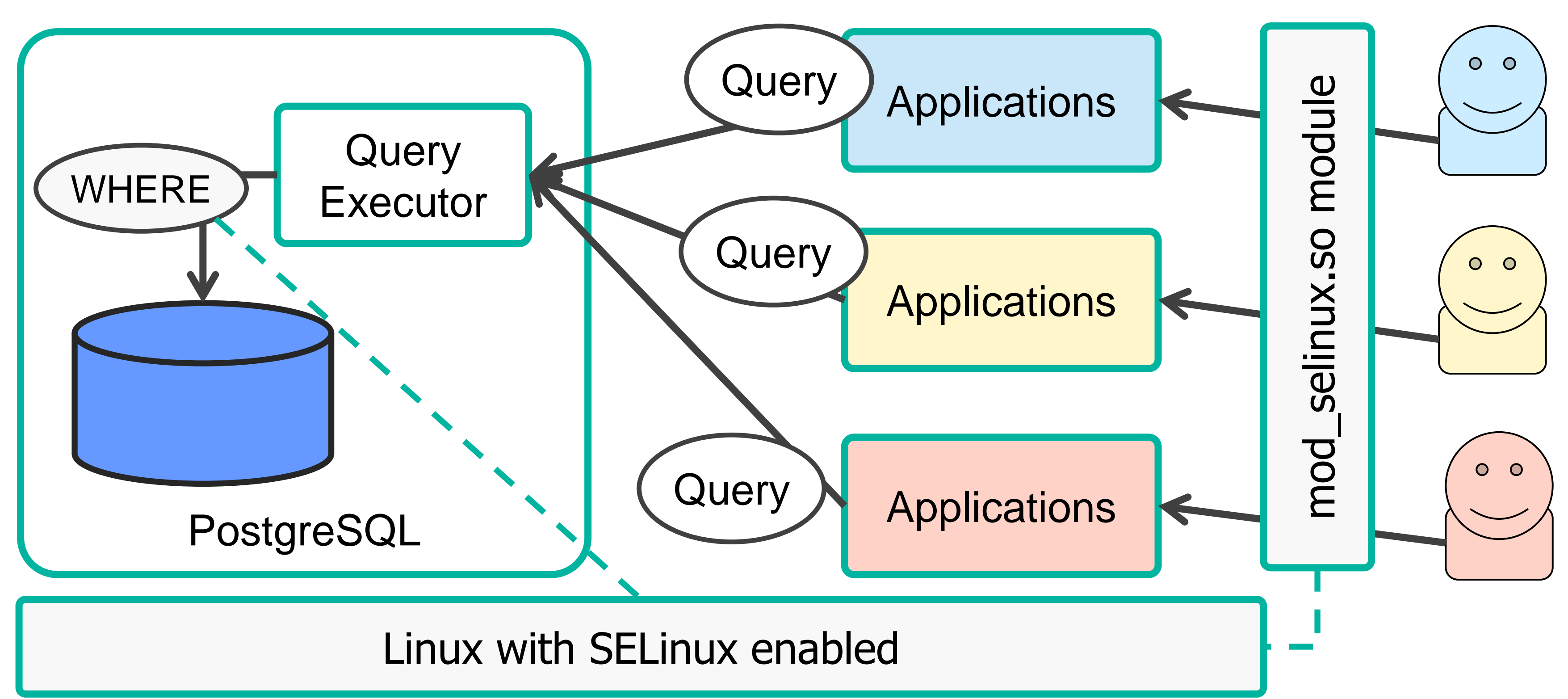
Empowered by Innovation NEC

Motivation (1/2) – Responsibility of access control

+WHERE
+WHERE
+WHERE
Query Executor
PostgreSQL
Query
Applications
Query
Applications
Query
Applications

- A case when a shared table is accessed by multiple users
- Not easy to ensure applications are bug/vulnerability free
- Move the responsibility of correct access control from Application to RDBMS ➔ centralization & consistency

Motivation (2/2) – Integration with SELinux



- Per user privileges of application instance on its start-up time
- Access control decision based on a centralized security policy
- Integration of access control between OS and DBMS

Empowered by Innovation NEC

Is WHERE-clause a 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)
```


Empowered by Innovation NEC

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)
```

'<' is more expensive than f_leak()

Empowered by Innovation NEC

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 '%abc%';
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 ~~ '%abc%'::text)
(7 rows)
```

f_leak() takes arguments come from t_left only

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)
```

Symptom

- Query optimization reorders the sequence of

Solution

- If purpose of the view is security, qualifiers should not be moved across the sub-query border.

Empowered by Innovation NEC

Security Barrier (1/2)

```
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 (2/2)

```
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 ...

- Prevention of user given qualifier into views with security_barrier attribute
- Advantage: qualifiers shall be evaluated according to user's intention
- Disadvantage: may not optimized query execution plan, instead

Empowered by Innovation NEC

Trade-off between performance and security

```
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)
```

- Query should be index-scannable using id=100
- Due to security_barrier attribute, sequential scan on “employee” first, then evaluation of “id=100”

Leakproof Function (1/2)

Leakproof attribute

- It shows the marked function is definitely safe.
- Thus, no side effects if it would be pushed down.

```
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
```


Empowered by Innovation NEC

Leakproof Function (2/2)

```
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)
```

Some functions are LEAKPROOF in the default

- Example) Equivalent operator between integers

```
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")     but performance loss!
   3 - filter("B"."Y"<>'bbb')
```

Towards v9.4 development cycle

Features in v9.2

- security_barrier attribute of VIEW
- leakproof attribute of FUNCION

Features in v9.3

- Row-level security discussion was time-over! (;_;)

Features in v9.4

- ALTER TABLE ... SET ROW SECURITY (...) statement
- Writer side checks
- 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
```

- <expression> (performing as a security policy) shall be appended on the query specified by <cmd>
- It is guaranteed that security policy is evaluated earlier than user given qualifiers.

```
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)
```

Sub-Query with security_barrier

- Replacement of table reference by a simple table scan with security barrier attribute and qualifiers of security policy
- Database superuser is an exception

How does RLS work? (1/2)

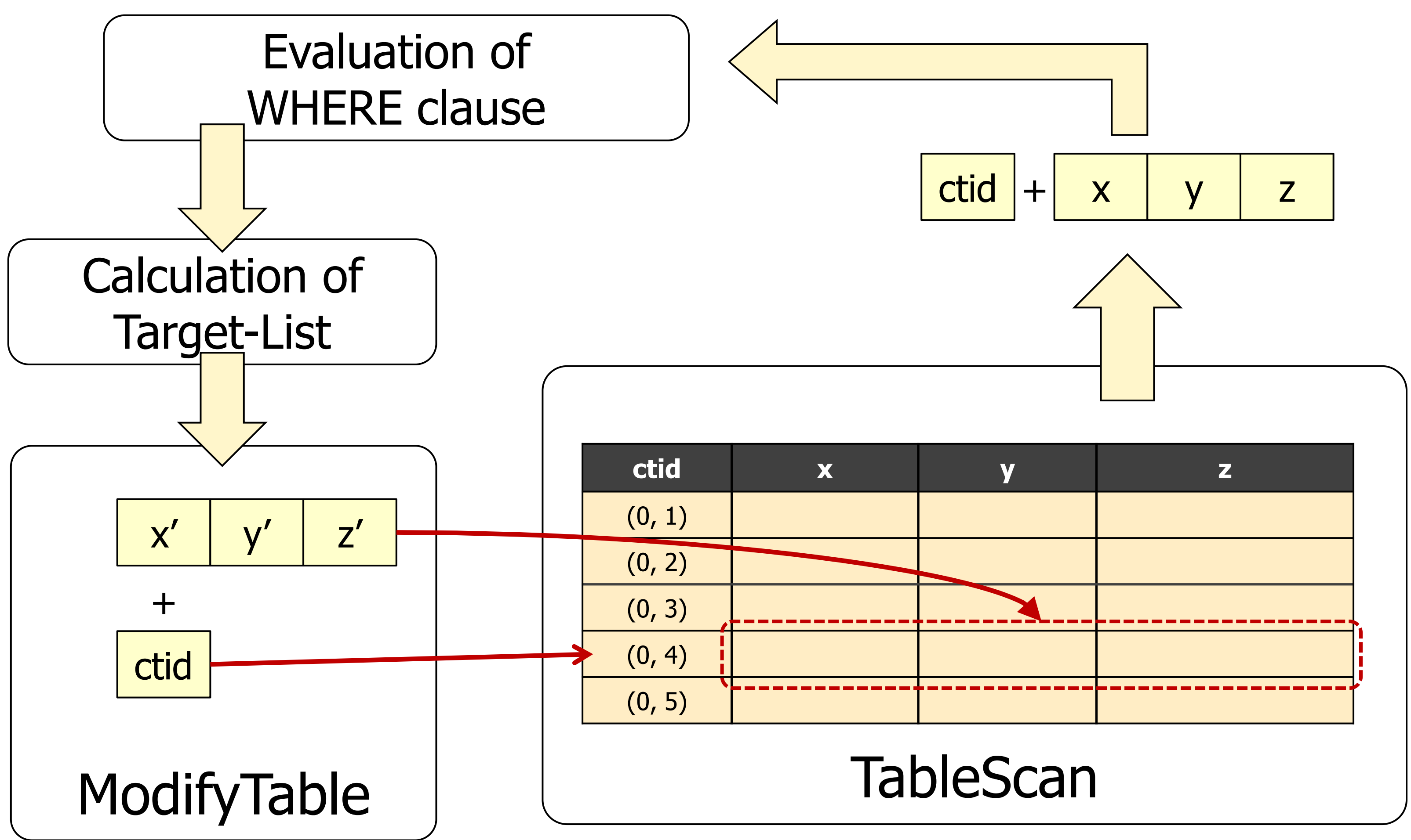
```
postgres=> ALTER TABLE t
        SET ROW SECURITY FOR ALL TO (owner = current_user);
ALTER TABLE

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"())
```

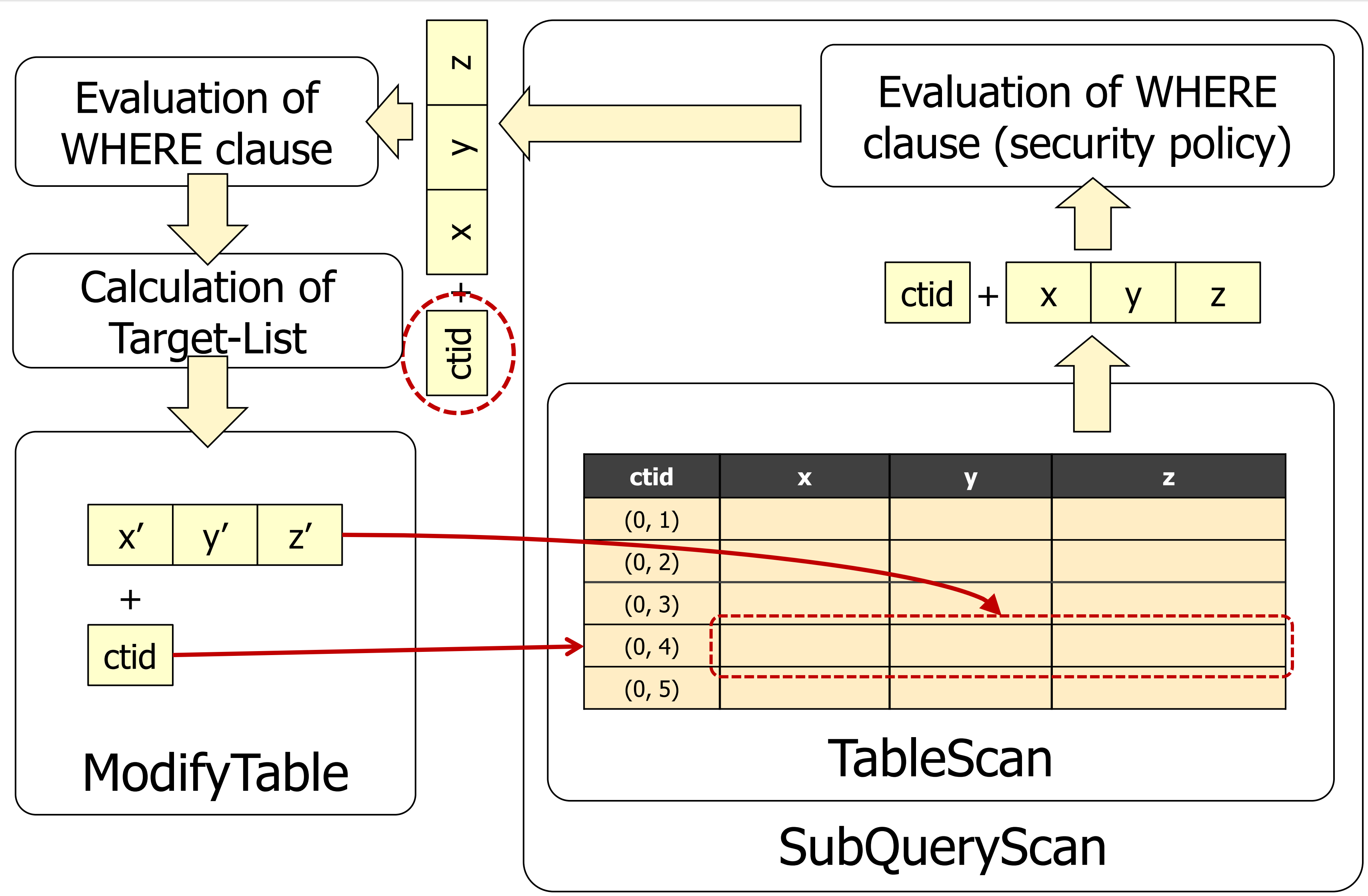
```
postgres=> EXPLAIN(costs off)
           DELETE FROM t WHERE f_leak(b);
                      QUERY PLAN
------------------------------------------------------
 Delete on t
   ->  Subquery Scan on t_1
         Filter: f_leak(t_1.b)
         ->  Index Scan using my_table_pkey on t t_2
           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

Table Update and RLS (1/2)



Empowered by Innovation NEC

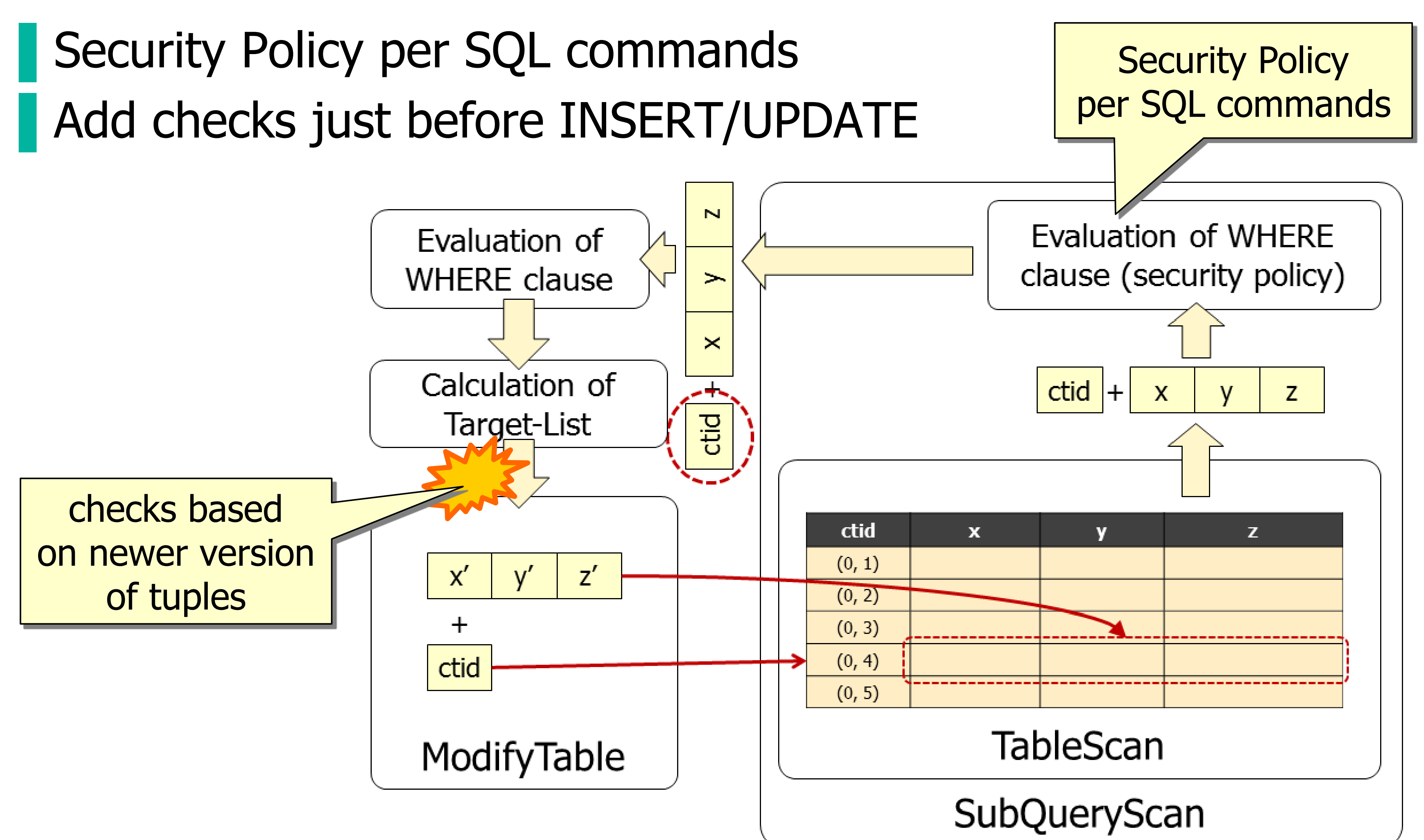
Table Update and RLS (2/2)



Empowered by Innovation NEC

Further development (1/2)

- Security Policy per SQL commands
- Add checks just before INSERT/UPDATE



Empowered by Innovation NEC

Futher development (2/2)

- Labal-based Row-level Security
  - access control functions according to SELinux policy
- Step to implementation
  - all the features of “standard” row-level security
  - security-label assignment on user’s table
  - enumerate type that can add items at run-time
  - enhancement of contrib/sepgsql

Resources

CommitFest:1st to v9.4

- https://commitfest.postgresql.org/action/commitfest_view?id=18

Git repository

- https://github.com/kaigai/sepgsql/tree/rowsec

Wikipage

- http://wiki.postgresql.org/wiki/RLS

Any Questions?

Empowered by Innovation