# Compiling PL/SQL Away

**CIDR 2020**

Christian Duta • Denis Hirn • Torsten Grust
**University of Tübingen**

# How is Everyone Enjoying Their PL/SQL?

From time to time I seek comfort in the success stories that tell how users benefit from the technology we've created ...

# PL/SQL: Computation Close to the Data (↓ Robot Walk)

```
CREATE FUNCTION walk(origin coord, win int, loose int, steps int) RETURNS int AS
$$
DECLARE
 reward   int   = 0;
 location coord = origin;
 movement text  = '';
 roll     float;
BEGIN
 -- move robot repeatedly
 FOR step IN 1..steps LOOP
  -- where does the Markov policy send the robot from here?
  movement = (SELECT p.action
              FROM   policy AS p
              WHERE  location = p.loc);
  -- compute new location of robot,
  -- robot may randomly stray from policy's direction
  roll = random();
  location =
    (SELECT move.loc
     FROM   (SELECT a.there AS loc,
                    COALESCE(SUM(a.prob) OVER lt, 0.0) AS lo,
                             SUM(a.prob) OVER leq      AS hi
             FROM   actions AS a
             WHERE  location = a.here AND movement = a.action
             WINDOW leq AS (ORDER BY a.there),
                    lt  AS (leq ROWS UNBOUNDED PRECEDING
                                EXCLUDE CURRENT ROW)
            ) AS move(loc, lo, hi)
     WHERE  roll BETWEEN move.lo AND move.hi);
  -- robot collects reward (or penalty) at new location
  reward = reward + (SELECT c.reward
                     FROM   cells AS c
                     WHERE  location = c.loc);
  -- bail out if we win or loose early
  IF reward >= win OR reward <= loose THEN
    RETURN step * sign(reward);
  END IF;
 END LOOP;
 -- draw: robot performed all steps without winning or losing
 RETURN 0;
END;
$$ LANGUAGE PLPGSQL;
```

# PL/SQL: ½ Imperative PL + ½ SQL (↓ N-Body Simulation)

```
CREATE FUNCTION force(b body, theta float) RETURNS point AS
$$
 DECLARE
 force      point := point(0,0);
 G CONSTANT float := 6.67e-11;
 Q          barneshut[];
 node       barneshut;
 children   barneshut[];
 dist       float;
 dir        point;
 grav       point;
BEGIN
 -- enter Barnes-Hut tree at the root
 node = (SELECT t
         FROM   barneshut AS t
         WHERE  t.node = 0);
 Q = array[node];
 -- iterate while there are Barnes-Hut nodes to consider
 WHILE cardinality(Q) > 0 LOOP
  node = Q[1];
  Q    = Q[2:];
  dist = node.center->b.pos;
  dir  = node.center - b.pos;
  grav = point(0,0);
  -- bodies separated by walls do not affect each other
  IF NOT EXISTS (SELECT 1
                 FROM   walls AS w
                 WHERE  (b.pos <= b.pos ## w.wall) <>
                        (node.center <= node.center ## w.wall)) THEN
   grav = (G * b.mass * node.mass / dist^2) * dir;
  END IF;
  -- Barnes-Hut optimization: approximate effect of distant bodies
  IF (node.node IS NULL) OR (width(node.bbox) / dist < theta) THEN
   force = force + grav;
  ELSE
   -- inspect area at higher resolution: descend into subtrees
   children = (SELECT array_agg(t)
               FROM   barneshut AS t
               WHERE  t.parent = node.node);
   Q = Q || children;
  END IF;
 END LOOP;
 -- return aggregated force on body
 RETURN force;
END;
$$ LANGUAGE PLPGSQL STABLE STRICT;
```

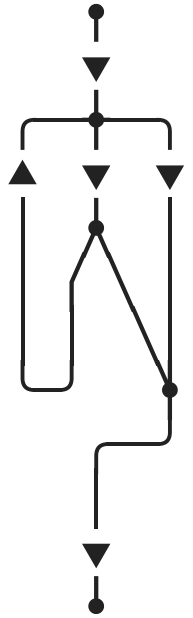
# Elements of PL/SQL: Stateful Variables

```
CREATE FUNCTION f(…) RETURNS τ AS
$$
  DECLARE
    v1 τ1;
    v2 τ2;
  BEGIN
    =====
    v1 = ==========;
    v2 = =====v1==;
    =======
    v1 = ==v1===v2====;
    ====
  END;
$$
```

- Statement sequencing (straight-line control flow)
- Variable references and variable updates

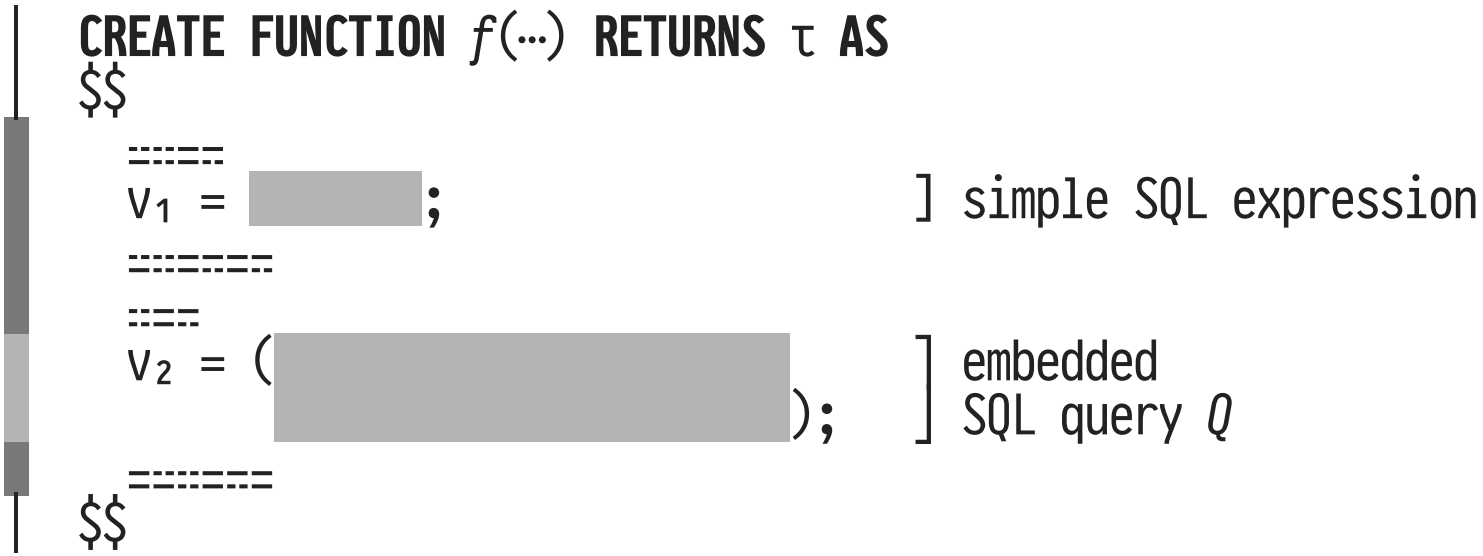
# Elements of PL/SQL: Complex and Iterative Control Flow

```
CREATE FUNCTION f(…) RETURNS τ AS
$$
  =====
  WHILE ==== LOOP
  | ========
  | IF ==== THEN
  | | ====
  | ELSE
  | | EXIT;
  | END IF;
  | =====
  END LOOP;
  ====
$$
```



- Arbitrarily complex control flow (via `IF…THEN…ELSIF`, `CASE…WHEN`, `LOOP`, `WHILE`, `FOR`, `EXIT`, `CONTINUE`, …)

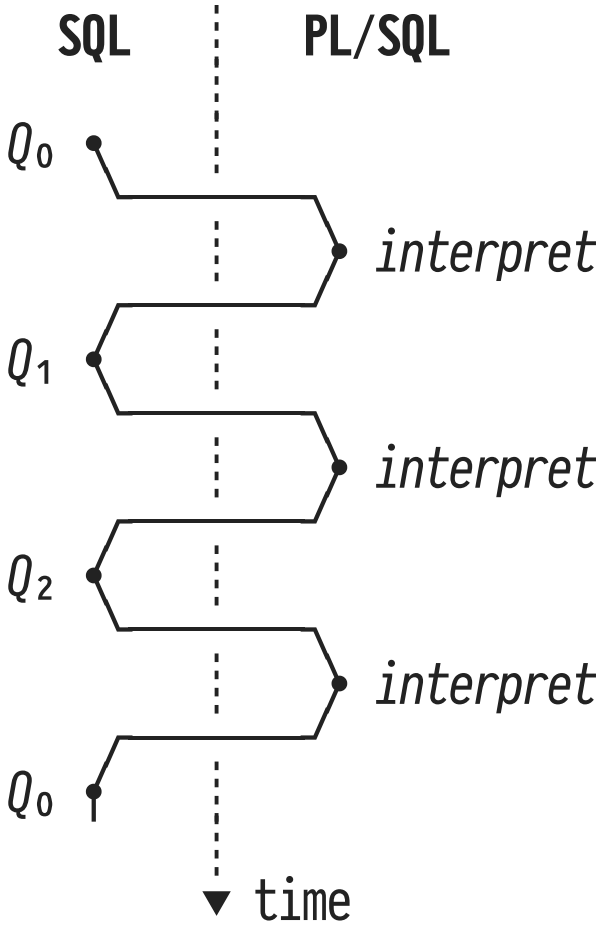
# Elements of PL/SQL: Embedded SQL Expressions




- PL/SQL expressions:
  1. “Simple” SQL expressions (evaluated w/o planning)
  2. Embedded SQL queries (require planning + execution)

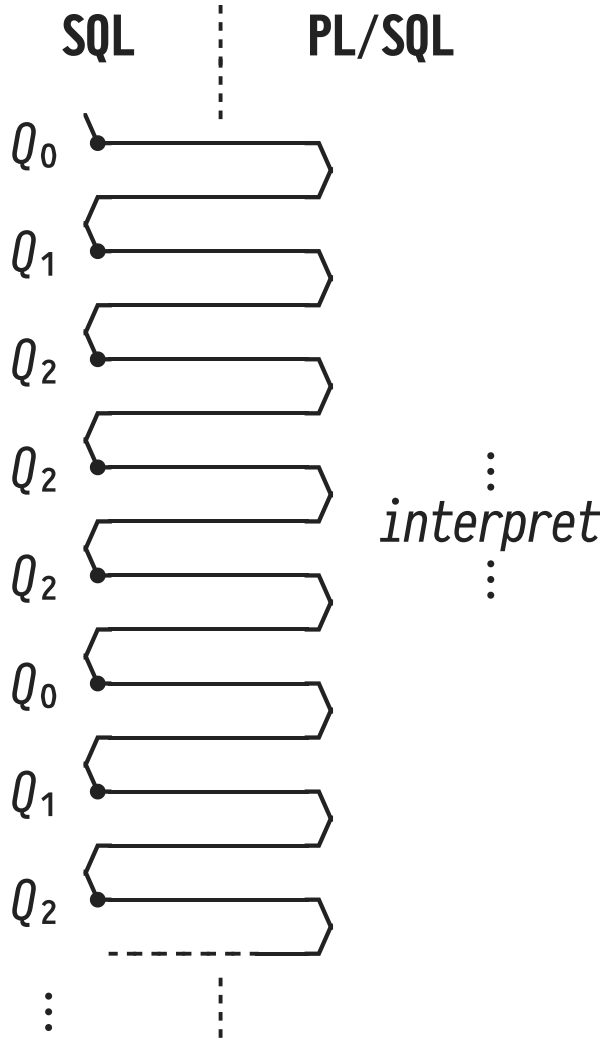
# From SQL to PL/SQL And Back Again

```
CREATE FUNCTION f(…) RETURNS τ AS
$$
  =====
  v1 = (                          );   ] embedded SQL Q1
  ==============
  ============================
    ==============
    ==============
  ====
    =======
    v2 = (                        );   ] embedded SQL Q2
    =======
  ============
$$

SELECT f(…);                           ] top-level SQL Q0
```

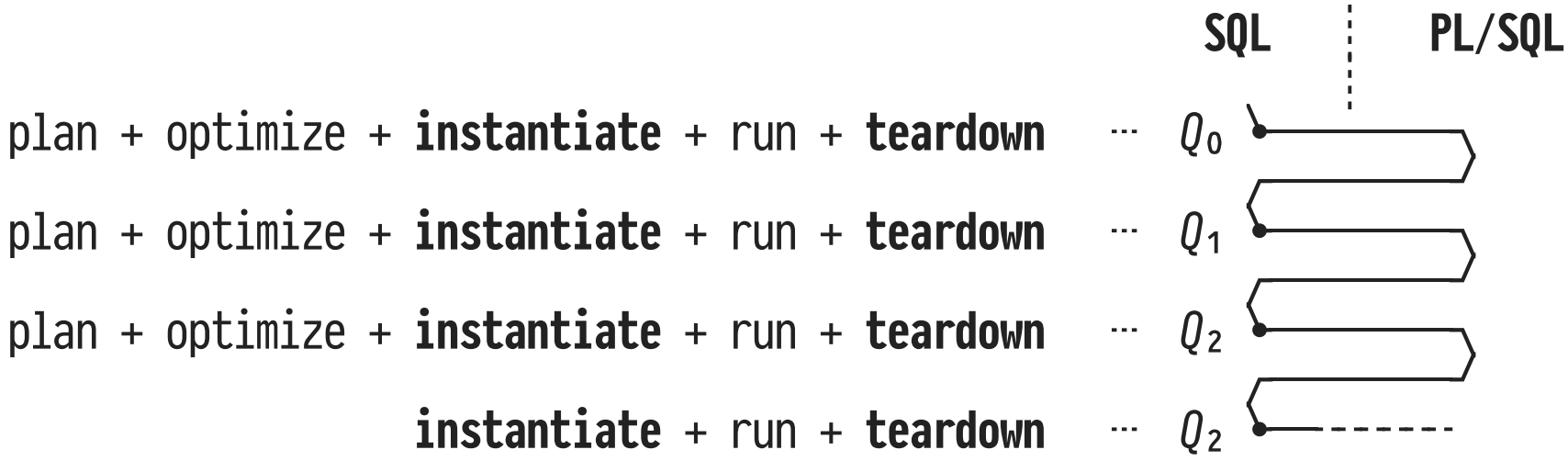


# SQL↔PL/SQL (And Again, Again, Again, Again, Again, …)

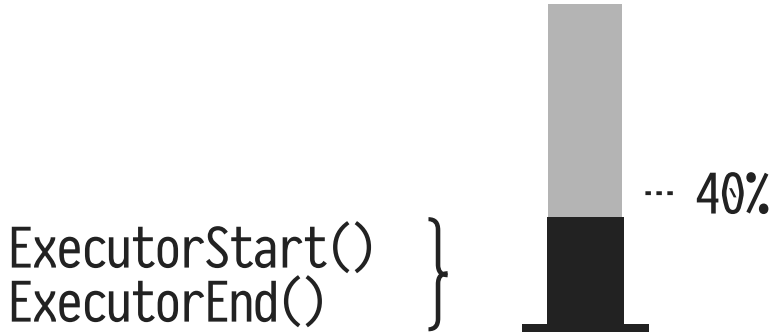
```
CREATE FUNCTION f(…) RETURNS τ AS
$$
  ======
  v1 = ( ▒▒▒▒▒▒▒▒▒▒ );          ] embedded SQL Q1
  ================
  WHILE ==== LOOP
    ================
      ================
    ====
      =======
      v2 = ( ▒▒▒▒▒▒▒▒▒▒▒▒▒▒ );  ] embedded SQL Q2
  LOOP END
  ============
$$

SELECT f(…,t,…);                ] top-level SQL Q0
FROM   T AS t
```



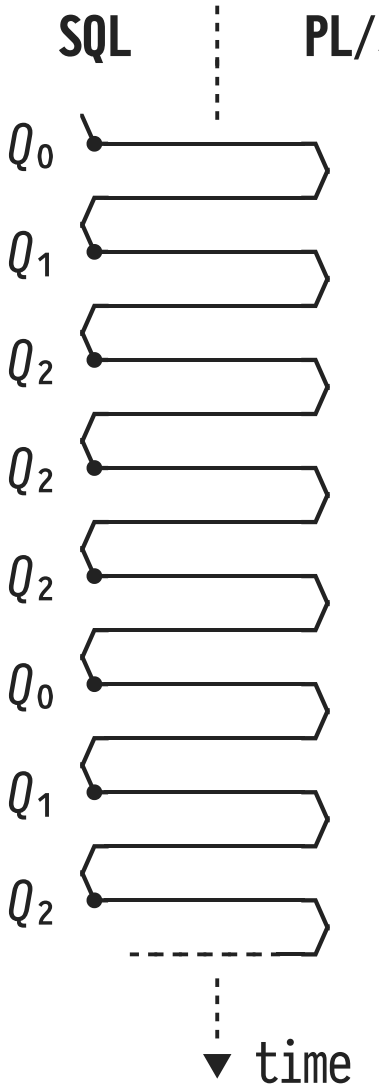
# SQL↔PL/SQL Context Switches Are Costly

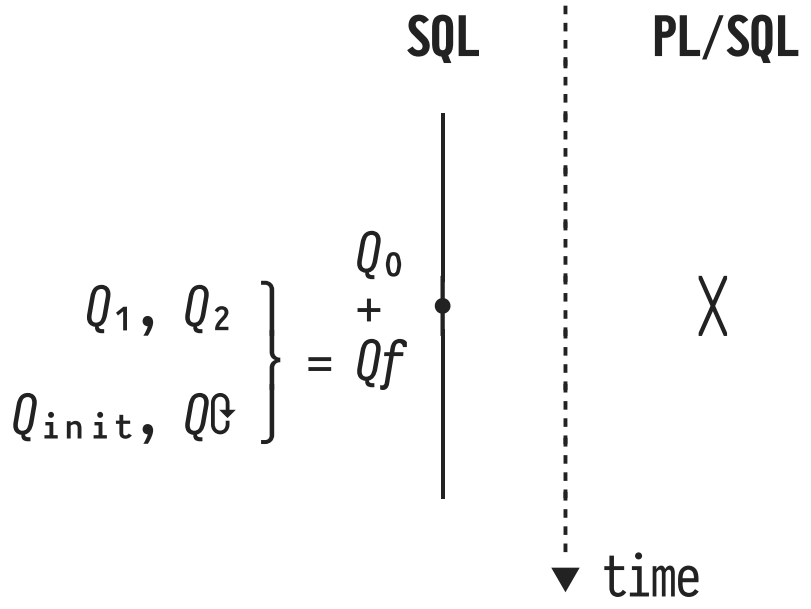



## PostgreSQL 11.3 Runtime Profile



# Compiling PL/SQL Away




**Main Goal:**

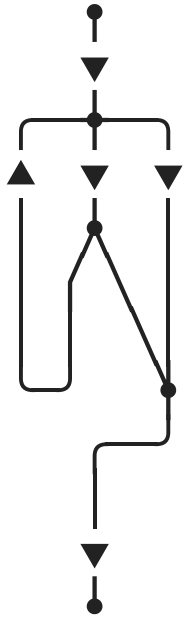
- Compile PL/SQL UDF *f* into plain SQL query *Qf*
- Inline *Qf* into top-level $Q_0$ at *f*'s call sites: plan, optimize, instantiate, run, teardown **once**

**No SQL↔PL/SQL context switches**

# If *f* Is Iterative, *Qf* Is Recursive

**PL/SQL** ➧ **Plain SQL**

**CREATE FUNCTION** *f*(…)
**RETURNS** τ **AS**
$$



$$

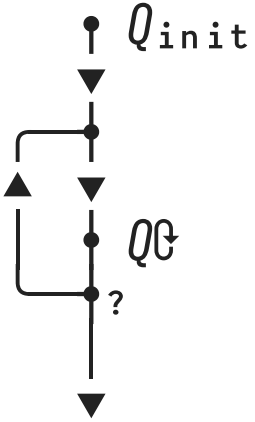
**WITH RECURSIVE** *run* **AS**

$Q_{init}$ [ ]

**UNION ALL**

$Q_{\circlearrowright}$ [ ]

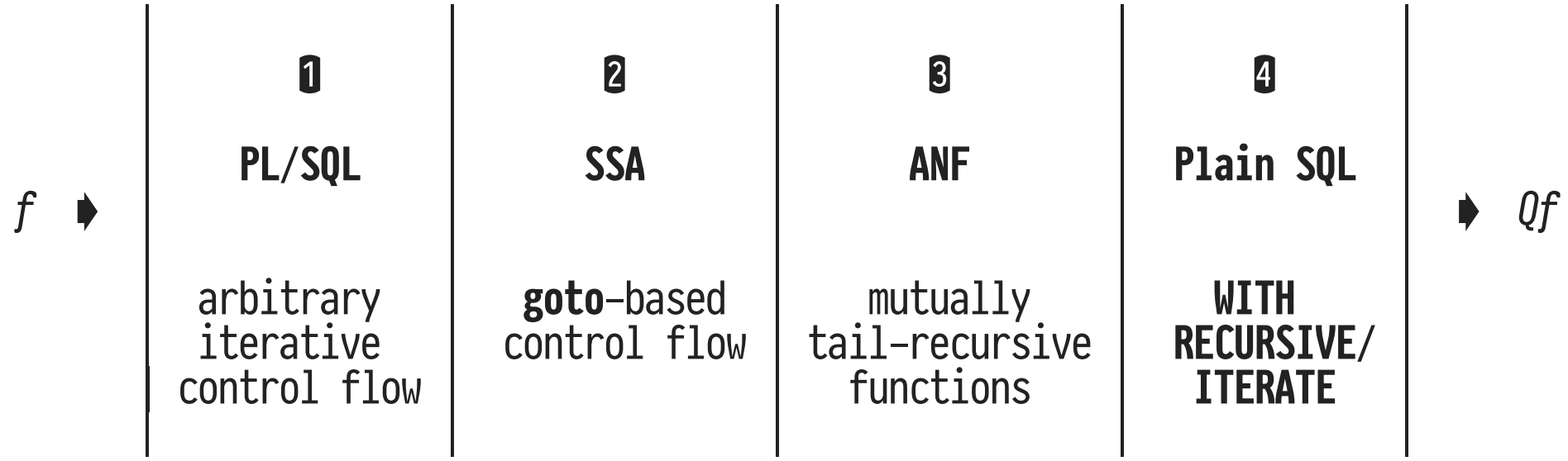
*Qf*




**Challenge:**

Map arbitrary control flow into **WITH RECURSIVE**

# PL/SQL to Plain SQL: Left to Right




(⚠ Compiler folks tend to go right to left ⧫ instead...)

# ❶ From PL/SQL to Single Static Assignment Form

**PL/SQL** ($x^n$) ➧ **SSA**

```
CREATE FUNCTION f(x int, n int)
RETURNS int AS
$$
  DECLARE
    i int = 0;
    p int = 1;
  BEGIN
    WHILE i < n LOOP
      p = p * x;
      i = i + 1;
    END LOOP;
    RETURN p;
  END;
$$
```

```
f(x,n):
        i₀ ← 0;
        p₀ ← 1;
 while: i₁ ← ϕ(i₀,i₂);
        p₁ ← ϕ(p₀,p₂);
        if i₁ < n then
          goto loop;
        else
          goto exit;
  loop: p₂ ← p₁ * x;
        i₂ ← i₁ + 1;
        goto while;
  exit: return p₁;
```

- Control flow expressed via **goto**, variables assigned once

# ❷ From SSA to Administrative Normal Form

**SSA** ➧ **ANF** ⟲

```
f(x,n):
        i₀ ← 0;
        p₀ ← 1;
while:  i₁ ← ϕ(i₀,i₂);
        p₁ ← ϕ(p₀,p₂);
        if i₁ < n then
          goto body;
        else
          goto exit;
 body:  p₂ ← p₁ * x;
        i₂ ← i₁ + 1;
        goto while;
 exit:  return p₁;
```

```
f(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
    while(i₀,p₀,x,n)

while(i₁,p₁,x,n) =
  let t₀ = i₁ ⩾ n in
   if t₀ then p₁                    [← exit]
         else body(i₁,p₁,x,n)

body(i₁,p₁,x,n) =
  let p₂ = p₁ * x in
   let i₂ = i₁ + 1 in
    while(i₂,p₂,x,n)
```

- Mutually recursive functions, **tail calls** only

# ❸ From Mutual to Direct Recursion

**ANF ⟲**

```
f(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
    while(i₀,p₀,x,n)

while(i₁,p₁,x,n) =
  let t₀ = i₁ ⩾ n in
   if t₀ then p₁
         else loop(i₁,p₁,x,n)

loop(i₁,p₁,x,n) =
  let p₂ = p₁ * x in
   let i₂ = i₁ + 1 in
    while(i₂,p₂,x,n)
```

➧

**ANF ↻**

```
f(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
    run(1,i₀,p₀,x,n)

run(fn,i₁,p₁,x,n) =
  if fn = 1 then
   let t₀ = i₁ ⩾ n in
    if t₀ then p₁
          else run(2,i₁,p₁,x,n)
  else
   let p₂ = p₁ * x in
    let i₂ = i₁ + 1 in
     run(1,i₂,p₂,x,n)
```

- Single recursive function *run*(), **tail calls** only

# ❸ From Mutual to Direct Recursion

**ANF** 🔁

```
f(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
    while(i₀,p₀,x,n)

while(i₁,p₁,x,n) =
  let t₀ = i₁ ⩾ n in
   if t₀ then p₁
         else body(i₁,p₁,x,n)

body(i₁,p₁,x,n) =
  let p₂ = p₁ * x in
   let i₂ = i₁ + 1 in
    while(i₂,p₂,x,n)
```

➧

**ANF** ↻

```
f(x,n) =
  let i₀ = 0 in
   let p₀ = 1 in
    run(i₀,p₀,x,n)

run(i₁,p₁,x,n) =
  let t₀ = i₁ ⩾ n in
   if t₀ then p₁
         else
          let p₂ = p₁ * x in
           let i₂ = i₁ + 1 in
            run(i₂,p₂,x,n)
```

- Single recursive function *run*(), **tail calls** only

# ❹ From Tail Recursion to WITH RECURSIVE

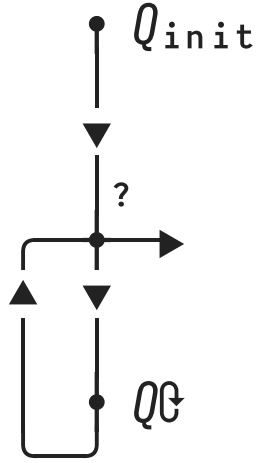
**ANF ↻**

```
f(x,n) =
  let i0 = 0 in
   let p0 = 1 in
    run(i0,p0,x,n)

run(i1,p1,x,n) =
  let t0 = i1 ⩾ n in
   if t0 then p1
         else
          let p2 = p1 * x in
           let i2 = i1 + 1 in
            run(i2,p2,x,n)
```

$Q_{init}$

] ?

$Q↻$




- Resulting tail recursion now matches the restricted control flow implemented by **WITH RECURSIVE** ✔

# ❹ From Tail Recursion to WITH RECURSIVE

**ANF Ⓖ** ➧ **Plain SQL**

```
f(x,n) =
 let i₀ = 0 in
  let p₀ = 1 in
   run(i₀,p₀,x,n)
```

$Q_{init}$

```
run(i₁,p₁,x,n) =
 let t₀ = i₁ ⩾ n in
  if t₀ then p₁
        else
         let p₂ = p₁ * x in
          let i₂ = i₁ + 1 in
           run(i₂,p₂,x,n)
```

$Q_{Ⓖ}$

```
WITH RECURSIVE
run("call?",i₁,p₁,x,n,result) AS (
  SELECT true, 0, 1, x, n, □
 UNION ALL
  SELECT iter.*
  FROM   run, LATERAL (
           SELECT false, □, □, □, □, p₁
           WHERE  i₁ >= n
             UNION ALL
           SELECT true, i₁+1, p₁*x, x, n, □
           WHERE  i₁ < n
         ) AS iter("call?",i₁,p₁,x,n,result)
  WHERE  run."call?")
TABLE run;
```

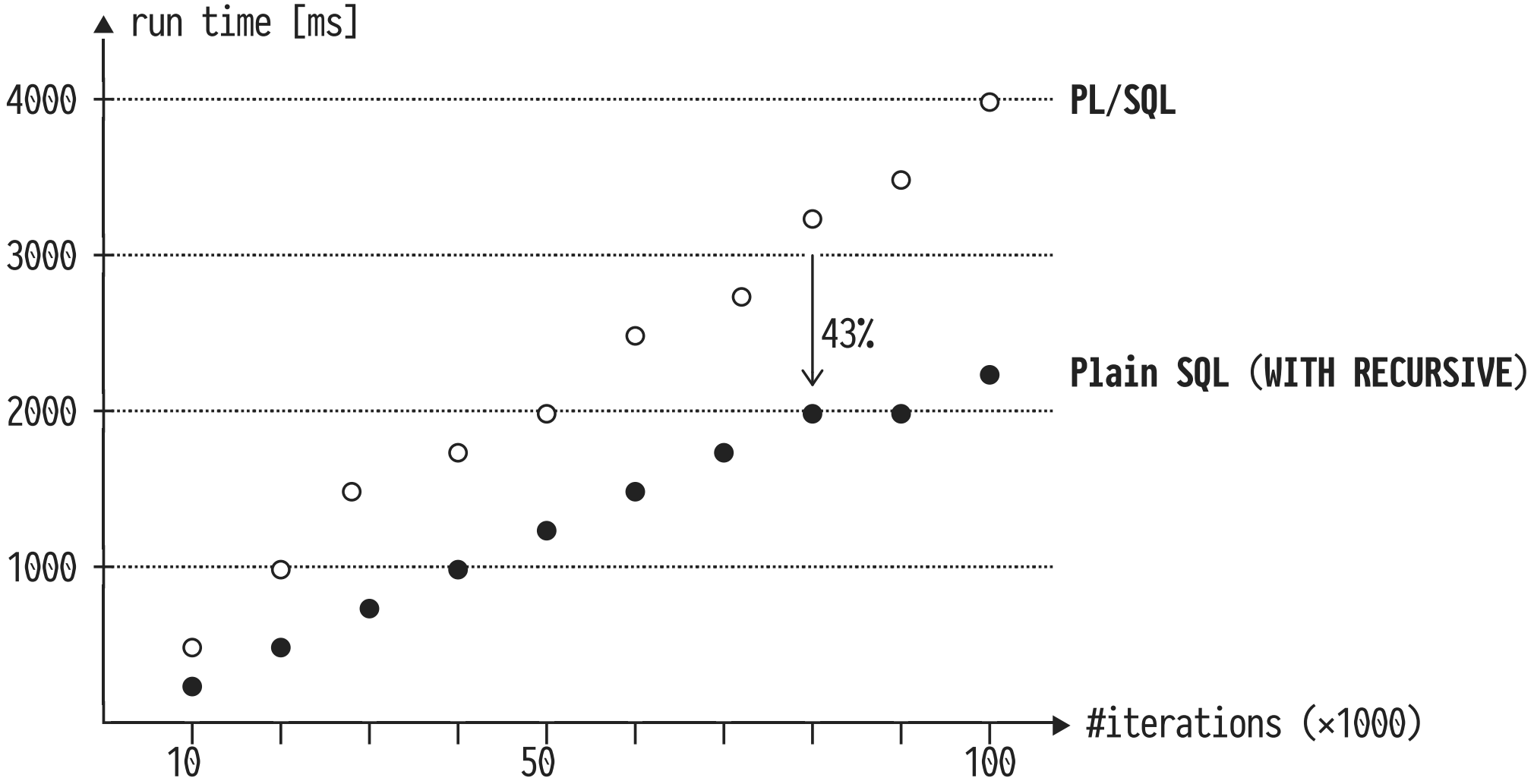
# Invoking a PL/SQL Function ≡ Running the Recursive CTE

**Table** *run* (for $f(2,5) = 2^5$)

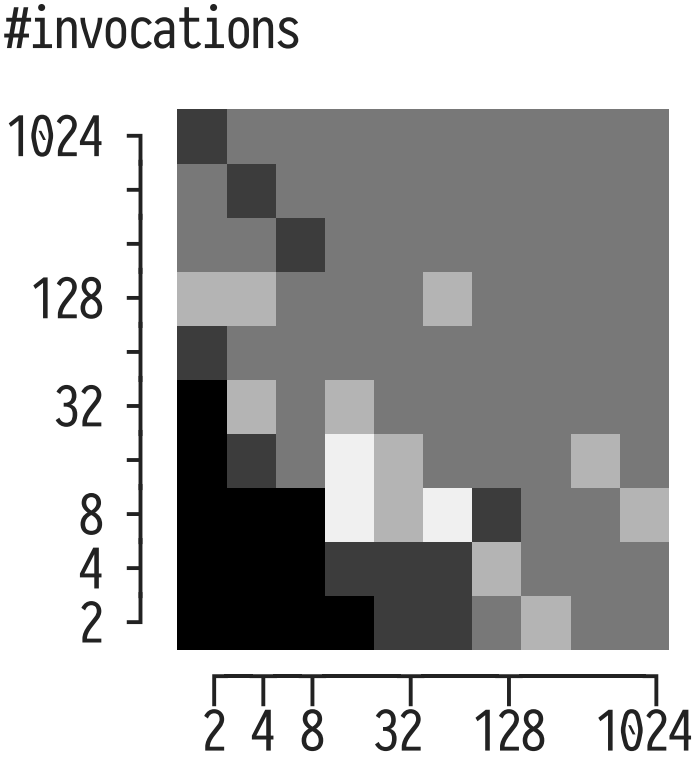
| call? | $i_1$ | $p_1$ | x | n | result |
|---|---|---|---|---|---|
| true | 0 | 1 | 2 | 5 | ▢ |
| true | 1 | 2 | 2 | 5 | ▢ |
| true | 2 | 4 | 2 | 5 | ▢ |
| true | 3 | 8 | 2 | 5 | ▢ |
| true | 4 | 16 | 2 | 5 | ▢ |
| true | 5 | 32 | 2 | 5 | ▢ |
| false | ▢ | ▢ | ▢ | ▢ | 32◀ |

- Look at this table either as...
  - ... the SSA function's **trace of variable states**, or
  - ... the ANF function's **call stack**

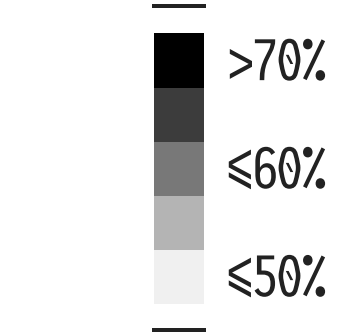
# SQL Recursion vs. PL/SQL Iteration (Intra-Function)

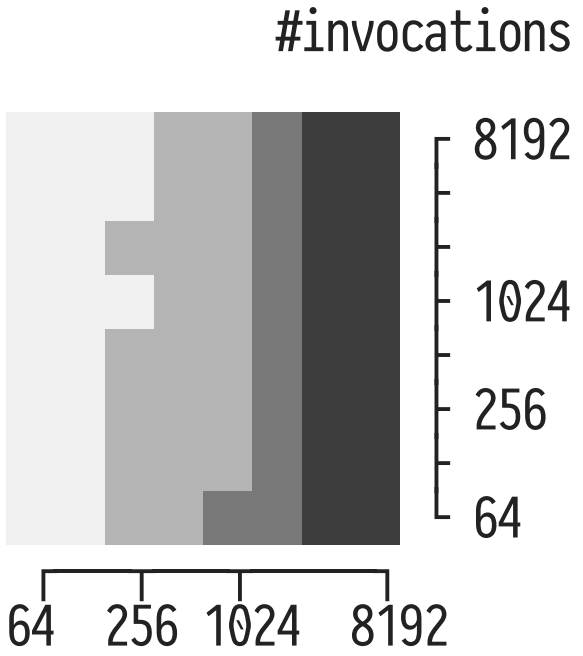


# Scaling the Number of PL/SQL↔SQL Context Switches




Markov DP-based Robot Walk

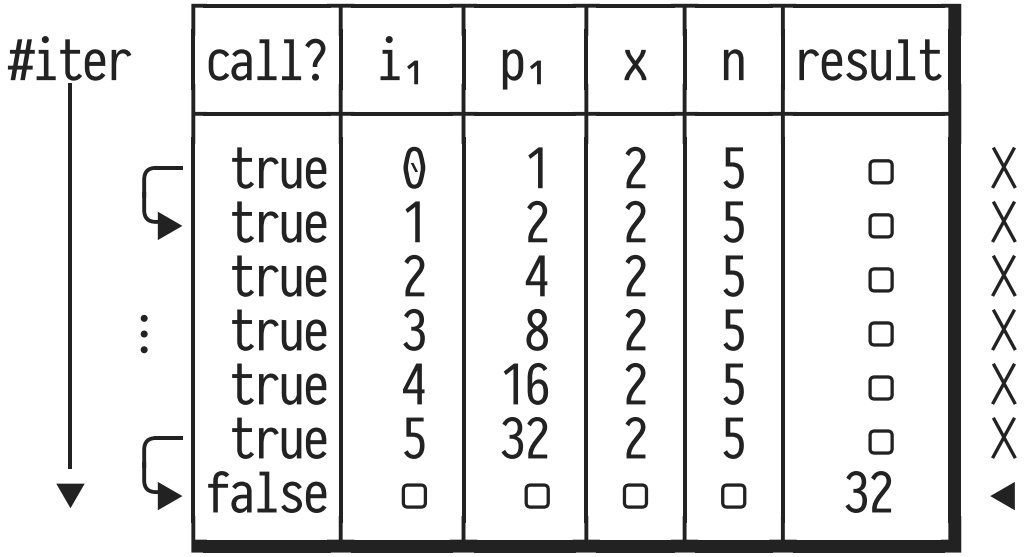



N-Body Simulation

# When WITH RECURSIVE Does Too Much

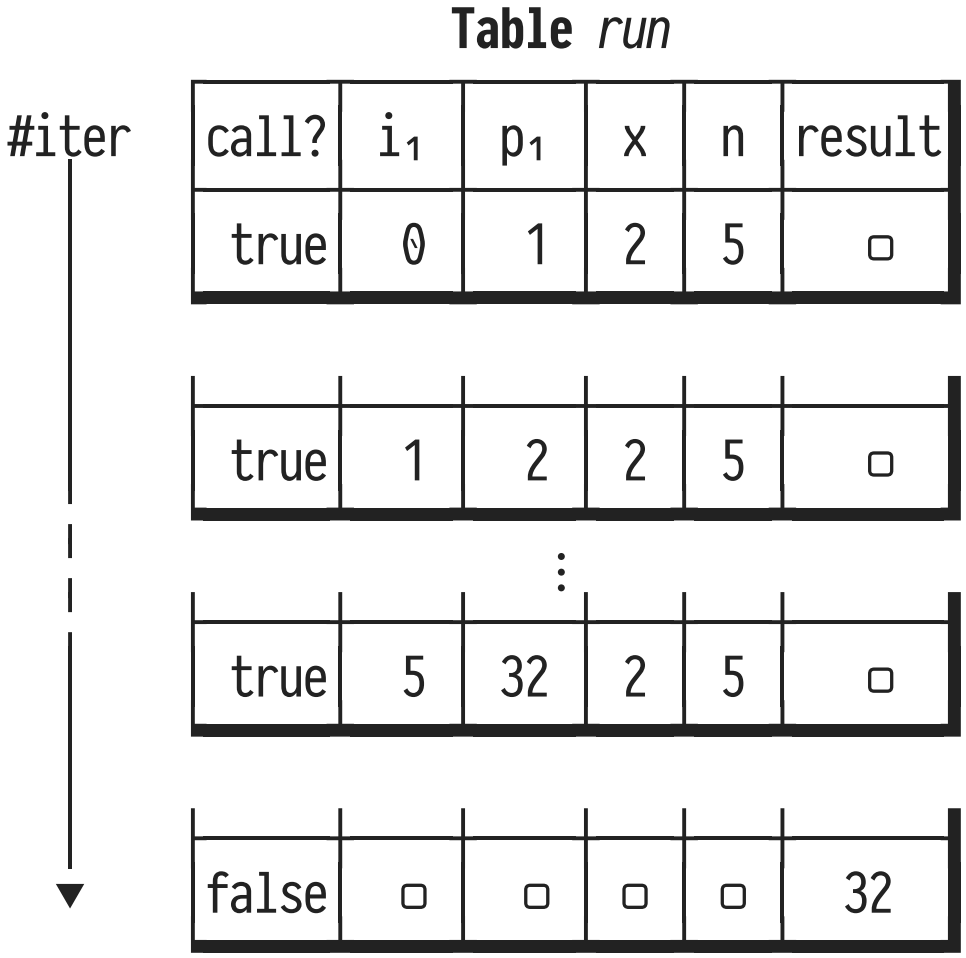
**Table** *run*




```
WITH RECURSIVE run AS (
  ⋮
)
SELECT result
FROM   run
WHERE  NOT "call?"
```

- Since tail recursion *does not look upwards the stack*, a single-row “stack” suffices

# If We Had “WITH TAIL RECURSIVE”…

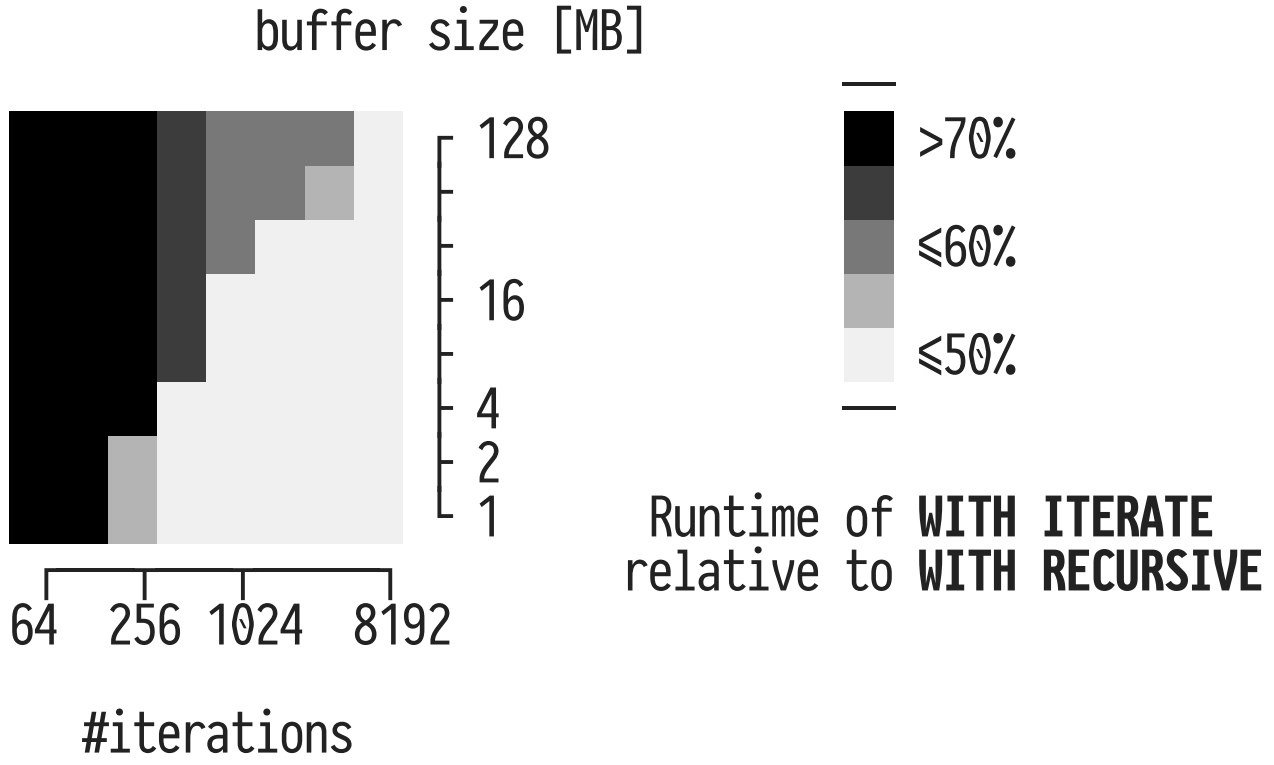



```
WITH ITERATE run AS (
  ⋮
)
SELECT result
FROM   run
WHERE  NOT "call?"
```

||| **WITH ITERATE:**

- Singleton table *run*
- Allocates no buffer space

# WITH ITERATE: Space-Efficient Iteration




- Local change to PostgreSQL, two implementation avenues

## Compiling PL/SQL Away

- Once compiled, **no traces of PL/SQL remain**
  - Thus, zero PL/SQL↔SQL context switches occur
- **Source-to-source compilation** *on top* of the RDBMS
  - Applies to Oracle and SQL Server just as well
- Run compilation chain…
  - …up to ANF: derive plain SQL UDFs from PL/SQL
  - …all the way: execute PL/SQL on RDBMSs that implement SQL:1999 but do not support UDFs at all (**SQLite3**)

# Compiling PL/SQL Away

**CIDR 2020**

Christian Duta • Denis Hirn • Torsten Grust
**University of Tübingen**

@Teggy | db.inf.uni-tuebingen.de/team/grust