

SQL_Warm_Up_Report

lanz

April 2021

1 Paper Understanding

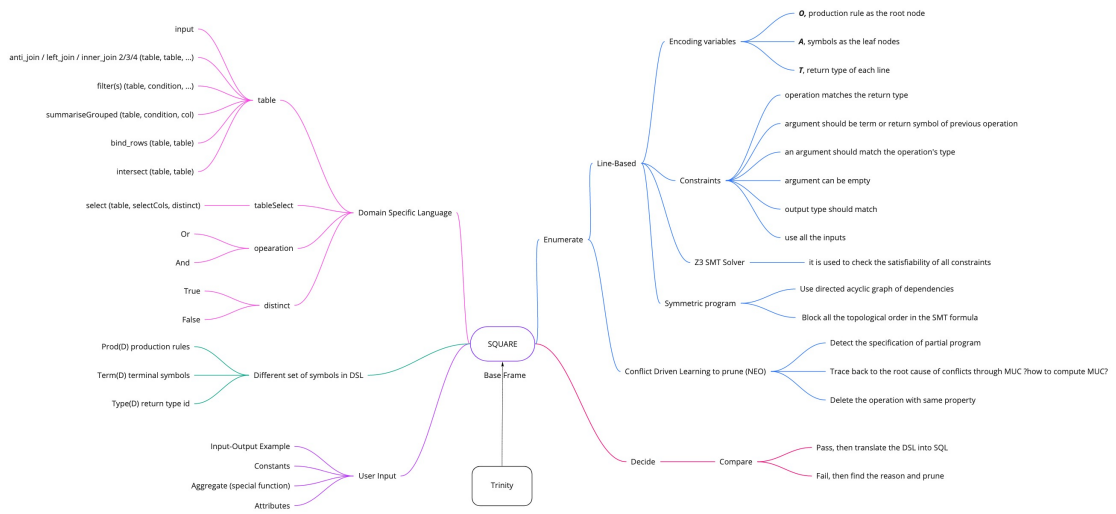


Figure 1: Conception of SQUARE

I read several papers about SQUARE and the tools it used and drew a mind map to clarify the concept and process of how it works.

For the domain specific language, in the SQUARE, there are four types of variables, table, tableSelect, operation and distinct. There are many functions, which are offered by R, such that they returns different types of variables. For example, *_join() function returns a table, bind_rows() function returns a table and select() returns a tableSelect.

The key idea of SQUARE is to enumerate all possible combination of different operations and then pick the one that outputs correctly given the input. To make it efficient, they change the frame into line base, add constraints of each operation, use SMT solver to detect the constraints, use NEO to detect the conflict of partial program and then give up the operation with same property.

The whole process of a single generation should be as followed, for example,

```
max_num_line = ?
max_arity = ?
for num_line in range(max_num_line):
    enumerate num_line operations and max_arity arguments for each operation in the domain specific
    space:
    if Z3 SMT solver checks constraints not pass:
        continue
    if the output is correct:
        output the num_line operations
```

```

else:
    use Neo to get the root cause of the conflict
    block the operations with the same property from being placed at the same position in this
        loop

```

2 Play with SQUARE

After successfully installing SQUARE, I tried several test cases in the folder 55-tests/. Then, I rewrite the generated SQL query into PostgreSQL format in order to use EXPLAIN query to calculate the cost of this SQL query. Here are some examples.

2.1 Testcase1

The generated SQL:

```

SELECT DISTINCT 'S_name'
FROM
    (SELECT 'F_key',
        'F_name',
        'C_name',
        'S_key',
        'S_name',
        'level'
    FROM
        (SELECT 'F_key',
            'F_name',
            'C_name',
            'S_key'
        FROM
            (SELECT 'F_key',
                'F_name',
                'C_name'
            FROM 'input2' AS 'LHS'
            INNER JOIN 'input0' AS 'RHS' ON ('LHS'. 'F_key' = 'RHS'. 'F_key')) AS 'LHS'
            INNER JOIN 'input1' AS 'RHS' ON ('LHS'. 'C_name' = 'RHS'. 'C_name')) AS 'LHS'
            INNER JOIN 'input3' AS 'RHS' ON ('LHS'. 'S_key' = 'RHS'. 'S_key'))
WHERE ('level' = 'JR'
    AND 'F_name' = 'faculty1');

```

The manually transformed PostgreSQL:

```

SELECT DISTINCT S_name
FROM
    (SELECT F_key, F_name, C_name, LHS.S_key, S_name, level
    FROM
        (SELECT F_key, F_name, RHS.C_name, S_key
        FROM
            (SELECT LHS.F_key, F_name, C_name
            FROM input2 AS LHS
            INNER JOIN input0 AS RHS ON (LHS.F_key = RHS.F_key)) AS LHS
            INNER JOIN input1 AS RHS ON (LHS.C_name = RHS.C_name)) AS LHS
            INNER JOIN input3 AS RHS ON (LHS.S_key = RHS.S_key)) AS F00
WHERE (level = 'JR' AND F_name = 'faculty1');

```

The optimal PostgreSQL query:

```

SELECT DISTINCT S_name
FROM input0 i0

```

```

INNER JOIN input2 i2 ON i0.F_key = i2.F_key
INNER JOIN input1 i1 ON i0.C_name = i1.C_name
INNER JOIN input3 i3 ON i1.S_key = i3.S_key
WHERE (level = 'JR' AND F_name = 'faculty1');

```

```

Unique (cost=54.82..54.82 rows=1 width=118)
-> Sort (cost=54.82..54.82 rows=1 width=118)
    Sort Key: rhs.s_name
-> Nested Loop (cost=26.41..54.81 rows=1 width=118)
    Join Filter: ((rhs_2.c_name)::text = (rhs_1.c_name)::text)
-> Hash Join (cost=13.78..27.93 rows=3 width=118)
    Hash Cond: ((rhs_2.f_key)::text = (rhs_1.f_key)::text)
-> Seq Scan on input0 rhs_2 (cost=0.00..13.00 rows=300 width=236)
-> Hash (cost=13.75..13.75 rows=2 width=118)
    Seq Scan on input2 rhs (cost=0.00..13.75 rows=2 width=118)
    Filter: ((f_name)::text = 'faculty1'::text)
-> Materialize (cost=12.64..26.79 rows=2 width=236)
    Hash Join (cost=12.64..26.79 rows=2 width=236)
    Hash Cond: ((rhs_1.s_key)::text = (rhs.s_key)::text)
-> Seq Scan on input1 rhs_1 (cost=0.00..13.00 rows=300 width=236)
-> Hash (cost=12.62..12.62 rows=1 width=236)
    Seq Scan on input3 rhs (cost=0.00..12.62 rows=1 width=236)
    Filter: ((level)::text = 'JR'::text)
(18 rows)

```

Figure 2: testcase1_generated

```

Unique (cost=54.82..54.82 rows=1 width=118)
-> Sort (cost=54.82..54.82 rows=1 width=118)
    Sort Key: i3.s_name
-> Nested Loop (cost=26.41..54.81 rows=1 width=118)
    Join Filter: ((i3.c_name)::text = (i1.c_name)::text)
-> Hash Join (cost=13.78..27.93 rows=3 width=118)
    Hash Cond: ((i3.f_key)::text = (i2.f_key)::text)
-> Seq Scan on input0 i3 (cost=0.00..13.00 rows=300 width=236)
-> Hash (cost=13.75..13.75 rows=2 width=118)
    Seq Scan on input2 i2 (cost=0.00..13.75 rows=2 width=118)
    Filter: ((f_name)::text = 'faculty1'::text)
-> Materialize (cost=12.64..26.79 rows=2 width=236)
    Hash Join (cost=12.64..26.79 rows=2 width=236)
    Hash Cond: ((i1.s_key)::text = (i3.s_key)::text)
-> Seq Scan on input1 i1 (cost=0.00..13.00 rows=300 width=236)
-> Hash (cost=12.62..12.62 rows=1 width=236)
    Seq Scan on input3 i3 (cost=0.00..12.62 rows=1 width=236)
    Filter: ((level)::text = 'JR'::text)
(18 rows)

```

Figure 3: testcase1_optimal

For this testcase 1, since the generated SQL query is very similar to the optimal one, the performance is generally the same.

2.2 Testcase4

The generated SQL:

```

SELECT 'S_name'
FROM
  (SELECT 'S_name',
    'meets_at',
    COUNT() AS 'n'
  FROM
    (SELECT 'C_name',
      'meets_at',
      'S_key',
      'S_name'
    FROM
      (SELECT 'C_name',
        'meets_at',
        'S_key'
      FROM 'input0' AS 'LHS'
      INNER JOIN 'input1' AS 'RHS' ON ('LHS'. 'C_name' = 'RHS'. 'C_name')) AS 'LHS'
      INNER JOIN 'input2' AS 'RHS' ON ('LHS'. 'S_key' = 'RHS'. 'S_key'))
    GROUP BY 'S_name',
      'meets_at')
WHERE ('n' > 2.0
  OR 'n' = 2.0)

```

The manually transformed PostgreSQL:

```

SELECT S_name
FROM
  (SELECT S_name, meets_at, COUNT(*) AS n
  FROM
    (SELECT C_name, meets_at, LHS.S_key, S_name
  FROM
    (SELECT LHS.C_name, meets_at, S_key
  FROM input0 AS LHS
  INNER JOIN input1 AS RHS ON (LHS.C_name = RHS.C_name)) AS LHS
  INNER JOIN input2 AS RHS ON (LHS.S_key = RHS.S_key)) AS FOO
  GROUP BY S_name, meets_at) AS FOO
WHERE (n > 2.0 OR n = 2.0);

```

The optimal PostgreSQL query:

```
SELECT S_name
FROM
(SELECT S_name, meets_at, COUNT(*) AS n
FROM (input0 i0
INNER JOIN input1 i1 on i0.C_name = i1.C_name
INNER JOIN input2 i2 on i1.S_key = i2.S_key) AS F00
GROUP BY S_name, meets_at) AS F00
WHERE (n = 2.0);
```

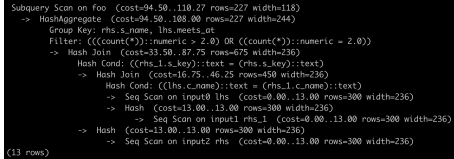


Figure 4: testcase4.generated

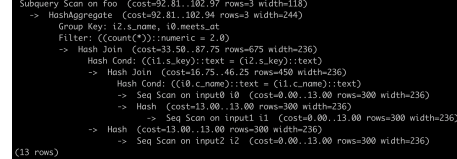


Figure 5: testcase4_optimal

For this testcase 4, the optimal solution cost slightly less than the generated query due to the condition of SELECT S_name. The optimal solution selects fewer results than the generated queries.

2.3 Some Expensive Testcases

Some testcases don't give a SQL solution so I checked their R query to see whether it is of huge cost.

2.3.1 Testcase 11

Generated R query:

```
RET_DF1618 <- left_join(input1, input0)
RET_DF1619 <- anti_join(select(input0,S_name), select(RET_DF1618, S_name))
RET_DF1620 <- RET_DF1619 %>% ungroup() %>% select(S_name) %>% distinct()
```

Optimal R query:

```
anti_join(student,enrolled) %>% select(S_name)
```

The larger cost is mainly due to the extra left_join. It is unnecessary to first left_join two tables because the next operation only select S_name from the joined table.

2.3.2 Testcase 14

Generated R query:

```
RET_DF193 <- input0 %>% group_by(S_key) %>% summarise(n = n())
RET_DF194 <- filter(RET_DF193, n == max(n))
RET_DF195 <- inner_join(inner_join(inner_join(input1, input0), input2), RET_DF194)
RET_DF196 <- RET_DF195 %>% ungroup() %>% select(S_name) %>% distinct()
```

Optimal R query

```
inner_join(parts,catalog) %>% inner_join(suppliers) %>%
group_by(sname) %>% summarise(n=n()) %>%
filter(n == max(n)) %>% select(sname)
```

Among the three tables, there is some information that relates with each other, such as S_name and S_key. The larger cost is due to the inner_join of an unnecessary table, which is produced by grouping by S_key.

The machine may use some unnecessary columns, which are related with the required column, to extract information.

From the examples above, I think the main reason for it should be that the machine tends to include more data than it needs. In this way, machine can compute the correct answer because all the data is selected but need more space and time to extract useful data.

3 My Idea on Generating Low-cost Query Efficiently

3.1 Running Cost during Enumeration

It is quite like doing a TSP problem. The idea is to brute-force enumerate all possible combination but with branch-and-bound method. For example, we can have a running cost when enumerating, when the running cost is greater than the current minimum cost, it can be ignored. But it should still consume much time. To make it more efficiently, we can have something similar to the TSP Heuristics. The heuristic is to calculate the lower bound of the cost of the unadded operations. With this lower-bound cost, the program should run faster.

3.2 Rate Operations Based on Cost

The other idea is to rate each operation by their average cost and then greedily enumerate these operations in the increasing order of their cost. For example, if it is line based SQUARE, when choosing the root operation for each line, we should first consider the operation that costs less given the inputs and the tables that previous operations returned.

3.3 Rate Operations and Arguments Based on Type and Size of Returned Tables

From the examples above, it seems a problem that the program give a solution with some unnecessary operations. To solve this, the arguments of each operations should be closer to the final output in order to avoid using some unnecessary but related data (testcase14) and the operations that returned fewer results should be preferred because it can prevent the machine giving solutions with unnecessary operations.