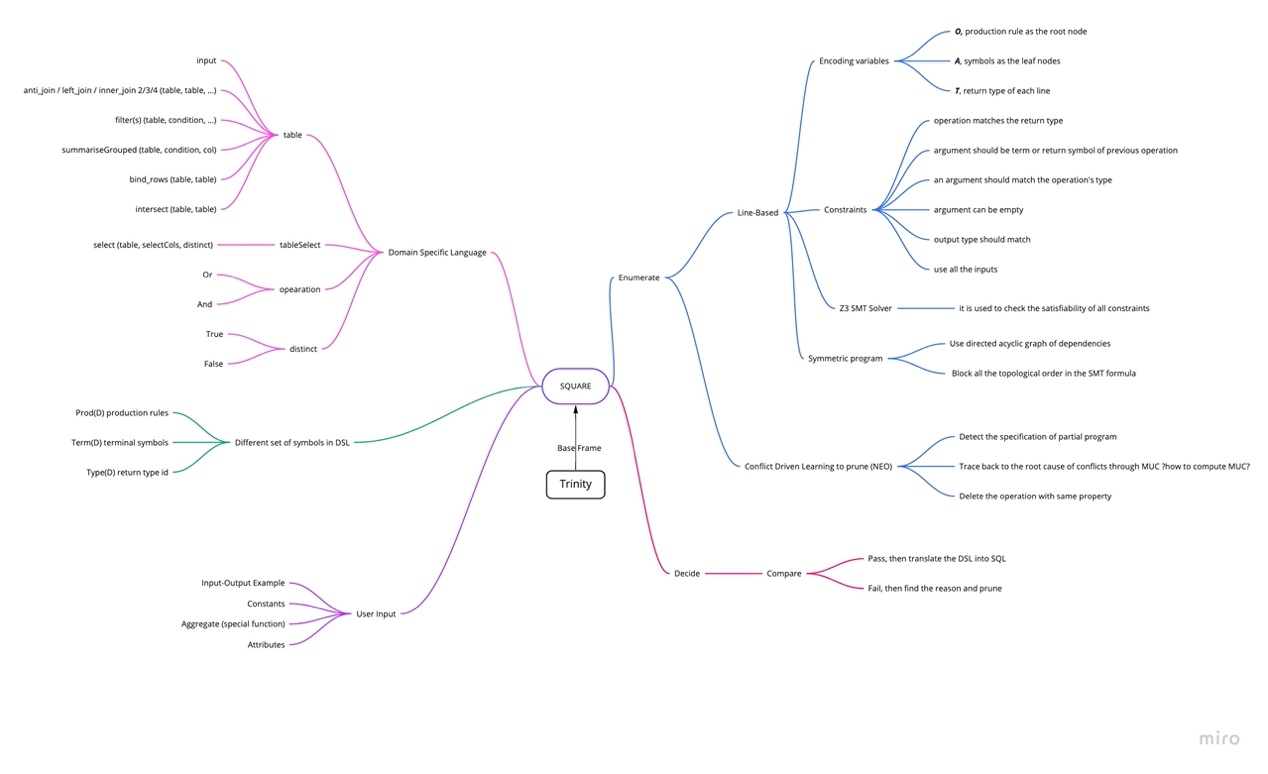
Warm-up exercise report (SQL)



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

postion in this loop

Try to play the SQUARE and evaluate the cost through PostgreSQL

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.

Testcase 1,

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

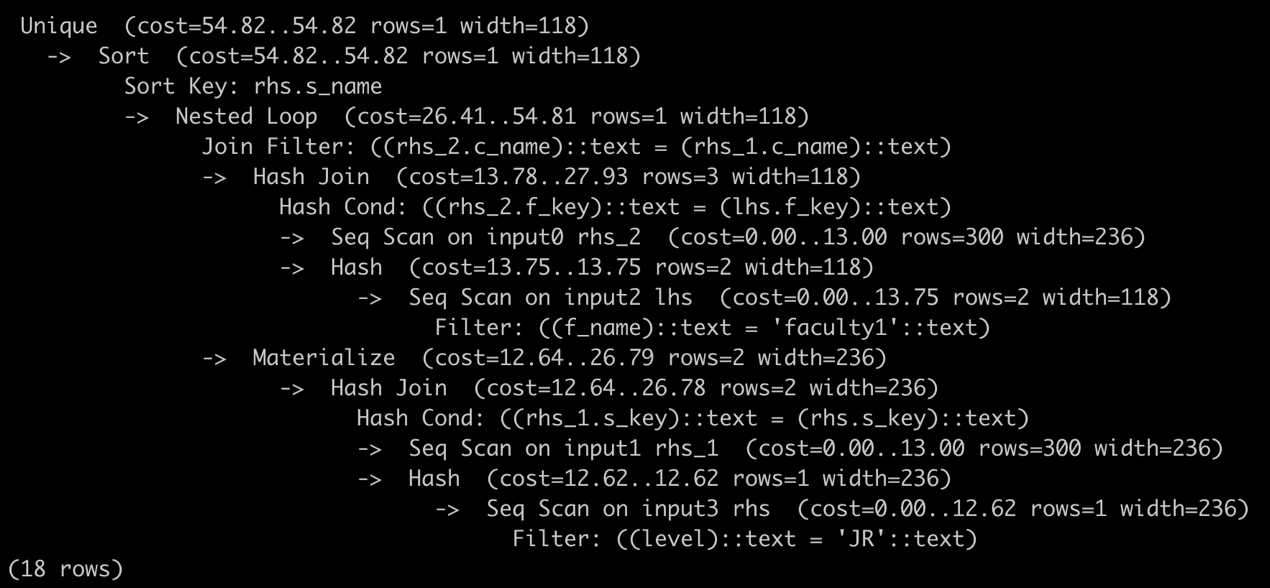
FROM input2 AS LHS

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 FOO

WHERE (level = 'JR' AND F\_name = 'faculty1’);



The optimal PostgreSQL:

SELECT DISTINCT S\_name

FROM input0 i0

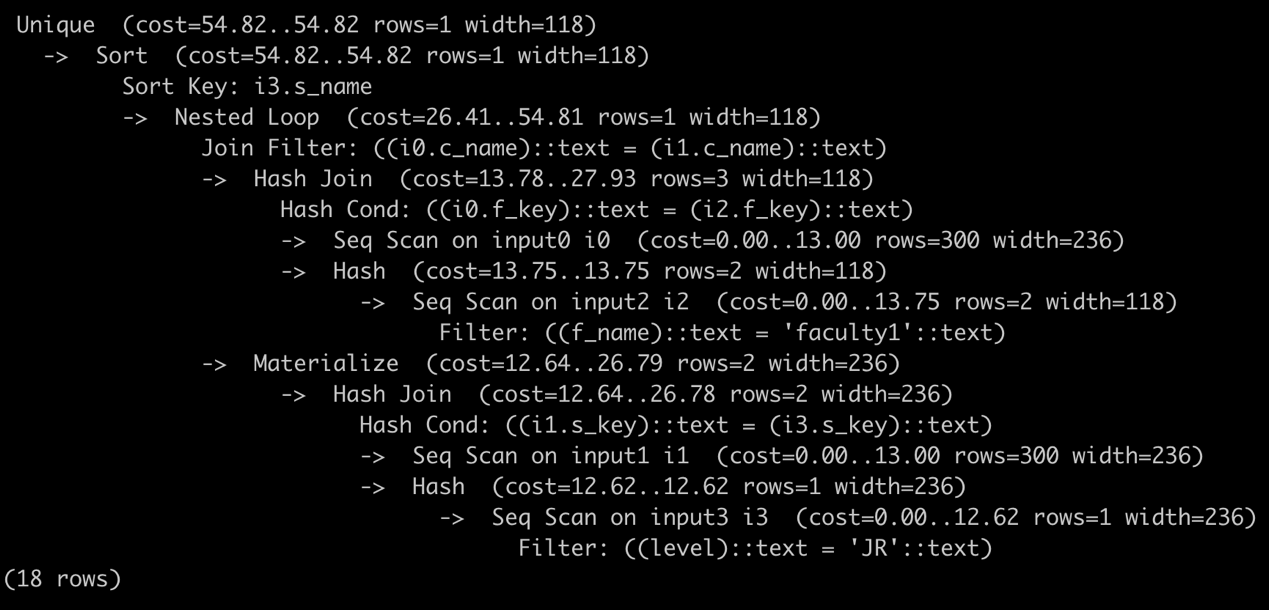
INNER JOIN input2 i2 ON i0.F\_key = i2.F\_key

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');

Since the queries are generally the same, the resulting costs are the same.



Testcase4:

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)

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

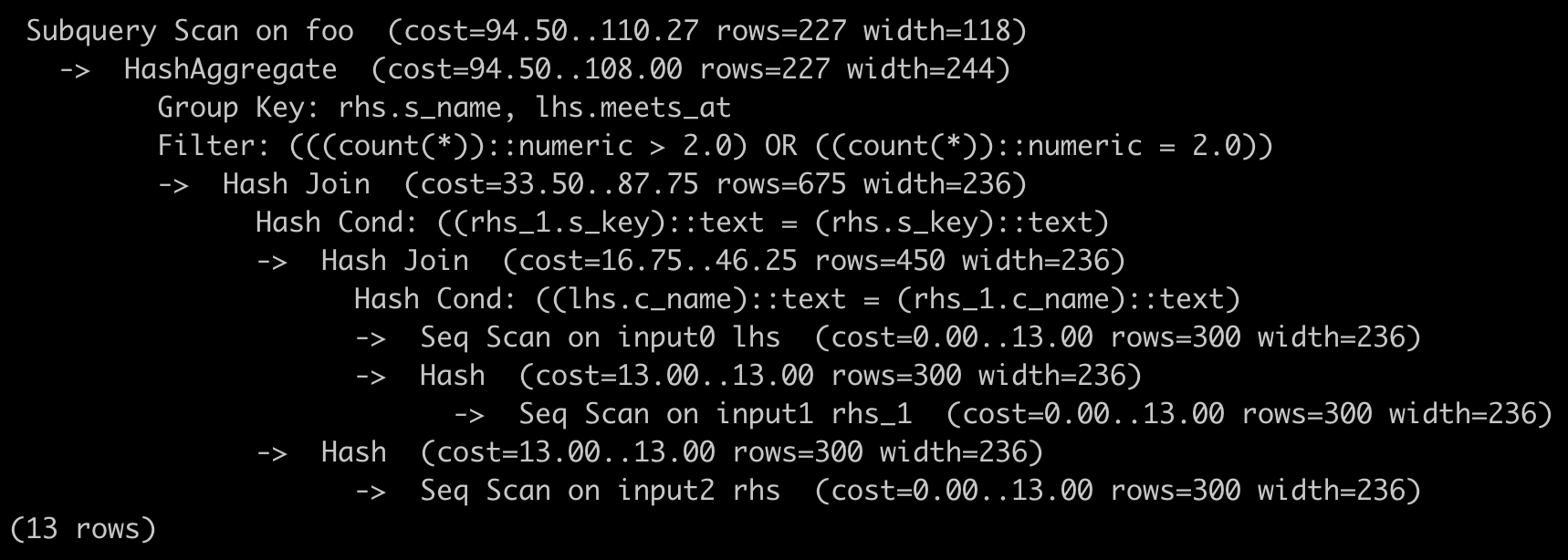
FROM input0 AS LHS

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);



Optimal PostgreSQL:

SELECT S\_name

FROM

(SELECT S\_name, meets\_at, COUNT(\*) AS n

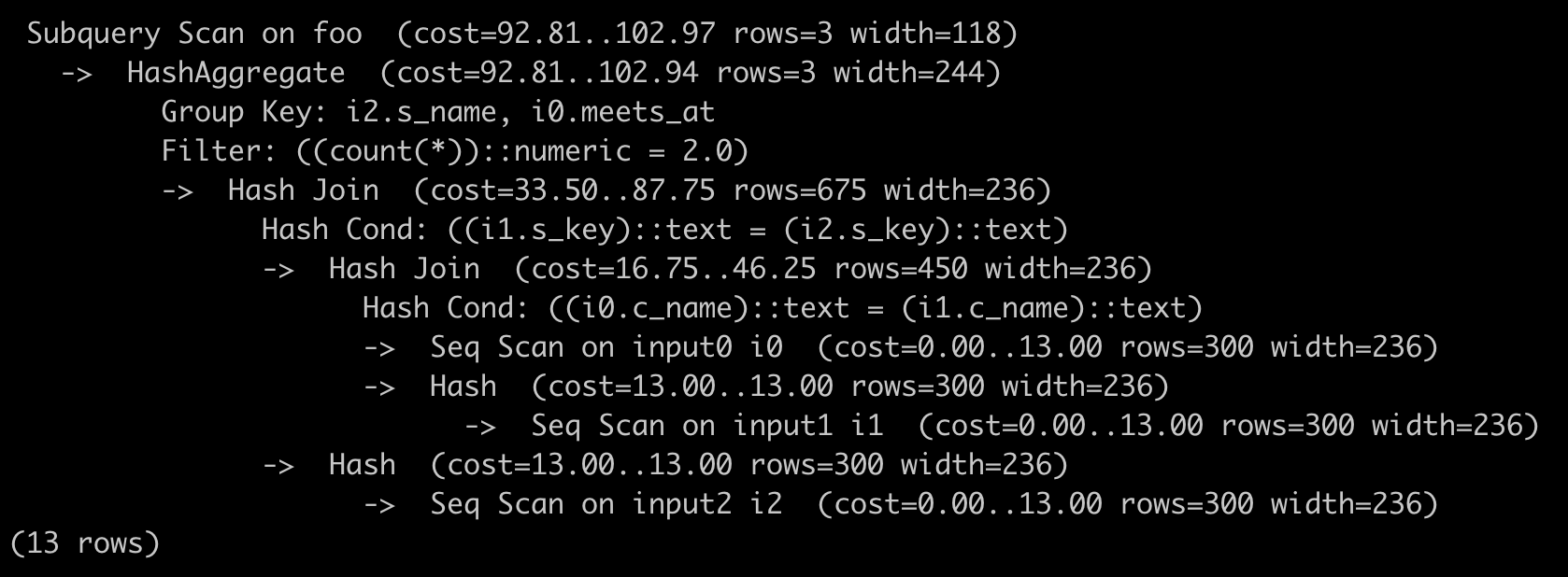
FROM (input0 i0

INNER JOIN input1 i1 on i0.C\_name = i1.C\_name

INNER JOIN input2 i2 on i1.S\_key = i2.S\_key) AS FOO

GROUP BY S\_name, meets\_at) AS FOO

WHERE (n = 2.0);



Although some testcases don’t give a SQL solution, I check their R query to see whether it is of huge cost.

For example, testcase4

Generated:

RET\_DF4837 <- inner\_join(inner\_join(input0, input1), input2)

RET\_DF4838 <- RET\_DF4837 %>% group\_by(S\_name, meets\_at) %>% summarise(n = n())

RET\_DF4839 <- RET\_DF4838 %>% ungroup() %>% filter(n > 2 | n == 2)

RET\_DF4840 <- RET\_DF4839 %>% ungroup() %>% select(S\_name)

Optimal:

inner\_join(class,enroll) %>% inner\_join(student) %>%

group\_by(S\_name,meets\_at) %>% summarize(n = n()) %>%

filter(n == 2) %>% select(S\_name)

The cost is a little larger than the optimal. Mainly because of the filter having n > 2 besides n == 2. As a result, the returned data is more than necessary and the machine need more memory to store and more time to search.

Testcase11:

Generated:

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:

anti\_join(student,enrolled) %>% select(S\_name)

The larger cost is due to the extra left\_join.

Testcase14:

Generated:

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:

inner\_join(parts,catalog) %>% inner\_join(suppliers) %>%

group\_by(sname) %>% summarise(n=n()) %>%

filter(n == max(n)) %>% select(sname)

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.

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.

My idea of generating low-cost query efficiently:

1. 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.
2. 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.