Command-line Parser

The Goal:

to take a sql statement and generate a parse tree

input: sql statement

```
select lname, fname from authors
where lname = "Garcia Marquez"
      and
      Country = "Colombia"
```

output: Parse Tree

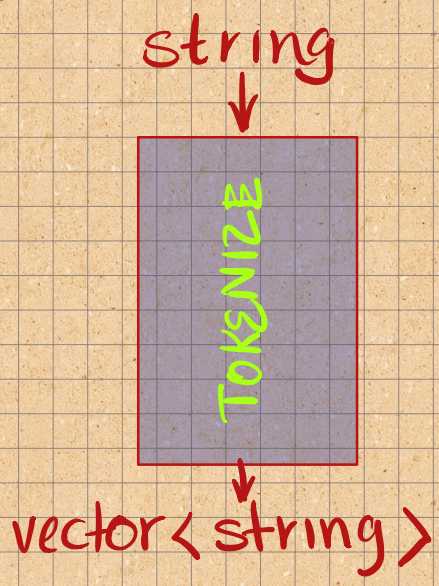
Command	select
table	authors
fields	lname fname
where	yes
Condition	lname = "Garcia Marquez" and Country = "Colombia"

the input is a string object or a cstring

the output is a multimap of string and string

Tokenize

we will use our string tokenizer to convert the input to a queue of strings.

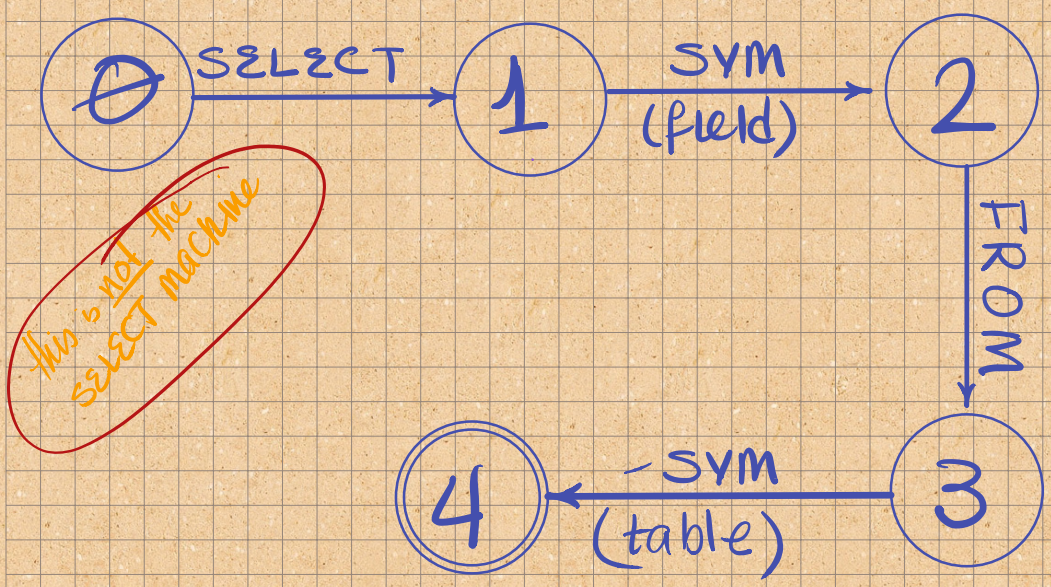


To Do

1. start with a state machine that represents all the legal commands we will handle:
   - make / create table
   - insert into
   - select
     - fields / *
     - with or without conditions

consider the simplest select command

select lname from authors



2. build an adjacency matrix:

state#s not stored

state	success	SELECT	FROM	STAR		SYM
∅	∅	1	-1			-1
1	∅	-1	-1	??		2
2	∅	-1	3			-1
3	∅	-1	-1			4
4	1					

3. mapping keywords

string	cnts
"select"	SELECT
"from"	FROM

- A map is used to map keyword strings to their corresponding column numbers in the adjacency matrix
  - example:

    "select" → 1 OR enum SELECT

    "from" → 2 OR enum FROM

    etc

- any string not in this map will be designated as SYM

select lname, fname from authors

- alternative (temporary) solution: write a function that will take the keyword string and return int column number (using if/else)

No!

input: select lname, fname from authors where lname = "Garcia Marquez" and country = colombia

Flow

input_q:

'select'
'lname'
';'
'fname'
'from'
'authors'
'where'
...

tokenize

step1

string

Keywords

String	enum
"Select"	SELECT
"From"	FROM
"*"	STAR
Else	SYM

step2

if the map does not contains() the token, then the token is a symbol

int

state →

state	success	SELECT	FROM	STAR		SYM
∅	∅	1	-1			-1
1	∅	-1	-1	??		2
2	∅	-1	3			-1
3	∅	-1	-1			4
4	1					

this is NOT the complete table

update state

step3

step4

keys

value list

Parse Tree

Command	Select
table	authors
fields	lname fname
where	yes
Condition	lname = "Garcia Marquez" and Country = "Colombia"

update parse tree

4. Build the parse tree: get_parse_tree

```
for each element in the input queue:
    and as long as we have not reached
    a fail state:
        token = next item in queue
        state = next_state:
                table[state][column_no]

        switch (state)
            1:  // token that brought you here
                // was "select"
                ptree["command"] += token

            2:  // your token is the field
                // name
                ptree["fields"] += token

            3 : // from : we dont care about
                        this token

            4 : // your token was the
                    table name
                ptree["table"] += token

if state is a success state,
                        return true
else reset ptree
     return false
```

The parser class

member variables:

ptree // multimap<string, string>
keywords map
adjacency table
fail boolean flag

inputq : holds tokens
enum keys: {ZERO, SELECT, FROM, WHERE, ... }
these are the adjacency matrix column numbers

input:

select lname from authors

input_q:

select
lname
from
authors

member functions

- CTOR
  - default:
  - one argument (char* s)
    - copy str to internal buffer
    - initialize "make" adjacency table
    - build keyword list
    - set_string
- set_string
  - tokenize input buffer into input_q
  - calls get_parse_tree
  - sets/resets fail flag

output: Parse Tree

Command	Select
table	authors
fields	lname fname
where	yes
condition	lname = "Garcia Marquez" and Country = "Colombia"

- get_column
  - maps tokens to keyword constants like SELECT, FROM, or SYMBOL
- get_parse_tree
  - sets ptree
  - returns true if ends at a success state false otherwise

- build_keyword_list()
  - builds the keyword map used to map keyword strings to adjacency matrix column numbers
- init & make table functions
- Accessors & Mutators:
  - bool fail()
  - mmap parse_tree()

bare minimum main:

```
⋮
parser p("select last from student");

multimap<string,string> ptree = p.ptree()
if ptree["command"] == "select"

    string table_name = ptree["table"]
    table t(table_name)
    cout << t.select(ptree["fields"]

p.set_string("select fname from student")
```

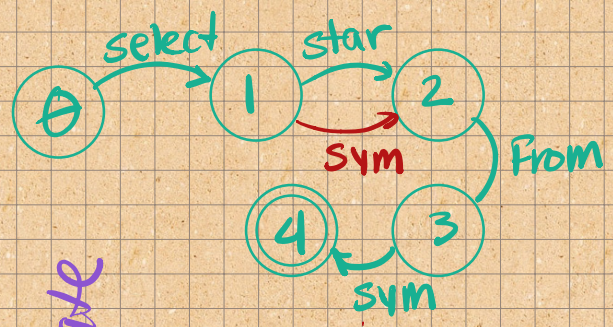
Work flow:

"select * from emp ← cstring

select	*	from	emp

← tokenized array/vector of strings

select last from emp



KeyWord	Column
"select"	SELECT
"*"	STAR
"from"	FROM
'	
if ! Contains, sym	

state

	success	select	star	from		sym
0	0	1	-1	-1	-1	-1
1	0	-1	2	-1	-1	2
2	0	-1	-1	3	-1	-1
3	0	-1	-1	-1	-1	4
4	1	-1	-1	-1	-1	-1

void get_parse_tree(input Q)

pop the Q
find column for this token
find your next state

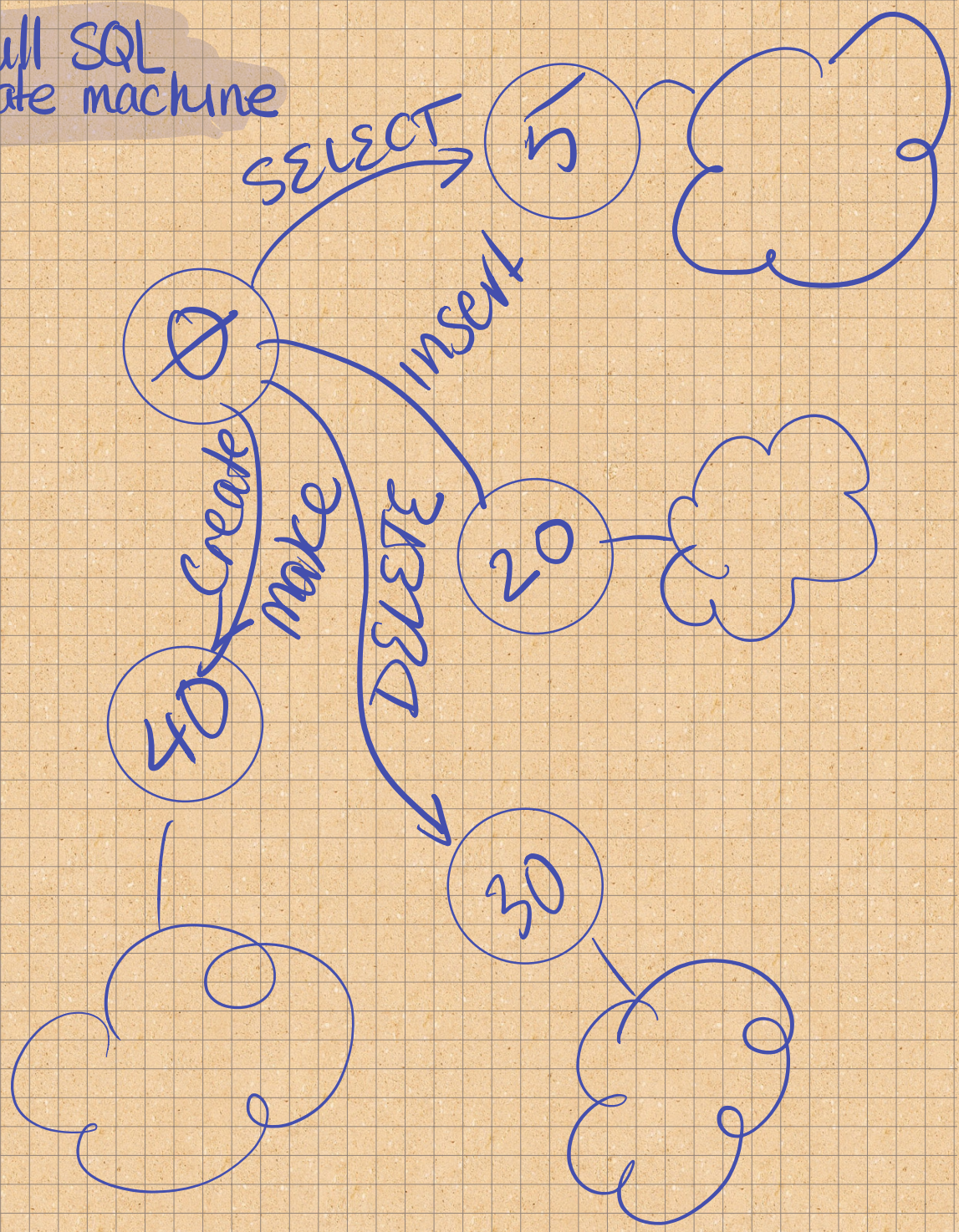
switch

store appropriate values in ptree based on new state (& the token who brought you here.

Parse Tree:

command	"SELECT"
fields	"*"
table_name	"EMP"

Full SQL state machine



Example SQL Statements

select

filter columns

select last, first from students

select last from student

select * from student

~~select *, first, last~~

select * from student where

- last = Jones
- last < Collins
- last <= Collins

filter rows

- last = "Arana" or first="Alexis"

  every Arana & every Alexis

- last = "Arana" and first="Alexis"
- last = Arana and (first = Alexis or first = Canara)

select [field list]
from [table name]
where [condition]

insert

insert into student values
Bud, Caldwell, CS, 128000, CA

field values are in the same order as field names were presented when the table was created

```
insert into [table_name]
values [value_list]
```

more examples:

"select * from emp;
everything from lable"
emp:
all rows & all columns

select last, first, age from emp;
filter columns.
all the rows

select last, first from emp
where last = "Malone"
filters both rows & columns:
only last & first cols
only rows matching last
name Malone

select * from emp where
last = Malone and
first = Deza or
first = "Mary ann"

all columns
rows wher name is Deza Malone
or first name is "Mary ann".
precedence of operators

- insert into emp -elds
  Bud, caldwell, CS, MI

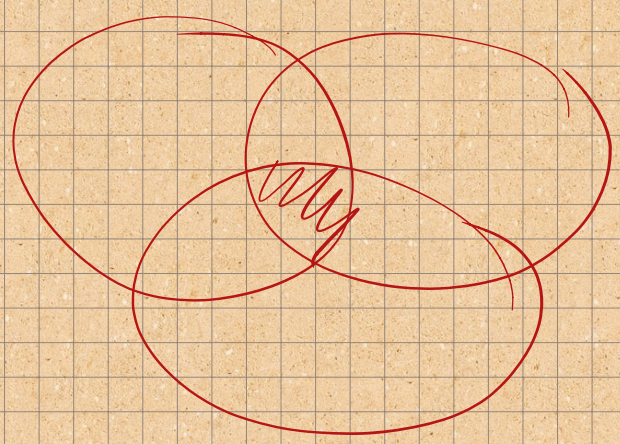
create table emp fields
first, last, major, state

Table(name, vector of field names)

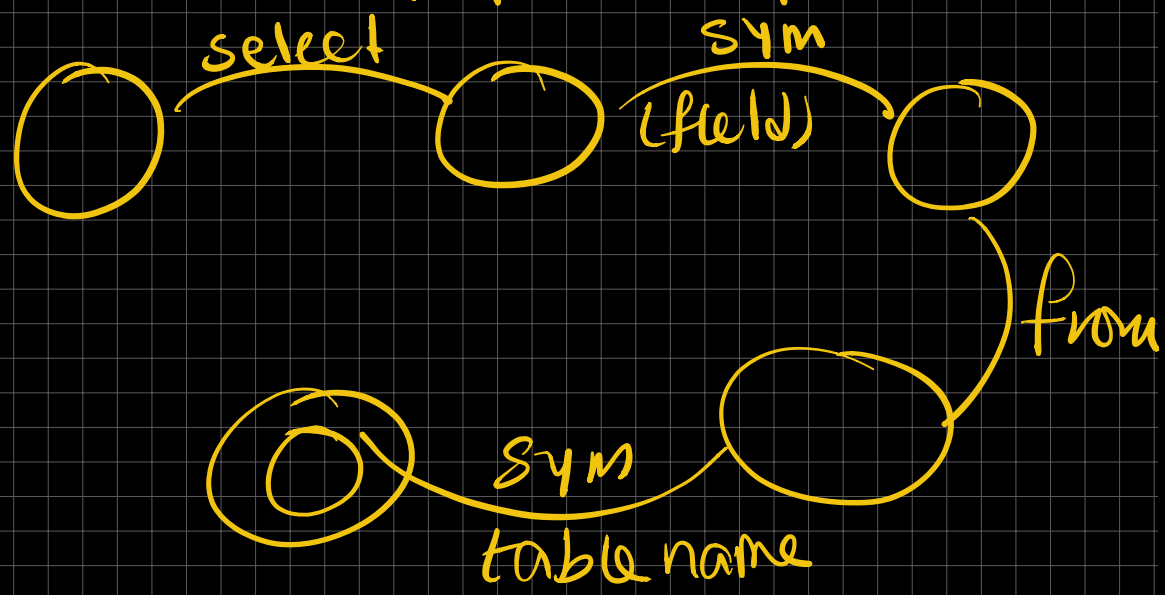
- Make Table

- Drop table emp

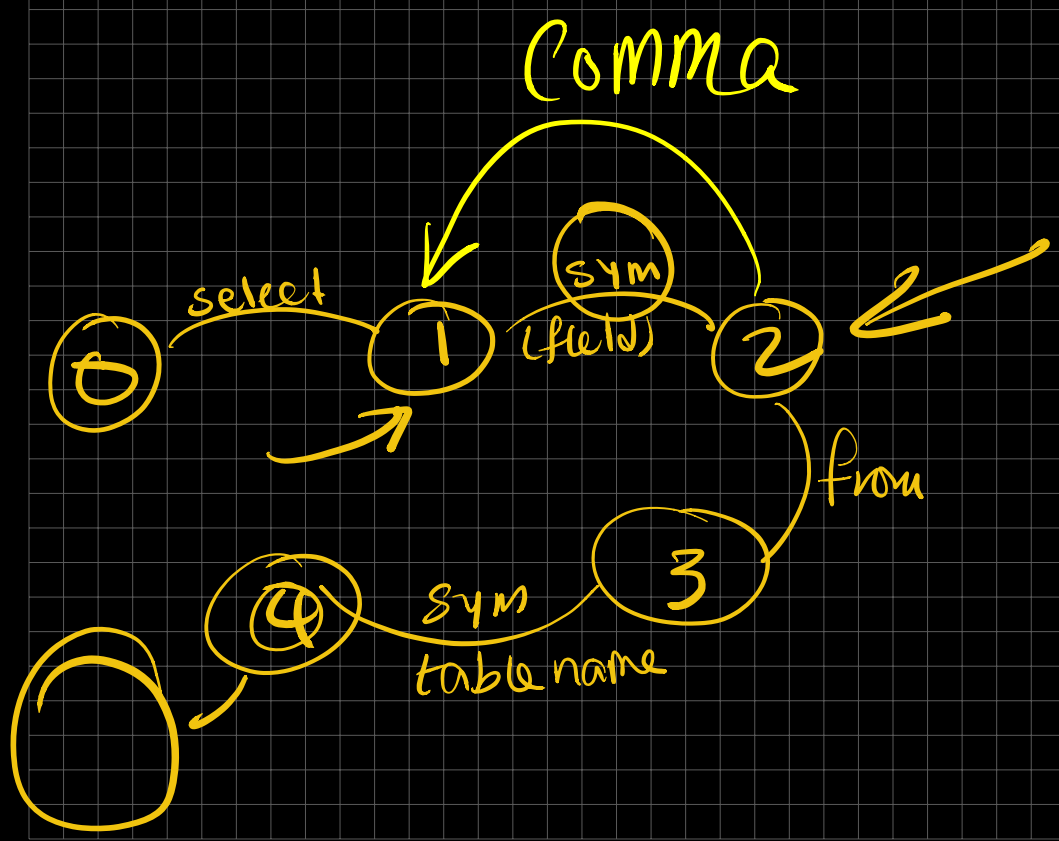
1. command line
2. SQL class
   manager
3. complex conditions
   shunting yard
   RPN
   { union
   { Intersection



select last from emp



select last, first from emp



mark_cells (2, COMMA, 1)

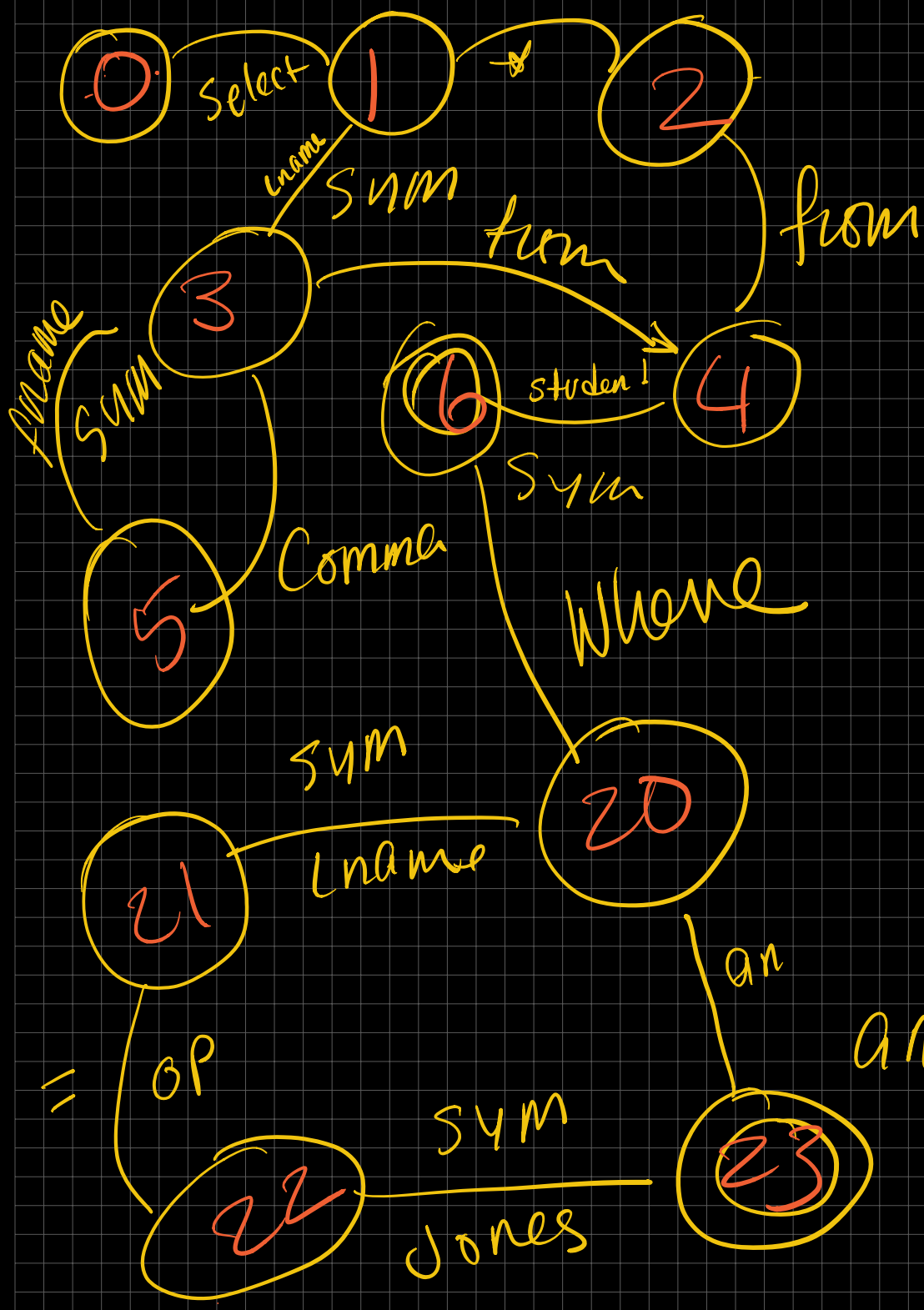
table,

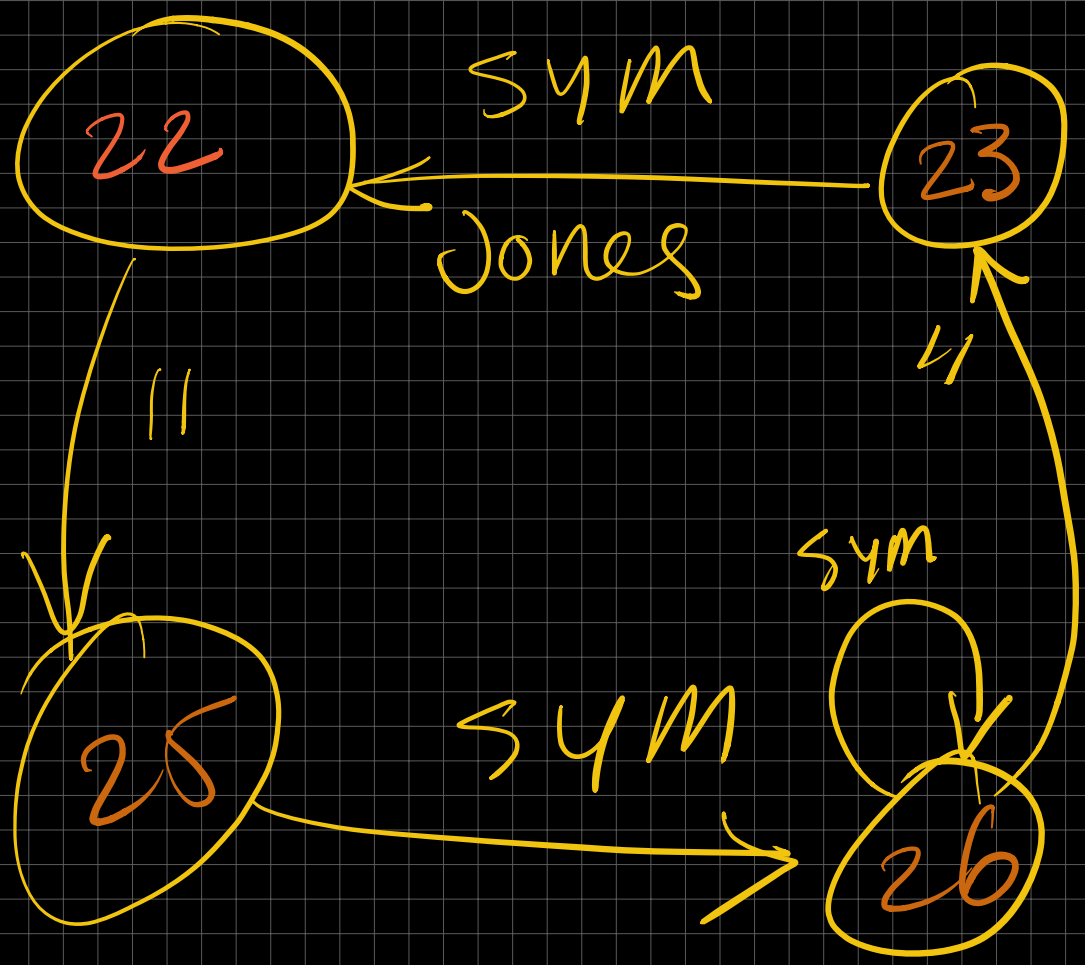
Case 2:

Ptree["fields"] += t

last	first	age

Select * from student

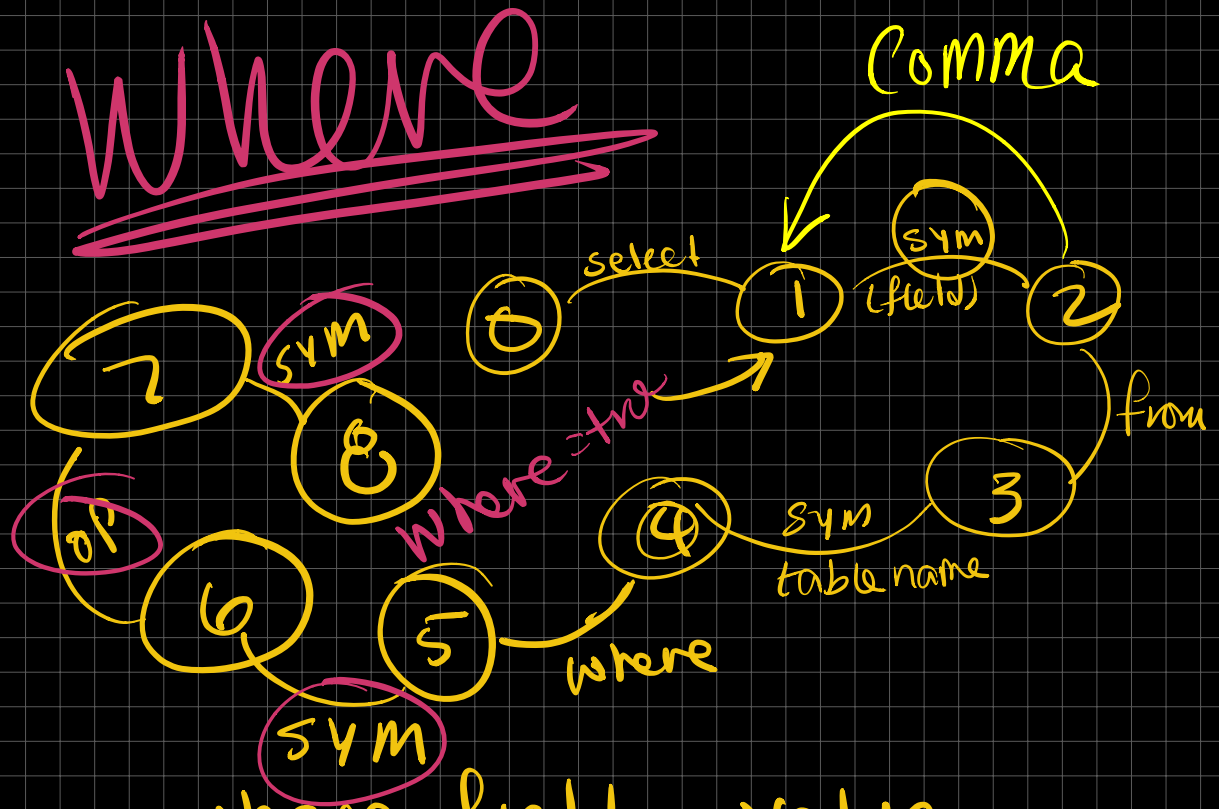




insert into employee
values smith, "maryann", 23, CS

last = "De La Torre" "

where



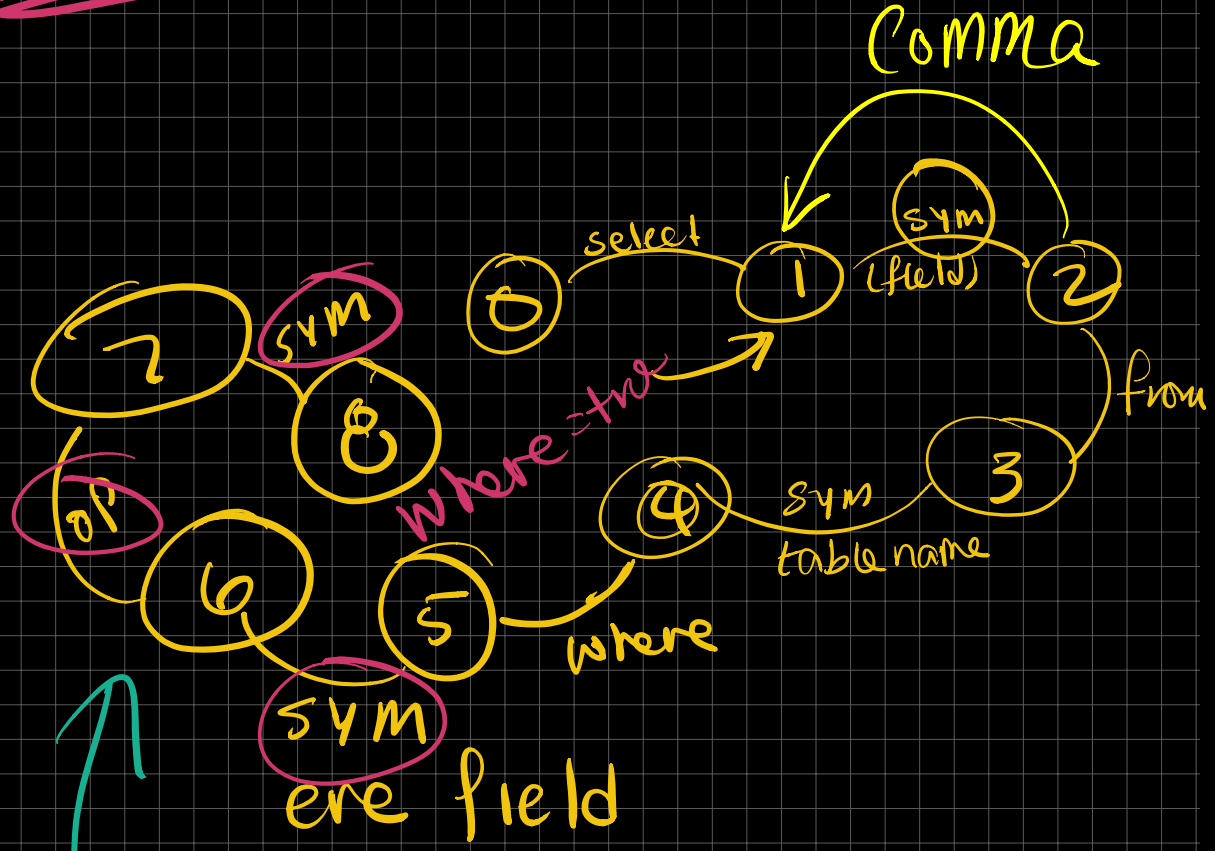
... where field = value

ptree["cond"] t = t

"cond"			
	Last	=	Harris

mpair

where*



error checking for invalid sql statements

sym

5

4

where

last = Harris and
first = Richard