Grammar For ASM

Operations

There are 2 kinds of basic operation which is supported by the ASM:

- **Unitary Operator**
  - count, sum, min, max, average, distinct, not
- **Binary Operator**
  - <, >, <=, >=, =, ==, neq, +, -, *, /, and, or, in, cross, search

Noticed that "=" here is the **assignment** operation, which would be introduced in the detail part.

Besides, the ASM also support two special operations commonly used in database: **Selection & Projection**

- For **Selection** you can use

$$(\text{Query})\ \text{where}\ (\text{Query})$$

  to achieve

- For **Projection**, you can use

$$(\text{Query}).\text{attribute}$$

  to achieve

Noticed that the Query here can be arbitrary list, which means the query like

$$((\text{MOVIE cross ROLE})\ \text{where}\ (\text{self.M\_title} = \text{self.ROLE\_title})).\text{M\_year}$$

are supported by our ASM. (Here is another essential operation in database: **equijoin**)

Therefore, **our ASM is highly flexible.**

Details

Here are some detail about several special operation

Selection

For single list to be selected, you could neglect the bracket for the list name, but you should always use bracket for the query after "where"

e.g.

$$\begin{aligned}
&\text{MOVIE where (Query)} \quad --- \quad \checkmark \\
&\text{MOVIE where Query} \quad --- \quad \times \\
&\text{MOVIE corss ROLE where (Query)} \quad --- \quad \times
\end{aligned}$$

When you want to use some attribute of the data which are selected, you could use $self.attribute$ to access the attribute of the data. The action scope of $self$ is in the (Query) except the selection query nested in (Query)

e.g.

$$\text{MOVIE where } ((\text{self.M\_title} = \text{StarWars}) \text{ and } (\text{self.M\_year} = 1977))$$

$$\text{PERSON where } (\text{self.PERSON\_id in } ((\text{ROLE where self.ROLE\_title} = \text{Matrix}).\text{ROLE\_id})$$

the previous $self$ represent data in list PERSON, and the latter one represent data in list $ROLE$

cross

cross here represent Cartesian product of lists, i.e. merge two list

e.g.

M_title	M_year	runtime
Matrix	1999	136
StarWar	1977	121

$$\text{cross}$$

ROLE_id	ROLE_title	ROLE_year
1	Matrix	1999
2	3Body	2999

$$=$$

M_title	M_year	runtime	ROLE_id	ROLE_title	ROLE_year
Matrix	1999	136	1	Matrix	1999
Matrix	1999	136	2	3Body	2999
StarWar	1977	121	1	Matrix	1999
StarWar	1977	121	2	3Body	2999

then you could use **selection** to achieve **equijoin**, which is **essential** in database query.

$$(\text{MOVIE cross ROLE}) \text{ where } ((\text{self.M\_title} = \text{self.ROLE\_title}) \text{ and } (\text{self.year} = \text{self}))$$

the result would be

M_title	M_year	runtime	ROLE_id	ROLE_title	ROLE_year
Matrix	1999	136	1	Matrix	1999

Notice that the attribute name of two lists should **be different**.

Projection

there are two kinds of projection, all achieved by "." operator

projection of one entry, i.e. self, is just one attribute data of self

projection of a list would be a list of attribute data of the projected data.

e.g.

MOVIE =

M_title	M_year	runtime
Matrix	1999	136
StarWar	1977	121

$$\text{MOVIE.M\_year} \Rightarrow [1999, 1977]$$
$$\text{MOVIE where (self.M\_year} = 1977) \Rightarrow$$
$$\text{the self.M\_year would be 1999 or 1977 in one loop}$$

You'd better add bracket for the projection of nested list if you what to process the projected data afterwards, because there is no priority of operations.

e.g.

$$(\text{MOVIE where (runtime} > 100)).\text{M\_year} \quad --- \quad \checkmark$$
$$1999 \text{ in } (\text{MOVIE where (runtime} > 100)).\text{M\_year} \quad --- \quad \times$$
$$1999 \text{ in } ((\text{MOVIE where (runtime} > 100)).\text{M\_year}) \quad --- \quad \checkmark$$

Search

the search here would access index structure in database in order to get data faster then traverse.

It is a binary operation, and its grammar is

$$\text{@A search } \{\text{data:1,year:10}\}$$

the dictionary here is the attribute and the corresponding value of the required data

it is equivalent to selection:

$$\text{A where } ((\text{self.data} = 1) \text{ and } (\text{self.year} = 10))$$

but maybe faster

a small example:

$$(\text{A x B}) \text{ where } (\text{self.data in } ((\text{@A search } \{\text{data:1,year:10}\}).\text{data}))$$

Noticed that because A is one of the table keyword, if we do not add a "@" before it to make it a constant string, it would output a lists of A, which would make search failed.

Assignment

we can store the results of queries using variables, and access variables in the subsequence query.

The grammar is simple: $A = Query$

where A is the name of the variable, but A cannot be a keyword (name of a list/operations)

Then we can use A in the subsequent query.

Notice that variables can only used to store tables!

An example:

$$\begin{gathered}\mathrm{B} = (\mathrm{RESTRICTION\ where\ (self.RESTRICTION\_description} == \mathrm{R18}))\\ \mathrm{C} = (\mathrm{B\ cross\ ROLE})\\ \mathrm{PERSON\ where\ (not\ (self.PERSON\_id\ in\ (A.ROLE\_id)))}\end{gathered}$$

Attention

- The attribute name of two lists which would be crossed should **be different**
- There is **no priority** for operations, so you'd better **add bracket for every operand.**
- if you want a constant string, please **add @ before string**,
  - example1: @18 instead of 18, because 18 would be convert to float.
  - example2: @RESTRICTION instead of RESTRICTION, because the latter one would be a list of entries and could not be used for searching using index structure.