What's RADB and how does it work?

- A simple **Relational Algebra (RA) interpreter** written in Python 3
- It implements RA queries by **translating them into SQL** and executing them on the underlying database system through SQLAlchemy.
- RADB is packaged with SQLite, so you can use RADB as a standalone RA database system. Alternatively, you can use RADB as an RA front-end to connect to other database servers from various vendors.
Basic Usage
RADB Language Usage -- Selection

Selection: `\select_{\text{condition}} \ input\_relation`

For example, to select Drinker tuples with name Amy or Ben, we can write:

```
\select_{\text{name}='Amy' \ or \ name='Ben'} \ Drinker;
```

String literals should be enclosed in single quotes. Comparison operators $\leq$, $\lt$, $\geq$, $\gt$, and $\neq$ (inequality) work as expected on strings, numbers, and dates. For string match you can use the `like` operator; e.g.:

```
\select_{\text{name \ like \ 'A%'}} \ Drinker;
```

finds all drinkers whose name start with “$A$”, where `%` is a wildcard character that matches any number of characters. Finally, you can use boolean connectives `and`, `or`, and `not` to construct more complex conditions. More features are available; see Data Types and Operators for details.
RADB Language Usage -- Projection

**Projection**: `\project_{attr_list} input_relation`

Here, `attr_list` is a comma-separated list of expressions that specifies the output attributes. For example, to find out what beers are served by Talk of the Town (but without the price information), you can write:

```sql
\project_{bar, beer} \select_{bar='Talk of the Town'} Serves;
```

You can also use an expression to compute the value of an output attribute; e.g.:

```sql
\project_{bar, 'Special Edition' || beer, price+1} Serves;
```

Note that `||` concatenates two strings.
RA DB Language Usage -- Theta-Join

**Theta-Join:** `input_relation_1 \join_{\text{cond}} input_relation_2`

For example, to join `Drinker(name, address)` and `Frequents(drinker, bar, times_a_week)` relations together using drinker name, you can write:

```
Drinker \join_{\text{name=drinker}} \text{Frequents};
```

Syntax for `\text{cond}` is similar to the case of `\text{select}`.

You can prefix references to attributes with names of the relations that they belong to, which is sometimes useful to avoid confusion (see Relation Schema and Attribute References for more details):

```
Drinker \join_{\text{Drinker.name=Frequents.drinker}} \text{Frequents};
```
Natural join: \texttt{input\_relation\_1 \join input\_relation\_2}

For example, to join \texttt{Drinker(name, address)} and \texttt{Frequents(drinker, bar, times\_a\_week)} relations together using drinker name, we can write \texttt{Drinker \join \rename\_\{name, bar, times\_a\_week\} Frequents};. Natural join will automatically equate all pairs of identically named attributes from its inputs (in this case, name), and output only one attribute per pair. Here we use \texttt{\rename} to create two name attributes for the natural join; see notes on \texttt{\rename} below for more details.

\begin{itemize}
\item \texttt{drinker(name, address)}
\item \texttt{bar(name, address)}
\item \texttt{beer(name, brewer)}
\item \texttt{frequents(drinker, bar, times\_a\_week)}
\item \texttt{likes(drinker, beer)}
\item \texttt{serves(bar, beer, price)}
\end{itemize}
**RADB Language Usage -- Cross Product**

**Cross product:** `input_relation_1 \cross input_relation_2`

For example, to compute the cross product of *Drinker* and *Frequents*, you can write:

```
Drinker \cross Frequents;
```

In fact, the following two queries are equivalent:

```
\select_{\{Drinker.name=Frequents.drinker\}}
(Drinker \cross Frequents);
```

```
Drinker \join_{\{Drinker.name=Frequents.drinker\}} Frequents;
```
Set union, difference, and intersection:

\[ \text{input}_1 \cup \text{input}_2 \]

\[ \text{input}_1 \setminus \text{input}_2 \]

\[ \text{input}_1 \cap \text{input}_2 \]

For a trivial example, the set union, difference, and intersection between Drinker and itself, should return the contents of Drinker itself, an empty relation, and again the contents of Drinker itself, respectively.

TABLE SCHEMAS

- drinker(name, address)
- bar(name, address)
- beer(name, brewer)
- frequents(drinker, bar, times_a_week)
- likes(drinker, beer)
- serves(bar, beer, price)
RADB Language Usage -- Rename

**Rename:**

\rename_{new\_attr\_names} input\_relation

This form of the rename operator renames the attributes of its input relation to those in new\_attr\_names, a comma-separated list of names.

\rename_{new\_rel\_name: \ *} input\_relation

This form of the rename operator gives a new relation name to its input relation (the attribute names remain the same). For example:

\begin{verbatim}
rename {s1:*} Serves
   join {s1.beer=s2.beer and s1.price=s2.price}
rename {s2:*} Serves;
\end{verbatim}

\rename_{new\_rel\_name: new\_attr\_names} input\_relation

This form of the rename operator allows you to rename both the input relation as well as its attributes.

**TABLE SCHEMAS**

- **drinker**(name, address)
- **bar**(name, address)
- **beer**(name, brewer)
- **frequents**(drinker, bar, times_a_week)
- **likes**(drinker, beer)
- **serves**(bar, beer, price)

\rename_{drinker, address} drinker
\rename_{d1: drinker, address} drinker
More useful tips
Nested Queries

- Build a complex query by nesting: you can feed a subquery as an input relation to another relational operator (using parentheses to enclose the subquery as necessary to avoid ambiguity), e.g.: \( \text{select}_{\{\text{condition}\}} (\text{project}_{\{\text{attr\_list}\}} \text{input\_relation\_1}) \text{\ join} \text{input\_relation\_2} ; \)
Views

RA lets you define “views,” which may be thought of as temporary, relation-valued variables holding the results of relational algebra expressions. To define a view, use the syntax:

```
view_name : - view_definition_query;
```
RADB Language Documentation

To find more details about this language, and how to use radb, please find this link:

https://users.cs.duke.edu/~junyang/radb/

(RADB is an in-house Duke product developed by Prof. Jun Yang!)