RA: A Relational Algebra Interpreter

Introduction

RA is a simple relational algebra interpreter written in Java. It is built on top of an SQL-based relational database system. It implements relational algebra queries by translating them into SQL queries and executing them on the underlying database system through JDBC. RA is packaged with SQLiteJDBC, so you can use RA as a standalone relational-algebra database system. Alternatively, you can use RA as a relational-algebra frontend to other database systems.

Currently Supported Database Systems

- SQLite (SQLite JDBC v056): JDBC driver is included in the JAR.
- PostgreSQL 8.4: JDBC driver is included in the JAR.
- MySQL 5: JDBC driver is included in the JAR.
- IBM DB2 9.7: JDBC driver is included in the JAR.

Getting RA

The current version of RA is 2.2b (released Aug. 15, 2014). You can download one of the following:

- A .tar.gz file with everything including source code.
- A .zip file with everything including source code.
- Just the .jar file. But we also recommend you download sample.db (sample SQLite database), sample.properties (RA connection properties file), and sample.ra (RA script for creating/restoring the sample database), and put them in the same directory as the .jar file.

Or follow the project on GitHub!

Using RA

To run RA, just type "`java -jar ra.jar`" in the directory containing `ra.jar`. RA will then by default run on the SQLite database file `sample.db` (on which query examples below are based). To see more options of running RA, type "`java -jar ra.jar -h`". Options `-i` (read commands/queries from a file), `-o` (log output to a file), and `-v` (turn on verbose mode) are all pretty useful.

If you want to connect to another database or a different database system, you will find it convenient to write a connection properties file of your own. An example is the file `sample.properties`. Once you have rolled your own properties file, say *`PROP_FILE`*, run RA using "`java -jar ra.jar PROP_FILE`"; add `-P` if you prefer typing your password at runtime instead of storing it in the properties file.

Once you are in RA, you will see the `ra>` prompt. For help, type `\help;`. You exit RA by issuing the `\quit;` command. Use the `\list;` command to see what relations are available for query in your database.

- *ADVANCED USERS:* Using `\sqlexec{`*`statement`*`};`, you can send an SQL command to the underlying database. This command allows you to manipulate the database in ways (e.g., updating it) that you cannot with just relational algebra. For example, the SQLite database file `sample.db` was in fact created by running "`java -jar ra.jar sample.properties -i sample.ra`", where `sample.ra` makes heavy use of `\sqlexec`.

RA supports command-line input history and editing using arrow keys: Up/Down recall previous/next lines, and Left/Right move within the current line.

The simplest relational query you can write is one that returns the content of a relation: Just type *`relName`*`;`, where *`relName`* is the name of the relation. Note that every command/operator should start with a backslash (`\`), and every query/command should be terminated by a semicolon (`;`).

For most database systems that RA runs on (except MySQL), relation and attribute names are case-insensitive per SQL standard. For example, `drink` is just as good as `DRINK`. Attributes can be of a variety of types. Details are not important; just beware that types such as `INTEGER`, `SMALLINT`, `FLOAT`, `REAL`, `DOUBLE`, `DECIMAL`, `NUMERIC` are for numbers, and `CHAR` and `VARCHAR` are for strings.

Here is an example of a complex query, which returns beers liked by those drinkers who do not frequent James Joyce Pub:

```
\project_{beer} (

  ((\project_{name}          // all drinkers
     Drinker)
   \diff
   (\rename_{name}           // rename so we can diff
      \project_{drinker}     // drinkers who frequent JJP
        \select_{bar = 'James Joyce Pub'}
          Frequents))

  \join_{drinker = name}     /* join with Likes to find beers */

  Likes

);
```

The syntax is insensitive to white space, and it is fine to enter a query on multiple lines; RA will number the lines (beyond the first one) you enter for the current query. C/C++/Java-style comments (`//` and `/*...*/`) are supported.

RA supports the following relational algebra operators:

- `\select_{`*`cond`*`}` is the relational selection operator. For example, to select `Drinker` tuples with name Amy or Ben, we can write `\select_{name = 'Amy' or name = 'Ben'} Drinker;`. Syntax for *cond* follows SQL. Note that string literals should be enclosed in ***single*** quotes, and you may use boolean operators `and`, `or`, and `not`. Comparison operators `<=`, `<`, `=`, `>`, `>=`, and `<>` work on both string and numeric types. For string match you can use the SQL `LIKE` operator; e.g., `\select_{name like 'A%'} drinker;` finds all drinkers whose name start with A, as `%` is a wildcard character that matches any number of characters.
- `\project_{`*`attr_list`*`}` is the relational projection operator, where *`attr_list`* is a comma-separated list of attribute names. For example, to find out what beers are served by Talk of the Town (but without the price information), we can write `\project_{bar, beer} (\select_{bar = 'Talk of the Town'} Serves);`.
- `\join_{`*`cond`*`}` is the relational theta-join operator. For example, to join `Drinker(name, address)` and `Frequents(drinker, bar, times_a_week)` relations together using drinker name, we can write `Drinker \join_{name = drinker} Frequents;`. Syntax for *cond* again follows SQL; see notes on `\select` for more details.
- `\join` is the relational natural join operator. For example, to join `Drinker(name, address)` and `Frequents(drinker, bar, times_a_week)` relations together using drinker name, we can write `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 `\rename` to create two `name` attributes for the natural join; see notes on `\rename` below for more details.
- `\cross` is the relational cross product operator. For example, to compute the cross product of `Drinker` and `Frequents`, we can write `Drinker \cross Frequents;`.
- `\union`, `\diff`, and `\intersect` are the relational union, difference, and intersect operators. For a trivial example, to compute the union, difference, and intersection between `Drinker` and itself, we can write `Drinker \union Drinker;`, `Drinker \diff Drinker;`, and `Drinker \intersect Drinker;`, which would return `Drinker` itself, an empty relation, and `Drinker` itself, respectively.
- `\rename_{`*`new_attr_name_list`*`}` is the relational rename operator, where *`new_attr_name_list`* is a comma-separated list of new names, one for each attribute of the input relation. For example, to rename the attributes of relation `Drinker` and compute the cross product of `Drinker` and itself, we can write `\rename_{name1, address1} Drinker \cross \rename_{name2, address2} Drinker;`.

Known Issues/Limitations

- `\rename` only supports renaming of attributes; it does not support renaming of the input relation.
- `\union`, `\diff`, and `\intersect` check the compatibility of input relations by checking the types of each pair of attributes in the order they appear in the input relations. Therefore, `(\project_{name, address} Drinker) \diff (\project_{address, name} Drinker)` will not return an empty table, because the ordering of attributes is important. This behavior is consistant with SQL (which departs from the "purest" relational model where the ordering of attributes is unimportant).
- Error messages may not be especially meaningful. Recall that RA translates relational algebra expressions into SQL queries. RA expressions with ill-formed selection/join conditions or attribute name lists simply result in incorrect SQL queries. In these cases, RA just passes back the error messages from the underlying database system, without attempting to create RA-specific messages.
- For some database systems, RA does not allow a relational algebra expression (or subexpression) to produce a result relation with identically named attributes, because these attributes cannot be distinguished. Depending on the underlying database system, you may get an error message like *column "name" specified more than once* (from PostgreSQL), or *the statement does not include a required column list* (from DB2).
  In particular, `\cross` and `\join_{`*`cond`*`}` (theta-join) do not allow input relations with identically named attributes. For example, `Drinker \cross Drinker;` is illegal because it would generate a relation with two `name` and two `address` attributes. Therefore, the attributes need to be renamed before the cross product. Also, `\project` cannot output the same attribute more than once.
  SQLite has no problem with identically named attributes, because it automatically assigns attributes different names.
- As a general rule, RA may signal an error whenever it has trouble determining how to name the attributes of a result relation. This behavior is database-dependent. For example, consider `Bar \union Beer;`. Never mind what this query means, but it is arguably legal because both relations have two attributes with compatible string types. PostgreSQL, SQLite, and MySQL have no problem with this query, but DB2 will complain that their attribute names are different. In any case, `\rename_{A, B} Bar \union \rename_{A, B} Beer;` will always work. Also, `\project_{price+1} Serves` does not work on DB2, not because it does not support arithmetic operators, but because RA does not know how to name the resulting attribute. On the other hand, PostgreSQL will accept this query and assign a default attribute name `?column?`, though it will still bark at `\project_{price+1, price+2} Serves`. SQLite and MySQL happily take both, by automatically assigning different names to output attributes.

Maintained by Jun Yang