Rax : Behavioral-Data Exporation

Jan-Mark Wams
Coders Co.
<jms@codersco.com>

Gosia Wrzesińska
Coders Co.
<gosia@codersco.com>

May 2014

Executive Summary

In a traditional analytics environment, data scientists spend a lot of time working around the limitations of their tools. Their most beloved tools, such as Excel or the R programming language, are unable to process large amounts of data. Tools that can deal with big data, such as relational database systems or Hadoop clusters, lack an interface that would allow data scientists to properly express their analytical questions. SQL lacks the expression power to deal with complex questions which quickly results in unreadable and unmaintanable queries. Hadoop's MapReduce or Spark require parallel programming skills which data scientists usually lack, forcing them to seek help from programmers.

Rax brings the expression power of mathematical expressions to big data. Rax is a programming language for behavioral-data exploration and analysis. The language is a mix of relational, regular, and temporal algebra. Its expressive power lets you create complex queries in simple and compact statements. At the same time, Rax processes the data directly where they reside – in your database. With Rax, you do not need to move your data between different data-analysis tools and you can utilize the performance of modern database systems to process very large datasets. Rax makes data scientists more productive by drastically simplifying the data-analysis process and making it less error-prone. With Rax, subtle behavior patterns in your data can finally be unlocked.

1 Introduction

Rax (Relational Algebra with eXtensions) is a programming language designed to make it easy to explore and analyze behavioral data, such as the who-where-when-and-what of watching TV, tweeting, or web-shop visits. Rax is a mix of relational, regular, and temporal algebra and shares some features with query languages like SQL and modeling languages like R. As *data* are turning into *Big Data*, it is getting harder to analyze them using traditional tools such as Excel or R because the data size is too large to fit in memory. On the other hand, languages and tools created for manipulating large datasets, such as SQL or MapReduce do not have enough expression power to deal with complex analytic questions. Especially when it comes to behavioral data, data analysts have two alternatives. They can either work in close concert with an SQL, Pig or MapReduce specialist to program a complex analysis using one of these tools. Or they can reduce the dataset size by aggregating and/or sampling it in SQL, and then export it to a tool with more power of expression, such as R. Both approaches have huge disadvantages. The first approach usually results in a lot of synchronization and communication overhead between the analyst and the programmer. It regularly leads to turf wars between analytic and software-development departments. The second approach significantly reduces the accuracy of the results and introduces a lot of manual, error-prone labor in copying the data between the tools. It also makes it hard to automate the data-analysis processes. Rax offers a third alternative: it empowers data analysts to explore and analyze behavioral data directly where they reside - in your database. Unlike SQL or MapReduce, Rax syntax feels intuitive and familiar to analysts and modelers, as it resembles mathematical notation. As we say in Rax:

```
{$} : greeting := {"Hello"} \/ {"Behavioral-Data", "Exploration"} \/ {"!"};
`print greeting * " ";
> "Hello Behavioral-Data Exploration !"
```

The Rax system has been designed to take advantage of parallelism, column stores, server-side processing, and multiple database systems. Rax is also designed to be easily extensible. Rax is the culmination of a process that started in 2001. The current version three is a total rewrite of version two and was started in 2005. It has been designed in cooperation with several econometric specialists.

2 The Problem

We have found out that math-inclined data analysts, or *modelers* as we refer to them, tend to be pragmatic when their model has to be translated into a program. They use all kinds of tools strung together to create a rudimentary program. Modelers often work with programmers, or *coders* as we refer to them, to create, cleanup, and extend these rudimentary programs. There seem to be several recurring problems:

De Facto Programming Model

Modelers often implement their models themselves using old code snippets, instruction texts, and personal habits. Coders have to incorporate the resulting amalgam of Pig, SQL, Excel macros, SPSS, Octave, and R into the rest of the program using the likes of C++, C#, and Python. This often leads to what we refer to as "spaghetti code." Moreover, modeling companies often have trouble hiring coders, forcing the modelers to do even more coding. All modelers we talked to reported that, in all previous modeling institutions they had worked at, such an unstructured approach was used.

SQL: De Facto Standard

Most modelers use SQL for data processing in their mix of tools to explore and manipulate data.

SQL is a seemingly simple language, yet extremely difficult when it comes to the details. (Actually SQL is a collection of incompatible data definition and processing languages.) Handling NULL for example is far from trivial, leading to code like this:

```
SELECT * FROM T  -- May contain NULLs.
    WHERE IFNULL(A = B, FALSE) OR (A IS NULL AND B IS NULL);
```

Further, to be able to get maximum performance from a SQL database system, one must apply many tricks. The most common one, is combining multiple steps of an analysis into a single, large and unwieldy SQL query. This complicates writing SQL to the point where you need an SQL expert to complement a modeler.

Understanding Time

With behavioral data, time plays a very important role. Yet, time-related data is especially hard to analyse. The main culprit is the fact that time concepts are confusing. While humans can handle time intuitively, passing these intuitions to a computer program is hard. For example, implementing a growth rate using a relative duration of "twelve months" or an absolute duration of "fifty two weeks" can influence the outcome dramatically. Even to the human brain these distinctions can be confusing to understand and explain. Most programming tools sport a host of functions to help convert time to numbers and back. This usually leads to fewer but more subtle errors.

Miscellaneous Coding Woes

During conversations with modelers, we found out that subjects like lexical-versus-dynamic-scoping are seldomly understood well. The behaviour of the bash code below surprised most modelers including the ones that actually used bash.

```
x=1
function g () { echo $x; x=2; }
function f () { local x=3; g; echo $x; }
x=4
f                             # will print '3' then '2'.
echo $x                       # $x will be '4'.
```

3 The Solution

In 2001, Pointlogic, an international modeling company, sought to remedy some of the problems described above. On top of that, they wanted to speed up their development process and solve problems related to system-imposed limits like the maximum number of rows or columns and excessive execution times. In 2005, after the development of previous in-house tools reached a dead-end, a fresh start was made, resulting in the first version of Rax.

Practical Needs

The solution had to be triple fast: implementing a model should be fast, models should execute fast, and learning how to build a model should be fast. On top of that, the solution had to be future-proof. These practical needs led to several technical necessities.

Fast To Code

In order to allow fast development, a DSL (Domain Specific Language) was called for that was *strong typed*, had *lexical scoping* and *closures*. Also, the syntax should not hinder code locality. Since Rax forces a clear separation between the model and rest of the code, Rax code is highly reusable, speeding up development also.

Fast Execution

In order to allow fast execution, the design had to support parallelism, column stores, server-side processing, and multiple database engines. Therefore, the language needed to be *functional* with *closures*, should have *separate parse and execute phases*, and be *modular*.

Fast To Learn

In order to flatten the learning curve, the design had to be *orthogonal*, have a *math-like syntax*, offer an *interactive mode* plus *simple type aggregates* and again have *lexical scoping*. Note that the concept of closures could be considered hard to grasp but in practice it fitted the expected behaviour of code better. The behaviour of the Rax code below did raise some eyebrows, but eventually was preferred by the modelers.

```
#: x := 1;
#<-#: g <- { `print x; };      // x cannot be assigned to.
#<-#: f <- { #:x := 3; #:dummy := g(0); `print x; };
x := 4;
#:dummy := f(0);              // will print '1' then '3'.
`print x;                     // x will be '4'.
```

4 The Design

Much can be explained about functional languages and their properties. The same goes for the pros and cons of closures, strong typing, and so on. This, however, falls outside the scope of this introductory paper. Below just a few highlights of the design:

Astronomical Time

In order to make it as easy as possible to work with time, Rax implements five different temporal types. Point in *time*, *absolute* duration, *relative* duration, time *interval*, and time interval *modulo*. Combining exact temporal types with strong type checking proved to be effective in catching temporal errors because it forces modelers to express what they mean exactly.

In addition to the temporal types, Rax implements a number of temporal operators which make dealing with time-related data easier and more intuitive than SQL. For example, the *temporal and* (`@&`) operator allows to easily compute the overlap between two sets of intervals. This is very useful when computing, for example ratings of TV programs, like in the below example.

```
{[#:channel_nr, $:program_title, |:air_time]}: tv_programming := {
  [1, "Product X ad on channel 1", (|)"2013-11-06T20:30/2013-11-06T20:32"],
  [2, "Product X ad on channel 2", (|)"2013-11-06T20:03/2013-11-06T20:05"],
  [2, "Scary horror movie",        (|)"2013-11-06T20:05/2013-11-06T22:00"]};
{[#:respondent_id, #:channel_nr, |:viewing_period]}: viewing_data := {
  [1, 1, (|)"2013-11-06T19:59/2013-11-06T20:31"],
  [2, 1, (|)"2013-11-06T17:13/2013-11-06T22:07"],
  [2, 2, (|)"2013-11-06T22:07/2013-11-06T22:31"],
  [3, 2, (|)"2013-11-06T19:18/2013-11-06T22:31"],
  [4, 1, (|)"2013-11-06T19:59/2013-11-06T21:07"],
  [4, 2, (|)"2013-11-06T21:59/2013-11-06T22:31"]};
&: total_num_respondents := # ! project [.respondent_id] viewing_data;
`print "Num respondents: " + ($)total_num_respondents;
{[#:channel_nr, $:program_title, |:viewing_time, #:respondent_id, #:ch_nr]}:
  viewings :=
    tv_programming @& :[.channel_nr#1 == .channel_nr#2] viewing_data;
{[#:channel_nr, $:program_title, &:rating]}: ratings :=
  project [.#2, .#3, .#1/total_num_respondents]
  Gcount [.respondent_id]:[.channel_nr, .program_title] viewings;
`print ratings;
```

Another example of a specialized temporal operator is *temporal union*: `@\/` which allows the unification of short events into longer ones, useful for example for converting clickstreams into website visits.

Relational Algebra

There is no explicit table type in Rax. By convention, a set of tuples is referred to as a "table." This allows modelers to think of data as mathematical objects. For example, since a table is just a set, it can be partitioned into a set of (non-overlapping) tables. Below is an example of how partitions have to be used in Oracle SQL with analytic functions:

```
-- Find the top two ranking earners per department.
-- Using PARTITION BY and the rank() analytic function:
SQL> SELECT *
  2  FROM (SELECT deptno, empno, ename, sal,
  3               rank() OVER (PARTITION BY deptno
  4                            ORDER BY sal DESC) "rank"
  5        FROM emp)
  6  WHERE "rank" <= 2
  7  ORDER BY deptno, "rank"
  8  /
```

In Rax, it is possible to write a filter function and apply that to a partition, i.e., a set of tables. The resulting set of sets can then be unified, as demonstrated below:

```
// Find the top two ranking earners.
\emp <- \emp: topSalary <- { out := in[1..2]![-.SAL]; };

// Partition on DEPTNO, get the top earners, unite.
\emp: topEarners := \/ topSalary(partition[.DEPTNO] emp);
```

Without in-depth knowledge of both SQL and Rax it might be hard to see why the second one would be preferred by modelers, but the Rax version closely matches how math-inclined modelers think and is therefore preferred. Also in Rax it is far easier to store and print (or plot) intermediate data, like the set of sets "`partition[.DEPTNO] emp`" mentioned above. This allows piecemeal development or debugging and it improves understanding of the code.

Integration with SQL databases

Rax runs atop an SQL-based relational database system, translating data-heavy operations into SQL queries. Rax can run on top of many different database systems. Its syntax and functionality are not dependent on the specific database system used. In other words, `Rax/Redshift` can execute exactly the same Rax code as `Rax/MySQL` and `Rax/Azure`. The subtle differences between various SQL dialects are hidden from Rax coders.

Because Rax operations are essentially running in a database system, Rax can operate directly on the data that already reside in the database. No expensive ETL operation is needed[1]. The Rax statement below maps the data residing in a database system to a Rax set:

```
{[#:RespondentId, |:Timeslot, #:ChannelId]}:
  TvExposures :=
    import [
      (#)"RESPONDENT_ID",
      (|)[(@)"START_TIMESLOT", (@)"END_TIMESLOT"],
      (#)"CHANNEL_ID"
    ]
    "TV_EXPOSURES";
```

Four columns from the `TV_EXPOSURES` table, `RESPONDENT_ID`, `START_TIMESLOT`, `END_TIMESLOT` and `CHANNEL_ID`, are mapped onto a 3-attribute Rax set `TvExposures`. Note that two columns, `START_TIMESLOT` and `END_TIMESLOT` are mapped to a single attribute, `Timeslot`. This is because Rax has a special type for time slots, while SQL does not. Do not be mislead by the `import` keyword. This is basically a mapping operation. No data copying is taking place. It is also very easy to export numbers produced by Rax back to the SQL database:

```
`export ReachPerChannel, "REACH_PER_CHANNEL";
```

The statement above exports the results of a Rax TV-reach computation back to the SQL database, to the `REACH_PER_CHANNEL` table.

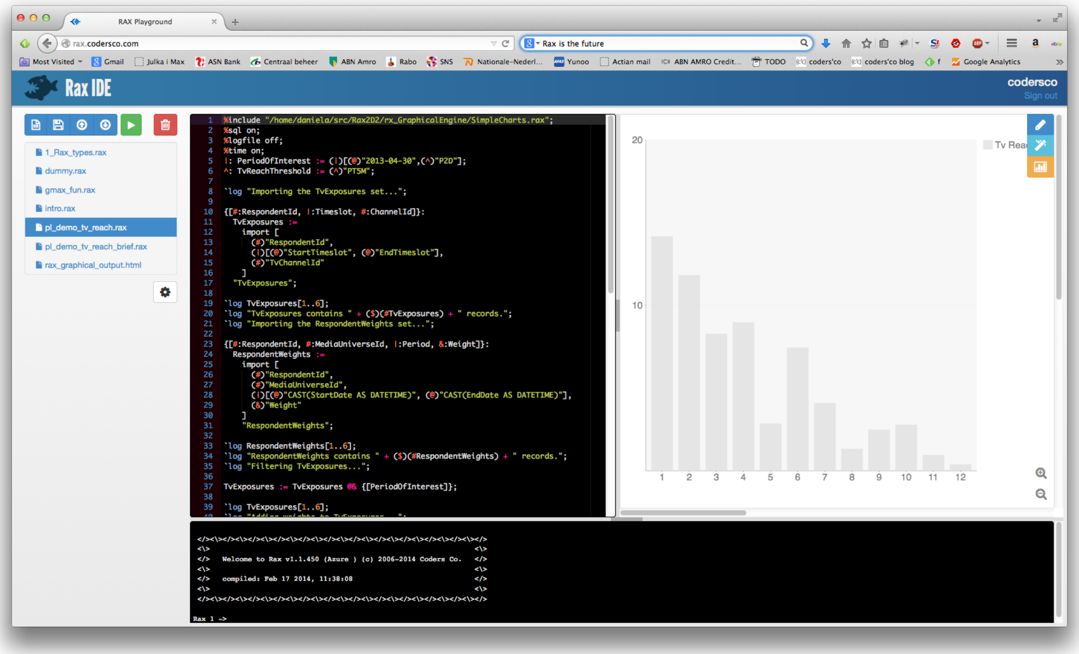
For optimal performance, Rax uses lazy execution. Rax set expressions are translated into SQL snippets and combined into a single, nested SQL query. Only when you actually want to see the results (i.e., when you use the `` `print `` procedure or generate a plot) is the SQL query executed. Therefore, you can split a complex analysis into small, easy-to-understand steps and still enjoy the performance advantages of combining multiple steps into a single SQL query. Debugging calculations written in Rax is very easy too: by inserting a few `print` statements in between the calculation steps, you will be able to easily see the intermediate results. Rax will automatically switch to executing simple SQL statements one-by-one, rather than combining them into a single SQL statement.

5 The Current State

Rax runs on Windows and Unices (including OS X). Rax supports many different database systems from simple in-core systems like SQLite, through traditional single-node systems like MySQL, PostgreSQL and SQL Server to large, parallel in-cloud databases like Amazon Redshift, Microsoft Azure and Snowflake.

[1]That said, Rax also allows importing data from files in various formats, such as CSV or XML

Rax has a graphical, browser-based IDE with built-in graphical output (see figure 1). The IDE contains a script editor with syntax highlighting and auto-completion and a window for executing Rax statements interactively. The IDE allows to you to connect to any SQL database (supported by Rax): just enter an ODBC connection string. You can also upload and import CSV and XML files with data. The IDE also lets you have a peek at the generated SQL queries.



```
%include "/home/daniela/src/Rax2D2/rx_GraphicalEngine/SimpleCharts.rax";
%sql on;
%logfile off;
%time on;
|: PeriodOfInterest := (|)[(@)"2013-04-30",(^)"P2D"];
^: TvReachThreshold := (^)"PT5M";

`log "Importing the TvExposures set...";

{[#:RespondentId, |:Timeslot, #:ChannelId]}:
  TvExposures :=
    import [
      (#)"RespondentId",
      (|)[(@)"StartTimeslot", (@)"EndTimeslot"],
      (#)"TvChannelId"
    ]
  "TvExposures";

`log TvExposures[1..6];
`log "TvExposures contains " + ($)(#TvExposures) + " records.";
`log "Importing the RespondentWeights set...";

{[#:RespondentId, #:MediaUniverseId, |:Period, &:Weight]}:
  RespondentWeights :=
    import [
      (#)"RespondentId",
      (#)"MediaUniverseId",
      (|)[(@)"CAST(StartDate AS DATETIME)", (@)"CAST(EndDate AS DATETIME)"],
      (&)"Weight"
    ]
    "RespondentWeights";

`log RespondentWeights[1..6];
`log "RespondentWeights contains " + ($)(#RespondentWeights) + " records.";
`log "Filtering TvExposures...";

TvExposures := TvExposures @& {[PeriodOfInterest]};

`log TvExposures[1..6];
```

```
</><\></><\></><\></><\></><\></><\></><\></><\></><\></><\></><\></>
<\>                                                              <\>
</>   Welcome to Rax v1.1.450 (Azure ) (c) 2006-2014 Coders Co.   </>
<\>                                                              <\>
</>   compiled: Feb 17 2014, 11:38:08                             </>
<\>                                                              <\>
</><\></><\></><\></><\></><\></><\></><\></><\></><\></><\></><\></>

Rax 1 ->
```

Figure 1: Rax IDE

6 The roadmap

In the near future, we are planning to add more statistical and machine-learning functionality to Rax. One approach we are considering is by integrating R into Rax, so that users can call R functions directly from Rax environment. This approach, however, will limit the machine-learning functionality to small, in-memory datasets (as R works in principle only on in-memory data). Another approach is to use the lower-level APIs provided by many database systems to implement statistical and machine-learning algorithms operating on data residing in the database system.

Further, the graphical capabilities of Rax will be extended to provide more types of charts and more chart formats, such as interactive JavaScript charts. There are also many improvements planned for the IDE, in particular a better interactive mode.

The above extensions will allow modelers to use Rax as a replacement of the current amalgam of data-processing tools. It is our ambition to make Rax the one-stop-shop of behavioral-data analysis.

7 Summary

Rax makes data analysis and modeling faster and easier by providing a natural habitat to modelers. Rax makes modelers less dependent on programmers and database experts because it automatically translates modeling concepts to efficient database instructions. Also, Rax hides the difference between different database systems. Code that works in Rax/MySQL will work unaltered on Rax/Redshift or any other Rax system.

With Rax you can ask more complex questions about your data. It takes far less time to get results and there are less errors in your findings. With Rax you need less people to do the same work or do more work with the same people.