12-2013

The relational algebra toolkit: A user-friendly approach to presenting and processing relational algebra queries on the web

Jeremy J. Miller
University of Texas-Pan American

THE RELATIONAL ALGEBRA TOOLKIT:

A USER-FRIENDLY APPROACH TO

PRESENTING AND PROCESSING

RELATIONAL ALGEBRA QUERIES

ON THE WEB

A Thesis

by

JEREMY J. MILLER

Submitted to the Graduate School of
The University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2013

Major Subject: Computer Science

THE RELATIONAL ALGEBRA TOOLKIT:

A USER-FRIENDLY APPROACH TO

PRESENTING AND PROCESSING

RELATIONAL ALGEBRA QUERIES

ON THE WEB

A Thesis
by
JEREMY J. MILLER

COMMITTEE MEMBERS

Dr. Artem Chebotko
Chair of Committee

Dr. Christine Reilly
Committee Member

Dr. Wendy Lawrence-Fowler
Committee Member

Dr. Xiang Lian
Committee Member

December 2013

ABSTRACT

Miller, Jeremy J., The Relational Algebra Toolkit: A User-Friendly Approach to Presenting and Processing Relational Algebra Queries on the Web. Master of Science (MS), December, 2013, 108 pp., 4 tables, 18 figures, 9 references, 56 titles.

Relational algebra is the mathematical basis of tools to express and optimize queries on relational data. However, there exists no convenient way to directly use, express, store, render, visualize, and execute relational algebra over the Web. Educators and practitioners are limited to creating relational algebra expressions using TeX or equation editors which cannot execute them.

This work presents a solution to this issue: the Relational Algebra Toolkit (RAT). Relational data and queries are represented using purpose-built XML vocabularies, to be both machine-processable and serializable. Encoded relational algebra expressions can be rendered as parenthetical expressions and as syntax trees, translated to SQL, reordered, and executed on encoded data in a JavaScript-enabled Web browser. These services are invoked as prescribed by the user, and the results are inserted into a Web page. RAT has been used by a number of universities internationally in undergraduate and graduate database courses, with favorable student feedback.

DEDICATION

The completion of this thesis would not have been possible without the quite literally self-sacrificing commitment of my mother, Debi Miller, to ensuring that her children have the best possible opportunities to succeed in whatever way makes them happy. I can never thank you enough for your love and support.

ACKNOWLEDGMENTS

I’d like to thank the Academy. The Science Academy of South Texas, that is, for showing me that even in a morass of intellectual apathy like the Lower Rio Grande Valley, there is always an opportunity to make a difference in the education of young minds. To that end, I would like to also thank Dr. Artem Chebotko and UTPA’s Department of Computer Science for allowing me to make my own small contributions to this field, not least of which is this thesis. The elevation of this thesis from the rank of “senior design project” would not have been possible without the patient counsel of Dr. Chebotko.

Thanks must also be extended to Nathan Arnold and Hussein Bakka, who developed RAML, the original RAT website, RAML Edit, and the initial concepts that evolved into this thesis. I would also like to thank Daniel Rebollar and Peter J. Vasquez, Sr., for the development of RDML. And of course Fadi Damaj, for his work on the website and documentation.

I would also like to take this opportunity to thank Microsoft Corporation for taking six years to catch up to Opera™ browser with regard to supporting SVG, without whom the amount of work required to implement Visualize would have been greatly reduced.

TABLE OF CONTENTS

Page

ABSTRACT.. iii

DEDICATION.. iv

ACKNOWLEDGMENTS... v

TABLE OF CONTENTS.. vi

LIST OF TABLES.. ix

LIST OF FIGURES... x

CHAPTER I. INTRODUCTION... 1

Statement of Problem... 2

Existing Techniques... 2

CHAPTER II. RELATED WORK... 4

CHAPTER III. PROPOSED SOLUTION... 6

System Architecture.. 7

XML Core.. 7

Services Layer... 8

Management Layer.. 10

Application Layer.. 10

RAML: Relational Algebra Markup Language... 11

Schema and Specification... 11

Relation and Attribute Specifiers.. 12

Boolean Conditions.. 13

Relational Algebra Operations... 15

A Note on Order of Operations.. 19

RAML Edit.. 20

RAML Font... 21

RDML: Relational Database Markup Language.. 23

Schema and Specification... 23

Relation Instance and Data Tuples... 24

Relation Schema – Attributes and Constraints................................ 25

Creating RDML Documents... 27

RATTAIL: RAT Automated Instruction Language.. 28

Service Descriptions.. 31

Validate... 31

Render... 32

Visualize... 33

Tabulate.. 35

Translate... 35

Reorder... 36

Execute... 37

Management Layer.. 40

Roles of RAT Manager... 40

Handling Service Outputs... 42

CHAPTER IV: DEPLOYMENT AND EVALUATION.. 44

Examples of Use.. 45

Evaluation... 47

CHAPTER V: CONCLUSIONS... 48

Future Work... 49

REFERENCES.. 50

APPENDIX A: XML SCHEMA FOR RAML.. 52

APPENDIX B: XML SCHEMA FOR RDML.. 60

APPENDIX C: XML SCHEMA FOR RATTAIL... 64

APPENDIX D: ALGORITHMS FOR RAML VALIDATOR............................. 66

APPENDIX E: ALGORITHMS FOR RDML VALIDATOR............................. 75

APPENDIX F: ALGORITHMS FOR RENDER.. 85

APPENDIX G: ALGORITHMS FOR VISUALIZE... 93

APPENDIX H: ALGORITHMS FOR TABULATE... 105

BIOGRAPHICAL SKETCH.. 108

LIST OF TABLES

Page

Table 1: RAML Font Symbols.. 21

Table 2: Browser Support for Render, by Version.. 33

Table 3: Browser Support for Visualize, by Version.. 35

Table 4: Tabulate Output Sample... 35

LIST OF FIGURES

Page

Figure 1: Layered Architecture of RAT....... 7

Figure 2: Flow of Data Between RAT Manager and Services....... 9

Figure 3: Root Element of RAML....... 11

Figure 4: RAML Relations and Attributes....... 12

Figure 5: Booleans....... 13

Figure 6: Projection and Selection....... 15

Figure 7: Attribute Renaming....... 16

Figure 8: Operations Without Boolean Conditions....... 17

Figure 9: Joins....... 17

Figure 10: Relation Assignment....... 18

Figure 11: Screenshot of RAML Edit....... 20

Figure 12: RDML Base Elements....... 24

Figure 13: RDML Schema Element....... 25

Figure 14: RDML Foreign Key....... 26

Figure 15: RATTAIL Schema....... 28

Figure 16: Render Output Sample....... 33

Figure 17: Visualize Output Sample....... 34

Figure 18: Translate Output Sample....... 35

CHAPTER I

INTRODUCTION

Relational database management systems – hereafter called RDBMSes – are based on the principles laid out by Edgar Codd in his 1969 paper that proposed a radically new paradigm for the organization and management of data that he termed the relational model [Codd 1969]. Part of this model is the use of relational algebra as a means by which to request specific information from such a database. To this day, relational algebra is part of the curricula for many educational courses about relational databases. However, unlike the SQL family of query languages that were developed around the exact same principles, it remains frustratingly difficult for many students, educators, and practitioners to work with relational algebra directly on a computer.

Current relational database management systems are designed such that users are meant to use SQL to interface with a relational database. With respect to queries, relational algebra can be used an alternative to the "SELECT" statements of SQL, since relational algebra is what provides the procedural mathematical basis for the optimization of declarative SQL queries. However, although relational algebra is used internally by relational database management systems to *optimize* SQL statements, users cannot use it *directly* in place of SQL.

Statement of Problem

There is a need for a software framework that allows the direct and convenient use of relational algebra as an alternative to SQL. Such a framework must also address the difficulties in expressing relational algebra on the Web.

Existing Techniques

It seems that to construct relational algebra expressions, or to empirically verify relational algebra expressions for correctness, or to actually execute them on sample data, or to reorganize them to produce results-equivalent expressions, the dominant technique is simply to work it out by hand. This is tedious, error-prone, and lacks the elegance of computing.

Existing techniques for the *digital* representation of relational algebra expressions include the use of TeX-based tools or equation editors, which retain the mathematical structure of the expression content. However, TeX-based tools typically have a fairly steep learning curve, and there are many different equation editors, each with their own underlying format.

One promising alternative is to encode relational algebra expressions in HTML or MathML using sub- and superscripts as well as fonts that define all the necessary symbols. This would additionally allow them to be included in the content of Web pages and certain word processing documents. However, there are so many stylistic differences between HTML programmers that it is hardly a portable solution to do this manually. MathML – despite being a World Wide Web Consortium Recommendation since 1998 [W3C Math Working Group] – lacks widespread Web browser support [Deveria 2013].

Furthermore, both HTML and MathML would require the use of a font that supports all the necessary symbols for relational algebra operators. The symbols for outer joins, in particular, are not implemented by any font that can be reasonably expected to be already on the user platform (e.g. Arial, Helvetica, Times). Fonts that attempt to implement as much of the Unicode standard as possible (e.g. Code2000, FreeSerif) are either impractically large in file size or are not licensed for commercial use.

CHAPTER II

RELATED WORK

Many of the concepts that underlie the RAT framework are innovative, in the sense that I was unable to find any existing implementations of them. For instance, I could not find a generally accepted XML vocabulary that captures the data definition language of relational databases, nor one that encodes algebraic expressions in a way that emphasizes the operations that are being encoded (rather than merely their symbols).

This work was originally conceived out of a desire to be able to display relational algebra expressions in a Web page, as one might find on the webpage of an educational course on database design. Similarly, one of the original purposes of MathML was to provide “much-needed support” for incorporating mathematical expressions (of all sorts) into Web pages [MathML 1.0 Specification, 1998]. However, the only native Web browser support for MathML is for so-called “presentation markup,” not “content markup”; that is, one can design MathML encodings that emphasize the operation being encoded, but the browser may not interpret it as the writer had intended. Furthermore, native Web browser support for MathML is not even universal as of this writing.

The W3C Math Working Group recommends the use of equation editors that generate MathML markup, which can then be embedded into XHTML (provided you are using Mozilla Firefox®) or HTML5 and rendered by Web browsers. This is the kind of elegance to be desired in a solution. However, not all equation editors use MathML; some are based on LaTeX or proprietary formats. All of these formats still require the existence of a font that will render all the symbols that are part of the encoded expressions [LaTeX Project Team, 2011]. I could not find a suitable font that supported the outer join symbols of relational algebra, even though these symbols have been defined by the Unicode standard since 2002 [Unicode Consortium]. There are, of course, specialized mathematical fonts abound, but few freeware ones, and even fewer that are good-looking at smaller font sizes. We also found freeware fonts like Code2000 and FreeSerif that attempt to support all the symbols defined by Unicode; however, such fonts are on the order of megabytes in size.

Although it may be possible to use MathML or LaTeX coupled with a purpose-built font to incorporate relational algebra into Web pages, it is certainly not easy to find utilities that make it easy to learn, teach, and actually work with relational algebra on a computer. These are the most interesting features of the Relational Algebra Toolkit.

CHAPTER III

PROPOSED SOLUTION

In response to the problems and the limitations of existing (partial) solutions as covered in the previous two chapters, I present a novel solution: The Relational Algebra Toolkit, hereafter abbreviated RAT.

RAT consists mainly of a JavaScript-based HTML pre-processor that invokes and handles the results of various utility services based on user-provided workflow scripts. These services are invoked on relational algebra expressions or relational data (or both) encoded in purpose-built XML vocabularies. Also incorporated are a font, a relational algebra expression editor that creates the relevant XML encoding on-the-fly, and a server-side XML database for user-driven storage of particularly interesting XML documents relevant to the usage of RAT.

System Architecture

RAT is a layered system. An XML core enables communication between a layer of utility services and a managerial service that forms its own layer, with Web browsers forming the outer shell and the execution platform for the system. For a graphical representation of this, see Fig. 1.

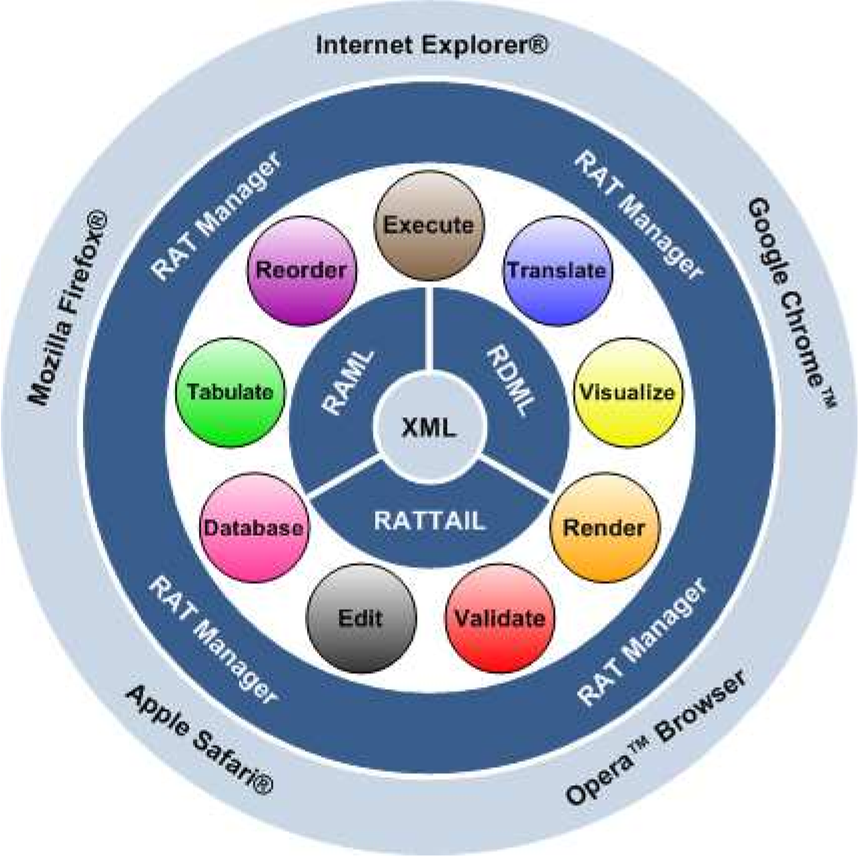


Figure 1. Layered Architecture of RAT.

XML Core

Forming the core of the RAT are two markup languages and a scripting language, each of which can be extended as needed, being formally defined using XML Schema. The first and most fundamental of these is RAML (pronounced like “camel”) – the (R)elational (A)lgebra (M)arkup (L)anguage – which encodes relational algebra expressions so that they have an unambiguous order of operations, allowing for straightforward evaluation.

Some of the services at the next layer require relational data upon which to operate. Such data must be encoded in RDML, the (R)elational (D)atabase (M)arkup (L)anguage. This language is described in detail

The third and final language is used to encode workflow scripts that consist of requests to the management layer to provide specific services (indirectly) upon RAML or RDML documents. This scripting language is known as RATTAIL, the (R)elational (A)lgebra (T)oolki(T) (A)utomated (I)nstruction (L)anguage.

Services Layer

At the services layer is the collection of utility services that truly differentiate RAT from existing solutions. There is a sort of what-you-see-is-what-you-get equation editor for relational algebra expressions – known as RAML Edit – that displays the RAML markup equivalent to the parenthetical expression being constructed by the user in real-time.

Differing from the other services in that it is meant to be explicitly server-side, RAT Database is a repository of RAML and RDML documents that users found particularly helpful or interesting and which they think might be of interest to other users.

Managed services. Both RAML Edit and RAT Database are special in that they do not communicate with the management layer at all; users interact with them directly. The remaining services can be thought of as stored procedures that are available for invocation by the management layer upon request. These managed services provide the following functionalities:

- *Validate* RAML and RDML documents against their respective schemas
- *Render* a RAML document as a conventional parenthetical expression
- *Visualize* the order of operations in a RAML expression by rendering its syntax tree
- *Tabulate* an RDML document as an XHTML table
- *Translate* a RAML expression into an equivalent SQL statement
- *Reorder* a RAML expression to arrive at an equivalent expression that would produce the same results but using a different order of operations
- *Execute* a RAML expression as a query over RDML-encoded relational data.

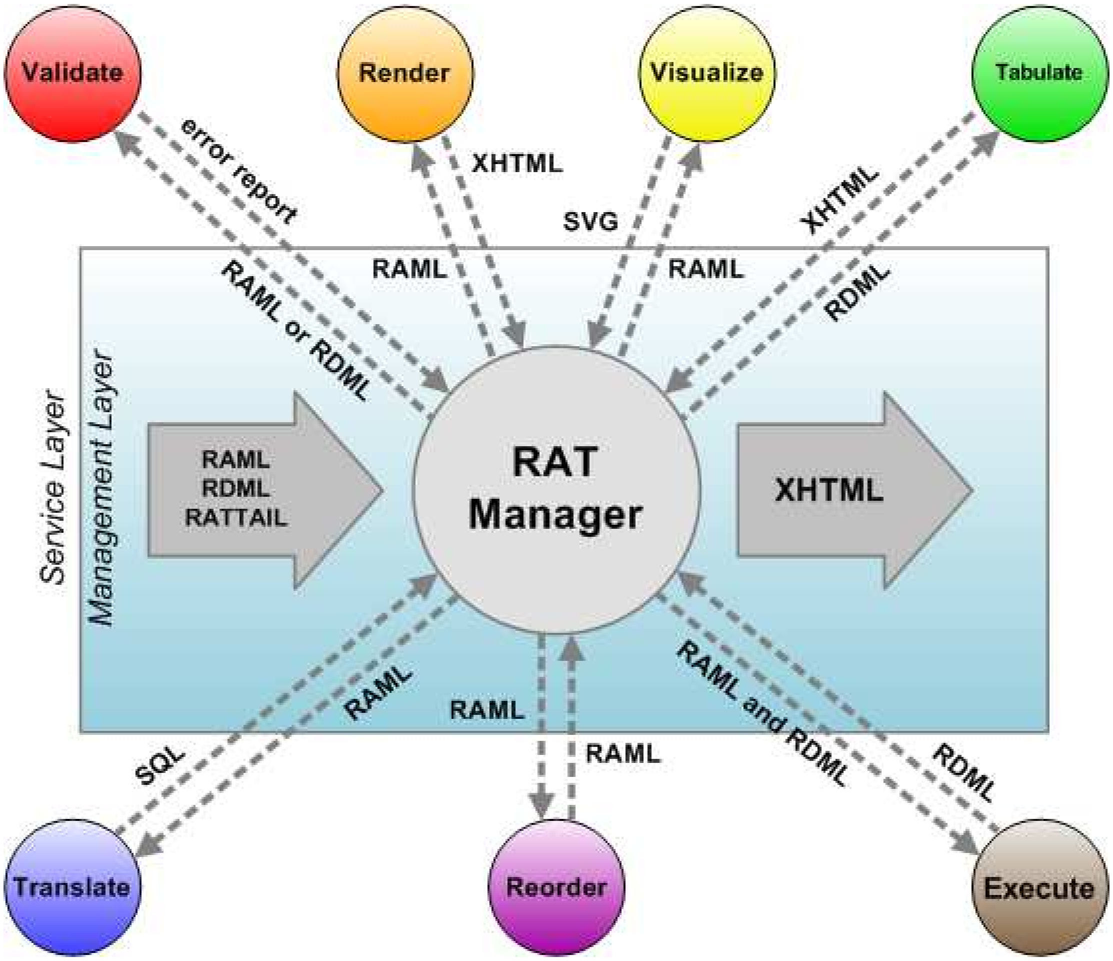


Figure 2. Flow of Data Between RAT Manager and Services.

Management Layer

When a Web page that uses the RAT framework is loaded by a user agent (assumed to be a Web browser), the management layer – whose sole occupant is a transparent, client-side servlet named RAT Manager – is initialized. It seeks out references to RATTAIL documents in the (X)HTML, fetches them using Ajax, and interprets them as a global workflow script. This script consists of instructions that request specific services on specific RAML and RDML documents, and which additionally specify (by *id*) a DOM element in the Web page that should be used to store the results. For each instruction, RAT Manager fetches and parses the specified input documents, passing them to the freshly dispatched services that were requested. When the service completes, the results are passed back to RAT Manager, which then injects them into the Web page at the specified DOM element. When all service requests have been satisfied, RAT Manager terminates. Fig. 2 illustrates this relationship between RAT Manager and services.

Application Layer

The outermost layer of the RAT framework consists of the user's Web browser, along with the RAT website and various tutorials and instructions that we have created for RAT users. Conceptually, the application layer would also include any user application that makes use of the RAT framework in any way.

RAML: Relational Algebra Markup Language

The Relational Algebra Markup Language is an XML vocabulary that provides a means by which to encode relational algebra expressions in a way that leaves the order of operations unambiguous. A RAML document is a tree of nested sub-expressions, the “leaf nodes” of which represent relations (tables). The remaining nodes each specify a relational algebra operation – such as projection, selection, or join – along with its parameters and sub-expression operands.

RAML was designed to be a faithful implementation of Edgar Codd’s original intent for relational algebra [Codd 1990], lacking the aggregation and post-processing (“count”, “avg”, “order by”, “group by…having”) clauses that are provided by standard SQL. Every service in the RAT framework is designed to handle relational algebra on an expression-by-expression basis, each of which is implemented as a RAML document.

Schema and Specification

A copy of the XML Schema document that formally defines RAML can be found in Appendix A. The following RAML markup example is one of the simplest possible relational algebra expressions: a single relation.

```
<?xml version="1.0"?>
<raml xmlns="http://www.cs.panam.edu/2010/RAML">
<expression>
   <relation>
      <name>Student</name>
   </relation>
</expression>
</raml>
```

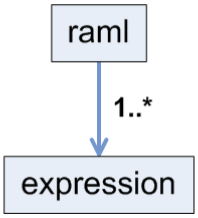


Figure 3. Root Element of RAML.

The root of a RAML document is a <raml> element (Fig. 3), which has at least one child <expression> element, each of which encapsulates a valid relational algebra expression. The schema supports multiple root-level <expression>s, which may be useful as an alternative to keeping track of dozens of individual RAML documents.

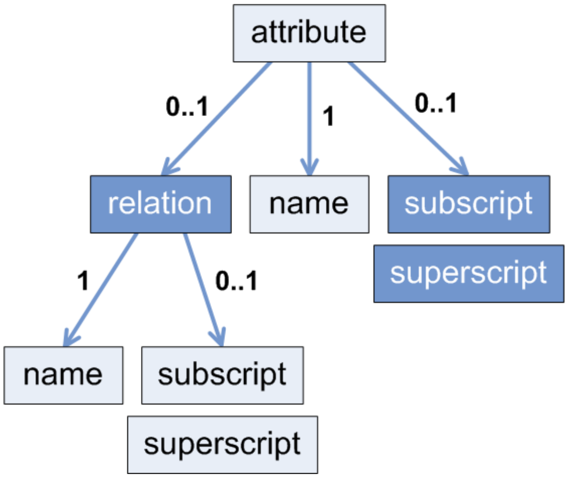


Figure 4. RAML Relations and Attributes.

Relation and Attribute Specifiers. As seen in Fig. 4, relations are encoded using the <relation> tag. The schema allows flexible naming of relations. The set of naming elements for a particular <relation> can only consist of a <name> (required), optionally followed by a <subscript> or a <superscript>. If both are to be specified, the <subscript> must come before the <superscript>. This allows encoding relations that have names like R_A or R^1. The naming elements are XML elements rather than element attributes to bypass the problem that user agents do not need to honor the order in which an element's attributes are specified. ("Note that the order of attribute specifications in a start-tag or empty-element tag is not significant." [XML 1.0 Specification])

Following the convention of having tag names be self-describing, attributes (of a relation schema, not an XML element) that are referenced in a relational algebra expression are specified with an <attribute> element (Fig. 4). RAML allows the same naming flexibility with attributes as it does with relations; its child elements are a required <name> followed optionally by a <subscript> or <superscript>, or both. In addition, the relation to which the attribute belongs can optionally be specified, by preceding the <name> with a <relation> sibling.

Boolean Conditions. These are fundamental to joins and selections. For clarity, it is required that they be enclosed in a <boolean> element. It is worth noting that <boolean> is the most complex element in the RAML schema. The following several paragraphs are illustrated by Fig. 5. Two examples of Boolean expressions are shown below; the one on the left represents “age is at least 18,” and the one on the right means “first name is not ‘John’.”

```
<boolean>
   <gte/>
   <attribute>
      <name>age</name>
   </attribute>
   <number>18</number>
</boolean>
```

```
<boolean>
   <neq/>
   <attribute>
      <name>firstName</name>
   </attribute>
   <string>John</string>
</boolean>
```

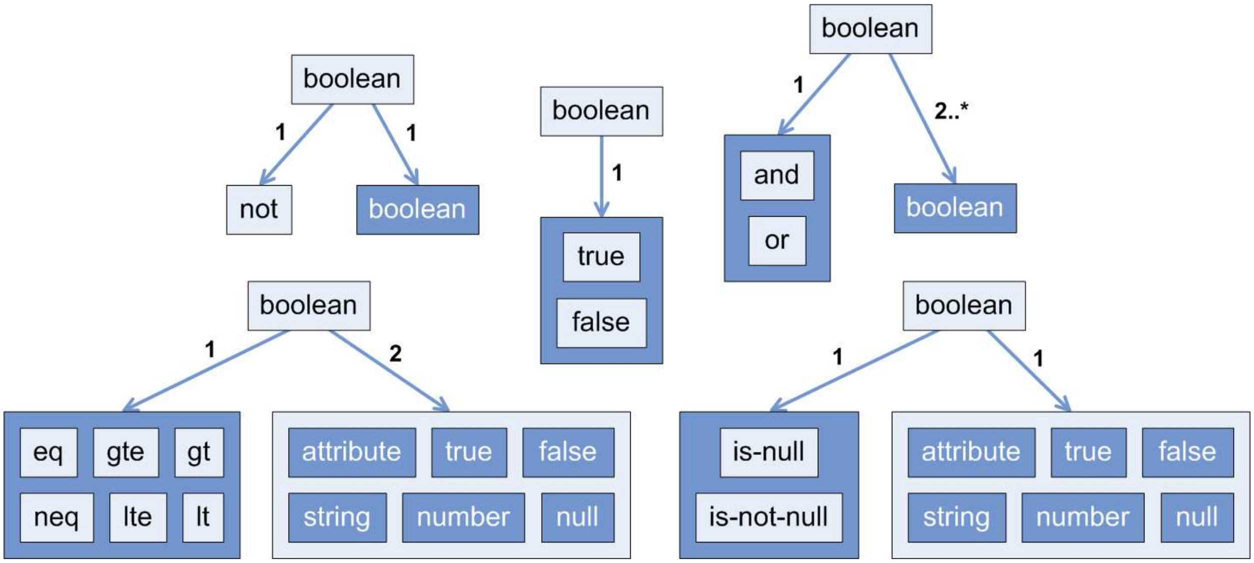


Figure 5. Booleans.

Boolean operators have their own elements. The unary value-existence operator <is-not-null/> and its complement <is-null/> are specified by the RAML schema. The familiar binary value-comparison operators are represented as follows: “equal” is <eq/>, “not equal” is <neq/>, “less than” is <lt/>, “less than or equal to” is <lte/>, “greater than” is <gt/>”, “greater than or equal to” is <gte/>. None of these elements have text content or child elements; their operands are encoded as their immediately following sibling element(s).

The operands of a Boolean expression have their own tag names as well. The simplest are the literal Boolean values <true/> and <false/>, as well as the universal placeholder <null/> which signifies the absence of any particular value. Numeric and character-string values are encoded as the text content of <number> and <string> elements, respectively. The RAML schema does not restrict the length, format, or even content of these two elements. The general idea with respect to encoding literal values is to use <number> for character data that can be parsed as a number, <true/> and <false/> for Boolean values, and <string> for everything else. The previous examples show that an attribute (of a relation schema) can be used as an operand as well; this would be encoded with an <attribute> element.

Logical connectives are encoded as follows: disjunction is <or/>, conjunction is <and/>, and negation is <not/>. The <not/> element is unary, and it must be followed by a <boolean> element. <and/> and <or/> must be followed two or more <boolean>s, in the following style:

```
<boolean>
   <and/>                        <boolean>
   <boolean>. . .</boolean>         <not/>
   <boolean>. . .</boolean>         <boolean>. . .</boolean>
   <boolean>. . .</boolean>      </boolean>
</boolean>
```

Expressing a chained connective in this way – in operation-prefix notation – makes it less likely that a complex Boolean expression will be encoded ambiguously or nonsensically. The alternative would be to have an <and/> or <or/> element between each pair of adjacent <boolean> operands. It is easier to reorder the Boolean sub-expressions when they are immediate sibling elements of each other. Most importantly, in this form, the order of operations is explicit. For instance, in the expression “(a = b ∧ b = c) ∨ c = d,” the “or” clause is at a higher level in the expression than the “and” clause. This means that the RAML-encoded version must be a binary <or/> with one of its <boolean> operands containing a binary <and/>.

Relational Algebra Operations. I have so far described how to encode literal values, specified attributes and relations, and Boolean expressions. Now to the relational operations.

Projection of attributes is represented with the <projection/> element (Fig. 6), which is followed by at least two sibling elements. The final one must be an <expression> element, and the rest are all <attribute>s. This example projects the first and last names of all students:

```
<expression>
   <projection/>
   <attribute><name>firstName</name></attribute>
   <attribute><name>lastName</name></attribute>
   <expression>
      <relation><name>Students</name></relation>
   </expression>
</expression>
```

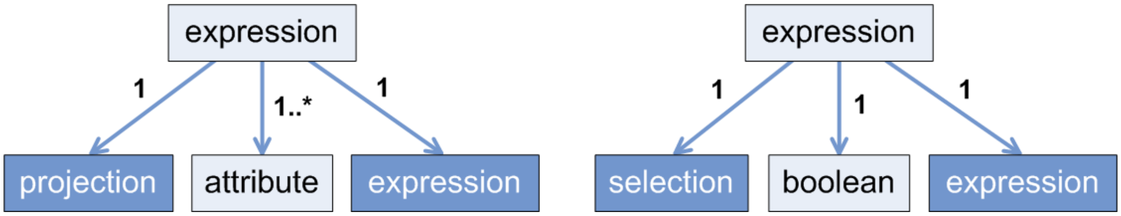


Figure 6. Projection and Selection.

Conditional selection or restriction of tuples is done with the <selection/> element, followed by a required <boolean> element that defines the selection condition, followed by another <expression> (Fig. 6). The following example selects only those students for which the attribute “age” has a numeric value of at least 21.

```
<expression>
   <selection/>
   <boolean><gte/>
      <attribute>
          <name>age</name>
      </attribute>
      <number>21</number>
   </boolean>
   <expression>
      <relation>
          <name>Students</name>
      </relation>
   </expression>
</expression>
```

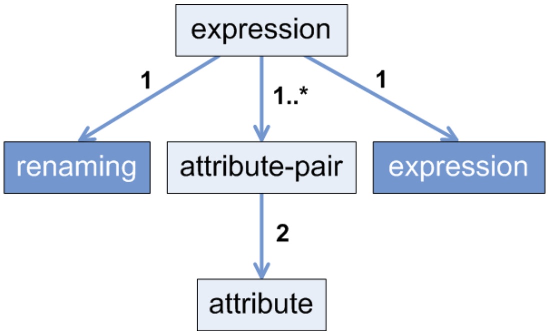


Figure 7. Attribute Renaming.

Renaming of attributes in an intermediate query result is accomplished through the <renaming/> element (Fig. 7). It is followed by at least two siblings, the last of which is an <expression>. The others are all <attribute-pair> elements, each representing a one-to-one mapping of attribute names. Each <attribute-pair> element has two <attribute> child elements. The following example renames *firstName* and *lastName* to *fn* and *ln*.

```
<expression>
   <renaming/>
   <attribute-pair>
      <attribute><name>firstName</name></attribute>
      <attribute><name>fn</name></attribute>
   </attribute-pair><attribute-pair>
      <attribute><name>lastName</name></attribute>
      <attribute><name>ln</name></attribute>
   </attribute-pair>
   <expression>
      <relation><name>Students</name></relation>
   </expression>
</expression>
```

Most of the other relational operators have a common encoding format (Fig. 8): The operation-specific element, followed by two or more <expression> elements. The specific elements for these operations have self-describing tag names, and the currently supported ones are <union/>, <intersection/>, <set-difference/>, <division/>, <cross-product/> (sometimes called cross join or Cartesian product), <outer-union/> (also called union-join), (left) <anti-join/>, (left) <semi-join/>, (inner) <join/>, <left-join/>, <right-join/>, and (outer) <full-join/>.

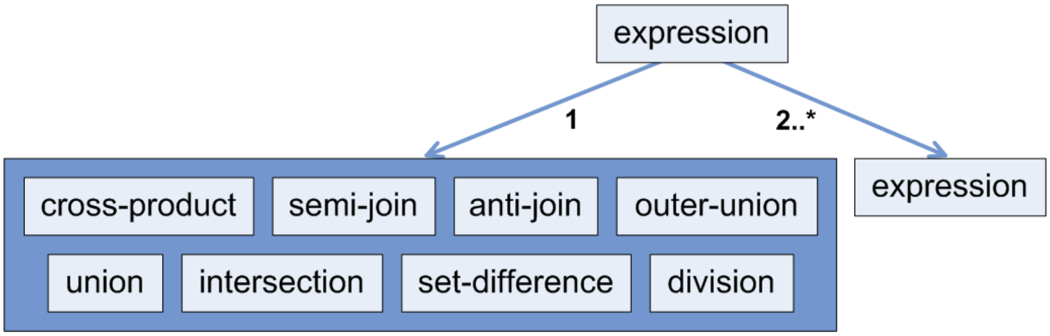


Figure 8. Operations Without Boolean Conditions.

The ability to chain multiple <expression> operands allows the user to specify that there is not meant to be any particular order in which the operand relations are used. In real-world databases, where relations expand and contract, a database management system will not always optimize a particular SQL statement the same way every time. The order of operands, in effect, does not matter unless the writer of the SQL statement wants it to matter.

There is an additional aspect to the join operations that RAML addresses: a join condition can be specified. The <join/>, <left-join/>, <right-join/>, and <full-join/> elements can all be encoded like the other operations, in which case they are interpreted as *natural* joins. However, if an explicit join condition is specified, then a <boolean> element must precede the <expression> siblings, and in that case there can only be two <expression>s. Fig. 9 illustrates this syntax. The following example requests all the entries in the relation Transcript for which the professor is Alan Turing, assuming that a professor's name is stored in relation Professors.

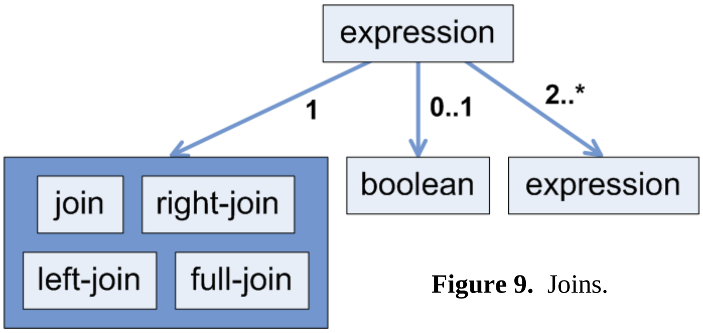


Figure 9. Joins.

```
<expression>
   <join/>
   <boolean><eq>
      <attribute>
         <relation><name>Professors</name></relation>
         <name>name</name>
      </attribute>
      <string>Alan Turing</string>
   </boolean>
   <expression>
      <relation><name>Transcript</name></relation>
   </expression><expression>
      <relation><name>Professors</name></relation>
   </expression>
</expression>
```

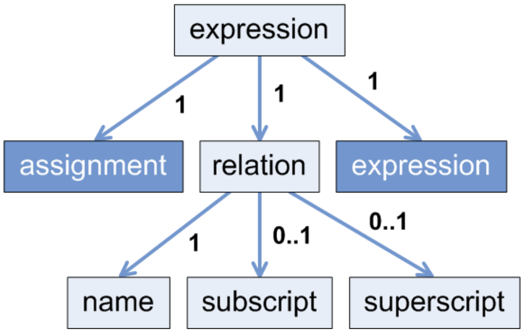


Figure 10. Relation Assignment.

The last operation and element specified by RAML is <assignment/> (Fig. 10), whereby the result of a query or sub-query is assigned an identifier, making it appear to be an actual relation. If a RAML document contains multiple root-level <expression>s, any assignments made in one expression will not be made available to subsequent expressions. This mirrors SQL in that sub-query aliases used in one SQL statement are only defined within that statement; they cannot be used in later statements. An example assignment, “R_3 ← R_1 U R_2”, is shown below.

```
<expression>
   <assignment/>
   <relation>
      <name>R</name><sub>3</sub>
   </relation>
   <expression>
      <union/>
      <expression>
  <relation><name>R</name><sub>1</sub></relation>
      </expression><expression>
         <relation><name>R</name><sub>2</sub></relation>
      </expression>
   </expression>
</expression>
```

A Note on Order of Operations

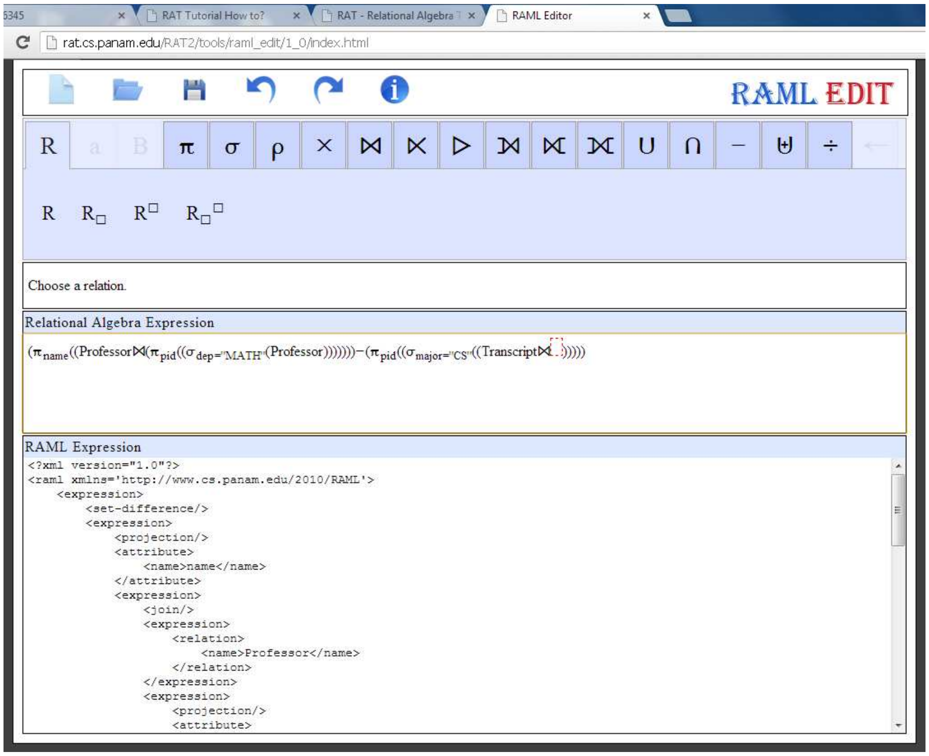
It is worth noting that the order of operations is unambiguous in a RAML-encoded expression. This is not true of arithmetic expressions, which follow a near-universally accepted but otherwise arbitrary order (parentheses, exponentiation, multiplication and division, addition and subtraction). Parenthesized clauses within an algebraic expression must be evaluated before non-parenthesized clauses, so parentheses can be used to explicitly state an intended order of operations. RAML expressions are forced to have an explicit order, because every clause and sub-clause is an <expression> element, and the overall RAML expression is a tree of nested <expression> elements. For example, consider the expression “A ∪ B ⋈ C,” in which the order of operations is ambiguous. This cannot be encoded ambiguously, due to the way RAML is defined. The only options are to explicitly encode it as “(A union B) ⋈ C” or as “A union (B ⋈ C)”. The encodings are shown below.

```
<expression>
  <join/>
  <expression>
    <union/>
    <expression>
      <relation>
        <name>A</name>
      </relation>
    </expression><expression>
      <relation>
        <name>B</name>
      </relation>
    </expression>
  </expression><expression>
    <relation>
      <name>C</name>
    </relation>
  </expression>
</expression>
```

```
<expression>
  <union/>
  <expression>
    <relation>
      <name>A</name>
    </relation>
  </expression><expression>
    <join/>
    <expression>
      <relation>
        <name>B</name>
      </relation>
    </expression><expression>
      <relation>
        <name>C</name>
      </relation>
    </expression>
  </expression>
</expression>
```

RAML Edit

Because the functionality of the RAT framework hinges on the correctness of its input, we felt it well worth my time to develop an error-resistant means of creating RAML documents. Although one can use an XML editor or even a simple text editor to do this, such an approach is highly prone to simple human error. To solve this problem, we created a relational algebra expression editor in the style of MathType and similar mathematical equation editors, known as RAML Edit. A screenshot of its use in Google Chrome™ is shown in Fig. 11.


5345
RAT Tutorial How to?
RAT - Relational Algebra T
RAML Editor
rat.cs.panam.edu/RAT2/tools/raml_edit/1_0/index.html
RAML EDIT
R a B π σ ρ × ⋈ ⋉ ▷ ⟖ ⟕ ⟗ ∪ ∩ − ⊎ ÷ ←
R R□ R□ R□□

Choose a relation.

Relational Algebra Expression

$(\pi_{name}((Professor\bowtie(\pi_{pid}((\sigma_{dep='MATH'}(Professor)))))))-(\pi_{pid}((\sigma_{major='CS'}((Transcript\ltimes\square)))))$

RAML Expression

```
<?xml version="1.0"?>
<raml xmlns='http://www.cs.panam.edu/2010/RAML'>
    <expression>
        <set-difference/>
        <expression>
            <projection/>
            <attribute>
                <name>name</name>
            </attribute>
            <expression>
                <join/>
                <expression>
                    <relation>
                        <name>Professor</name>
                    </relation>
                </expression>
                <expression>
                    <projection/>
                    <attribute>
```

Figure 11. Screenshot of RAML Edit.

Using RAML Edit, what you see is what you get, which is why it and similar equation editors are known as WYSIWYG editors. Users have a point-and-click interface that all but removes the possibility of making syntax errors. The only time that the user is allowed to freely type anything is when specifying names of attributes or relations, or when using string literals or numbers as part of a Boolean expression. At all times during the use of RAML Edit, the equivalent RAML markup of the expression being constructed is displayed in a frame that makes up the bottom half of the interface. A working prototype of RAML Edit has been deployed at <http://rat.cs.panam.edu/RAT2/tools/raml_edit/1_0/index.html>.

When the expression is complete, users are meant to select and copy the content of the RAML markup frame, and then paste it in a text or XML editor. There is a "Save as a file" feature, but the deployed version of RAML Edit does not actually implement it. This is because the ability to save files to local storage is not allowed if done through client-side scripting, as I had originally intended. As it turns out, the ability to do this has historically been a security vulnerability [Dormann & Rafail, 2008].

Symbol	Meaning
∧	Logical AND
∨	Logical OR
¬	Logical NOT
≠	Not Equal
≤	Less or Equal
≥	Greater or Equal
	Projection
	Selection
	Renaming
×	Cross Product
⋈	Join
⋉	Semijoin
▷	Antijoin
⟕	Left Outer Join
⟖	Right Outer Join
⟗	Full Outer Join
∪	Union
∩	Intersection
−	Set Difference
⊎	Outer Union
÷	Division
←	Assignment

Table 1. RAML Font Symbols.

RAML Font

One of the main obstacles inherent in expressing relational algebra in Web pages (not to mention word processing documents) is font support. I was unable to find a lightweight, freeware font that implemented the symbols for outer joins. So I simply decided to develop my own and incorporate it into RAT. (Actually, it was originally created by Hussein Bakka; I just updated it.) The RAML Font is used to display the symbols shown in Table 1.

The RAML Font is available as a TrueType font that is configured to be installable. Interested parties may download the font from <http://rat.cs.panam.edu/RAT2/fonts/raml.ttf>. Installing the font will allow it to be rendered correctly in practically all browsers, and it can also be used in word processing documents.

Web pages that use RAT do *not* need to declare a ‘@font-face’ CSS style statement which would import the RAML Font. This is done at runtime by RAT Manager, which inserts all the necessary style statements before doing anything else. The Web browser will interpret the new markup and download the font, if it is not already locally installed. Pages that use the RAML Font but not the RAT framework should use the following CSS:

```
@font-face {
        font-family: "RAML Font";
        src:    url("http://rat.cs.panam.edu/RAT2/fonts/raml.eot?#iefix") format("eot"),
                url("http://rat.cs.panam.edu/RAT2/fonts/raml.ttf") format("truetype");
        font-style: normal;
        font-weight: normal;
}
```

RDML: Relational Database Markup Language

The Relational Database Markup Language is an XML vocabulary for expressing the structure, schema, and content of a relational database. It corresponds roughly to the data definition language of traditional RDBMSes, except that it is database-agnostic; it makes no attempt to mimic the style of any particular RDBMS. Because it is XML, it is designed to be easy both for humans to read and for programs to process. Because XML documents are well-structured plain text files, large RDML documents are highly compressible.

Schema and Specification

A copy of the XML Schema document that formally defines RDML can be found in Appendix B. Shown below is a very simple RDML document, describing a database named University that contains a single relation, named Students, whose schema specifies two attributes “name” (string) and “grade” (number) but contains no actual data tuples.

```
<rdml xmlns="http://www.cs.panam.edu/2011/RDML">
<database>
  <name>University</name>
  <relation>
    <name>Students</name>
    <schema>
      <attribute>
        <name>name</name>
        <domain>string</domain>
      </attribute>
      <attribute>
        <name>grade</name>
        <domain>number</domain>
      </attribute>
    </schema>
  </relation>
</database>
</rdml>
```

The root of an RDML document is an <rdml> element, which has at least one <database> child element. A <database> encapsulates a named collection of relations, which is the definition of a relational database. A single <database> may have any number of <relation> children, or none at all, which signifies an empty database. It also has a required <name> child.

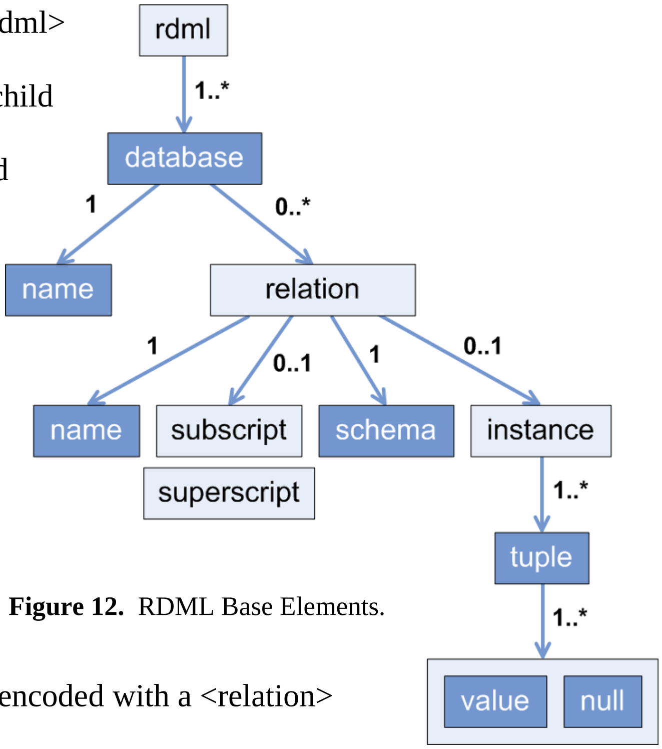


Figure 12. RDML Base Elements.

As illustrated in Fig. 12, each relation is encoded with a <relation> element, as in RAML, but <rdml:relation> is far more complex than <raml:relation>. Like the RAML version, an RDML <relation> requires its first child element to be a <name>, optionally followed by a <subscript> and/or <superscript>. This is where the similarities end. The next child element must be a <schema> element, which is so complex that it requires its own diagram (Fig. 13). The last child of a <relation> element is an optional (but expected) <instance> element.

Relation Instance and Data Tuples. The <instance> element (Fig. 12) reflects the existence of actual data in the relation. Each tuple in the relation is specified with the <tuple> element, and every <instance> must have at least one <tuple> child; this requirement ensures that if an <instance> exists, there is at least some real data in the relation. Each attribute of the relation must be accounted for in every tuple, even those that have no particular value or which are not applicable to the real-world entity that the tuple represents. If there are five attributes specified in the <schema> element, then there must always be five child elements for every <tuple> in the same relation. Each child element can be either a <value> or <null/> element, corresponding to either a data value or no content for a single attribute. A very simple example of an <instance> is shown below.

```
<instance>
   <tuple>
      <value>John Smith</value>
      <value>83.5</value>
   </tuple><tuple>
      <value>Jane Doe</value>
      <null/>
   </tuple>
</instance>
```

Relation Schema – Attributes and Constraints. The <schema> element roughly corresponds to the “CREATE TABLE” statement in SQL, defining the attributes and keys in the relation. Its complex nature is illustrated in Fig. 13.

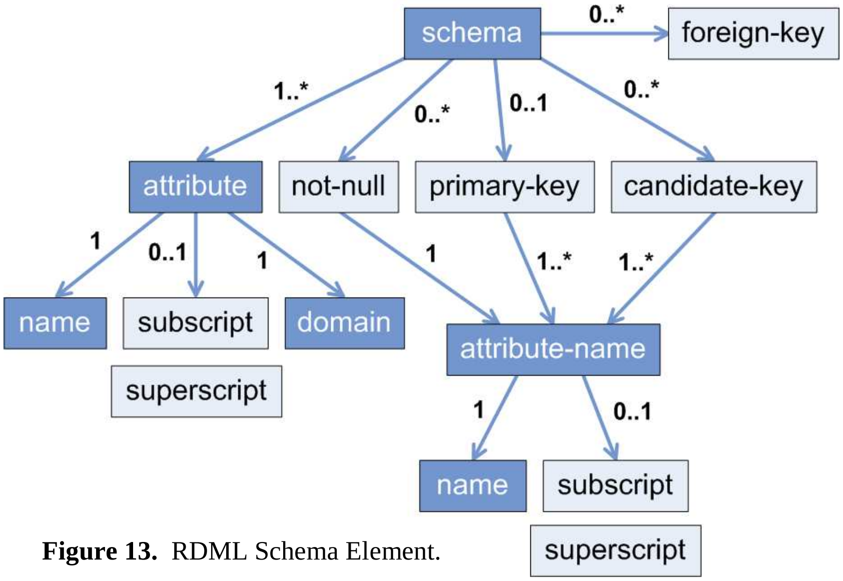


Figure 13. RDML Schema Element.

All of the <attribute> elements must be specified before anything else. Once again, there is an element in RDML that shares a name with a RAML element. They do correspond to each other, but the RDML version is slightly more complex. Each <attribute> element must have a <name> child, optionally followed by at most one <subscript> and at most one <superscript>, in that order. However, <rdml:attribute> has another element after the naming elements – a required <domain>, which specifies the data type of the values for that attribute. Applications that use RDML are free to define and enforce their own set of data types.

After all the <attribute>s have been defined under the <schema> element, its following sibling elements all make references to them. First are the <not-null> elements, of which there may be any number. Each one has exactly one <attribute-name> child (corresponding to one of its preceding sibling <attribute>s), each of which has exactly one <name> child, optionally followed by a <subscript> and/or <superscript>. The <not-null> elements are used to specify that certain attributes must always correspond to a <value> element in every tuple; never a <null/>.

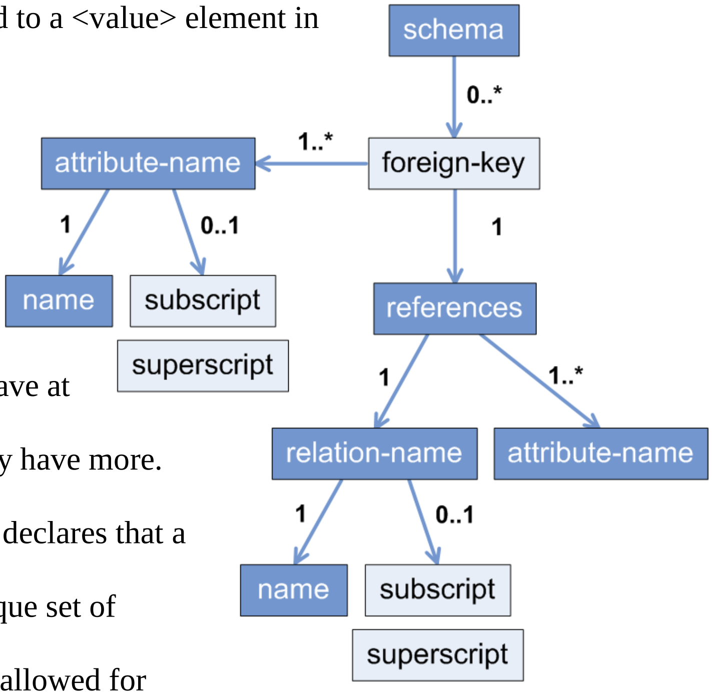


Figure 14. RDML Foreign Key.

The remaining elements all relate to key constraints. First is <primary-key>, of which there can be either zero or one. The <primary-key> element must have at least one <attribute-name> child, but may have more.

Next comes <candidate-key>, which declares that a specific set of attributes must have a unique set of <value>s (and/or <null>s, which are not allowed for any <primary-key> attribute) in every <tuple>.

There can be any number (or none) of <candidate-key>s in a relation, and an <attribute> can participate in more than one <candidate-key> as well as the <primary-key> or a <not-null>. The <candidate-key> element corresponds to the “UNIQUE” constraint of SQL.

Finally, there is <foreign-key> (Fig. 14), a set of attributes in relation R_1 such that (1) every corresponding set of values in R_1’s <instance> must also be present in the <instance> of another relation R_2, and (2) these referenced value sets in R2 must correspond to a set of attributes that form a <candidate-key> or the <primary-key> of R2. A schema may specify any number of <foreign-key>s. Each <foreign-key> of hypothetical relation R_1 specifies one or more of its own attributes (via <attribute-name>s) and – isolated inside of a <references> element – the full name of another relation R_2 (encapsulated in a <relation-name>) and one or more of R_2’s attributes. These two sets of attributes must agree in both number, respective value domains, and relative order. The RDML Validator verifies that there is the same number of <attribute-name>s as previous sibling elements of the <references> as there are as child elements of the <references>. It also verfies the existence of the referenced<attribute>s of R_1, the referenced <relation> R_2, and the referenced <attribute>s of R_2. For more information, consult Appendix D.

Creating RDML Documents

Regrettably, there is no RDML Editor to complement RAML Edit. Currently, the only way to create RDML documents is by using a text editor or – to at least avoid XML syntax errors if not RDML schema errors – an XML editor.

RATTAIL: RAT Automated Instruction Language

The managed services provided through the RAT framework need to be explicitly requested by the user. These service requests are written in a novel XML-based scripting language known as RATTAIL, the (R)elational (A)lgebra (T)oolki(T) (A)utomated (I)nstruction (L)anguage. RATTAIL is a deterministic, sequential workflow-scripting language with a very small number of instructions. It is formally defined using XML Schema.

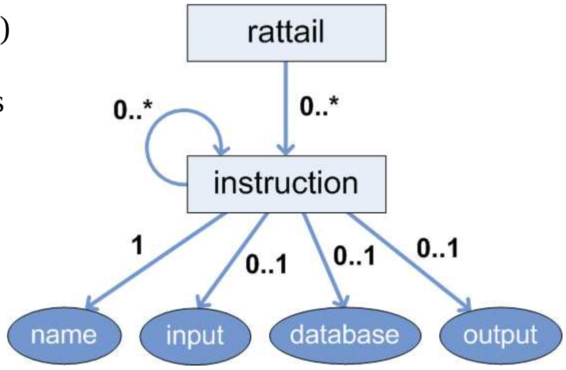


Figure 15. RATTAIL Schema.

A <rattail> element forms the root of a RATTAIL document. Its children are all <instruction> elements. Each <instruction> element has a required *name* attribute that refers unambiguously to one of the managed services, and optional *input*, *database*, and *output* attributes. Each instruction can have nested child <instruction> elements as well, making RATTAIL a highly extensible XML-based scripting language that could serve many other purposes outside of the RAT framework. It is also the simplest by far of the RAT languages (see Fig. 15). Its schema may be found in Appendix C.

The possible values for the *name* attribute of <instruction>s are restricted to “validate,” “render,” “tabulate,” “visualize,” “reorder,” “translate,” and “execute,” each corresponding to the name of a managed service.

The value of the *input* attribute must be a relative or absolute path to a RAML or RDML document. The “validate” instruction is special in that the input attribute can refer to either a RAML or RDML document, and RAT Manager will pass it to the appropriate XML validation service. The other instructions are designed to work only with either RAML or RDML specifically; “tabulate” requires RDML, whereas the others require RAML. If the *input* attribute is omitted, RAT Manager will dispatch the requested service on every RAML document that is referenced in the page in the following way:

```
<script type="application/raml+xml" src="sample.raml"></script>
```

In the given example, “sample.raml” is a RAML document, apparently located in the same directory as the Web page. Assuming the instruction currently being interpreted has the value “render” for its name attribute, RAT Manager will dispatch the Render service on that document. If the instruction is “tabulate”, RAT Manger will dispatch the Tabulate service on all RDML documents instead of all RAML documents; and if the instruction is “validate”, it will dispatch the RAML Validator on all RAML documents and the RDML Validator on all RDML.

Only when the instruction is “execute” will RAT Manager pay attention to the *database* attribute; otherwise it is ignored. It is a secondary input for the Execute service, which requires both RAML and RDML. Its value is a relative or absolute path to an RDML document. If it is omitted and the instruction is “execute”, the instruction will produce an error message.

The *output* attribute is used to specify where the results of the service request will be inserted into the Web page. Its value should correspond with the *id* attribute of some element in the Web page. The results will be wrapped in an HTML <p> element and then inserted as the immediately following sibling node of the referenced element, unless the referenced element is a <div>, in which case it will be inserted as its last child node. If there is no element found with an *id* equal to the value of the *output* attribute, a new one will be created and given that value as its *id*.

The example below shows how to encode instructions, complete with outputs.

```
<rattail xmlns="http://www.cs.panam.edu/2012/RATTAIL">
  <instruction name="visualize" input="q.raml" output="q_tree"/>
  <instruction name="translate" input="q.raml" output="q_sql"/>
  <instruction name="execute" input="q.raml" database="db.rdml"
               output="q_on_db"/>
</rattail>
```

As mentioned before, nested <instruction> elements are permitted by the RATTAIL schema, but RAT Manager will ignore them unless the parent instruction is “reorder”. The Reorder service manipulates the relational algebra it is given until it arrives at a permutation that would produce the same result if it were executed. The result is a new RAML expression, but nothing will be shown to the user unless further instructions are provided. In the following example, “q1.raml” is Reordered, and the resulting RAML is Rendered so that it can be seen.

```
<rattail>
  <instruction name="reorder" input="q1.raml" output="q1_reord">
    <instruction name="render" output="q1_reord_text"/>
  </instruction>
</rattail>
```

Suppose there are twenty equivalent permutations of “q1.raml”. The results will be embedded in twenty <p> elements whose *id*s are “q1_reord_text_1,” “q1_reord_text_2,” and so on. These twenty <p>s will be siblings of each other, and all will be the children of a new <div> whose *id* is “q1_reord”.

There can be multiples of any kind of instruction in a single RATTAIL document, which allows users to have service results be displayed at more than one location in the page. Alternatively, one can have individual RATTAIL-referencing <script> elements at each location where they want the result to be displayed, and omitting the *output* attribute. Without a value for *output*, RAT Manager will insert the result as the immediately following sibling of the <script>.

Service Descriptions

The second of the four layers of the RAT framework is comprised of the individual services being provided. In this section, each service is described in detail, except for the RAML Edit, which was described on pp.20.

Validate

One of the challenges of working with XML is ensuring that a particular sample of XML is in accordance with a particular schema. RAT provides this as a managed service, and the other services simply assume that their input is valid. If a user did not write their RAML using RAML Edit, it may be worthwhile to request the Validate service on it before using it as part of a published Web page. For RDML documents, it is always suggested that the Validate service be requested at least once before scripting other service requests on it.

```
<rattail>
  <instruction name="validate" input="q.raml" output="q_val"/>
  <instruction name="validate" input="db.rdml" output="db_val"/>
  <instruction name="execute" input="q.raml" database="db.rdml"
               output="q_on_db"/>
</rattail>
```

In the above example, “q.raml” is serviced with the RAML Validator. If the results are an empty string, then it was found to be valid; otherwise, the results are a detailed listing of the errors, which will be inserted as text into the Web page near the element whose *id* is “q_valid”. RAT Manager will flag “q.raml” as invalid input, and the “execute” instruction will not be honored. In place of service results, an error message will be displayed. The same would occur if “db.rdml” were found to be invalid by the RDML Validator.

If the “validate” instructions were not used and either the RAML or RDML input happened to be invalid, then the “execute” instruction would have been attempted and likely would have resulted in an error message that made less sense and was less helpful than the “validate” output would have been.

The reason behind making Validate its own service is threefold: in addition to being able to provide more focused and helpful error messages, automatically validating all input to a service would result in performance losses. Furthermore, if the user created their RAML documents using RAML Edit, it would be pointless to validate it because RAML Edit is incapable of producing invalid RAML.

Algorithms to support the RAML Validator can be found in Appendix D while those to support the RDML Validator are in Appendix E

Render

Database educators and relational algebra practitioners want to be able to express relational algebra on the Web. The most convenient way to do this is to have the Web browser render them as if they were part of the normal content of a Web page. Although this seems simple enough, there are two major design issues: the inconsistent handling of superscripts and subscripts between browsers, and the lackluster support for relational algebra symbols. CSS provides a fix for the former, and the RAML Font provides support for the latter (see pp.21-22).

When Render receives RAML as input, it translates the relational algebra into a conventional parenthetical expression. The intent is to produce a digital version of what the user would most likely come up with if they were to work it out on paper or a whiteboard. The result ends up being valid XHTML 1.0 markup, which RAT Manager inserts into the Web page. The Web browser then updates its rendering of the page, since the HTML source has now changed. A sample of what Render can output is shown in Fig. 16.

$$\begin{array}{l} \pi_{\text{Name}}(\text{Professor} \bowtie (\\ \qquad (\pi_{\text{Pid}}(\sigma_{\text{Dept = 'CSCI'}}(\text{Professor}))) - (\pi_{\text{Pid}}(\sigma_{\text{Major = 'MATH'}}(\text{Transcript} \bowtie \text{Student}))) \\ )) \end{array}$$

Figure 16. Render Output Sample.

I performed field testing to determine the level of Web browser support for Render. For all intents and purposes, Render enjoys universal support. Table 3 summarizes the results. The algorithms to support Render can be found in Appendix F.

Browser	**Without installing font**	**With font installed**
Google Chrome™	4.0 or later	All versions
Internet Explorer®	6 or later	6 or later
Mozilla Firefox®	3.5 or later	1.0 or later
Apple Safari®	3.1 or later	3.1 or later
Opera™ browser	10.0 or later	8.0 or later

Table 2. Browser Support for Render, by Version.

Visualize

Relational algebra expressions can be very difficult to visually analyze, even at moderate complexity. Fortunately, it is fairly straightforward to decompose them into syntax trees – as is the case with any algebraic expression – which are much less visually crowded and, for complex expressions, easier to read. Visualize allows its users to view any relational algebra expression in this less-cryptic form.

When Visualize receives RAML as input, it constructs SVG vector graphics and returns them in a form that can be dynamically embedded into a Web page. I used vector graphics instead of raster (bitmap) graphics because vector graphics are perfectly scalable – they can be shrunk or expanded with zero blurring or loss of visual information. I used SVG in particular because it is the de facto standard for vector graphics on the Web.

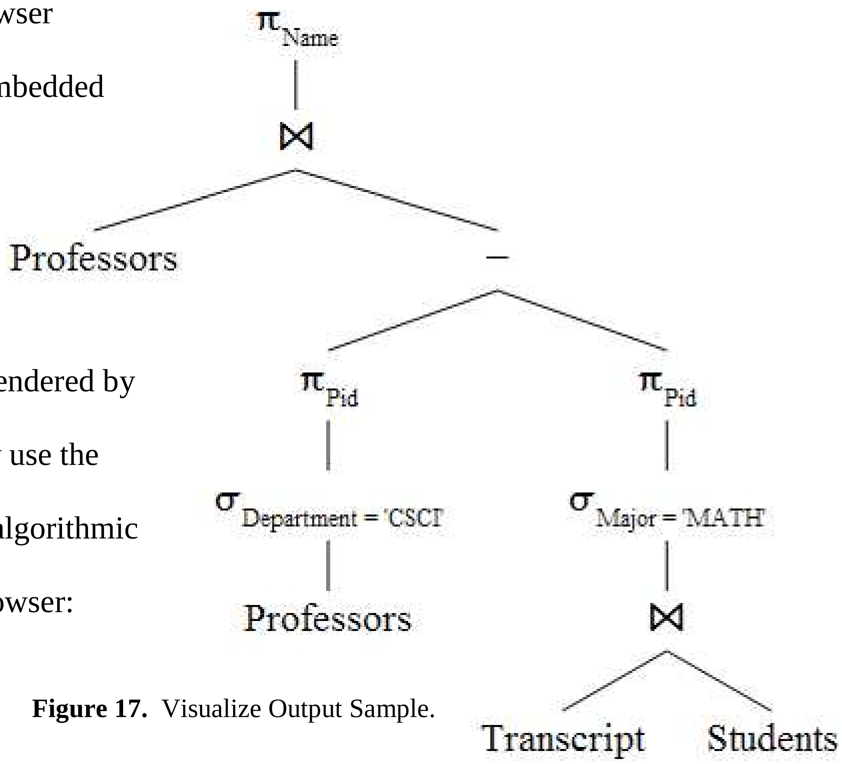


Figure 17. Visualize Output Sample.

Depending on the Web browser being used, support for SVG embedded in an (X)HTML <object> as a UTF-8-encoded data URI may be limited or nonexistent. If support is limited, the trees rendered by the browser might not correctly use the RAML Font. There is also an algorithmic issue that is unrelated to the browser: The overall width of the tree depends upon how the display width of each node is calculated. If the calculation is not very sophisticated, some trees will be far wider than they need to be. Nevertheless, the syntactic breakdown illustrated by these trees will still be correct. Table 4 summarizes the browser support for Visualize, with the focus on results being consistent. The algorithms behind Visualize may be found in Appendix G.

Browser	Without installing font	With font installed
Google Chrome™	4.0 or later	4.0 or later
Internet Explorer®	9 or later	9 or later
Mozilla Firefox®	*not supported*	3.0 or later
Apple Safari®	3.1 or later	3.1 or later
Opera™ browser	10.0 or later	9.0 or later

Table 3. Browser Support for Visualize, by Version.

Tabulate

Users of RDBMSes have come to expect their data to be presented as a table. To that end, the Tabulate service will accept an RDML document as input and render all the <relation> elements as XHTML <table>s. Because styling is expected to be highly dependent on the user, I decided to leave the styling of the output relatively bland and neutral. Table 3 shows a sample result from Tabulate.

Courses		
cid	**title**	**area**
3333	Data Structures & Algorithms	DB
3342	Internet Programming	WEB
4333	Database Design & Implementation	DB
6312	Advanced Internet Programming	WEB
6315	Applied Database Systems	DB
6333	Advanced Database Design & Implem	DB

Table 4. Tabulate Output Sample.

Translate

In order to better illustrate the connection between relational algebra and SQL, RAT provides a service called Translate which, given a RAML expression, constructs an SQL statement

```
SELECT Name
FROM Professor P
WHERE Department = 'CSCI'
  AND NOT EXISTS (
    SELECT Pid
    FROM Transcript T, Student
    WHERE Major = 'MATH'
      AND T.Pid = P.Pid
  );
```

Figure 18. Translate Output Sample.

that would give equivalent results. The relative order of the tuples in the result is not significant, and may vary between RDBMSes. A sample of Translate output is shown in Fig. 18.

Although the resulting SQL may not be optimal for any particular RDBMS, it follows a long-standing version of the ISO standard for SQL. Furthermore, RDBMSes will attempt to optimize the query anyway, which (to my amusement) would end up being done through manipulation of the underlying relational algebra.

Translate makes no attempt to optimize the query before returning a result. The main purpose of these services is to provide the tools to make teaching and learning relational algebra an easier undertaking. The ability to detect an inefficient query is certainly part of this. It is entirely up to the user to determine whether or not their query is optimal, although if they use Reorder they will likely have a much easier time doing so.

The algorithms for Translate have not yet been developed.

Reorder

One of the more tedious parts of optimizing queries by hand is coming up with permutations of that query that produce the same results. The reason for doing this is because two queries that produce the same results will require different amounts of computation and time to execute. This is what Reorder does as a service to its users, except that it produces *all* equivalent permutations.

Given a RAML expression, Reorder will manipulate the relational algebra until it arrives at a permutation that would produce results identical to those of the original expression if it were executed. It will keep doing this until it has exhausted all possibilities for manipulation.

Uniquely among RAT services, Reorder returns an array of RAML documents (actually, they are parsed DOM objects). Depending on the child instructions of the “reorder” instruction, these RAML permutations will each be piped to Render, Visualize, or Translate, or a combination of them. An example of a RATTAIL script that requests services on the permutations as well as the original permutation is shown below.

```
<rattail>
   <instruction name="visualize" output="original_tree"/>
   <instruction name="reorder" output="permuted_trees">
      <instruction name="visualize" output="equivalent_tree"/>
   </instruction>
</rattail>
```

The results of the above example will be organized in a hierarchy of XHTML <div> elements. If there are, say, three equivalent permutations of the given RAML, the results will look like this:

```
<div id="original_tree">. . .</div>
<div id="permuted_trees">
   <div id="equivalent_tree_1">. . .</div>
   <div id="equivalent_tree_2">. . .</div>
   <div id="equivalent_tree_3">. . .</div>
</div>
```

The algorithms for Reorder have not yet been developed.

Execute

Without a doubt the most interesting service is Execute, which evaluates a given RAML-encoded relational algebra query upon RDML-encoded relational data. The resulting relation is encoded as a new RDML database and automatically piped to the Tabulate service, whose results are passed back to RAT Manager.

Execute is unique in terms of its RATTAIL request, as it is the only instruction that requires a *database* attribute:

```
<rattail>
  <instruction name="execute" input="query.raml"
               database="students.rdml"
               output="query_on_students"/>
</rattail>
```

Execute is a two-stage XML interpreter that takes advantage of the fact that JavaScript has a meta-interpreter built into it – the *eval()* function. First, a modified depth-first search of the <expression> hierarchy of the given RAML input is done in order to construct a complex JavaScript function call whose parameters are the results of other function calls, each of which may branch into yet more function calls. The root-level function call is used as the right-hand side of an assignment statement to a variable that is declared in the code for this translation process (rather than being declared in the assignment statement). Once the RAML-to-function-call translation is complete, the assignment statement is passed as an argument to the *eval()* function. This executes the statement in a JavaScript interpreter. When complete, the value assigned to the local variable is retained, and we have our result, which is an RDML <relation>.

The functions that are called in the translated JavaScript are defined within the scope of the Execute service, and are invoked from within *eval()*. Each of these functions evaluates a relational algebra operation, receiving RDML as input and returning new (or transformed, in the case of selection, projection, renaming, or assignment) RDML as output.

The most basic of these functions is *load()*, which receives the string-serialized naming elements of a <raml:relation> as additional input and looks for a corresponding <rdml:relation> in its RDML input. If a match is found, *load()* returns a deep clone of the <instance> child of the matching <rdml:relation>. The set of naming elements (<name>, <subscript>, <superscript>) must be identical in both relative order and respective text content. For example, the RAML (left) and RDML (right) markup segments shown below do *NOT* refer to the same relation because the contents of their <superscript>s are not identical:

```
      <expression>                      <relation>
        <relation>                        <name>R</name>
          <name>R</name>                  <subscript>1</subscript>
          <subscript>1</subscript>        <superscript>a</superscript>
          <superscript>A</superscript>    <schema>. . .</schema>
        </relation>                       <instance>. . .</instance>
      </expression>                     </relation>
```

Regarding operations that can specify Boolean expressions as additional input (selections and non-natural joins), a similar check is performed whenever a <raml:attribute> is specified. The <schema> of the RDML input is first checked for the existence of an <rdml:attribute> with an identical set of naming elements. If none is found, the operation fails.

Chained operations like “A ⋈ B ⋈ C” are given an explicit order in the first stage; in this case, “*join*(*join*(*load*(A), *load*(B), null), *load*(C), null)”, where A and B are joined first, and the result is then joined with C. The third operand to *join()* in this example would generally be a string-serialized RAML <boolean> element, representing the join condition. Since the example is a chain of *natural* joins, there is no explicit condition.

Management Layer

Roles of RAT Manager

RAT is a service-oriented architecture – a central coordinating process waits for requests (in the case of RAT, it actively looks for them) and processes them. Each request specifies one of a collection of independent modular services. The coordinator is known as RAT Manager.

RAT Manager has several major duties: Setting up the runtime environment for the managed services, fetching all referenced documents, interpreting user-provided scripts, dispatching services as requested, and embedding the results into the Web page.

When a Web browser loads a Web page that utilizes RAT, RAT Manager is automatically invoked. The first thing that must be done is to make sure that the browser renders the results consistently. RAT Manager does this by creating a <style> element that defines a '@font-face' to import the RAML Font:

```
@font-face {
    font-family: "RAML Font";
    src:   url("http://rat.cs.panam.edu/RAT2/fonts/raml.eot?#iefix") format("eot"),
           url("http://rat.cs.panam.edu/RAT2/fonts/raml.ttf") format("truetype");
    font-style: normal;
    font-weight: normal;
}
```

Additional style rules are also included in this <style> element, mostly related to the positioning and relative font size of subscripts and superscripts. RAT Manager then inserts the <style> element into the <head> of the page.

jQuery. Because the jQuery framework is vital to the proper functioning of RAT, users who are designing a Web page that uses RAT must also import the jQuery framework, and it must be imported before "rat.js" is imported. The general idea is shown below.

```
<head>
...
<script type="text/javascript" src="jquery-1.9.1.min.js">
</script>
...
<script type="text/javascript" src="rat.js"></script>
...
</head>
```

There is no need to import the most recent version of jQuery. In fact, I recommend version 1.9.1, as this is the version that was used during all of the testing of the deployed prototype. Also, I do not recommend users to “upgrade” to jQuery 2.x unless they do not mind its lack of support for Internet Explorer® versions 6 through 8 [jQuery Foundation, 2013].

Next, RAT Manager looks for service requests. This is accomplished through the use of jQuery. All service requests must be located in RATTAIL documents, which must be referenced in the Web page via (X)HTML <script> elements *exactly* as follows, except for the underlined portion, which should be the URI of an existing RATTAIL document:

```
<script type="application/rattail+xml" src="sample.rattail">
</script>
```

RAT Manager seeks out all such <script> elements and caches the values of *src* (relative or absolute paths to RATTAIL documents) so that it can fetch and parse them.

While looking for RATTAIL documents, RAT Manager also caches the locations of all RAML and RDML documents that are referenced “stand-alone” as follows:

```
<script type="application/raml+xml" src="query.raml"></script>

<script type="application/rdml+xml" src="db.rdml"></script>
```

Through the use of Ajax, RAT Manager will have the Web browser attempt to fetch all the referenced RATTAIL documents. The “stand-alone” RAML and RDML documents are not fetched unless RAT Manager encounters an instruction that does not specify any particular input document, in which case the instruction is applied to all the RAML documents (or, if the instruction is “tabulate”, to all the RDML documents). Although Ajax is typically used for communicating transparently with a server, it may be possible – depending on the security features of the Web browser being used – to fetch documents stored locally on the user’s own computer, without needing to publish their Web page on an HTTP server. By design, the managed services of the RAT framework do not require active Internet access; they merely use the Web browser as an execution platform.

Handling Service Outputs

Once all the RATTAIL documents have been fetched and returned as parsed XML, RAT Manager will begin interpreting the scripts, instruction by instruction. For each instruction, when the requested service ever a service completes, its results are passed back to RAT Manager. RAT Manager then looks for the element in the Web page whose *id* equals the value of the instruction’s *output* attribute. If it is found and is a <div> element, the results will be inserted as its last child node. If it was found but not a <div>, the results will be inserted as its immediately following sibling node. If it was not found at all, RAT Manager will create a new <div>, assign it the specified id, inject the results into it, and insert the new <div> as the immediately following sibling node of whichever <script type="application/rattail+xml"> element referenced the RATTAIL document that contained this instruction.

In any case, the results of the service request will be inserted into the Web page, and the browser will then automatically re-render the modified portions, making the results visible to the user almost immediately. RAT Manager will patiently wait for each instruction to finish before dispatching the next one. This behavior may change in future visions of RAT, so as to benefit from parallel processing and perhaps out-of-order execution.

```
<instruction name="execute"
             input="query.raml" database="students.rdml"
             output="query_on_students"/>
```

The above instruction would be interpreted by fetching “query.raml” and “students.rdml”, then dispatching the Execute service using both as input. If either of the two documents do not happen to be valid against their respective schemas, the execution may fail. Once the service is finished, the result is passed back to RAT Manager. Since the service is Execute, the result is automatically piped to the Tabulate service, and its results are then injected into the DOM element in the Web page whose *id* equals “query_on_students”.

What are the postconditions of a Web page that uses RAT, after RAT Manager is done? First, the Web page can now use the RAML Font, whether or not there were any service requests in the page. This is of particular benefit to users who are only interested in the Render service. Second, all RATTAIL documents – provided they were properly referenced and were not invalid – have been processed, and the Web page now incorporates relational algebra in its content.

The Web page can now be saved in its current state as a new page, to avoid having RAT Manager re-process everything every time the page is loaded.

CHAPTER IV

DEPLOYMENT AND EVALUATION

Over the course of three and a half years, a partial implementation of RAT was developed with the intention of using it as an instructional aid for college database courses. As of this writing, RAT Manager, RAML Edit, the RAML and RDML Validators, and the services Render, Visualize, and Tabulate have been fully implemented. This implementation has been deployed and can be found at <http://rat.cs.panam.edu/RAT2/index.html>.

At present, the most feasible application of the Relational Algebra Toolkit is education. Working with relational algebra should not be substantially more time-consuming or frustrating than working with SQL, which I currently believe to be the case. To that end, the active deployment of RAT has been utilized in the curriculum for the graduate- and undergraduate-level Database Design and Development courses at the University of Texas – Pan American since Fall 2010, when taught by Dr. Artem Chebotko or Dr. Christine Reilly. There has also been correspondence related to RAT from professors in other universities internationally – the USA, Germany, and Malaysia, as of this writing.

Examples of Use

This section briefly describes two suggested uses of the RAT framework. The RAT website <http://rat.cs.panam.edu/RAT2/index.html> provides more information, as well as tutorials with a large number of examples.

The first step, common to most use cases, is to incorporate “rat.js” into a Web page. All of the managed RAT services – Validate, Render, Tabulate, Visualize, Reorder, Translate, and Execute – are distributed along with RAT Manager in this single JavaScript file.

One possible use of RAT is to have relational algebra expressions become part of the content of a Web page, as might occur when coordinating an educational course on databases. The suggested course of action, then, is:

1. Incorporate “rat.js” into the Web page as a <script>,
2. Use RAML Edit to obtain RAML encodings of the expression(s) in question,
3. Create XML documents out of the RAML encodings using a text or XML editor and simply copying and pasting,
4. Wherever you want each particular expression to appear in the page, place the following XHTML markup, one for each RAML document, replacing “DOCUMENT_PATH” with the path to the respective RAML document that encodes the expression:

```
<script type=”application/raml+xml" src="DOCUMENT_PATH"></script>
```

This is the simplest use case for RAT, as it does not even require the use of RATTAIL script. In the absence of RATTAIL scripts, RAT Manager will fallback to invoking the Render service on all the referenced RAML documents.

Another, more interesting use case involves the Execute service. The first three steps are the same as in the previous example, but then it becomes more complicated:

4. Create the RDML encoding of the intended sample data using a text editor.

5. Anywhere in the <body> of the page, place the following XHTML markup, one for each RAML document, replacing “DOCUMENT_PATH” with the path to the document:

```
<script type="application/raml+xml" src="DOCUMENT_PATH"></script>
```

6. Do the same for each RDML document, using the following template:

```
<script type="application/rdml+xml" src="DOCUMENT_PATH"></script>
```

7. Wherever you want the result to appear in the page, place an empty <div> element with a unique value for its *id* attribute.

8. Create an XML document containing the following RATTAIL script, replacing the values of the *input*, *database*, and *output* attributes with their intended values:

```
<?xml version="1.0" encoding="UTF-8"?>
<rattail xmlns="http://www.cs.panam.edu/2012/RATTAIL">
    <instruction name="execute" input="PATH_TO_RAML_DOCUMENT"
                 database="PATH_TO_RDML_DOCUMENT"
                 output="ID_CHOSEN_IN_STEP_7"/>
</rattail>
```

9. Anywhere in the <body> of the page, place the following XHTML markup, replacing “DOC_PATH” with the path to the RATTAIL document created in step 8:

```
<script type="application/rattail+xml" src="DOC_PATH"></script>
```

The results of the Execute service will be passed to Tabulate, and an XHTML table containing the results will be inserted into the <div> that was created in step 7.

Evaluation

In the undergraduate-level course in Fall 2013, Dr. Chebotko delivered a survey to the students to ascertain whether or not it was worthwhile to incorporate RAT into the curriculum. Out of 32 respondents, 26 had used RAT on an assignment. 16 respondents strongly agreed that "overall, the Relational Algebra Toolkit is a useful tool for learning relational algebra" (an additional 9 also agreed, though not strongly). With regard to specific services, 17 respondents strongly agreed that Visualize "helped [them] better understand [their] relational algebra expressions" (an additional 8 also agreed, though not strongly); and 17 respondents strongly agreed that RAML Edit "helped [them] learn relational algebra by only allowing [them] to enter expressions that contain correct syntax" (an additional 7 also agreed, though not strongly).

Furthermore, having used RAML Edit, 13 respondents strongly *dis*agreed that they "prefer to write relational algebra expressions by hand (on paper) as opposed to composing them using RAML Edit." An additional 4 respondents also disagreed, though not strongly, and only 1 respondent agreed, though not strongly. The remaining 8 respondents had a neutral opinion.

CHAPTER V

CONCLUSIONS

Inferring from my problem statement, I intend RAT to provide a means by which to *directly* and *conveniently* use relational algebra *as an alternative to SQL*. I have received correspondence from universities in the USA, Germany, and Malaysia that indicate RAT is being used internationally in college database courses. The deployed partial implementation of RAT does not incorporate the Execute, Translate, or Reorder services, so it remains to be seen whether or not they would improve the capability of RAT to solve the overall problem.

Nevertheless, given the survey results and the unsolicited interest in RAT from professors in universities that I had never even heard of beforehand, I deem RAT to be empirically proven useful in educational courses that cover relational algebra.

Future Work

The algorithms for the Execute service have been developed and may be found in Appendix H. However, they have not been implemented, let alone deployed on the RAT website. The algorithms for the Translate and Reorder services have not yet been developed. Complete implementation of the RAT framework would, I believe, more directly address the problem that relational algebra cannot be worked with as conveniently as SQL.

Apart from the managed services, RAT Database also needs to be implemented. Its inclusion on the RAT website would encourage communication and contribution from users, which would lead to a greater volume of constructive feedback, and ultimately the overall improvement of the entire RAT framework. It would be a user-account-controlled repository of RAML and RDML documents, perhaps even lesson plans for educational courses.

Given the empirical success of RAML Edit, I also feel that it would be greatly beneficial to develop similar editors for the creation of RDML and RATTAIL markup. Particularly important to learning the concepts of relational algebra is RAML Edit's refusal to let users create invalid relational algebra expressions. Survey respondents and former classmates who have used RAT have commented that this feature is either frustrating or enlightening, more often the latter.

Finally, I believe that RAML and RDML should support semantic content, specifically in the form of RDF-style annotations. This would provide viewers of a RAML document with a plain-English explanation of what the relational algebra expression is actually doing.

REFERENCES

Codd, E.F. (1969). *Derivability, redundancy, and consistency of relations stored in large data banks.* IBM Research Report, San Jose, California, RJ599: 1969.

Codd, E.F. (1990). *The relational model for database management.* Boston: Addison-Wesley.

Deveria, A. (2008). *Can I use...*. Retrieved 4 Dec. 2013 from http://caniuse.com/#feat=mathml

Dormann, W. & Rafail, J. (2008, Feb. 14). *Securing your web browser.* Carnegie Mellon University Software Engineering Institute. Retrieved 10 Sept. 2013 from http://www.cert.org/tech_tips/securing_browser/

jQuery Foundation (2013, January). *Browser support | jQuery*. The jQuery Foundation. Retrieved 4 Dec. 2013 from http://jquery.com/browser-support/

LaTeX Project Team (2011, July 31). *LaTeX 2e for authors*. Retrieved 4 Dec. 2013 from http://www.latex-project.org/guides/usrguide.pdf

Unicode Consortium (2002, March). *Miscellaneous Mathematical Symbols-A*. The Unicode Standard, Version 3.2.0, defined by: The Unicode Standard, Version 3.0 as amended by the *Unicode Standard Annex #28: Unicode 3.2*. Retrieved 10 Nov. 2011 from http://www.unicode.org/charts/PDF/Unicode-3.2/

World Wide Web Consortium (2008, Nov. 26). *Extensible Markup Language (XML) 1.0 (fifth edition)*. World Wide Web Consortium. Retrieved 10 Sept. 2013 from http://www.w3.org/TR/xml/#sec-starttags

W3C Math Working Group (1998, April 7). *Mathematical Markup Language (MathML) 1.0 specification*. World Wide Web Consortium. Retrieved 4 Dec. 2013 from http://www.w3.org/TR/1998/REC-MathML-19980407/

APPENDIX A

APPENDIX A

XML SCHEMA FOR RAML

```
<xs:schema targetNamespace="http://www.cs.panam.edu/2010/RAML"
           xmlns="http://www.cs.panam.edu/2010/RAML"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">

    <xs:complexType name="emptytype">
        <xs:restriction base="xs:string">
            <xs:maxLength value="0"/>
        </xs:restriction>
    </xs:complexType>

    <xs:complexType name="relationtype">
        <xs:sequence>
            <xs:choice>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="nm" type="xs:string"/>
            </xs:choice>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="subscript" type="xs:string"/>
                <xs:element name="sub" type="xs:string"/>
            </xs:choice>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="superscript" type="xs:string"/>
                <xs:element name="sup" type="xs:string"/>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="attrtype">
        <xs:sequence>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="relation" type="relationtype"/>
                <xs:element name="r" type="relationtype"/>
            </xs:choice>
            <xs:choice>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="nm" type="xs:string"/>
            </xs:choice>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="subscript" type="xs:string"/>
                <xs:element name="sub" type="xs:string"/>
            </xs:choice>
            <xs:choice minOccurs="0" maxOccurs="1">
                <xs:element name="superscript" type="xs:string"/>
                <xs:element name="sup" type="xs:string"/>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="attribute-pairtype">
        <xs:sequence>
            <xs:choice minOccurs="2" maxOccurs="2">
                <xs:element name="attribute" type="attrtype"/>
                <xs:element name="a" type="attrtype"/>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="booleantype">
        <xs:choice>
            <xs:choice>
                <xs:element name="true" type="emptytype"/>
                <xs:element name="t" type="emptytype"/>
                <xs:element name="false" type="emptytype"/>
                <xs:element name="f" type="emptytype"/>
            </xs:choice>
            <xs:sequence>
                <xs:choice>
                    <xs:element name="lt" type="emptytype"/>
                    <xs:element name="lte" type="emptytype"/>
                    <xs:element name="gt" type="emptytype"/>
                    <xs:element name="gte" type="emptytype"/>
                    <xs:element name="eq" type="emptytype"/>
                    <xs:element name="neq" type="emptytype"/>
                </xs:choice>
                <xs:choice minOccurs="2" maxOccurs="2">
                    <xs:element name="number" type="xs:double"/>
                    <xs:element name="num" type="xs:double"/>
                    <xs:element name="string" type="xs:string"/>
                    <xs:element name="str" type="xs:string"/>
                    <xs:element name="attribute" type="attrtype"/>
                    <xs:element name="a" type="attrtype"/>
                    <xs:element name="true" type="emptytype"/>
                    <xs:element name="t" type="emptytype"/>
                    <xs:element name="false" type="emptytype"/>
                    <xs:element name="f" type="emptytype"/>
                    <xs:element name="null" type="emptytype"/>
                    <xs:element name="n" type="emptytype"/>
                </xs:choice>
            </xs:sequence>
            <xs:sequence>
                <xs:choice>
                    <xs:element name="is-null" type="emptytype"/>
                    <xs:element name="isn" type="emptytype"/>
                    <xs:element name="is-not-null" type="emptytype"/>
                    <xs:element name="isnn" type="emptytype"/>
                </xs:choice>
            <xs:choice>
                <xs:element name="number" type="xs:double"/>
                <xs:element name="num" type="xs:double"/>
                <xs:element name="string" type="xs:string"/>
                <xs:element name="str" type="xs:string"/>
                <xs:element name="attribute" type="attrtype"/>
                <xs:element name="a" type="attrtype"/>
                <xs:element name="true" type="emptytype"/>
                <xs:element name="t" type="emptytype"/>
                <xs:element name="false" type="emptytype"/>
                <xs:element name="f" type="emptytype"/>
                <xs:element name="null" type="emptytype"/>
                <xs:element name="n" type="emptytype"/>
            </xs:choice>
        </xs:sequence>
        <xs:sequence>
            <xs:element name="not" type="emptytype"/>
            <xs:choice>
                <xs:element name="boolean" type="booleantype"/>
                <xs:element name="bn" type="booleantype"/>
            </xs:choice>
        </xs:sequence>
        <xs:sequence>
            <xs:choice>
                <xs:element name="or" type="emptytype"/>
                <xs:element name="and" type="emptytype"/>
            </xs:choice>
            <xs:choice minOccurs="2" maxOccurs="unbounded">
                <xs:element name="boolean" type="booleantype"/>
                <xs:element name="bn" type="booleantype"/>
            </xs:choice>
        </xs:sequence>
    </xs:choice>
</xs:complexType>

<xs:complexType name="exprtype">
    <xs:choice>
        <xs:choice>
            <xs:element name="relation" type="relationtype"/>
            <xs:element name="r" type="relationtype"/>
        </xs:choice>
        <xs:sequence>
            <xs:choice>
                <xs:element name="projection" type="emptytype"/>
                <xs:element name="pn" type="emptytype"/>
            </xs:choice>
            <xs:choice minOccurs="1" maxOccurs="unbounded">
                <xs:element name="attribute" type="attrtype"/>
                <xs:element name="a" type="attrtype"/>
            </xs:choice>
            <xs:choice>
                <xs:element name="expression" type="exprtype"/>
                <xs:element name="exp" type="exprtype"/>
            </xs:choice>
        </xs:sequence>
```

```
<xs:sequence>
    <xs:choice>
        <xs:element name="selection" type="emptytype"/>
        <xs:element name="sn" type="emptytype"/>
    </xs:choice>
    <xs:choice>
        <xs:element name="boolean" type="booleantype"/>
        <xs:element name="bn" type="booleantype"/>
    </xs:choice>
    <xs:choice>
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="renaming" type="emptytype"/>
        <xs:element name="rg" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element name="attribute-pair"
                    type="attribute-pairtype"/>
        <xs:element name="ap" type="attribute-pairtype"/>
    </xs:choice>
    <xs:choice>
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="cross-product" type="emptytype"/>
        <xs:element name="cp" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="join" type="emptytype"/>
        <xs:element name="jn" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element name="boolean" type="booleantype"/>
        <xs:element name="bn" type="booleantype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
```

```
        <xs:element name="semi-join" type="emptytype"/>
        <xs:element name="sj" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="anti-join" type="emptytype"/>
        <xs:element name="aj" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="left-join" type="emptytype"/>
        <xs:element name="lj" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element name="boolean" type="booleantype"/>
        <xs:element name="bn" type="booleantype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="right-join" type="emptytype"/>
        <xs:element name="rj" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element name="boolean" type="booleantype"/>
        <xs:element name="bn" type="booleantype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="full-join" type="emptytype"/>
        <xs:element name="fj" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element name="boolean" type="booleantype"/>
        <xs:element name="bn" type="booleantype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="union" type="emptytype"/>
        <xs:element name="un" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="intersection" type="emptytype"/>
        <xs:element name="in" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="set-difference"
                    type="emptytype"/>
        <xs:element name="sd" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="outer-union" type="emptytype"/>
        <xs:element name="ou" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
</xs:sequence>
<xs:sequence>
    <xs:choice>
        <xs:element name="division" type="emptytype"/>
        <xs:element name="dn" type="emptytype"/>
    </xs:choice>
    <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element name="expression" type="exprtype"/>
        <xs:element name="exp" type="exprtype"/>
    </xs:choice>
```

```
            </xs:sequence>
            <xs:sequence>
                <xs:choice>
                    <xs:element name="assignment" type="emptytype"/>
                    <xs:element name="at" type="emptytype"/>
                </xs:choice>
                <xs:choice>
                    <xs:element name="relation" type="relationtype"/>
                    <xs:element name="r" type="relationtype"/>
                </xs:choice>
                <xs:choice>
                    <xs:element name="expression" type="exprtype"/>
                    <xs:element name="exp" type="exprtype"/>
                </xs:choice>
            </xs:sequence>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="ramltype">
        <xs:sequence>
            <xs:choice minOccurs="1" maxOccurs="unbounded">
                <xs:element name="expression" type="exprtype"/>
                <xs:element name="exp" type="exprtype"/>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>

    <xs:element name="raml" type="ramltype"/>

</xs:schema>
```

APPENDIX B

APPENDIX B

XML SCHEMA FOR RDML

```
<xs:schema targetNamespace="http://www.cs.panam.edu/2011/RDML"
           xmlns="http://www.cs.panam.edu/2011/RDML"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">

    <xs:simpleType name="emptytype">
        <xs:restriction base="xs:string">
            <xs:maxLength value="0"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="tupletype">
        <xs:choice minOccurs="1" maxOccurs="unbounded">
            <xs:element name="value" type="xs:string"/>
            <xs:element name="null" type="emptytype"/>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="instancetype">
        <xs:sequence>
            <xs:element name="tuple" type="tupletype"
                        minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:group name="namegrp">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="subscript" type="xs:string"
                        minOccurs="0" maxOccurs="1"/>
            <xs:element name="superscript" type="xs:string"
                        minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:group>

    <xs:complexType name="attrtype">
        <xs:sequence>
            <xs:group ref="namegrp"/>
            <xs:element name="domain" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
```

```
<xs:complexType name="reftype">
    <xs:group ref="namegrp"/>
</xs:complexType>

<xs:complexType name="foreign-keytype">
    <xs:sequence>
        <xs:element name="attribute-name" type="reftype"
                    minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="references">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="relation-name"
                                type="reftype"/>
                    <xs:element name="attribute-name"
                                type="reftype" minOccurs="1"
                                maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="schematype">
    <xs:sequence>
        <xs:element name="attribute" type="attrtype"
                    minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="not-null"
                    minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:element name="attribute-name" type="reftype"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="primary-key" minOccurs="0" maxOccurs="1">
            <xs:complexType>
                <xs:element name="attribute-name" type="reftype"
                            minOccurs="1" maxOccurs="unbounded"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="candidate-key"
                    minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:element name="attribute-name" type="reftype"
                            minOccurs="1" maxOccurs="unbounded"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="foreign-key" type="foreign-keytype"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

```
    <xs:complexType name="relationtype">
        <xs:sequence>
            <xs:group ref="namegrp"/>
            <xs:element name="schema" type="schematype"/>
            <xs:element name="instance" type="instancetype"
                        maxOccurs="1" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="databasetype">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="relation" type="relationtype"
                        minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="rdmltype">
        <xs:sequence>
            <xs:element name="database" type="databasetype"
                        minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:element name="rdml" type="rdmltype"/>

</xs:schema>
```

APPENDIX C

APPENDIX C

XML SCHEMA FOR RATTAIL

```
<xs:schema xmlns="http://www.cs.panam.edu/2012/RATTAIL"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.cs.panam.edu/2012/RATTAIL"
      elementFormDefault="qualified"
      attributeFormDefault="qualified">

     <xs:annotation><xs:documentation>
RATTAIL – (R)elational (A)lgebra (T)oolki(T)
          (A)utomated (I)nstruction (L)anguage
     </xs:documentation></xs:annotation>

     <xs:complexType name="instrtype">
         <xs:attribute name="name" use="required">
              <xs:simpleType>
                  <xs:restriction base="xs:string">
                       <xs:enumeration value="validate"/>
                       <xs:enumeration value="render"/>
                       <xs:enumeration value="visualize"/>
                       <xs:enumeration value="reorder"/>
                       <xs:enumeration value="translate"/>
                       <xs:enumeration value="execute"/>
                       <xs:enumeration value="data-render"/>
                  </xs:restriction>
              </xs:simpleType>
         </xs:attribute>
         <xs:attribute name="input" type="xs:anyURI"
                       use="optional"/>
         <xs:attribute name="database" type="xs:anyURI"
                       use="optional"/>
         <xs:attribute name="output" type="xs:ID" use="optional"/>
     </xs:complexType>

     <xs:complexType name="scripttype">
         <xs:sequence>
              <xs:element name="instruction" type="instrtype"
                          minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
     </xs:complexType>

     <xs:element name="rattail" type="scripttype"/>

</xs:schema>
```

APPENDIX D

APPENDIX D

ALGORITHMS FOR RAML VALIDATOR

```
algorithm RAML Validator
input (src): DOM element (a parsed RAML document)
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is a <raml> element then
        error_string = “”
        for each child element kid of src do
            error_string += validate_expression(kid)
        end for each
    end if
    return “\nRoot element must be <raml>.”
end
```

```
algorithm validate_expression
input (src): DOM element
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is not an <expression> element then
        return “\nExpected <expression> in place of ‘” + tag name of src + “’.”
    end if
    firstborn = first child element of src
    if firstborn does not exist then
        return “\n(expression): Requires a child element.”
    end if
    switch the element type of firstborn:
        case <relation>:
            return validate_relation_raml(firstborn)
        case <join>, <left-join>, <right-join>, <full-join>:
            return validate_op_conditional(src)
        case <cross-product>, <union>, <intersection>, <set-difference>, <division>,
                <semi-join>, <anti-join>, <outer-union>:
            return validate_op_unconditional(src)
        case <selection>:       return validate_selection(src)
        case <projection>:      return validate_projection(src)
        case <renaming>:        return validate_renaming(src)
        case <assignment>:      return validate_assignment(src)
        case else:
            return “\n(expression): Invalid child element ‘” + tag name of firstborn + “’.”
    end switch
    end if
end
```

algorithm validate_relation_raml
input (*src***):** DOM element whose tag name is “relation” or “r”
output: A string listing the syntax errors in *src*. The string will be empty if *src* is valid.

```
begin
    if src has no child elements or if first child element is not a <name> then
        return “\n(relation): First child must be <name>.”
    end if
    return validate_name(child elements of src)
end
```

algorithm validate_name
input (*src***):** array of DOM elements, the first of which has tag name “name” or “nm”
output: A string listing the syntax errors in *src*. The string will be empty if *src* is valid.

```
begin
    next_index = 1
    if src contains more than one element then
        if src[1] is a <subscript> then
            if src contains more than two elements then
                next_index = 2
            else
                return “”
            end if
        end if
        if src[next_index] is a <superscript> then
            if src contains more than (next_index+1) elements then
                return “(“ + tag name of src[next_index] + “): Cannot be followed by a ‘”
                    + tag name of src[next_index+1] + “’.”
            end if
            return “”
        end if
    end if
    return “”
end
```

```
algorithm validate_assignment
input (src): DOM element with tag name “expression” or “exp”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has more than 3 child elements then
        return “\n(assignment): Siblings must be a <relation> and an <expression>.”
    end if
    error_string = validate_relation(second child element of src)
    error_string += validate_expression(third child element of src)
    return error_string
end
```

```
algorithm validate_selection
input (src): DOM element with tag name “expression” or “exp”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has more than 3 child elements then
        return “\n(selection): Siblings must be a <boolean> and an <expression>.”
    end if
    error_string = validate_boolean(second child element of src)
    error_string += validate_expression(third child element of src)
    return error_string
end
```

```
algorithm validate_projection
input (src): DOM element with tag name “expression” or “exp”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has fewer than 3 child elements then
        return “\n(projection): Siblings must be <attribute>s and an <expression>.”
    end if
    error_string = “”
    for each child element att of src except the first and last ones do
        if att is not an <attribute> then
            return error_string + “\n(projection): Expected <attribute> in place of ‘”
                + tag name of att + “’.”
        end if
        error_string += validate_attribute(att)
    end for each
    if last child element of src is not an <expression> then
        error_string += “(projection): Last child must be an <expression>.”
    else
```

```
        error_string += validate_expression(last child element of src)
    end if
    return error_string
end
```

```
algorithm validate_renaming
input (src): DOM element with tag name “expression” or “exp”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has fewer than 3 child elements then
        return “\n(renaming): Siblings must be <attribute-pair>s and an <expression>.”
    end if
    error_string = “”
    for each child element attpair of src except the first and last ones do
        if attpair is not an <attribute-pair> then
            return error_string + “\n(renaming): Expected <attribute-pair> in place of ‘”
                + tag name of attpair + “’.”
        end if
        error_string += validate_attribute_pair(att)
    end for each
    if last child element of src is not an <expression> then
        error_string += “(renaming): Last child must be an <expression>.”
    else
        error_string += validate_expression(last child element of src)
    end if
    return error_string
end
```

```
algorithm validate_attribute_pair
input (src): DOM element with tag name “attribute-pair” or “ap”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src does not have exactly 2 child elements then
        return “\n(attribute-pair): Must have exactly two <attribute> children.”
    end if
    error_string = validate_attribute(first child element of src)
    error_string += validate_attribute(second child element of src)
    return error_string
end
```

```
algorithm validate_attribute
input (src): DOM element
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is not an <attribute> then
        return “\n(” + tag name of src + “): Expected <attribute>.”
    else if src does not have at least one child element then
        return “\n(attribute): Requires a <name>.”
    end if
    error_string = “”
    kids = child elements of src
    if kids[0] is a <relation> then
        error_string += validate_relation(kids[0])
        kids = all child elements of src except the first one
    end if
    if kids is empty or if kids[0] is not a <name> then
        error_string += “\n(attribute): Requires a <name>.”
    else
        error_string += validate_name(kids)
    end if
    return error_string
end
```

```
algorithm validate_op_unconditional
input (src): DOM element with tag name “expression” or “exp”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has fewer than 3 child elements then
        return “\n(” + tag name of first child of src + “): Requires at least 2 <expression>s.”
    end if
    error_string = “”
    for each child element expr of src except the first one do
        if expr is not an <expression> then
            return “\n(” + tag name of first child of src
                + “): Siblings must all be <expression>s.”
        end if
        error_string += validate_expression(expr)
    end for each
    return error_string
end
```

```
algorithm validate_op_conditional
input (src): DOM element with tag name “expression” or “exp”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has fewer than 3 child elements then
        return “\n(” + tag name of first child element of src
            + “): Requires at least two <expression>s.”
    end if
    error_string = “”
    offset = 2
    if second element child of src is a <boolean> then
        error_string += validate_boolean(second child element of src)
        if src has fewer than (offset + 2) child elements then
            return error_string + “\n(” + tag name of first child element of src
                + “): Requires at least two <expression>s.”
        end if
    else
        offset = 1
    end if
    for i from offset to number of child elements of src do
        if the i-th child element of src is not an <expression> then
            return “\n(” + tag name of first child element of src
                + “): Siblings must all be <expression>s.”
        end if
        error_string += validate_expression(expr)
    end for
end
```

```
algorithm validate_simple_boolean
input (src): DOM element with tag name “true”, “false”, “null”, “string”, “number”, “attribute”
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src has child elements then
        return “\n(” + tag name of src + “): Cannot have child elements.”
    end if
    switch tag name of src:
        case <true>, <false>, <null>, <string>, <number>:
            return “”
        case <attribute>:  return validate_attribute(src)
        case else:
            return “\n(” + tag name of src
                + “): Expected <true>, <false>, <null>, <string>, <number>, or <attribute>.”
    end switch
    return error_string
end
```

```
algorithm validate_boolean
input (src): DOM element
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is not a <boolean> then
        return "\n(" + tag name of src + "): Expected <boolean>."
    end if
    firstborn = first child element of src
    switch number of child elements of src:
        case 1:
            if firstborn is not a <true>, <false>, or <null> then
                return "\n(boolean): Invalid <boolean> beginning with '"
                    + tag name of firstborn + "'."
            else
                return validate_simple_boolean(firstborn)
            end if
        case 2:
            if firstborn is a <not> then
                if second child element of src is not a <boolean> then
                    return "\n(not): Sibling must be a <boolean>."
                else
                    return validate_boolean(second child element of src)
                end if
            else if firstborn is an <is-null> or <is-not-null> then
                error_string = validate_simple_boolean(second child element of src)
                if error_string is not empty then
                    return "\n(" + tag name of firstborn
                        + "): Expected <attribute> in place of '"
                        + tag name of second child element of src + "'."
                else
                    return ""
                end if
            else
                return "\n(boolean): Invalid <boolean> beginning with '"
                     + tag name of firstborn + "'."
            end if
        case 3:
            if firstborn is <eq>, <neq>, <lt>, <lte>, <gt>, or <gte> then
                error_string = validate_simple_boolean(second child element of src)
                error_string += validate_simple_boolean(third child element of src)
            else if firstborn is <and> or <or> then
```

```
                error_string = validate_boolean(second child element of src)
                error_string += validate _boolean(third child element of src)
            else
                error_string = “\n(boolean): Invalid <boolean> beginning with ‘”
                    + tag name of firstborn + “’.”
            end if
            return error_string
        case else:
            if firstborn is <and> or <or> then
                error_string = “”
                for each child element kid of src except the first one do
                    error_string += validate _boolean(kid)
                end for
            else
                error_string = “\n(boolean): Invalid <boolean> beginning with ‘”
                    + tag name of firstborn + “’.”
            end if
            return error_string
    end switch
    return error_string
end
```

APPENDIX E

APPENDIX E

ALGORITHMS FOR RDML VALIDATOR

```
algorithm RDML Validator
input (src): DOM element (a parsed RDML document)
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is an <rdml> element then
        error_string = “”
        for each child element kid of src do
            error_string += validate_database(kid)
        end for each
    end if
    return “\nRoot element must be <rdml>.”
end
```

```
algorithm validate_database
input (src): DOM element
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is not a <database> element then
        return “\nExpected <database> in place of ‘” + tag name of src + “’.”
    else if src has no child elements or if first child element is not a <name> then
        return “\n(database): Requires a <name>.”
    end if
    error_string = “”
    relation_names = empty array
    for each child element kid of src except the first one do
        error_string += validate_relation_attribute(kid, false, relation_names)
    end for each
end
```

```
algorithm validate_relation_attribute
input (src): DOM element whose tag name is “relation” or “attribute”
input (isAttrNotReln): Boolean flag; true if src is an “attribute”, false if it is a “relation”
input (familyNames): array of the names of all <relation>s encountered so far in this <database>,
   or of all the <attribute>s encountered so far in this <schema>. On exit, will contain another
   element, but only if the name of src is unique.
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if !isAttrNotReln and if src is not a <relation> then
        return “\nExpected <relation> in place of ‘” + tag name of src + “’.”
    end if
    // If we are validating an attribute, by precondition we already know src is an <attribute>.
    if isAttrNotReln then
        this_name = “attribute”
        this_needs = “domain”
    else
        this_name = “relation”
        this_needs = “schema”
    end if

    // Locate the <domain> or <schema> sibling.
    if src has fewer than 2 child elements then
        return “\n(” + this_name + “): Requires a <name> and a <” + this_needs + “>.”
    else if first child element of src is not a <name> then
        return “\n(” + this_name + “): First child must be <name>.”
    end if

    if src has more than 2 child elements then
        if second child element of src is a <subscript> then
            if src has more than 3 child elements then
                if third child element of src is a <superscript> then
                    need_index = 4
                else need_index = 3
                end if
            end if
        else if second child element of src is a <superscript> then
            need_index = 3
        else need_index = 2
        end if
    else need_index = 2
    end if
    if this_needs != tag name of need_index-th child element of src then
        return “\n(” + this_name + “): Children must be: <name> (required), <subscript> ”
            + “(optional), <superscript> (optional), <” + this_needs + “> (required).”
    end if
    // At this point we know where the required <schema> or <domain> are.
    if !isAttrNotReln then
        error_string = validate_schema(need_index-th child element of src)
        if src has more than need_index child elements then
            error_string += validate_instance((need_index+1)-th child element of src)
        end if
        if error_string is not empty then
            return error_string
        end if
    end if

    // Serialize the full name of this <relation> or <attribute>.
    namingElem = first child element of src
    myName = “<name>” + text content of namingElem + “</name>”
    for i from 0 to need_index do
        namingElem = next sibling element of namingElem
        myName += “<” + tag name of namingElem + “>”
        myName += text content of namingElem
        myName += “</” + tag name of namingElem + “>”
    end for

    // Ensure that the full name is not already present in familyNames.
    for each string cand in familyNames do
        if myName = cand then
            return “\nFound duplicate ” + this_name + “ name ” + myName + “!”
        end if
    end for each
    // This name does not exist in familyNames, so it is valid.
    Push myName onto familyNames
end
```

```
algorithm validate_instance
input (src): DOM element
output: A string listing the syntax errors in src. The string will be empty if src is valid.
begin
    if src is not an <instance> then
        return “\n(relation): Expected <instance> in place of ‘” + tag name of src + “’.”
    else if src does not have child elements then
        return “\n(instance): Must have at least one <tuple> child.”
    end if
    for each child element tup of src do
        if tup is not a <tuple> then
            return “(instance): Children must all be &lt;tuple&gt;s.”
        end if
        for each child element val of tup do
            if val is not a <value> or <null> then
                return “\n(tuple): Children can only be <value> or <null>.
            end if
        end for each
    end for each
    return error_string
end
```

algorithm validate_schema
input (*src***):** DOM element whose tag name is “schema”
output: A string listing the syntax errors in *src*. The string will be empty if *src* is valid.

```
begin
    child = first element child of src
    error_string = “”
    // First handle the attributes.
    attNames = empty array
    while child is an <attribute> do
        error_string += validate_relation_attribute(child, true, attNames)
        child = next element sibling of child
    end while
    if child is same node as first element child of src then
        return “\n(schema): Requires at least one <attribute>.”
    else if child is null then
        return error_string
    end if

    // Now handle the <not-null>s.
    while child is a <not-null> do
        error_string += validate_constraint(child, attNames)
        child = next element sibling of child
    end while
    if child is null then
        return error_string
    end if

    // Now handle the <not-null>s.
    while child is a <not-null> do
        error_string += validate_constraint(child, attNames)
        child = next element sibling of child
    end while
    // Now handle the <primary-key>, if any.
    if child is a <primary-key> then
        error_string += validate_constraint(child, attNames)
        child = next element sibling of child
    end if
    if child is null then
        return error_string
    end if

    // Now handle the <candidate-key>s.
    while child is a <candidate-key> do
        error_string += validate_constraint(child, attNames)
        child = next element sibling of child
    end while
    if child is null then
        return error_string
    end if

    // Now handle the <foreign-key>s.
    while child is a <foreign-key> do
        error_string += validate_foreign_key(child, attNames)
        child = next element sibling of child
    end while

    if child is not null then
        return "\n(schema): Extra elements found after <foreign-key>s."
    end if
end
```

algorithm validate_attrOrReln_name
input (*src***):** DOM element
input (*isAttrNotReln***):** Boolean flag; true if *src* is an “attribute-name”, false if a “relation-name”
output: A serialized version of the <attribute-name> or <relation-name>, if valid.
If not valid, output is an empty string.

```
begin
    if src has no child elements or if first child element of src is not a <name> then
        return “”
    end if
    fullName = “<name>” + text content of first child element of src + “</name>”
    if src has more than 1 child element then
        if second child element of src is a <superscript> then
            if src has more than 2 child elements then
                return “”
            end if
            fullName += “<superscript>” + text content of second child element of src
                + “</superscript>”
        else if second child element of src is a <subscript> then
            fullName += “<subscript>” + text content of second child element of src
                + “</subscript>”
            if src has more than 2 child elements then
                if third child element of src is not a <superscript>
                  or if src has more than 3 child elements then
                    return “”
                end if
                fullName += “<superscript>” + text content of third child element of src
                    + “</superscript>”
            end if
        else
            return “”
        end if
    end if
    return fullName
end
```

```
algorithm validate_constraint
input (src): DOM element with tag name “not-null”, “primary-key”, “candidate-key”
input (validNames): array of the names of all <attributes>s declared in this <schema>.
  This function will make no changes to the array.
output: A string listing the syntax errors in src. The string will be empty if src is valid.
```

```
begin
    if src has no child elements then
        return “\n(” + tag name of src + “): Requires an <attribute-name>.”
    else if src is a <not-null> and if src has more than 1 child element then
        return “\n(not-null): Can only be applied to one attribute at a time.”
    end if

    for each child element attref of src do
        if attref is not an <attribute-name> do
            return “\n(” + tag name of src + “): Children must all be <attribute-names>.”
        end if
        myName = validate_attOrReln_name(attref, true)
        if myName is empty then
            return “\n(”+ tag name of src + “): Invalid <attribute-name> child.”
        end if
        // Check validNames for the existence of myName
        found = false
        for each string name in validNames do
            if name = myName then
                found = true
                break for each
            end if
        end for each
        if !found then
            return “\n(” + tag name of src + “): Referenced attribute ”
                + myName + “ not found in <schema>.”
        end if
    end for each
    return error_string
end
```

```
algorithm validate_foreign_key
input (src): DOM element with tag name “foreign-key”
input (validNames): array of the names of all <attributes>s declared in this <schema>.
  This function will make no changes to the array.
output: A string listing the syntax errors in src. The string will be empty if src is valid.
----------------------------------------------------------------------------------------
begin
    if src has fewer than 2 child elements then
        return “\n(foreign-key): Requires at least one <attribute-name> and a <references>.”
    end if
    error_string = “”
    for each child element kid of src do
        if kid is an <attribute-name> then
            break for each
        end if
        attName = validate_attOrReln_name(kid, true)
        if attName is empty then
            // Validation failed
            error_string += “\n(foreign-key): Invalid <attribute-name> child.”
        else
            // Search for the existence of the named <attribute>.
            found = false
            for each string name in validNames do
                if name = attName then
                    found = true
                    break for each
                end if
            end for each
            if !found then
                error_string += “\n(foreign-key): Referenced attribute ”
                    + attName + “ not found in <schema>.”
                break for each
            end if
        end if
    end for each
```

```
if error_string is not empty then
    return error_string
end if
// All referencING attributes exist in the schema.

// Now process the <references>.
references = last child element of src
if references is not a <references> then
    return “\n(foreign-key): Last child must be a <references>.”
else if first element child of references is not a <relation-name> then
    return “\n(references): First child must be a <relation-name>.”
end if
refRelName = validate_attOrReln_name(first element child of references, false)
if refRelName is empty then
    return “\n(references): Invalid <relation-name> child.”
end if

// Locate the referenced relation.
remoteRelation = null
for each <relation> child reln of the <database> great-grandparent of src do
    relnName = serialization of the naming elements of reln
    // relnName has a value like “<name>R</name><subscript>1</subscript>”
    if relnName = refRelName then
        remoteRelation = reln
        break for each
    end if
end for each
if remoteRelation is null then
    return “\n(foreign-key): Referenced relation ” + refRelName
        + “ could not be found in ancestor <database>.”
end if

// Now obtain the <attribute> children of the referenced relation.
remoteSchema = first (and hopefully only) <schema> child of remoteRelation
remoteAtts = all <attribute> child elements of remoteRelation

// Iterate through the referencED attribute children of the <references>.
for each child element attr of references except the first one do
    if attr is not an <attribute-name> then
        return “\n(references): Children must be a <relation-name> ”
            + “followed by one or more <attribute-name>s.”
    else
        refAttName = validate_attOrReln_name(attr, true)
        if refAttName is empty then
            return “\n(references): Invalid <attribute-name> child.”
            end if

            // Locate the referenced attribute in the referenced relation.
            found = false
            for each <attribute> child remAtt of remoteSchema do
                remAttName = serialization of the naming elements of remAtt
                // remAttName has a value like “<name>title</name>”
                if remAttName = refAttName then
                    found = true
                    break for each
                end if
            end for each
            if !found then
                return “\n(foreign-key): Referenced attribute ” + refAttName
                    + “ could not be found in referenced <relation>.”
            end if
        end if
    end for each
    return “”
end
```

APPENDIX F

APPENDIX F

ALGORITHMS FOR RENDER

```
algorithm Render
input (src): DOM element
output: A string that can be parsed as XHTML.
begin
    res = null
    try do
        // Call the root-level subprocedure.
        res = rend_root(src)
    end try
    catch any error do
        res = "Render failed: " + thrown error message
    end catch
    return res
end
```

```
algorithm rend_root
input (src): DOM element, expected to have tag name "raml"
output: A string that can be parsed as XHTML.
begin
    str = '�' // Placeholder error-indicating character
    if src has any <expression> child elements do
        str = rend_expression(first <expression> child element of src, false)
        for each remaining <expression> child kid of src do
            str += "<br /><br />"
            str += rend_expression(kid, false)
        end for each
    end if
    return str
end
```

```
algorithm rend_expression
input (src): DOM element, expected to have tag name “expression”
input (parenthesize): Integer flag; if the first child element is a <relation>, or if parenthesize is
   0, the output will not be enclosed in parenthesis; otherwise, output will be enclosed in parens.
output: A string that can be parsed as XHTML.
begin
    firstborn = first child element of src
    if firstborn does not exist then
        return “( )”
    end if
    res = “”
    switch the element type of firstborn:
        case <relation>:
            res = rend_relation(firstborn)
        case <join>, <left-join>, <right-join>, <full-join>:
            res = rend_conditional(src)
        case <cross-product>, <union>, <intersection>, <set-difference>, <division>,
                <semi-join>, <anti-join>, <outer-union>:
            res = rend_unconditional(src)
        case <selection>:       res = validate_selection(src)
        case <projection>:      res = validate_projection(src)
        case <renaming>:        res = validate_renaming(src)
        case <assignment>:      res = validate_assignment(src)
        case else:
            res = ‘�’ // Placeholder error-indicating character
    end switch
    if firstborn is not a <relation> then
        if parenthesize != 0 then
            return “(“ + res + “)”
        end if
    end if
    return res
end
```

```
algorithm rend_projection
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    if src has at least one <attribute> child element then
        res = “π<sub>” // Unicode 0x03C0
        res += rend_attribute(first <attribute> child of src)
```

```
            for each remaining <attribute> child element attr of src do
                res += “, ” + rend_attribute(attr)
            end for each
            res += “</sub>”
        end if
        res += rend_expression(first <expression> child of src, 1)
        return res
end
```

```
algorithm rend_renaming
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
        if src has at least one <attribute-pair> child element then
            res = “ρ<sub>” // Unicode 0x03C1
            res += rend_attribute_pair(first <attribute-pair> child of src)

            for each remaining <attribute-pair> child element attpair of src do
                res += “, ” + rend_attribute_pair(attpair)
            end for each
            res += “</sub>”
        end if
        res += rend_expression(first <expression> child of src, 1)
        return res
end
```

```
algorithm rend_attribute_pair
input (src): DOM element with tag name “attribute-pair”
output: A string that can be parsed as XHTML.
begin
        res = rend_attribute(first <attribute> child element of src)
        res += “→” // Unicode 0x2192
        res += rend_attribute(second <attribute> child element of src)
        return res
end
```

```
algorithm rend_assignment
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
        res = “(“ + rend_relation(first <relation> child of src)
        res += “ ← ” // Unicode 0x2190
```

```
    res += rend_expression(first <expression> child of src, 0) + “)”
    return res
end
```

```
algorithm rend_relation
input (src): DOM element with tag name “relation”
output: A string that can be parsed as XHTML.
begin
    return rend_name(child elements of src)
end
```

```
algorithm rend_attribute
input (src): DOM element with tag name “attribute”
output: A string that can be parsed as XHTML.
begin
    res = “”
    if the first child element of src is a <relation> then
        res += rend_relation(first child element of src)
        kids = all but the first child element of src
    else
        kids = child elements of src
    end if
    res += rend_name(kids)
    return res
end
```

```
algorithm rend_name
input (src): array of DOM elements
output: A string that can be parsed as XHTML.
begin
    res = “”
    for each element elem in src do
        txt = text content of elem
        switch element type of elem:
            case <name>:          res += txt
            case <subscript>:     res += “<sub>” + txt + “</sub>”
            case <superscript>:   res += “<sup>” + txt + “</sup>”
            case else:
                return res
        end switch
    end for each
    return res
end
```

```
algorithm rend_unconditional
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    switch element type of first element child of src:
        case <union>:            joiner = “ ⋃ ”    // Unicode 0x22C3, 0x222A
        case <intersection>      joiner = “ ⋂ ”    // Unicode 0x22C2, 0x2229
        case <set-difference>    joiner = “ − ”    // Unicode 0x2212
        case <cross-product>     joiner = “ × ”    // Unicode 0xD7, 0x2A09, 0x2A2F
        case <division>          joiner = “ ÷ ”    // Unicode 0xF7
        case <outer-union>       joiner = “ ⨄ ”    // Unicode 0x2A04, 0x228E
        case <anti-join>         joiner = “ ▷ ”    // Unicode 0x25B7
        case <semi-join>         joiner = “ ⋉ ”    // Unicode 0x22C9
    end switch
    res = rend_expression(first <expression> child of src, 1)
    for each remaining <expression> child operand of src do
        res += joiner + rend_expression(operand, 1)
    end for each
    return res
end
```

```
algorithm rend_conditional
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    switch element type of first element child of src:
        case <join>:             joiner = “ ⋈ ”    // Unicode 0x22C8, 0x2A1D
        case <left-join>         joiner = “ ⟕ ”    // Unicode 0x27D5
        case <right-join>        joiner = “ ⟖ ”    // Unicode 0x27D6
        case <full-join>         joiner = “ ⟗ ”    // Unicode 0x27D7
    end switch
    res = rend_expression(first <expression> child of src, 1)

    // If a boolean condition exists, ignore all but the first two operands.
    if src has a <boolean> child element then
        res += joiner + “<sub>”
            + rend_boolean(first <boolean> child of src) + “</sub>”
        res += rend_expression(second <expression> child of src, 1)
    else
        for each remaining <expression> child operand of src do
            res += joiner + rend_expression(operand, 1)
        end for each
    end if
    return res
end
```

```
algorithm rend_selection
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    if src has a <boolean> child element then
        res = “σ<sub>” // Unicode 0x03C3
        res += rend_boolean(first <boolean> child of src)
        res += “</sub>”
    end if
    res += rend_expression(first <expression> child of src, 1)
end
```

```
algorithm rend_simple_boolean
input (src): DOM element with tag name “true”, “false”, “null”, “string”, “number”, “attribute”
output: A string that can be parsed as XHTML.
begin
    switch element type of src:
        case <attribute>:       return rend_attribute(src)
        case <string>:          return ‘"’ + text content of src + ‘"’
        case <number>:          return text content of src
        case <true>:            return “true”
        case <false>:           return “false”
        case <null>:            return “null”
    end switch
    return ‘�’ // Placeholder error-indicating character
end
```

```
algorithm rend_boolean
input (src): DOM element
output: A string that can be parsed as XHTML.
begin
    if src is not a <boolean> then
        return “”
    end if
    firstborn = first child element of src
    switch element type of firstborn:
        case <true>, <false>, <null>:
            return rend_simple_boolean(firstborn)
```

```
case <not>:
    res = “¬(” // Unicode 0x00AC
    res += rend_boolean(first <boolean> child of firstborn)
    res += “)”
    return res
case <is-null>, <is-not-null>:
    res = rend_simple_boolean(next element sibling of firstborn)
    if firstborn is an <is-null> then
        res += “ is null”
    else
        res += “ is not null”
    end if
    return res
case <eq>, <neq>, <lt>, <gt>, <lte>, <gte>:
    res = rend_simple_boolean(first following element sibling of firstborn)
    switch element type of firstborn:
        case <eq>:     res += “ = ”
        case <neq>:    res += “ ≠ ” // Unicode 0x2260
        case <lt>:     res += “ < ”
        case <lte>:    res += “ ≤ ” // Unicode 0x2264
        case <gt>:     res += “ > ”
        case <gte>:    res += “ ≥ ” // Unicode 0x2265
    end switch
    res += rend_simple_boolean(second following sibling of firstborn)
    return res
case <and>, <or>:
    if firstborn is an <and> then
        joiner = “ ∧ ” // Unicode 0x2227, 0x22C0
    else
        joiner = “ ∨ ” // Unicode 0x2228, 0x22C1
    end if
    bln1 = first <boolean> sibling element of firstborn
    res = viz_boolean(bln1)
    if first child element of bln1 is an <and> or <or> then
        res = ‘(’ + res + ‘)’
    end if
    for each remaining <boolean> sibling clause of firstborn do
        strClause = viz_boolean(clause)
        if first child element of clause is an <and> or <or> then
            strClause = ‘(’ + strClause + ‘)’
        end if
        res += joiner + strClause
    end for each
    return res
```

APPENDIX G

APPENDIX G

ALGORITHMS FOR VISUALIZE

```
algorithm Visualize
input (src): DOM element
output: A string that can be parsed as XHTML.
begin
    res = null
    try do
        // Call the root-level subprocedure.
        res = viz_root(src)
    end try
    catch any error do
        throw error “(Visualize) RAML-to-SVG failed: ” + caught error message
    end catch

    // STEP 2: The <text> elements need to be positioned within the SVG image.
    rootNode = result of parsing an XML DOM-tree object out of res
    rootNode = rootNode.documentElement
    try do
        position = viz_positionText(rootNode, 10, 20, 0)
        width = position[0]
        height = position[1]
    end try
    catch any error do
        throw error “(Visualize) Node label positioning failed: ” + caught error
    end catch
    // Some of the rightmost and/or bottommost parts of the tree might not display.
    // Provide extra padding below and to the right.
    width += 25
    height += 10

    // STEP 3: The <line> elements need to be positioned within the SVG image.
    try do
        viz_positionLines(rootNode)
    end try
    catch any error do
        throw error “(Visualize) Line positioning failed: ” + caught error
    end catch

    // STEP 4: The SVG image needs to be encapsulated and given style information.
    // Note the absence of a <!DOCTYPE>. The SVG Working Group condemns them
    //    because their use encourage DTD-based markup validation, which is known to
    //    give false negatives in addition to being namespace-unaware.
    svgHead = “<?xml version="1.0" encoding="UTF-8" standalone="yes"?>”
        + “<svg xmlns="http:// www.w3.org/2000/svg" ”
        +      “version="1.1" baseProfile="basic" ”
        +      “width="” + width + “px" height="” + height + “px">”
        + “<defs><style type="text/css"><![CDATA[”
        + “@font-face { font-family:"RAML Font" ”
        +   “src:url("http:// rat.cs.panam.edu/RAT2/fonts/raml.ttf")”
        +      “ format("truetype") }”
        + “text{ text-anchor: middle; font-size: 20px; ”
        +       “font-family: "RAML Font","Times New Roman",serif; }”
        + “tspan { font-size:60% }”
        + “line { stroke:black }”
        + “]]></style></defs>”

    // STEP 5: Serialize the completed SVG document to a UTF-8-encoded string.
    // NOTE: encodeURIComponent is defined in the ECMAScript standard. It converts all
    //    characters that have any kind of special meaning in a URI (
    svgStr = result of serializing rootNode to a string
    svgUtf = svgHead + svgStr  + “</svg>”
    svgUtf = encodeURIComponent(svgUtf)

    // STEP 6: Wrap up the results in serialized XHTML.
    svgHtml = “<object type="image/svg+xml" width="” + width
        + “" height="” + height + “" data="image/svg+xml;charset=UTF-8,”
        + svgUtf + “"></object>”
    return svgHtml
end
```

---

algorithm viz_root
input (*src***):** DOM element, expected to have tag name “raml”
output: A string that can be parsed as XHTML.

```
begin
    str = ‘�’ // Placeholder error-indicating character
    if src has any <expression> child elements do
        str = viz_expression(first <expression> child element of src, true)
        // Visualize does not support multiple root-level <expression>s
    end if
    return str
end
```

```
algorithm viz_expression
input (src): DOM element, expected to have tag name “expression”
input (isRoot): Boolean flag;
output: A string that can be parsed as XHTML.
begin
    firstborn = first child element of src
    if firstborn does not exist then
        return “( )”
    end if
    res = “<g><text>”
    switch the element type of firstborn:
        case <relation>:
            res = viz_relation(firstborn) + “</text>”
        case <join>, <left-join>, <right-join>, <full-join>:
            res = viz_conditional(src)
        case <cross-product>, <union>, <intersection>, <set-difference>, <division>,
                <semi-join>, <anti-join>, <outer-union>:
            res = viz_unconditional(src)
        case <selection>:      res = validate_selection(src)
        case <projection>:     res = validate_projection(src)
        case <renaming>:       res = validate_renaming(src)
        case <assignment>:     res = validate_assignment(src)
        case else:
            res = ‘�</text>’ // Placeholder error-indicating character
    end switch
    if !isRoot then
        res += “<line />”
    end if
    res += “</g>”
    return res
end
```

```
algorithm viz_projection
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    if src has at least one <attribute> child element then
        res = “π<tspan dy="0.4em">” // Unicode 0x03C0
        res += viz_attribute(first <attribute> child of src)
        for each remaining <attribute> child element attr of src do
            res += “, ” + viz_attribute(attr)
        end for each
        res += “</tspan></text>”
    end if
    res += viz_expression(first <expression> child of src, false)
    return res
end
```

```
algorithm viz_renaming
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    if src has at least one <attribute-pair> child element then
        res = “ρ<tspan dy="0.4em">” // Unicode 0x03C1
        res += viz_attribute_pair(first <attribute-pair> child of src)

        for each remaining <attribute-pair> child element attpair of src do
            res += “, ” + viz_attribute_pair(attpair)
        end for each
        res += “</tspan></text>”
    end if
    res += viz_expression(first <expression> child of src, false)
    return res
end
```

```
algorithm viz_attribute_pair
input (src): DOM element with tag name “attribute-pair”
output: A string that can be parsed as XHTML.
begin
    res = viz_attribute(first <attribute> child element of src)
    res += “→” // Unicode 0x2192
    res += viz_attribute(second <attribute> child element of src)
    return res
end
```

```
algorithm viz_assignment
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    res = “(“ + viz_relation(first <relation> child of src)
    res += “ ← ” // Unicode 0x2190
    res += viz_expression(first <expression> child of src, false) + “)”
    return res
end
```

```
algorithm viz_relation
input (src): DOM element with tag name “relation”
output: A string that can be parsed as XHTML.
begin
    return viz_name(child elements of src)
end
```

```
algorithm viz_attribute
input (src): DOM element with tag name “attribute”
output: A string that can be parsed as XHTML.
begin
    res = “”
    if the first child element of src is a <relation> then
        res += viz_relation(first child element of src)
        kids = all but the first child element of src
    else
        kids = child elements of src
    end if
    res += viz_name(kids)
    return res
end
```

```
algorithm viz_name
input (src): array of DOM elements
output: A string that can be parsed as XHTML.
begin
    res = “”
    for each element elem in src do
        txt = text content of elem
        switch element type of elem:
            case <name>:          res += txt
            case <subscript>:     res += “<tspan dy="0.4em">” + txt + “</tspan>”
            case <superscript>:   res += “<tspan dy="0.4em">” + txt + “</tspan>”
            case else:
                return res
        end switch
    end for each
    return res
end
```

```
algorithm viz_unconditional
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    switch element type of first element child of src:

        case <union>:          res = “∪</text>”  // Unicode 0x22C3, 0x222A
        case <intersection>    res = “∩</text>”  // Unicode 0x22C2, 0x2229
        case <set-difference>  res = “−</text>”  // Unicode 0x2212
        case <cross-product>   res = “×</text>”  // Unicode 0xD7, 0x2A09, 0x2A2F
        case <division>        res = “÷</text>”  // Unicode 0xF7
        case <outer-union>     res = “⊎</text>”  // Unicode 0x2A04, 0x228E
        case <anti-join>       res = “▷</text>”  // Unicode 0x25B7
        case <semi-join>       res = “⋉</text>”  // Unicode 0x22C9
    end switch
    for each <expression> child operand of src do
        res += viz_expression(operand, false)
    end for each
    return res
end
```

```
algorithm viz_conditional
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    switch element type of first element child of src:
        case <join>:         res = “⋈”  // Unicode 0x22C8, 0x2A1D
        case <left-join>     res = “⟕”  // Unicode 0x27D5
        case <right-join>    res = “⟖”  // Unicode 0x27D6
        case <full-join>     res = “⟗” // Unicode 0x27D7
    end switch
    // If a boolean condition exists, ignore all but the first two operands.
    if src has a <boolean> child element then
        res += “<tspan dy="0.4em">”
            + viz_boolean(first <boolean> child of src) + “</tspan>”
    end if
    res += “</text>”
    for each <expression> child operand of src do
        res += joiner + viz_expression(operand, false)
    end for each
    return res
end
```

```
algorithm viz_selection
input (src): DOM element with tag name “expression”
output: A string that can be parsed as XHTML.
begin
    if src has a <boolean> child element then
        res = “σ<tspan dy="0.4em">” // Unicode 0x03C3
        res += viz_boolean(first <boolean> child of src)
        res += “</tspan>”
    end if
    res += viz_expression(first <expression> child of src, false)
    return res
end
```

```
algorithm viz_simple_boolean
input (src): DOM element with tag name “true”, “false”, “null”, “string”, “number”, “attribute”
output: A string that can be parsed as XHTML.
begin
    switch element type of src:
        case <attribute>:    return viz_attribute(src)
        case <string>:       return ‘"’ + text content of src + ‘"’
        case <number>:       return text content of src
        case <true>:         return “true”
        case <false>:        return “false”
        case <null>:         return “null”
    end switch
    return ‘�’ // Placeholder error-indicating character
end
```

```
algorithm viz_boolean
input (src): DOM element
output: A string that can be parsed as XHTML.
begin
    if src is not a <boolean> then
        return “”
    end if
    firstborn = first child element of src

    switch element type of firstborn:
        case <true>, <false>, <null>:
            return viz_simple_boolean(firstborn)
        case <not>:
            res = “¬(” // Unicode 0x00AC
            res += viz_boolean(first <boolean> child of firstborn)
            res += “)”
            return res
        case <is-null>, <is-not-null>:
            res = viz_simple_boolean(next element sibling of firstborn)
            if firstborn is an <is-null> then
                res += “ is null”
            else
                res += “ is not null”
            end if
            return res
        case <eq>, <neq>, <lt>, <gt>, <lte>, <gte>:
            res = viz_simple_boolean(first following element sibling of firstborn)
            switch element type of firstborn:
                case <eq>:      res += “ = ”
                case <neq>:     res += “ ≠ ” // Unicode 0x2260
                case <lt>:      res += “ < ”
                case <lte>:     res += “ ≤ ” // Unicode 0x2264
                case <gt>:      res += “ > ”
                case <gte>:     res += “ ≥ ” // Unicode 0x2265
            end switch
            res += viz_simple_boolean(second following sibling of firstborn)
            return res
        case <and>, <or>:
            if firstborn is an <and> then
                joiner = “ ∧ ” // Unicode 0x2227, 0x22C0
            else
                joiner = “ ∨ ” // Unicode 0x2228, 0x22C1
            end if
            bln1 = first <boolean> sibling element of firstborn
            res = viz_boolean(bln1)
            if first child element of bln1 is an <and> or <or> then
                res = ‘(’ + res + ‘)’
            end if
            for each remaining <boolean> sibling clause of firstborn do
                strClause = viz_boolean(clause)
                if first child element of clause is an <and> or <or> then
                    strClause = ‘(’ + strClause + ‘)’
                end if
                res += joiner + strClause
            end for each
            return res
    end switch
    return “”
end
```

algorithm viz_positionText
input (*thisG*): An SVG <g> element
input (*myX*, *myY*): Starting values for the ‘x’ and ‘y’ attributes being assigned to the <text> child of *thisG*
input (*parentWidth*): Minimum value for the amount of horizontal space that will be used to display this node’s text label
input (*oldHeight*): Minimum amount of vertical space needed to display all of this node’s child nodes
output (*childrenWidth*): Amount of horizontal space needed to properly display the entire sub-expression tree rooted at *thisG*
output (*newHeight*): Amount of vertical space needed for the same reason

```
begin
    newHeight = 0
    txt = first <text> child element of thisG
    thisWidth = amount of horizontal space needed to render txt
    // NOTE: Implementation of the above line may be very difficult to do well,
    //   due to the fact that Times New Roman is not a monospace font.

    if parentWidth > thisWidth then
        thisWidth = parentWidth
    end if

    // Recurse into each <g> child sub-tree to determine the amount
    //   of horizontal space needed to display them all.
    childrenWidth = 0
    for each <g> child element subtree of thisG do
        // Each child node’s text label is at least 40 pixels below this one.
        // The 60 is intentional; it includes the 20 pixels of the text label’s height.
        ret = viz_positionText(subtree, myX + childrenWidth,
                                        myY + 60, thisWidth)
        childrenWidth += ret[0]
        if newHeight < ret[1] then
            newHeight = ret[1]
        end if
    end for each

    // Assign the necessary attributes to position thisG’s text label.
    if childrenWidth = 0 then
        // This is a leaf node.
        myX += thisWidth / 2
        childrenWidth = thisWidth
    else
        // Get the horizontal midpoint coordinates of the text labels of
        //    both the leftmost and rightmost <g> child elements.
        lefty = first <text> child of the first <g> child of thisG
        righty = first <text> child of the last <g> child of thisG
        myX = (value of ‘x’ attribute of lefty) + (value of ‘x’ attribute of righty) / 2
    end if

    // Keep track of the height needed to fully display the tree.
    if myY > newHeight then
        newHeight = myY
    end if
    Set ‘x’ attribute of txt to be myX + “px”
    Set ‘y’ attribute of txt to be myY + “px”

    // Return the width and height needed to properly display
    //   this node and all its child nodes.
    return [childrenWidth, newHeight]
end
```

```
algorithm viz_positionLines
input (thisG): An SVG <g> element
output: Nothing
```

```
begin
    thisText = first <text> child of thisG
    thisLine = first <line> child of thisLine

    // Give starting positions to the <line> of each <g> child.
    // They will all have the same starting position.
    // The 10 is for vertical space between a line and text label.
    myX = (‘x’ attribute of thisText)
    myY = (‘y’ attribute of thisText) + 10      // font size / 2

    // If thisG’s text label includes a subscript, we need to
    //    make sure that the line below thisG’s text does not
    //    overlap onto the subscript.
    // We will also need to raise the line above thisG’s text
    //    label if it contains a superscript.
    hasSuperscript = false
    for each <tspan> child element tsp of thisText do
        // If the ‘dy’ attribute is positive, it’s a subscript.
        // Otherwise, it’s a superscript.
        if tsp has a ‘dy’ attribute then
            val = integer parsed out of the value of the ‘dy’ attribute of tsp
            if val < 0 then
                hasSuperscript = true
            else
```

```
                myY += 5      //font size / 4
            end if
        end if
    end for each

    myKids = all <g> child elements of thisG
    // Give the same starting position (bottom-center of thisG’s
    //    text label) to the <line> of every child <g> element.
    for each element subG of myKids do
        childLine = first <line> child of subG
        Set ‘x1’ attribute of childLine to myX + “px”
        Set ‘y1’ attribute of childLine to myY + “px”
    end for each

    //Give thisG’s own <line> child an ending position.
    if the parent node of thisG is a <g> element then
        // thisG “connects with” its parent’s <line>
        myY = (value of ‘y’ attribute of thisText) – 20;
        // If there is a superscript, we might need to subtract some.
        if hasSuperscript then
            myY -= 5 // font size / 4
        end if
        // ‘y2’ is the ‘y’ attribute of thisG’s <text> child, minus its height
        // ‘x2’ is the same as ‘x1’
        Set ‘x2’ attribute of thisLine to myX + “px”
        Set ‘y2’ attribute of thisLine to myY + “px”
    end if
    // Recurse into the <g> children of thisG so we can finish positioning their lines.
    for each element subG of myKids do
        viz_positionLines(subG)
    end for each
end
```

APPENDIX H

APPENDIX H

ALGORITHMS FOR TABULATE

```
algorithm Tabulate
input (src): DOM element with tag name “rdml”
output: A string that can be parsed as XHTML.
begin
    res = tabulate_database(first <database> child element of src)
    for each remaining <database> child element db of src do
        res += “<hr />”
        res += tabulate_database(db)
    end for each
    return res
end
```

```
algorithm tabulate_database
input (src): DOM element with tag name “database”
output: A string that can be parsed as XHTML.
begin
    wrapper = new XHTML <div> element
    if src has no child elements then
        return wrapper
    end if
    if src has a <name> child element then
        label = new XHTML <h2> element
        Set text content of label to be the text content of the <name> element
        Append label as a child element of wrapper
    end if
    for each <relation> child element reln of src do
        reldiv = new XHTML <div> element
        res = tabulate_relation(reln)
        if res is a DOM element then
            Append res as a child element of reldiv
        else //Something went wrong; res is an error string
            Set text content of relvid to be res
        end if
        Append reldiv as a child element of wrapper
    end for each
    return wrapper
end
```

```
algorithm tabulate_relation
input (src): RDML DOM element with tag name “relation”
output: A string that can be parsed as XHTML.
begin
    table = new XHTML <table> element
    table.className = “rdml_relation”
    thead = new XHTML <thead> element
    tbody = new XHTML <tbody> element
    Append thead as a child element of table
    Append tbody as a child element of table

    //Table name header
    row = new XHTML <tr> element
    cell = new XHTML <th> element
    name = tabulate_name(child elements of src)
    Set text content of cell to be name

    schema = first <schema> child element of src
    atts = all <attribute> child elements of schema
    Set value of “colspan” attribute of cell to be number of elements in atts
    Append cell as a child element of row
    Append row as a child element of thead

    //Column headers
    row = new XHTML <tr> element
    for each element att in atts do
        cell = new XHTML <th> element
        name = tabulate_name(child elements of att)
        Set text content of cell to be name
        Append cell as a child element of row
    end for each
    Append row as a child element of thead

    //Rows of data values
    instance = first <instance> child element of src
    for each <tuple> child element tup of instance do
        row = new XHTML <tr> element
        for each child element val of tup do
            cell = new XHTML <td> element
            if val is a <value> then
                Set text content of cell to be the text content of val
            else if val is a <null> then
                Set text content of cell to be “<em>null</em>”
            else
                Set text content of cell to be “”
            end if
            Append cell as a child element of row
        end for each
        Append row as a child element of tbody
    end for each
    return table
end
```

```
algorithm tabulate_name
input (src): array of DOM elements
output: A string that can be parsed as XHTML.
begin
    name = “<em>unnamed</em>”
    if the first element of src is a <name> then
        name = text content of first element of src
        if the second element kid2 of src is a <superscript> then
            name += “<sup>” + text content of kid2 + “</sup>”
        else if kid2 is a <subscript> then
            name += “<sub>” + text content of kid2 + “</sub>”
            if the third child element kid3 of src is a <superscript> then
                name += “<sup>” + text content of kid3 + “</sup>”
            end if
        end if
    end if
    return name
end
```

BIOGRAPHICAL SKETCH

Jeremy J. Miller earned his Bachelor of Science degree in Computer Science from the University of Texas – Pan American in 2011, and his Master of Science degree in Computer Science from the same university in 2013, graduating summa cum laude both times.

While pursuing his Master of Science degree, Mr. Miller worked for two years as an instructor of the university's entry-level computer literacy course, as well as being research assistant to Dr. Artem Chebotko in the Department of Computer Science. He interned at IBM in the summer of 2013, and became a professional software engineer the following year.

Mr. Miller has presented his research at the Hispanic Engineering, Science, and Technology (HESTEC) conference. His thesis, *The Relational Algebra Toolkit: A User-Friendly Approach to Presenting and Processing Relational Algebra Queries on the Web*, was supervised by Dr. Artem Chebotko and based on three years of contributions to the RAT project.

Those interested in contacting Mr. Miller may reach him at his permanent e-mail address: jeremy.millerscitech@gmail.com.