Logical Data Modeling: Part 5

Terry Halpin
INTI International University

This is the fifth article in a series on logic-based approaches to data modeling. The first article [5] briefly overviewed deductive databases, and illustrated how simple data models with asserted and derived facts may be declared and queried in LogiQL [2, 11, 13], a leading edge deductive database language based on extended datalog [1]. The second article [6] discussed how to declare inverse predicates, simple mandatory role constraints and internal uniqueness constraints on binary fact types in LogiQL. The third article [7] explained how to declare *n*-ary predicates and apply simple mandatory role constraints and internal uniqueness constraints to them. The fourth article [8] discussed how to declare external uniqueness constraints in LogiQL. The current article covers derivation rules in a little more detail, and shows how to declare inclusive-or constraints in LogiQL. The LogiQL code examples are implemented using the free, cloud-based REPL (Read-Eval-Print-Loop) tool for LogiQL that is accessible at https://developer.logicblox.com/playground/.

Some More Examples of Derivation Rules

Figure 1 shows a genealogy chart for the ancient Egyptian god Ra and nine of his descendants, who were collectively known as the Ennead or “Nine Gods” of Heliopolis. Each arrow depicts a parenthood relationship pointing from parent to child. According to the legend, Ra needed no consort, and simply spat or sneezed out Shu and Tefnut, and later created humanity from his own tears. Descendants other than the Ennead (e.g. Anubis is a son of Nepththys) are excluded from consideration.

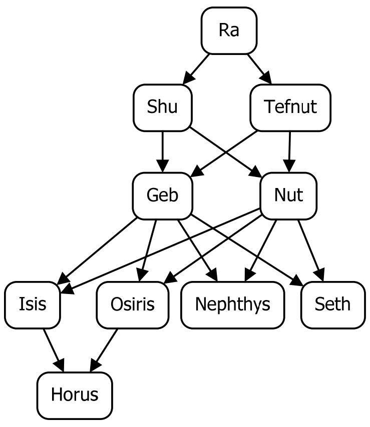
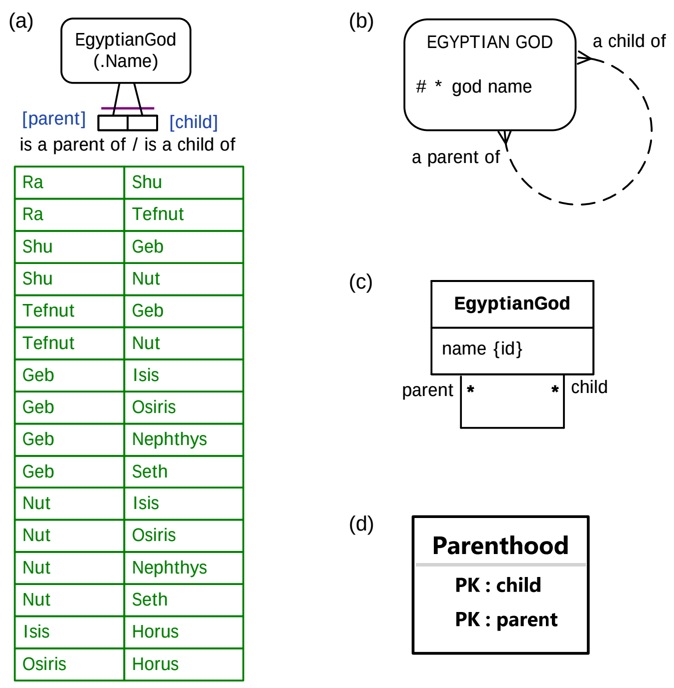


Figure 1 Parenthood chart of the Egyptian god Ra and the nine gods of Heliopolis.

Figure 2(a) models this example with a populated Object-Role Modeling (ORM) [9, 10] diagram, showing both a forward predicate reading (“is a parent of”) and an inverse predicate reading (“is a child of”) for the parenthood relationship. Mainly to generate convenient column names for the relational mapping, role names (“parent”, “child”) for the parenthood relationship are also provided. The example is also schematized in Figure 2(b) as an Entity Relationship diagram in Barker notation (Barker ER) [3], in Figure 2(c) as a class diagram in the Unified Modeling Language (UML) [14], and in Figure 2(d) as a relational database (RDB) schema.



Ra	Shu
Ra	Tefnut
Shu	Geb
Shu	Nut
Tefnut	Geb
Tefnut	Nut
Geb	Isis
Geb	Osiris
Geb	Nephthys
Geb	Seth
Nut	Isis
Nut	Osiris
Nut	Nephthys
Nut	Seth
Isis	Horus
Osiris	Horus

Figure 2 Data model for Figure 1 in (a) ORM, (b) Barker ER, (c) UML, and (d) relational database notation.

Although not shown in Figure 2, additional constraints apply to the parenthood relationship (e.g. each god has at most two parents, and no god may be its own parent). Such additional constraints will be covered later articles.

In the ORM schema in Figure 2(a), the preferred reference scheme for EgyptianGod is depicted by the popular reference mode “Name” shown in parentheses. The spanning uniqueness constraint over the parenthood fact type (shown as a bar over the constrained roles), indicates that the relationship is many-to-many. The absence of a mandatory role dot on the roles indicates that each role in the parenthood fact type is optional.

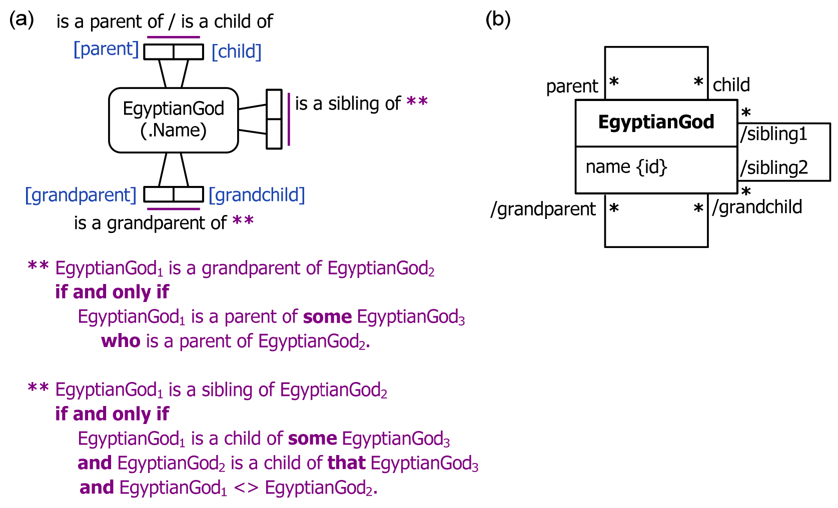
The Barker ER schema in Figure 2(b) depicts the primary reference scheme for EgyptianGod by prepending an octothrope “#” to the god name attribute, with an asterisk “*” to indicate the attribute is mandatory. The many-to-many nature of the parenthood relationship is depicted by a crowsfoot at each end, and the optionality of the roles is indicated by using a dashed line for the relationship.

The UML class diagram in Figure 2(c) depicts the primary identification for EgyptianGod by appending “{id}” to the name attribute. The multiplicity constraints shown as a star (“*”) at both ends of the parenthood association indicates the optional, many-to-many nature of the association.

Figure 2(d) shows the relational database schema diagram generated by the NORMA tool [4] from the ORM schema. Here, prepending “PK” to the child and parent attributes indicates that their combination provides the primary key of the table and hence each (parent, child) entry is unique.

Now suppose that it is also of interest to know who is a grandparent of whom, and which gods are siblings of one another. There is no need for us to explicitly assert this information as additional facts, since it can be derived from the parenthood facts. Figure 3(a) shows how to extend the ORM model to derive the required grandparenthood and siblinghood facts. Here, the *derived fact types* EgyptianGod is a grandparent of EgyptianGod and EgyptianGod is a sibling of EgyptianGod have been added, along with their *derivation rules*, which are displayed textually using the syntax of FORML (Formal ORM Language) [12]. Graphically, derived fact types are distinguished in ORM by appended at least one asterisk “*” to their predicate reading(s).

The FORML derivation rule for grandparenthood should be easy to understand, so no further explanation is provided here. The FORML derivation rule for siblinghood is based on the Oxford dictionary definition which treats two entities as siblings so long as they have at least one parent in common. Similar to SQL, FORML uses “<>” for the inequality operator (shown as “≠” in mathematics).



** EgyptianGod$_1$ is a grandparent of EgyptianGod$_2$
if and only if
EgyptianGod$_1$ is a parent of **some** EgyptianGod$_3$
who is a parent of EgyptianGod$_2$.

** EgyptianGod$_1$ is a sibling of EgyptianGod$_2$
if and only if
EgyptianGod$_1$ is a child of **some** EgyptianGod$_3$
and EgyptianGod$_2$ is a child of **that** EgyptianGod$_3$
and EgyptianGod$_1$ <> EgyptianGod$_2$.

Figure 3 Adding derived associations for grandparenthood and siblinghood in (a) ORM and (b) UML.

By default, derived fact types in ORM are evaluated lazily, so derived facts are computed only on request (e.g. when querying a derived fact type) rather than on update (when the fact types used to perform the derivation are updated). This "*derived but not stored*" choice is indicated in ORM by marking the derived fact type with just a *single asterisk* "*". In contrast, by default LogiQL evaluates derived predicates eagerly, and stores the derived facts for later use. This "*derived and stored*" option is indicated in ORM by marking the derived fact type with a *double asterisk* "**", as shown in Figure 3(a).

The Barker ER notation does not support derived relationships, so is ignored for this extended model. Figure 3(b) adds derived associations for grandparenthood and siblinghood, prepending a slash "/" to the association role names to indicate their derived status. UML has no way to distinguish between the storage options for derivation that are supported in ORM. Derivation rules may be specified textually in OCL, but are omitted in Figure 3(b), since OCL rules are typically unintelligible to nontechnical users. In this case, the OCL rules for grandparenthood are readable (e.g. self.grandchild = self.child.child), but the OCL rules for siblinghood are opaque.

The extended model may be captured in an RDB schema by adding views for Parenthood and Siblinghood, but the SQL code for these views is omitted here. To cater for the derived and stored option, the views should be materialized.

The schema for the Figure 3 example may be coded in LogiQL as shown below. The right arrow symbol "->" stands for the material implication operator "→" of logic, and is read as "implies". The left arrow symbol "<-" stands for the inverse material implication operator "←" of logic, and is read as "if". Recall that LogiQL is case-sensitive, and each formula must end with a period. The first line declares EgyptianGod as an entity type whose instances may be referenced by god names that are coded as character strings. The colon ":" in hasGodName(g:gn) distinguishes hasGodName as a refmode predicate, so this predicate is injective (mandatory, 1:1).

The second line declares the typing constraints on the isParentOf predicate. The third line is a derivation rule to declare the isChildOf predicate as the inverse of the isParentOf predicate. The remaining lines specify the derivation rules for the isGrandparentOf and isSiblingOf predicates.

```
EgyptianGod(g), hasGodName(g:gn) -> string(gn).
isParentOf(g1, g2) -> EgyptianGod(g1), EgyptianGod(g2).
isChildOf(g1, g2) <- isParentOf(g2, g1).
isGrandparentOf(g1, g2) <-
    isParentOf(g1, g3), isParentOf(g3, g2).
isSiblingOf(g1, g2) <-
    isChildOf(g1, g3), isChildOf(g2, g3), g1 != g2.
```

Recall that LogiQL uses a comma "," for the and-operator ("&") and "!=" for the inequality-operator ("≠"), and that variables in the head of LogiQL rules (before "<-") are implicitly universally quantified, and variables that occur only in the body of a rule (after "<-") are implicitly existentially quantified. Hence the grandparenthood and siblinghood rules are equivalent to the following logical formulae:

$\forall g_1,g_2$ [g_1 isaGrandparentOf $g_2 \leftarrow \exists g_3$($g_1$ isaParentOf g_3 & g_3 isaParentOf g_2)].
$\forall g_1,g_2$ [g_1 isaSiblingOf $g_2 \leftarrow \exists g_3$($g_1$ isaChildOf g_3 & g_2 isaChildOf g_3 & $g_1 \neq g_2$)].

Since no other rules are provided for the isaGrandparentOf and isaSiblingOf predicates, LogiQL's closed world assumption effectively promotes the if-operator "←" to an if-and-only-if-operator ("≡"), so these rules correspond precisely to the FORML rules shown in Figure 3(a).

To enter the schema in the free, cloud-based REPL tool, use a supported browser such as Chrome, Firefox or Internet Explorer to access the website https://repl.logicblox.com. Alternatively, you can access https://developer.logicblox.com/playground/, then click the "Open in new window" link to show a full screen for entering the code.

Schema code is entered in one or more *blocks* of one or more lines of code, using the *addblock* command to enter each block. After the "/>" prompt, type the letter "a", and click the addblock option that then appears. This causes the addblock command (followed by a space) to be added to the code window. Typing a single quote after the addblock command causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your block of code (see Figure 4).

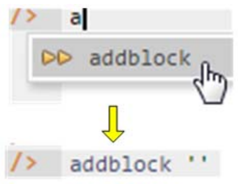
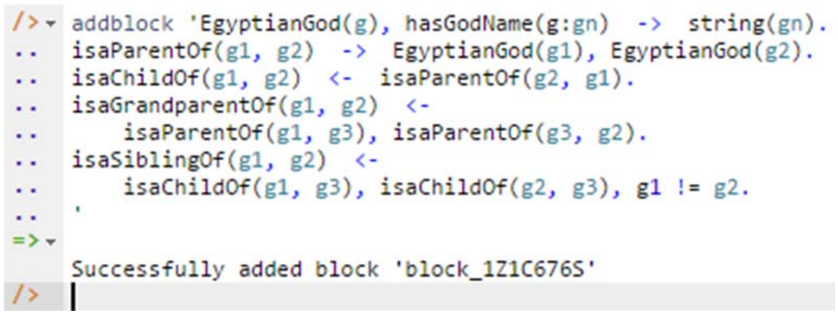


Figure 4 Invoking the addblock command in the REPL tool.

Now copy the seven lines of schema code provided above to the clipboard (e.g. using Ctrl+C), then paste it between the quotes (e.g. using Ctrl+V), and then press the Enter key. You are now notified that the block was successfully added, and a new prompt awaits your next command (see Figure 5). By default, the REPL tool also appends an automatically generated identifier for the code block. Alternatively, you can enter each line of code directly, using a separate addblock command for each line.



```
/> addblock 'EgyptianGod(g), hasGodName(g:gn)  ->  string(gn).
.. isaParentOf(g1, g2)  ->  EgyptianGod(g1), EgyptianGod(g2).
.. isChildOf(g1, g2)  <-  isaParentOf(g2, g1).
.. isaGrandparentOf(g1, g2)  <-
..     isaParentOf(g1, g3), isaParentOf(g3, g2).
.. isaSiblingOf(g1, g2)  <-
..     isChildOf(g1, g3), isChildOf(g2, g3), g1 != g2.
.. '
=>
   Successfully added block 'block_1Z1C676S'
/>
```

Figure 5 Adding a block of schema code.

The data in Figure 2(a) may be entered in LogiQL using the following delta rules. A delta rule of the form +*fact* inserts that fact. Recall that *plain, double quotes* (i.e. ",") are needed here, not single quotes or smart double quotes. Hence it's best to use a basic text editor such as WordPad or NotePad to enter code that will later be copied into a LogiQL tool.

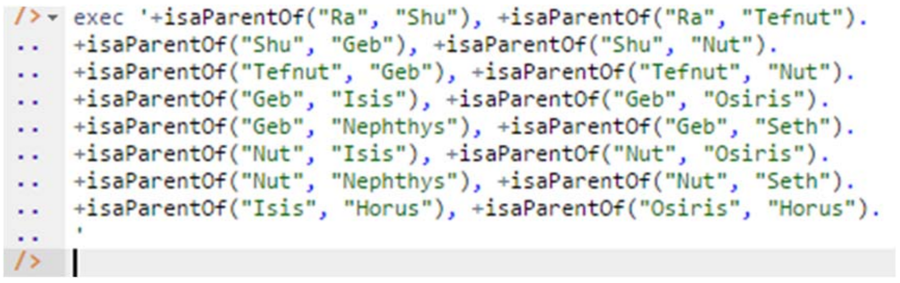
```
+isaParentOf("Ra", "Shu"), +isaParentOf("Ra", "Tefnut").
+isaParentOf("Shu", "Geb"), +isaParentOf("Shu", "Nut").
+isaParentOf("Tefnut", "Geb"), +isaParentOf("Tefnut", "Nut").
```

```
+isaParentOf("Geb", "Isis"), +isaParentOf("Geb", "Osiris").
+isaParentOf("Geb", "Nephthys"), +isaParentOf("Geb", "Seth").
+isaParentOf("Nut", "Isis"), +isaParentOf("Nut", "Osiris").
+isaParentOf("Nut", "Nephthys"), +isaParentOf("Nut", "Seth").
+isaParentOf("Isis", "Horus"), +isaParentOf("Osiris", "Horus").
```

There is no need to add delta rules simply to insert instances of the entity type EgyptianGod, because those facts are implied by facts in which they particpate. For example, the assertion +isaParentOf("Ra", "Shu") implies the assertions +EgyptianGod("Ra") and +EgyptianGod("Shu"). However, you may add those facts separately if you wish.

Delta rules to add or modify data are entered using the exec (for 'execute') command rather than the addblock command. To invoke the exec command in the REPL tool, type "e" and then select exec from the drop-down list. A space character is automatically appended. Typing a single quote after the exec command and space causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your delta rules.

Now copy the lines of data code provided above to the clipboard (e.g. using Ctrl+C), then paste it between the quotes (e.g. using Ctrl+V), and then press the Enter key. A new prompt awaits your next command (see Figure 6).

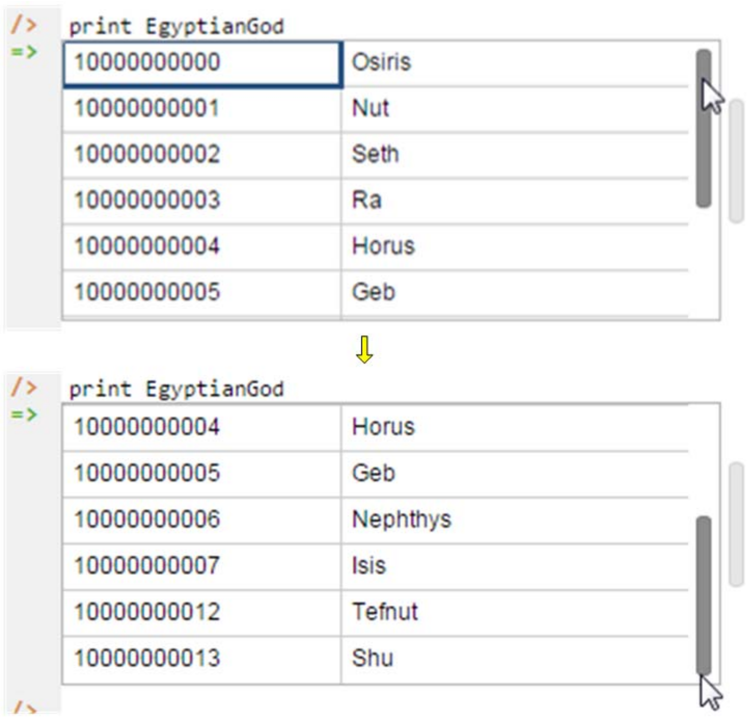


```
/> exec '+isaParentOf("Ra", "Shu"), +isaParentOf("Ra", "Tefnut").
..  +isaParentOf("Shu", "Geb"), +isaParentOf("Shu", "Nut").
..  +isaParentOf("Tefnut", "Geb"), +isaParentOf("Tefnut", "Nut").
..  +isaParentOf("Geb", "Isis"), +isaParentOf("Geb", "Osiris").
..  +isaParentOf("Geb", "Nephthys"), +isaParentOf("Geb", "Seth").
..  +isaParentOf("Nut", "Isis"), +isaParentOf("Nut", "Osiris").
..  +isaParentOf("Nut", "Nephthys"), +isaParentOf("Nut", "Seth").
..  +isaParentOf("Isis", "Horus"), +isaParentOf("Osiris", "Horus").
..  '
/>
```

Figure 6 Adding the data below the program code.

Now that the data model (schema plus data) is stored, you can use the *print* command to inspect the contents of any predicate. For example, to list all the recorded Egyptian gods, type "p" then select print from the drop-down list, and then type a space followed by "E", then select EgyptianGod from the drop-down list and press Enter. Alternatively, type "print EgyptianGod" yourself and press Enter.

Currently, the query result window in REPL tool displays at most six rows at a time, so only six of the ten god names are displayed, as shown in the top part of Figure 7. Drag the window's scroll bar down to see the rest of the result, as shown in the bottom part of Figure 7. By default, the REPL tool prepends a column listing automatically generated, internal identifiers for the returned entities.



```
/> print EgyptianGod
=> 1000000000  Osiris
   1000000001  Nut
   1000000002  Seth
   1000000003  Ra
   1000000004  Horus
   1000000005  Geb
```

```
/> print EgyptianGod
=> 1000000004  Horus
   1000000005  Geb
   1000000006  Nephthys
   1000000007  Isis
   1000000012  Tefnut
   1000000013  Shu
/>
```

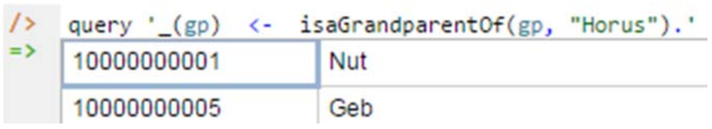
Figure 7 Using the print command and the scroll bar to list the extension of a predicate.

Similarly, you can use the print command to print the extension of the isaParentOf, isachildOf, isaGrandparentOf and isaSiblingOf predicates.

As discussed in previous articles, to perform a general query, you need to specify a derivation rule to compute the facts requested by the query. For example, the following query may be used to list the grandparents of Horus. Here the rule's head _(gp) uses an anonymous predicate to capture the result derived from the rule's body. Since the variable *gp* is a head variable, it is implicitly universally quantified. Since the derived isaGrandparentOf predicate has already been defined, the query is short and simple.

```
_(gp) <- isaGrandparentOf(gp, "Horus").
```

In LogiQL, queries are executed by appending their code in single quotes to the *query* command. To do this in the REPL tool, type "q", choose "query" from the drop-down list, type a single quote, then copy and paste the above LogiQL query code between the quotes and press Enter. The relevant query result is now displayed as shown in Figure 8. By default, the REPL tool prepends a column listing automatically generated, internal identifiers for the returned entities.



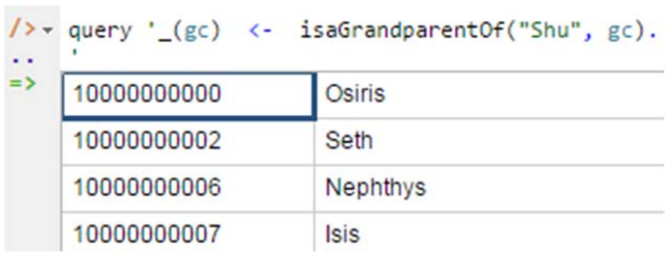
```
/> query '_(gp) <- isaGrandparentOf(gp, "Horus").'
=> 1000000001  Nut
   1000000005  Geb
```

Figure 8 Using a query to list the grandparents of Horus.

The isaGrandparentOf predicate has two arguments, and it is just as easy to query for the second argument. For example, the following query may be used to list the grandchildren of Shu.

```
_(gc) <- isaGrandparentOf("Shu", gc).
```

Entering this query causes the relevant query result to be displayed as shown in Figure 9. By default, the REPL tool prepends a column listing automatically generated, internal identifiers for the returned entities.

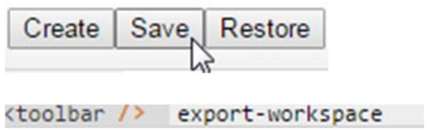


```
/> query '_(gc) <- isaGrandparentOf("Shu", gc).
..
=>
```

10000000000	Osiris
10000000002	Seth
10000000006	Nephthys
10000000007	Isis

Figure 9 Using a query to list the grandchildren of Shu.

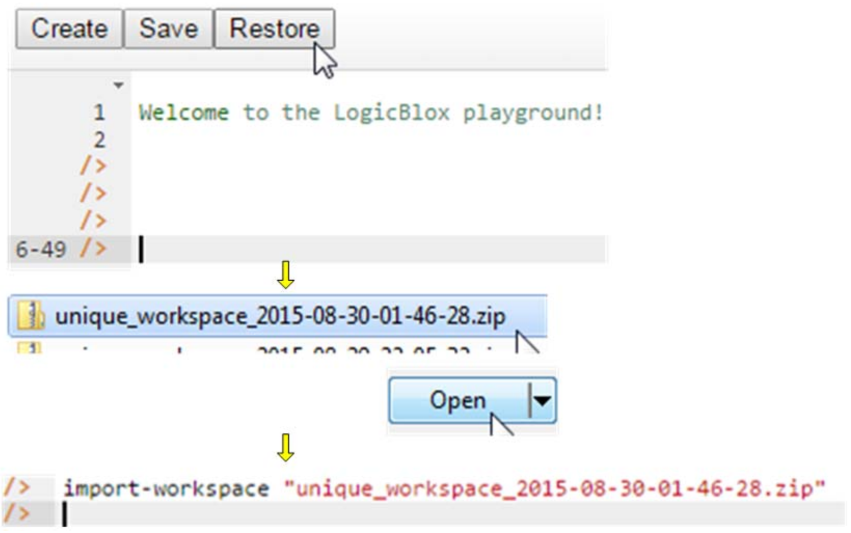
The REPL playground has limited screen space available, and after a while online access to the workspace can time out, so let's now save the workspace for later use by pressing the *Save* tab at the top of the screen. This exports the workspace to a zip file in your computer's normal download folder (see Figure 10).



```
<toolbar /> export-workspace
```

Figure 10 Saving your workspace.

Now exit REPL by closing the REPL window in your browser. Then re-access the REPL website at https://repl.logicblox.com, press the *Restore* tab at the top of the screen, then select the workspace zip file in your download folder and press Open to import it as your current workspace (see Figure 11).



```
Welcome to the LogicBlox playground!
```

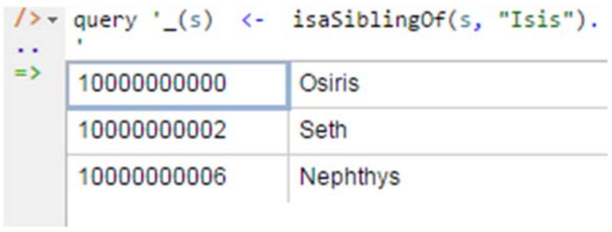
```
/> import-workspace "unique_workspace_2015-08-30-01-46-28.zip"
/>
```

Figure 11 Restoring your workspace.

You now have a fresh screen, with your workspace loaded. Now issue the following query to list the siblings of Isis.

```
_(s) <- isaSiblingOf(s, "Isis").
```

Entering this query causes the relevant query result to be displayed as shown in Figure 12. As usual, the REPL tool prepends a column listing automatically generated, internal identifiers for the returned entities.



```
/> query '_(s)  <-  isaSiblingOf(s, "Isis").
..  '
=>
10000000000   Osiris
10000000002   Seth
10000000006   Nephthys
```

Figure 12 Using a query to list the siblings of Isis.

Inclusive-Or Constraints

Table 1 Names and recorded titles of Ra and the Ennead

God Name	*God Title*
Ra	Sun God
Shu	Air God
Tefnut	Moisture God
Geb	Earth God
Nut	Sky Goddess
Isis	Mother Goddess
Osiris	God of the dead
Nephthys	
Seth	God of storms and chaos
Horus	Sky God

Now let's expand our model by including the titles of some of the Egyptian gods, as indicated in Table 1. Note that each god title applies to only one god. We do not record any title for Nephthys, so for our purposes it is *optional* for a god to have a title. Nevertheless, as indicated in our parenthood graph in Figure 1, it is *mandatory* for each of our Egyptians gods to *play at least one of the roles in the parenthood relationship*. This is known in ORM as an *inclusive-or constraint*, since each god must play the role of being a parent, or having a parent, or both.

Figure 13(a) models this expanded example as a populated ORM schema. The two uniqueness constraint bars on the roles of the EgyptianGod has GodTitle fact type indicate that the relationship is one-to-one. The *circled dot* connected to the junction of the roles in the parenthood fact type depicts the inclusive-or constraint. The NORMA tool [4] verbalizes the uniqueness and inclusive-or constraints as follows:

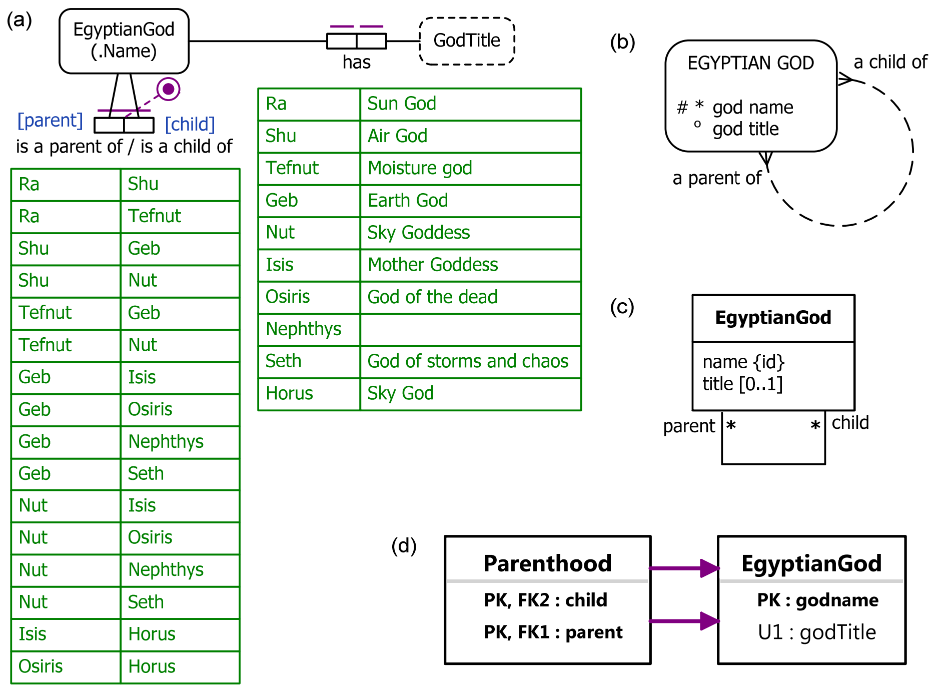
Each EgyptianGod has **at most one** GodTitle.
For each GodTitle, **at most one** EgyptianGod has **that** GodTitle.

Each EgyptianGod is a parent of **some** EgyptianGod
or is a child of **some** EgyptianGod.

Figure 13(b) models the expanded example in Barker ER notation. The "°" prepended to the god title attribute indicates its optionality. The Barker ER graphical notation is not able to capture the uniqueness constraint on the god title attribute (i.e. each title applies to only one god) or the inclusive-or constraint.

Figure 13(c) models the expanded example in UML. Here the [0..1] multiplicity constraint on the title attribute ensures that each Egyptian god has at most one title. The UML graphical notation is not able to capture the uniqueness constraint on the god title attribute or the inclusive-or constraint. However, these constraints could be specified textually in OCL.

Figure 13(d) displays the relational schema diagram generated by NORMA from the ORM schema. Here the optionality of the godTitle attribute is indicated by non-bold type, and the uniqueness of non-null entries for this attribute is indicated by the U1 annotation. The two arrows depict foreign key constraints. The inclusive-or constraint is not captured graphically, but can be coded textually in SQL.



Ra	Shu
Ra	Tefnut
Shu	Geb
Shu	Nut
Tefnut	Geb
Tefnut	Nut
Geb	Isis
Geb	Osiris
Geb	Nephthys
Geb	Seth
Nut	Isis
Nut	Osiris
Nut	Nephthys
Nut	Seth
Isis	Horus
Osiris	Horus

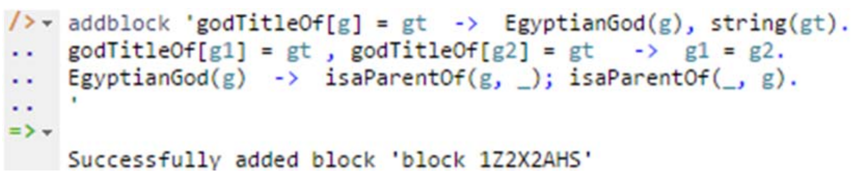
Ra	Sun God
Shu	Air God
Tefnut	Moisture god
Geb	Earth God
Nut	Sky Goddess
Isis	Mother Goddess
Osiris	God of the dead
Nephthys	
Seth	God of storms and chaos
Horus	Sky God

Figure 13 Expanded data model in (a) ORM, (b) Barker ER, (c) UML, and (d) relational database notation.

The additional aspects of the expanded schema may be coded in LogiQL as follows. The first line uses the square bracket notation to declare the functional nature of the godTitleOf predicate (so each god has at most one title), as well as the types of its arguments. The second line declares the inverse functional nature of the godTitleOf predicate (so each title applies to at most one god). The third line declares the inclusive-or constraint, using an underscore "_" for anonymous variables and a semicolon ";" for the inclusive-or operator (each god is a parent of something or has something as a parent).

```
godTitleOf[g] = gt  ->  EgyptianGod(g), string(gt).
godTitleOf[g1] = gt , godTitleOf[g2] = gt   ->  g1 = g2.
EgyptianGod(g)  ->  isaParentOf(g, _); isaParentOf(_, g).
```

Use the addblock command to add this code to the workspace in the usual way, as shown in Figure 14.



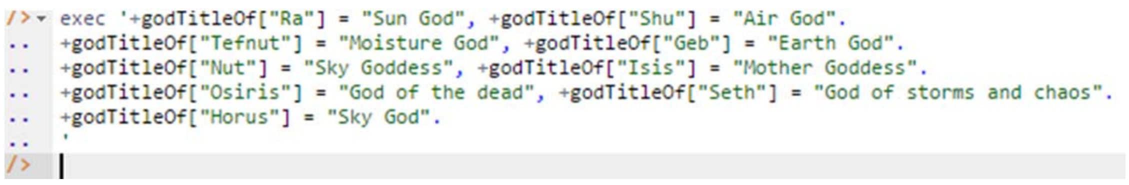
```
/> addblock 'godTitleOf[g] = gt  ->  EgyptianGod(g), string(gt).
.. godTitleOf[g1] = gt , godTitleOf[g2] = gt   ->  g1 = g2.
.. EgyptianGod(g)  ->  isaParentOf(g, _); isaParentOf(_, g).
.. '
=>
   Successfully added block 'block_1Z2X2AHS'
```

Figure 14 Adding the extra block of code to the workspace.

The additional god title data may be coded in LogiQL using the following delta rules.

```
+godTitleOf["Ra"] = "Sun God", +godTitleOf["Shu"] = "Air God".
+godTitleOf["Tefnut"] = "Moisture God", +godTitleOf["Geb"] = "Earth God".
+godTitleOf["Nut"] = "Sky Goddess", +godTitleOf["Isis"] = "Mother Goddess".
+godTitleOf["Osiris"] = "God of the dead", +godTitleOf["Seth"] = "God of storms and chaos".
+godTitleOf["Horus"] = "Sky God".
```

Now use the exec command in the usual way to enter the delta rules (see Figure 15).



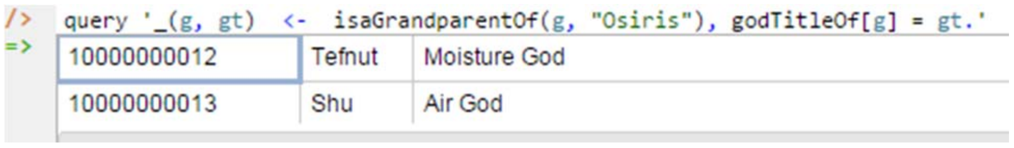
```
/> exec '+godTitleOf["Ra"] = "Sun God", +godTitleOf["Shu"] = "Air God".
..  +godTitleOf["Tefnut"] = "Moisture God", +godTitleOf["Geb"] = "Earth God".
..  +godTitleOf["Nut"] = "Sky Goddess", +godTitleOf["Isis"] = "Mother Goddess".
..  +godTitleOf["Osiris"] = "God of the dead", +godTitleOf["Seth"] = "God of storms and chaos".
..  +godTitleOf["Horus"] = "Sky God".
..  '
/>
```

Figure 15 Adding the god title facts to the workspace.

Using a comma "," for the logical "and" operator, the following query may now be used to list the name and title of the grandparents of Osiris.

```
_(g, gt)  <-  isaGrandparentOf(g, "Osiris"), godTitleOf[g] = gt.
```

Entering this query causes the relevant result to be displayed as shown in Figure 16. As usual, the REPL tool prepends a column listing automatically generated, internal identifiers for the returned entities.



```
/>  query '_(g, gt)  <-  isaGrandparentOf(g, "Osiris"), godTitleOf[g] = gt.'
=>
```

10000000012	Tefnut	Moisture God
10000000013	Shu	Air God

Figure 16 Querying the expanded workspace to list the names and titles of the grandparents of Osiris.

Conclusion

The current article discussed derivation rules in a little more detail, explained how to save and restore workspaces, and showed how to declare inclusive-or constraints in LogiQL. Future articles in this series will examine how LogiQL can be used to specify business constraints and rules of a more advanced nature. An introductory online tutorial for LogiQL and the REPL tool is available at the website https://developer.logicblox.com/content/docs4/tutorial/repl/section/split.html. Further coverage of LogiQL may be found in [11].

References

1. Abiteboul, S., Hull, R. & Vianu, V. 1995, *Foundations of Databases*, Addison-Wesley, Reading, MA.
2. Aref, M., Cate, B., Green T., Kimefeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. & Washburn, G. 2015, 'Design and Implementation of the LogicBlox System', *Proc. 2015 ACM SIGMOD*

International Conference on Management of Data, ACM, New York. http://dx.doi.org/10.1145/2723372.2742796.
3. Barker, R. 1990, *CASE*Method: Entity Relationship Modelling*, Addison-Wesley, Wokingham.
4. Curland, M. & Halpin, T. 2010, ‘The NORMA Software Tool for ORM 2’, P. Soffer & E. Proper (Eds.): *CAiSE Forum 2010*, LNBIP 72, pp. 190-204, Springer-Verlag Berlin Heidelberg 2010.
5. Halpin, T. 2014, ‘Logical Data Modeling: Part 1’, *Business Rules Journal*, Vol. 15, No. 5 (May, 2014), URL: http://www.BRCommunity.com/a2014/b760.html.
6. Halpin, T. 2014, ‘Logical Data Modeling: Part 2’, *Business Rules Journal*, Vol. 15, No. 10 (Oct., 2014), URL: http://www.BRCommunity.com/a2014/b780.html.
7. Halpin, T. 2015, ‘Logical Data Modeling: Part 3’, *Business Rules Journal*, Vol. 16, No. 1 (Jan., 2015), URL: http://www.BRCommunity.com/a2015/b795.html.
8. Halpin, T. 2015, ‘Logical Data Modeling: Part 4’, *Business Rules Journal*, Vol. 16, No. 7 (July, 2015), URL: http://www.BRCommunity.com/a2015/b820.html.
9. Halpin, T. 2015, *Object-Role Modeling Fundamentals*, Technics Publications, New Jersey.
10. Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, *2nd edition*, Morgan Kaufmann, San Francisco.
11. Halpin, T. & Rugaber, S. 2014, *LogiQL: A Query language for Smart Databases*, CRC Press, Boca Raton. http://www.crcpress.com/product/isbn/9781482244939#.
12. Halpin, T. & Wijbenga, J. 2010, ‘FORML 2’, *Enterprise, Business-Process and Information Systems Modeling*, eds. I. Bider et al., LNBIP 50, Springer-Verlag, Berlin Heidelberg, pp. 247–260.
13. Huang, S., Green, T. & Loo, B. 2011, ‘Datalog and emerging applications: an interactive tutorial’, *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, ACM, New York. http://dl.acm.org/citation.cfm?id=1989456.
14. Object Management Group 2013, *OMG Unified Modeling Language (OMG UML)*, version 2.5 RTF Beta 2. Available online at: http://www.omg.org/spec/UML/2.5/Beta2/PDF/.
15. OMG, 2012, *OMG Object Constraint Language (OCL), version 2.3.1*. Retrieved from http://www.omg.org/spec/OCL/2.3.1/.