Key LogiQL Concepts

LogiQL and its associated execution engine provide you with a powerful way to address your computational needs. Its power arises from its ability to efficiently deal with large amounts of data while avoiding many of the low-level implementation details found in other languages.

The language contains a number of intertwined features, and some connections between concepts might not be immediately obvious as you read sequentially through this book. To ease your way, this section provides a brief introduction to the language's key concepts that you will learn about in subsequent chapters. Each of the mentioned concepts is highlighted in bold and also appears in the Glossary to this book. Be aware that many of these terms have other senses. Further clarification is provided when the concepts are discussed in the chapters.

WORKSPACES

LogiQL programs manage **workspaces** (also called *databases*) that contain both your program and your data. In particular, workspaces hold collections of facts, each of which is concerned with a predicate. In logic, **predicates** are either properties that may be held by individual things or relationships that may apply to multiple things. In a workspace, a collection of facts associated with a predicate is called that predicate's *population*. Sometimes the distinction between the logical predicate and its population is glossed over by referring to the stored predicate population simply as a *predicate*.

You can think of a stored predicate as a named table. The **facts** of a predicate population correspond to table rows, and each row comprises a tuple of **data elements**. There are two types of data elements stored in LogiQL workspaces. Built-in **primitive types** include strings, numbers, and datetimes. In addition, programmers can define their own entity types, elements of which are guaranteed to be distinct. Moreover, an entity

TABLE 1 British Monarchy `houseOf` Predicate

Monarch	House
Anne	Stuart
George I	Hanover
George II	Hanover
George III	Hanover
George IV	Hanover
William IV	Hanover
Victoria	Hanover
Edward VII	Saxe-Coburg and Gotha
George V	Windsor
Edward VIII	Windsor
George VI	Windsor
Elizabeth II	Windsor

type may have a corresponding programmer-visible reference scheme (**refmode**) that can be used to identify the individual elements of that type.

All of the tuples in a predicate population have the same length (*arity*). Thinking again in terms of tables, this says that all rows in a given table have the same number of data elements. Hence, a column in a table consists of all of the data elements occupying the corresponding position in the facts of the predicate. Moreover, all of the elements of a column are of the same type. The data elements in a particular column of a predicate are said to fill that column's **role** in the predicate.

Table 1 contains an example that you will see again in Chapter 1. It illustrates the `houseOf` predicate relating British monarchs to their houses. `Monarch` and `House` are entity types referenced by their names, which are elements of type `string`.

There are 12 facts in this predicate, and each fact fills two roles in the predicate population, one for the monarch and one for the house. A LogiQL program could be used to add these facts to the `houseOf` predicate and store the results in a workspace.

LOGIC PROGRAMS

To manipulate the predicates in a workspace, you write logic programs. A **logic program** is a set of **clauses**, each of which makes a claim about the facts in a predicate. There are three kinds of clauses: fact assertions/retractions (facts), constraints, and derivation rules (*rules*). A **fact assertion** is

used to add a fact to a predicate's population, and a **fact retraction** is used to delete a fact from a predicate's population. **Constraints** can be used to declare predicates or limit the facts that can populate them. **Derivation rules** are used for programmatically altering a workspace, typically by deriving new facts.

A clause comprises a *head* and/or a *body*, both of which contain atoms, possibly combined by operators. An **atom** consists of a predicate name and a parenthesized list of arguments, each of which corresponds to one of the predicate's roles. In addition, an atom may be adorned with a *delta modifier* to control changes to the contents of the corresponding predicate.

LogiQL's syntax enables a programmer to specify two key properties of each rule: whether or not the predicate that the rule computes is an intensional database (IDB) predicate or an extensional database (EDB) predicate, and whether or not the rule computes an aggregation. EDB predicates are populated either by fact assertions, which have an empty body and a head containing only delta-modified atoms, or *delta rules*, that specify existing contributing predicates in their bodies. A program's *delta logic* comprises its fact assertions/retractions and its delta rules. IDB predicates, in contrast, are populated by *IDB rules*, in which neither head nor body atoms may be delta modified. *Aggregation rules* have their own syntax that includes mention of one of the built-in functions for aggregating data.

Here are some examples of how these concepts are expressed in LogiQL. The first example illustrates how you could declare the `Monarch` entity type, along with its `hasMonarchName` refmode.

```
Monarch(m), hasMonarchName(m:s) -> string(s).
```

Here is how you can assert the existence of a specific `Monarch` entity:

```
+Monarch("George VI").
```

If you wished to ensure that each such monarch must belong to a house, you could specify the following constraint, using "_" to mean "something."

```
Monarch(m) -> houseOf[m] = _.
```

If you wanted the LogiQL engine to automatically add facts about grandparents every time you added facts about parents, you could use the following IDB rule to derive the `isGrandParentOf` predicate.

```
isGrandparentOf(p1, p2) <- isParentOf(p1, p3),
    isParentOf(p3, p2).
```

Alternatively, if you wished to manage the `isGrandParent` predicate yourself, you could use the following delta rule:

```
+isGrandparentOf(p1, p2) <-
    +isParentOf(p1, p3), isParentOf(p3, p2) ;
    isParentOf(p1, p3), +isParentOf(p3, p2).
```

Finally, here is an aggregation rule you could use to compute the number of monarchs:

```
nrMonarchs[] = n <- agg<<n = count()>> Monarch(_).
```

WORKSPACE ORGANIZATION AND MANAGEMENT

A LogiQL program manipulates two kinds of predicates—EDB predicates and IDB predicates. *EDB predicates* normally are used to hold the facts that you explicitly enter into the workspace with fact assertions or facts you wish to manage yourself using delta rules. The term *extensional database (EDB)* is used to describe the set of all EDB predicate populations. In contrast, the facts populating IDB predicates are computed for you by your logic program with its IDB rules. The *intensional database (IDB)* is the set of IDB predicates stored in a workspace.

The logic program in a workspace comprises a set of **blocks**, each of which, in turn, is a set of related clauses residing in a source file. The process of loading a block into a workspace is called **installation**, and when you install a block you can designate it as *active* or *inactive*. Active blocks are used to automatically update the predicates in a workspace, whereas inactive blocks are available for on-demand use.

Program evaluation is broken into atomic units called *transactions*, each of which has two stages. The *initial stage* is used for processing queries and for on-demand evaluation of inactive blocks. This is typically the stage at which fact assertions/retractions are applied. During the *final stage*, active blocks are evaluated by continually interpreting all active program rules until a fixed point is reached; that is, until no further changes to the workspace occur. If at any time during evaluation a constraint is violated, the current transaction aborts and the content of the workspace reverts to its state before the transaction began. When a non-aborting transaction completes, it is said to *commit*.

Here is an example of a short LogiQL program consisting of two blocks. The first, an active block, provides a schema declaring the `Person` entity type and the `isParentOf` and `isAncestorOf` predicates. It also includes a constraint disallowing a person from being an ancestor of himself or herself.

```
Person(p), hasPersonName(p:n) -> string(n).
isParentOf(p1, p2) -> Person(p1), Person(p2).
isAncestorOf(p1, p2) -> Person(p1), Person(p2).
isAncestorOf(p1, p2) <- isParentOf(p1, p2).
isAncestorOf(p1, p2) <- isParentOf(p1, p3),
    isAncestorOf(p3, p2).
!isAncestorOf(p, p).
```

The second, inactive, block is responsible for populating the `isParentOf` predicate with three facts:

```
+isParentOf("Doctor Who", "Doc Brown").
+isParentOf("Doc Brown", "Merlin").
+isParentOf("Merlin", "Doctor Who").
```

The above program will not run as expected. Can you tell why? It is because the given facts and rules determine that Doctor Who is an ancestor of himself, but the constraint expressed in the schema disallows this. If you try to run this program, it will print an error message and abort.

PROGRAMMING IN LOGIQL

The first step in writing a LogiQL program is, of course, to understand the problem that you are trying to solve. Usually, the problem consists of inputting a data file into a workspace, where rules are applied to derive new facts that you can later query. You express your understanding of the problem by devising a schema in the form of a set of predicate declarations and accompanying constraints. Then you specify the rules you intend to use to compute the required results. You may also need to do some work preparing your input data for entry into the workspace. Naturally, you should also devise tests to ensure your program computes the correct result and performs efficiently. Hopefully, this book you are about to work through will prepare you to successfully accomplish these steps.