Appendix H: LogiQL and SQL

SQL IS A STANDARD language used for defining, manipulating, and querying relational databases. This appendix provides a brief discussion of how basic queries conveyed in SQL may be expressed in LogiQL.

Consider the following relational database, which includes two relational tables (Tables H.1 and H.2) describing countries in 2011. The country table lists the ISO two-letter code, name, and population of various countries. For discussion purposes, the population of Finland (5,396,292) is omitted simply to illustrate SQL's use of a null value to indicate that a data value is missing (e.g., because it is unknown or inapplicable). To save space, only a small number of countries are included. For those countries that have presidents, the president table lists the name, country, gender, and birth year of those presidents. Australia, Canada, and the United Kingdom have prime ministers instead of presidents, so they are not included in the president table.

Throughout the database, countries are standardly identified by their country codes. The entries in the first two columns of the tables are necessarily unique, so each of these columns is a *candidate key* for its table. The country table has *countryCode* as its *primary key* and *countryName* as an alternate key.

The president table has *presidentName* as its primary key, and *countryCode* as an alternate key. Note that the *countryCode* column appears in both tables. In fact, the *countryCode* column in the president table serves as a foreign key constraint ensuring that each country code in the president table also appears in the country table. A *foreign key* is a column, or list of columns, within a table, that is not the primary key, but the foreign key entries must also occur in a candidate key of some table.

TABLE H.1 Country Codes and Populations

countryCode	countryName	Population
AU	Australia	22,778,975
CA	Canada	34,482,779
DE	Germany	81,729,000
FI	Finland	
FR	France	65,300,000
GB	United Kingdom	62,300,000
IN	India	1,210,193,422
US	United States	312,702,000

TABLE H.2 Genders and Birth Years of Presidents

presidentName	countryCode	Gender	birthYear
Christian Wulff	DE	M	1959
Tarja Halonen	FI	F	1943
Nicolas Sarkozy	FR	M	1955
Pratibha Patil	IN	F	1934
Barack Obama	US	M	1961

Each row entry of a relational table is an ordered *n*-tuple of values, or *tuple* for short. Each tuple represents one or more atomic facts. For example, the first row of the country table stores the fact that the country with the code "AU" has the name "Australia," and also stores the fact that the country with the code AU has a population of 22,778,975. In contrast, LogiQL uses a separate predicate to store each kind of atomic fact. This enables the data to be stored without using *nulls*. For example, instead of setting the population of Finland to be null, LogiQL simply does not store a population fact for Finland. The schema for the sample relational database described above may be set out in LogiQL as follows. For simplicity, years and populations are modeled simply as numbers:

```
Country(c), hasCountryCode(c:cc) -> string(cc).
President(p), hasPresidentName(p:pn) -> string(pn).
Gender(g), hasGenderCode(g:gc) -> string(gc).
countryNameOf[c] = cn -> Country(c), string(cn).
populationOf[c] = n -> Country(c), int(n).
countryOf[p] = c -> President(p), Country(c).
genderOf[p] = g -> President(p), Gender(g).
birthyearOf[p] = y -> President(p), int](y).
```

```
countryTable(c, cn, ns) -> Country(c), string(cn),
    string(ns).
isPairedWith(mp, fp) -> President(mp), President(fp).
isHighlyPopulated(c) -> Country(c).
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
Country(c) -> countryNameOf[c] = _.
President(p) -> countryOf[p] = _, genderOf[p] = _,
    birthyearOf[p] = _.
```

If you ever wish to display LogiQL data in the form of a relational table, you can use a rule or query to conjoin the relevant facts into a single tuple. If the relational tuple includes a null, you can check for instances where the relevant fact does not exist and then display an empty string to represent the null. If the data type for a nullable column is not `string`, you can use a conversion function to coerce the data type to be `string`. For example, the following rule may be used to derive a predicate which displays like the relational country table:

```
countryTable(c, cn, ns) -> Country(c), string(cn),
    string(ns).
countryTable(c, cn, ns) <- countryNameOf[c] = cn,
    ((populationOf[c] = n, ns = int:string:convert[n]);
    (!populationOf[c] = _, ns = "")).
// Show the country code, country name, and a
// population string for each country. If the country
// population is recorded,display it,otherwise
// display an empty string for the population
```

The above program, together with the data, a derivation rule and query to display the relational tables, and some later queries, are accessible in the files `SQL1.logic` and `SQL1Data.logic`.

SQL is based partly on two formal notations called the relational algebra and the relational calculus. To construct SQL queries, you make use of several operations that have their historical roots in these notations. Relational *selection* is the operation of selecting those rows from a relation (which may be either asserted, or derived from other relations) that satisfy a specified condition. In SQL, this is achieved by including the condition in a **where**-clause. Relational *projection* is the operation of choosing just those columns of interest from a relation. In SQL, this is achieved by including the source relation(s) in a **from**-clause and the relevant columns in a **select**-list. For example, a query to list the names and birth years of the female presidents may be formulated in SQL:

```
select presidentName, birthyear
from President
where gender = 'F'
```

When run, this query returns the result displayed in Table H.3.

Relational algebra and LogiQL are both *set-oriented* languages. What this means is that a given tuple can appear at most once in a table. SQL, on the other hand, is *bag oriented*. This means that duplicate tuples are allowed. For example, the result of the following query to list the genders of presidents includes multiple occurrences of each gender code:

```
select gender
from President
```

The result of executing the query is shown in Table H.4.

SQL includes a **distinct** option to convert bags to sets, so the following query returns only a single occurrence of each gender. The same result is obtained from the LogiQL query: `_(g) <- genderOf[_]=g.`

```
select distinct gender
from President
```

TABLE H.3 Names and Birth Years of Female Presidents

presidentName	**birthYear**
Tarja Halonen	1943
Pranab Mukherje	1935

TABLE H.4 Genders (Only) of Presidents in SQL

Gender
M
F
M
F
M

The result of executing the query is shown in Table H.5.

SQL includes various relational operators for *joining* multiple relations into a single relation. The most expansive of these is the **cross join** operator, which outputs the *Cartesian product* of the input relations. Given any two sets *A* and *B*, the *Cartesian product* $A \times B$ is the set of all ordered pairs (x, y) where *x* belongs to *A* and *y* belongs to *B*. If *A* and *B* are relations, *x* and *y* are tuples.

For example, the following query lists all the possible ways of pairing a male president with a female president. Here **as**-clauses are used to introduce aliases for the president table (MP, male president; FP, female president). The aliases enable the table to be cross joined to itself (pairing each president with each president) before the **where**-clause condition filters out the unwanted rows from the Cartesian product. The aliases are necessary to distinguish the two occurrences of the president table in forming the join:

```
select MP.presidentName, FP.presidentName
from President as MP
    cross join President as FP
where MP.gender = 'M'
    and FP.gender = 'F'
```

The result of executing the query is shown in Table H.6.

TABLE H.5 Genders (Only) of Presidents in LogiQL

Gender
M
F

TABLE H.6 Male–Female Presidential Pairings

MP.presidentName	**FP.presidentName**
Christian Wulff	Tarja Halonen
Christian Wulff	Pratibha Patil
Nicolas Sarkozy	Tarja Halonen
Nicolas Sarkozy	Pratibha Patil
Barack Obama	Tarja Halonen
Barack Obama	Pratibha Patil

The following LogiQL rule computes the same set of ordered pairs:

```
isPairedWith(mp, fp) -> President(mp), President(fp).
isPairedWith(mp, fp) <- genderOf[mp] = "M",
    genderOf[fp] = "F".
// Pair each male president with each female president.
```

If we remove the **where**-clause from the previous SQL query, the full Cartesian product would be listed, pairing each of the five presidents with each president, resulting in 25 rows. The equivalent computation in LogiQL is shown below. Notice how much simpler the LogiQL code is compared with the SQL code:

```
isPairedWith(p1, p2) -> President(p1), President(p2).
isPairedWith(p1, p2) <- President(p1), President(p2).
// Pair each president with each president.
```

Now suppose that we wish to list each president as well as the ISO code and name of the country of which he/she is president. If you look back at the relational tables, you'll see that the relevant country codes are listed in both tables, but the presidents are listed in only the president table, while the country names are listed only in the country table.

The query may be formulated by forming the *natural join* of the two tables (matching rows in the country table with rows in the president table that have the same value for country code) and then projecting on the three required columns from the join result. In the SQL standard, this query may be formulated as shown below, using the **natural join** operator:

```
select presidentName, countryCode, countryName
from President natural join Country
```

Although included in the SQL standard since 1992, many commercial SQL dialects do not yet support this syntax. In this case, the query may be reformulated as a *conditional inner join* as follows. Here the join condition is stated in an **on**-clause, and the *countryCode* columns must be qualified by prepending their table name:

```
select presidentName, President.countryCode,
  countryName
from President join Country
on President.countryCode = Country.countryCode
```

The result of executing the query is shown in Table H.7.

The same results may be formulated in LogiQL:

```
result(p, c, cn) -> President(p), Country(c),
    string(cn).
result(p, c, cn) <- countryOf[p] = c,
    countryNameOf[c] = cn.
```

Notice that the join is achieved simply by using the same variable `c` in each conjunct. This ensures that the `c` value in `countryNameOf[c]=cn` matches the `c` value in `countryOf[p]=c` for each `(p,c,cn)` triple that satisfies the body condition.

$A \cup B$, the union of sets A and B, is the set of all elements that belong to either A or B. SQL includes the **union** operator to form the *union* of two compatible relations. As an example, consider the following two *m:n* relations, shown in Tables H.8 and H.9. Notice that their populations are properly compatible. A corresponding LogiQL schema is set out below the tables:

```
Person(p), hasPersonName(p:pn) -> string(pn).
Food(f), hasFoodName(f:fn) -> string(fn).
eats(p, f) -> Person(p), Food(f).
likes(p, f) -> Person(p), Food(f).
```

TABLE H.7 President Names, Country Codes, and Country Names

presidentName	**countryCode**	**countryName**
Christian Wulff	DE	Germany
Tarja Halonen	FI	Finland
Nicolas Sarkozy	FR	France
Pratibha Patil	IN	India
Barack Obama	US	United States

TABLE H.8 Eats Relation

personName	**foodName**
Spencer	Pizza
Spencer	Spinach
Spencer	Raisins
Terry	Mangoes
Terry	Pizza
Terry	Vegemite

TABLE H.9 Likes Relation

personName	**foodName**
Spencer	Pizza
Spencer	Raisins
Terry	Chocolate
Terry	Mangoes
Terry	Pizza

Since the types of the *Eats* and *Likes* relations are compatible, it is meaningful to construct their union. The following SQL query may be used to list who eats or likes what foods:

```
select personName, foodName
from Eats
union
select personName, foodName
from Likes
```

The result of executing the query is shown in Table H.10.

The following, disjunctive LogiQL query returns the same set of ordered pairs. In general, unions may be formulated in LogiQL by using the *inclusive-or* operator (';'). Again, the LogiQL query is simpler than the corresponding SQL query. The program, data, and query are accessible in the files `SQL2.logic` and `SQLData.logic`:

```
_(p, f) <- eats(p, f) ; likes(p, f).
```

Returning to our country and president tables, suppose we want to list the presidents of those countries that have a population of at least 100 million people. Although we need to access both tables to determine the answer, the final projection comes from just the president table. Hence, even though we could formulate the query using a join, it is also possible to formulate the query in SQL without a join by using a *subquery*:

```
select presidentName
from President
where countryCode in
(select countryName
 from Country
 where population > = 100000000)
```

TABLE H.10 Union of Eats and Likes Relations

personName	**foodName**
Spencer	Pizza
Spencer	Spinach
Spencer	Raisins
Terry	Chocolate
Terry	Mangoes
Terry	Pizza
Terry	Vegemite

TABLE H.11 Presidents of Highly Populated Countries

presidentName
Pratibha Patil
Barack Obama

The result of executing the query is shown in Table H.11.

Here the bracketed **select**-statement is a subquery to derive which countries have a population above 100 million. This subquery returns the set {'IN', 'US'}, and this intermediate result is used to transform the outer query into **select** *presidentName* **from** president **where** *countryCode* **in** {'IN', 'US'}, which is then run to return the final result.

In LogiQL the subquery to compute the highly populated countries may be specified as a derivation rule such as

```
isHighlyPopulated(c) -> Country(c).
isHighlyPopulated(c) <- populationOf[c] > = 100000000.
```

That computed predicate may then be referenced in the final query to return the presidents of highly populated countries. This code is included in the file `SQL1.logic`:

```
_(p) <- countryOf[p] = c, isHighlyPopulated(c).
```

SQL provides several aggregation functions, including **count**, **sum**, **min**, and **max** which roughly correspond, respectively, to the `count`, `total`, `min`, and `max` functions in LogiQL. SQL's **avg** function for computing averages may be emulated in LogiQL by dividing the sum computed by the `total` function by the number of elements computed by the `count` function.

SQL's **count** function may be applied to a single column, but when its argument is specified as an asterisk it counts all the rows in the specified bag or set. For example, the following SQL query may be used to count the number of female presidents:

```
select count(*) from President
where gender = 'F'
```

Result: 2

In LogiQL, we could derive this result using the following derivation rule:

```
nrFemalePresidents[] = n -> int(n).
nrFemalePresidents[] = n <-
    agg<<n = count()>> President(p), genderOf[p] = "F".
```

SQL's syntax for aggregation functions is usually simpler than that of LogiQL. The following SQL query may be used to list the name of the country (or countries) with the maximum population. Since China was omitted from our data, the query returns India:

```
select countryName from Country
where population =
    (select max(population) from Country)
```

Result: India

In LogiQL, the maximum population may be derived as follows:

```
maxPopulation[] = n -> int(n).
maxPopulation[] = n <- agg<<n = max(pop)>>
    populationOf[_] = pop.
```

The name of the country with the maximum population may now be listed using the following query:

```
_(cn) <- countryNameOf[c] = cn, populationOf[c] =
    maxPopulation[].
```

SQL includes a **group by** clause for partitioning a table into groups of rows, where each row in a specific group has the same value(s) for the specified grouping criterion/criteria. This may then be used to list properties that apply to each group as a whole, so the final query result has at most one row for each group. For example, the following SQL query lists the number of presidents for each gender, as well as the minimum birth year for each gender:

```
select gender, count(*), min(birthyear)
from President
group by gender
```

Result:

```
F 2 1934
M 3 1955
```

When executed, the **from**-clause chooses the president table, and then the **group by** clause partitions the president table into two groups, one for each gender, as depicted below. The **select**-clause is then executed to list for each gender the count of all the rows in its group and the minimum birth year in its group.

The result of executing the query is shown in Table H.12.

In LogiQL, the grouping criterion is used as an argument to the derived functions, and thus,

```
nrOfPresidentsOf[g] = n -> Gender(g), int(n).
nrOfPresidentsOf[g] = n <- agg<<n = count()>>
  genderOf[_] = g .
// Computes the number of presidents for each gender.
minBirthYearOf[g] = n -> Gender(g), int(n).
minBirthYearOf[g] = n <-
    agg<<n = min(y)>> genderOf[p] = g ,
    birthyearOf[p] = y.
// Computes the minimum birth year for each gender.
```

TABLE H.12 Illustration of Group by Clause

presidentName	countryCode	Gender	birthYear
Tarja Halonen	FI	F	1943
Pratibha Patil	IN	F	1934
Christian Wulff	DE	M	1959
Nicolas Sarkozy	FR	M	1955
Barack Obama	US	M	1961

The following query may now be used to return the required result:

```
_(g, np, mby) <-
    nrOfPresidentsOf[g] = np,
    minBirthYearOf[g] = mby.
```

The program, data, and query codes for the above grouping examples are accessible as `SQL3.logic` and `SQL3Data.logic`.

Now consider Table H.13. This table records, for each item, region, and quarter, the number of items sold in that region during that quarter. To save space, we include just two items and two regions, and limit the quarters to a single year. The key of this table is itemCode, region, and quarter. To help with later discussion, the table is displayed as partitioned into four groups, where all rows in any specific group share the same item and region.

The LogiQL schema for this table may be set out as follows;

```
Item(i), hasItemCode(i:c) -> string(c).
Region(r), hasRegionName(r:rn) -> string(rn).
Quarter(q), hasQuarterNr(q:qn) -> int(qn).
nrSoldOf_In_In_[i, r, q] = n -> Item(i), Region(r),
    Quarter(q), int(n).
hasRegionName(_:rn) -> rn = "East" ; rn = "West".
hasQuarterNr(_:qn) -> qn > = 1, qn <= 4.
```

TABLE H.13 Product Sales Report

itemCode	Region	Quarter	Number Sold
BBB	East	1	50
BBB	East	2	100
BBB	East	3	100
BBB	East	4	150
BBB	West	1	100
BBB	West	2	150
BBB	West	3	200
BBB	West	4	250
DL	East	1	20
DL	East	2	30
DL	East	3	40
DL	East	4	50
DL	West	1	50
DL	West	2	100
DL	West	3	100
DL	West	4	150

Now suppose we wish to list for each item and region the total number of items sold over all quarters. The query groups by multiple criteria, in this case item and region. The SQL query may be formulated as follows, yielding the result shown:

```
select itemCode, region sum(nrSold)
from Sale
group by itemCode, region
```

Result:

```
BBB   East  400
BBB   West  700
DL    East  140
DL    West  400
```

In LogiQL, sums are computed by the `total` function. To sum over the quarters for each item–region combination, we include the two grouping criteria as the arguments of the required function, which may be computed using the following derivation rule:

```
totalNrSoldOf_In_[i, r] = n -> string(i), string(r),
    int(n).
totalNrSoldOf_In_[i, r] = n <-
    agg<<n = total(qty)>>
    nrSoldOf_In_In_[i, r, _] = qty.
// Computes for each item and region combination
// the total number of items sold.
```

Querying the `totalNrSoldOf_In_` predicate now gives the same result set as output by the SQL query. The program, data, and query code for the above grouping example are accessible as `SQL4.logic` and `SQL4Data.logic`.