## 4.3 Declarative Programming

In addition to streams, data values are often stored in large repositories called databases. A database consists of a data store containing the data values along with an interface for retrieving and transforming those values. Each value stored in a database is called a *record*. Records with similar structure are grouped into tables. Records are retrieved and transformed using queries, which are statements in a query language. By far the most ubiquitous query language in use today is called Structured Query Language or SQL (pronounced "sequel").

SQL is an example of a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the *query interpreter* of the database system to design and perform a computational process to produce such a result.

This interaction differs substantially from the procedural programming paradigm of Python or Scheme. In Python, computational processes are described directly by the programmer. A declarative language abstracts away procedural details, instead focusing on the form of the result.

### 4.3.1 Tables

The SQL language is standardized, but most database systems implement some custom variant of the language that is endowed with proprietary features. In this text, we will describe a small subset of SQL as it is implemented in Sqlite. You can follow along by downloading Sqlite or by using this online SQL interpreter.

A table, also called a *relation*, has a fixed number of named and typed columns. Each row of a table represents a data record and has one value for each column. For example, a table of cities might have columns `latitude` `longitude` that both hold numeric values, as well as a column `name` that holds a string. Each row would represent a city location position by its latitude and longitude values.

| Latitude | Longitude | Name |
|---|---|---|
| 38 | 122 | Berkeley |
| 42 | 71 | Cambridge |
| 45 | 93 | Minneapolis |

A table with a single row can be created in the SQL language using a `select` statement, in which the row values are separated by commas and the column names follow the keyword "as". All SQL statements end in a semicolon.

```
sqlite> select 38 as latitude, 122 as longitude, "Berkeley" as name;
38|122|Berkeley
```

The second line is the output, which includes one line per row with columns separated by a vertical bar.

A multi-line table can be constructed by *union*, which combines the rows of two tables. The column names of the left table are used in the constructed table. Spacing within a line does not affect the result.

```
sqlite> select 38 as latitude, 122 as longitude, "Berkeley" as name union
   ...> select 42,             71,               "Cambridge"        union
   ...> select 45,             93,               "Minneapolis";
38|122|Berkeley
42|71|Cambridge
45|93|Minneapolis
```

A table can be given a name using a `create table` statement. While this statement can also be used to create empty tables, we will focus on the form that gives a name to an existing table defined by a `select` statement.

```
sqlite> create table cities as
   ...>   select 38 as latitude, 122 as longitude, "Berkeley" as name union
   ...>   select 42,             71,               "Cambridge"        union
   ...>   select 45,             93,               "Minneapolis";
```

Once a table is named, that name can be used in a `from` clause within a `select` statement. All columns of a table can be displayed using the special `select *` form.

```
sqlite> select * from cities;
38|122|Berkeley
42|71|Cambridge
45|93|Minneapolis
```

### 4.3.2 Select Statements

A `select` statement defines a new table either by listing the values in a single row or, more commonly, by projecting an existing table using a `from` clause:

```
select [column description] from [existing table name]
```

The columns of the resulting table are described by a comma-separated list of expressions that are each evaluated for each row of the existing input table.

For example, we can create a two-column table that describes each city by how far north or south it is of Berkeley. Each degree of latitude measures 60 nautical miles to the north.

```
sqlite> select name, 60*abs(latitude-38) from cities;
Berkeley|0
Cambridge|240
Minneapolis|420
```

Column descriptions are expressions in a language that shares many properties with Python: infix operators such as + and %, built-in functions such as `abs` and `round`, and parentheses that describe evaluation order. Names in these expressions, such as `latitude` above, evaluate to the column value in the row being projected.

Optionally, each expression can be followed by the keyword `as` and a column name. When the entire table is given a name, it is often helpful to give each column a name so that it can

be referenced in future `select` statements. Columns described by a simple name are named automatically.

```
sqlite> create table distances as
   ...>   select name, 60*abs(latitude-38) as distance from cities;
sqlite> select distance/5, name from distances;
0|Berkeley
48|Cambridge
84|Minneapolis
```

**Where Clauses.** A `select` statement can also include a `where` clause with a filtering expression. This expression filters the rows that are projected. Only a row for which the filtering expression evaluates to a true value will be used to produce a row in the resulting table.

```
sqlite> create table cold as
   ...>   select name from cities where latitude > 43;
sqlite> select name, "is cold!" from cold;
Minneapolis|is cold!
```

**Order Clauses.** A `select` statement can also express an ordering over the resulting table. An `order` clause contains an ordering expression that is evaluated for each unfiltered row. The resulting values of this expression are used as a sorting criterion for the result table.

```
sqlite> select distance, name from distances order by -distance;
84|Minneapolis
48|Cambridge
0|Berkeley
```

The combination of these features allows a `select` statement to express a wide range of projections of an input table into a related output table.

### 4.3.3 Joins

Databases typically contain multiple tables, and queries can require information contained within different tables to compute a desired result. For instance, we may have a second table describing the mean daily high temperature of different cities.

```
sqlite> create table temps as
   ...>   select "Berkeley" as city, 68 as temp union
   ...>   select "Chicago"         , 59         union
   ...>   select "Minneapolis"     , 55;
```

Data are combined by *joining* multiple tables together into one, a fundamental operation in database systems. There are many methods of joining, all closely related, but we will focus on just one method in this text. When tables are joined, the resulting table contains a new row for each combination of rows in the input tables. If two tables are joined and the left table has $m$ rows and the right table has $n$ rows, then the joined table will have $m \cdot n$ rows. Joins are expressed in SQL by separating table names by commas in the `from` clause of a `select` statement.

```
sqlite> select * from cities, temps;
38|122|Berkeley|Berkeley|68
38|122|Berkeley|Chicago|59
38|122|Berkeley|Minneapolis|55
42|71|Cambridge|Berkeley|68
```

```
42|71|Cambridge|Chicago|59
42|71|Cambridge|Minneapolis|55
45|93|Minneapolis|Berkeley|68
45|93|Minneapolis|Chicago|59
45|93|Minneapolis|Minneapolis|55
```

Joins are typically accompanied by a `where` clause that expresses a relationship between the two tables. For example, if we wanted to collect data into a table that would allow us to correlate latitude and temperature, we would select rows from the join where the same city is mentioned in each. Within the `cities` table, the city name is stored in a column called `name`. Within the `temps` table, the city name is stored in a column called `city`. The `where` clause can select for rows in the joined table in which these values are equal. In SQL, numeric equality is tested with a single `=` symbol.

```
sqlite> select name, latitude, temp from cities, temps where name = city;
Berkeley|38|68
Minneapolis|45|55
```

Tables may have overlapping column names, and so we need a method for disambiguating column names by table. A table may also be joined with itself, and so we need a method for disambiguating tables. To do so, SQL allows us to give aliases to tables within a `from` clause using the keyword `as` and to refer to a column within a particular table using a dot expression. The following `select` statement computes the temperature difference between pairs of unequal cities. The alphabetical ordering constraint in the `where` clause ensures that each pair will only appear once in the result.

```
sqlite> select a.city, b.city, a.temp - b.temp
   ...>        from temps as a, temps as b where a.city < b.city;
Berkeley|Chicago|10
Berkeley|Minneapolis|15
Chicago|Minneapolis|5
```

Our two means of combining tables in SQL, join and union, allow for a great deal of expressive power in the language.

### 4.3.4 Interpreting SQL

In order to create an interpreter for the subset of SQL we have introduced so far, we need to create a representation for tables, a parser for statements written as text, and an evaluator for parsed statements. The sql interpreter example includes all of these components, providing a simple but functional demonstration of a declarative language interpreter.

In this implementation, each table has its own a class, and each row in a table is represented by an instance of its table's class. A row has one attribute per column in the table, and a table is a sequence of rows.

The class for a table is created using the namedtuple function in the `collections` package of the Python standard library, which returns a new sub-class of `tuple` that gives names to each element in the tuple.

Consider the `cities` table from the previous section, repeated below.

```
sqlite> create table cities as
   ...>    select 38 as latitude, 122 as longitude, "Berkeley" as name union
   ...>    select 42,             71,               "Cambridge"        union
   ...>    select 45,             93,               "Minneapolis";
```

The following Python statements construct a representation for this table.

```
>>> from collections import namedtuple
>>> CitiesRow = namedtuple("Row", ["latitude", "longitude", "name"])
>>> cities = [CitiesRow(38, 122, "Berkeley"),
              CitiesRow(42,  71, "Cambridge"),
              CitiesRow(43,  93, "Minneapolis")]
```

The result of a `select` statement can be interpreted using sequence operations. Consider the `distances` table from the previous section, repeated below.

```
sqlite> create table distances as
   ...>   select name, 60*abs(latitude-38) as distance from cities;
sqlite> select distance/5, name from distances;
0|Berkeley
48|Cambridge
84|Minneapolis
```

This table is generated from the `name` and `latitude` columns of the `cities` table. This resulting table can be generated by mapping a function over the rows of the input table, a function that returns a `DistancesRow` for each `CitiesRow`.

```
>>> DistancesRow = namedtuple("Row", ["name", "distance"])
>>> def select(cities_row):
        latitude, longitude, name = cities_row
        return DistancesRow(name, 60*abs(latitude-38))
>>> distances = list(map(select, cities))
>>> for row in distances:
        print(row)
Row(name='Berkeley', distance=0)
Row(name='Cambridge', distance=240)
Row(name='Minneapolis', distance=300)
```

The design of our SQL interpreter generalizes this approach. A `select` statement is represented as an instance of a class `Select` that is constructed from the clauses of the select statement.

```
>>> class Select:
        """select [columns] from [tables] where [condition] order by [order]."""
        def __init__(self, columns, tables, condition, order):
            self.columns = columns
            self.tables = tables
            self.condition = condition
            self.order = order
            self.make_row = create_make_row(self.columns)
        def execute(self, env):
            """Join, filter, sort, and map rows from tables to columns."""
            from_rows = join(self.tables, env)
            filtered_rows = filter(self.filter, from_rows)
            ordered_rows = self.sort(filtered_rows)
            return map(self.make_row, ordered_rows)
        def filter(self, row):
            if self.condition:
                return eval(self.condition, row)
            else:
                return True
        def sort(self, rows):
            if self.order:
                return sorted(rows, key=lambda r: eval(self.order, r))
            else:
                return rows
```

The `execute` method joins input tables, filters and orders the resulting rows, then maps a function called `make_row` over those resulting rows. The `make_row` function is created in the `Select` constructor by a call to `create_make_row`, a higher-order function that creates a new class for the resulting table and defines how to project an input row to an output row. (A version of this function with more error handling and special cases appears in sql.)

```
>>> def create_make_row(description):
        """Return a function from an input environment (dict) to an output row.
        description -- a comma-separated list of [expression] as [column name]
        """
        columns = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
            else:
                expression, name = column, column
            expressions.append(expression)
            names.append(name)
        row = namedtuple("Row", names)
        return lambda env: row(*[eval(e, env) for e in expressions])
```

Finally, we need to define the `join` function that creates the input rows. Given an `env` dictionary contains existing tables (lists of rows) keyed by their name, the `join` function groups together all combinations of rows in the input tables using the product function in the `itertools` package. It maps a function called `make_env` over the joined rows, a function that converts each combination of rows into a dictionary so that it can be used to evaluate expressions. (A version of this function with more error handling and special cases appears in sql.)

```
>>> from itertools import product
>>> def join(tables, env):
        """Return an iterator over dictionaries from names to values in a row.
        tables -- a comma-separate sequences of table names
        env    -- a dictionary from global names to tables
        """
        names = tables.split(", ")
        joined_rows = product(*[env[name] for name in names])
        return map(lambda rows: make_env(rows, names), joined_rows)
>>> def make_env(rows, names):
        """Create an environment of names bound to values."""
        env = dict(zip(names, rows))
        for row in rows:
            for name in row._fields:
                env[name] = getattr(row, name)
        return env
```

Above, `row._fields` evaluates to the column names of the table containing the `row`. The `_fields` attribute exists because the type of `row` is a `namedtuple` class.

Our interpreter is complete enough to execute `select` statements. For instance, we can compute the latitude distance from Berkeley for all other cities, ordered by their longitude.

```
>>> env = {"cities": cities}
>>> select = Select("name, 60*abs(latitude-38) as distance",
                    "cities", "name != 'Berkeley'", "-longitude")
>>> for row in select.execute(env):
        print(row)
```

```
Row(name='Minneapolis', distance=300)
Row(name='Cambridge', distance=240)
```

The example above is equivalent to the following SQL statement.

```
sqlite> select name, 60*abs(latitude-38) as distance
   ...>        from cities where name != "Berkeley" order by -longitude;
Minneapolis|420
Cambridge|240
```

We can also store this resulting table in the environment and join it with the `cities` table, retrieving the longitude for each city.

```
>>> env["distances"] = list(select.execute(env))
>>> joined = Select("cities.name as name, distance, longitude", "cities, distances",
                    "cities.name == distances.name", None)
>>> for row in joined.execute(env):
        print(row)
Row(name='Cambridge', distance=240, longitude=71)
Row(name='Minneapolis', distance=300, longitude=93)
```

The example above is equivalent to the following SQL statement.

```
sqlite> select cities.name as name, distance, longitude
   ...>        from cities, distances where cities.name = distances.name;
Cambridge|240|71
Minneapolis|420|93
```

The full sql example program also contains a simple parser for `select` statements, as well as `execute` methods for `create table` and `union`. The interpreter can correctly execute all SQL statements included within the text so far. While this simple interpreter only implements a small amount of the full Structured Query Language, its structure demonstrates the relationship between sequence processing operations and query languages.

**Query Plans.** Declarative languages describe the form of a result, but do not explicitly describe how that result should be computed. This interpreter always joins, filters, orders, and then projects the input rows in order to compute the result rows. However, more efficient ways to compute the same result may exist, and query interpreters are free to choose among them. Choosing efficient procedures for computing query results is a core feature of database systems.

For example, consider the final select statement above. Rather than computing the join of `cities` and `distances` and then filtering the result, the same result may be computed by first sorting both tables by the `name` column and then joining only the rows that have the same name in a linear pass through the sorted tables. When tables are large, efficiency gains from query plan selection can be substantial.

### 4.3.5 Recursive Select Statements

Select statements can optionally include a `with` clause that generates and names additional tables used in computing the final result. The full syntax of a select statement, not including unions, has the following form:

```
with [tables] select [columns] from [names] where [condition] order by [order]
```

We have already demonstrated the allowed values for `[columns]` and `[names]`. `[condition]` and `[order]` are expressions that can be evaluated for an input row. The `[tables]` portion is a

comma-separated list of table descriptions of the form:

```
[table name]([column names]) as ([select statement])
```

Any `select` statement can be used to describe a table within `[tables]`.

For instance, the `with` clause below declares a table `states` containing cities and their states. The `select` statement computes pairs of cities within the same state.

```
sqlite> with
   ...>   states(city, state) as (
   ...>     select "Berkeley",  "California"    union
   ...>     select "Boston",    "Massachusetts" union
   ...>     select "Cambridge", "Massachusetts" union
   ...>     select "Chicago",   "Illinois"      union
   ...>     select "Pasadena",  "California"
   ...>   )
   ...> select a.city, b.city, a.state from states as a, states as b
   ...>        where a.state = b.state and a.city < b.city;
Berkeley|Pasadena|California
Boston|Cambridge|Massachusetts
```

A table defined within a `with` clause may have a single recursive case that defines output rows in terms of other output rows. For example, the `with` clause below defines a table of integers from 5 to 15, of which the odd values are selected and squared.

```
sqlite> with
   ...>   ints(n) as (
   ...>     select 5 union
   ...>     select n+1 from ints where n < 15
   ...>   )
   ...> select n, n*n from ints where n % 2 = 1;
5|25
7|49
9|81
11|121
13|169
15|225
```

Multiple tables can be defined in a `with` clause, separated by commas. The example below computes all Pythagorean triples from a table of integers, their squares, and the sums of pairs of squares. A Pythagorean triple consists of integers $a$, $b$, and $c$ such that $a^2 + b^2 = c^2$.

```
sqlite> with
   ...>   ints(n) as (
   ...>     select 1 union select n+1 from ints where n < 20
   ...>   ),
   ...>   squares(x, xx) as (
   ...>     select n, n*n from ints
   ...>   ),
   ...>   sum_of_squares(a, b, sum) as (
   ...>     select a.x, b.x, a.xx + b.xx
   ...>            from squares as a, squares as b where a.x < b.x
   ...>   )
   ...> select a, b, x from squares, sum_of_squares where sum = xx;
3|4|5
6|8|10
5|12|13
9|12|15
8|15|17
12|16|20
```

Designing recursive queries involves ensuring that the appropriate information is available in each input row to compute a result row. To compute Fibonacci numbers, for example, the input row needs not only the current but also the previous element in order to compute the next element.

```
sqlite> with
   ...>   fib(previous, current) as (
   ...>     select 0, 1 union
   ...>     select current, previous+current from fib
   ...>     where current <= 100
   ...>   )
   ...> select previous from fib;
0
1
1
2
3
5
8
13
21
34
55
89
```

These examples demonstrate that recursion is a powerful means of combination, even in declarative languages.

**Building strings**. Two strings can be concatenated into a longer string using the `||` operator in SQL.

```
sqlite> with wall(n) as (
  ....>   select 99 union select 98 union select 97
  ....> )
  ....> select n || " bottles" from wall;
99 bottles
98 bottles
97 bottles
```

This feature can be used to construct sentences by concatenating phrases. For example, one way to construct an English sentence is to concatenate a subject noun phrase, a verb, and an object noun phrase.

```
sqlite> create table nouns as
  ....>   select "the dog" as phrase union
  ....>   select "the cat"           union
  ....>   select "the bird";
sqlite> select subject.phrase || " chased " || object.phrase
  ....>        from nouns as subject, nouns as object
  ....>        where subject.phrase != object.phrase;
the bird chased the cat
the bird chased the dog
the cat chased the bird
the cat chased the dog
the dog chased the bird
the dog chased the cat
```

As an exercise, use a recursive local table to generate sentences such as, "the dog that chased the cat that chased the bird also chased the bird."

### 4.3.6 Aggregation and Grouping

The `select` statements introduced so far can join, project, and manipulate individual rows. In addition, a `select` statement can perform aggregation operations over multiple rows. The aggregate functions `max`, `min`, `count`, and `sum` return the maximum, minimum, number, and sum of the values in a column. Multiple aggregate functions can be applied to the same set of rows by defining more than one column. Only columns that are included by the `where` clause are considered in the aggreagation.

```
sqlite> create table animals as
  ....>   select "dog" as name, 4 as legs, 20 as weight union
  ....>   select "cat"        , 4        , 10           union
  ....>   select "ferret"     , 4        , 10           union
  ....>   select "t-rex"      , 2        , 12000        union
  ....>   select "penguin"    , 2        , 10           union
  ....>   select "bird"       , 2        , 6;
sqlite> select max(legs) from animals;
4
sqlite> select sum(weight) from animals;
12056
sqlite> select min(legs), max(weight) from animals where name <> "t-rex";
2|20
```

The `distinct` keyword ensures that no repeated values in a column are included in the aggregation. Only two distinct values of `legs` appear in the `animals` table. The special `count(*)` syntax counts the number of rows.

```
sqlite> select count(legs) from animals;
6
sqlite> select count(*) from animals;
6
sqlite> select count(distinct legs) from animals;
2
```

Each of these `select` statements has produced a table with a single row. The `group by` and `having` clauses of a `select` statement are used to partition rows into groups and select only a subset of the groups. Any aggregate functions in the `having` clause or column description will apply to each group independently, rather than the entire set of rows in the table.

For example, to compute the maximum weight of both a four-legged and a two-legged animal from this table, the first statement below groups together dogs and cats as one group and birds as a separate group. The result indicates that the maximum weight for a two-legged animal is 3 (the bird) and for a four-legged animal is 20 (the dog). The second query lists the values in the `legs` column for which there are at least two distinct names.

```
sqlite> select legs, max(weight) from animals group by legs;
2|12000
4|20
sqlite> select weight from animals group by weight having count(*)>1;
10
```

Multiple columns and full expressions can appear in the `group by` clause, and groups will be formed for every unique combination of values that result. Typically, the expression used for grouping also appears in the column description, so that it is easy to identify which result row resulted from each group.

```
sqlite> select max(name) from animals group by legs, weight order by name;
bird
dog
ferret
penguin
t-rex
sqlite> select max(name), legs, weight from animals group by legs, weight
  ....>   having max(weight) < 100;
bird|2|6
penguin|2|10
ferret|4|10
dog|4|20
sqlite> select count(*), weight/legs from animals group by weight/legs;
2|2
1|3
2|5
1|6000
```

A `having` clause can contain the same filtering as a `where` clause, but can also include calls to aggregate functions. For the fastest execution and clearest use of the language, a condition that filters individual rows based on their contents should appear in a `where` clause, while a `having` clause should be used only when aggregation is required in the condition (such as specifying a minimum `count` for a group).

When using a `group by` clause, column descriptions can contain expressions that do not aggregate. In some cases, the SQL interpreter will choose the value from a row that corresponds to another column that includes aggregation. For example, the following statement gives the `name` of an animal with maximal `weight`.

```
sqlite> select name, max(weight) from animals;
t-rex|12000
sqlite> select name, legs, max(weight) from animals group by legs;
t-rex|2|12000
dog|4|20
```

However, whenever the row that corresponds to aggregation is unclear (for instance, when aggregating with `count` instead of `max`), the value chosen may be arbitrary. For the clearest and most predictable use of the language, a `select` statement that includes a `group by` clause should include at least one aggregate column and only include non-aggregate columns if their contents is predictable from the aggregation.

*Continue*: 4.4 Logic Programming