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 mm rows and the right table has nn rows, then the joined table will have $m \cdot n$m·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 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.

4.3.5 Create Table and Drop Table

The `create table` statement creates a new table in our database. As we saw earlier, we can combine the `create table` statement with the `select` statement to give a name to an existing table, but we can also use the `create table` statement along with a list of column names to create an empty table. For each column, we can optionally include the `unique` keyword, which indicates that the column can only contain unique values, or the `default` keyword, which gives a default value for an item in the column. For the entire `create table` statement, including the optional `if not exists` clause will prevent an error if we attempt to create duplicate tables.

The `drop table` statement deletes a table from our database. Including the optional `if exists` clause will prevent an error if we attempt to drop a non-existing table.

```
sqlite> create table primes (n, prime);
sqlite> drop table primes;
sqlite> drop table if exists primes;
```

```
sqlite> create table primes (n unique, prime default 1);
sqlite> create table if not exists primes (n, prime);
```

4.3.6 Modifying Tables

The `insert into` statement allows us to add rows to a table in our database. In particular, we can insert values into all columns of our table, or we can add to one specific column, which will set the other columns to their default values. By combining the `insert into` and `select` statements, we can add the rows of an existing table to our table.

```
sqlite> insert into primes values (2, 1), (3, 1);
sqlite> select * from primes;
2|1
3|1
sqlite> insert into primes(n) values (4), (5);
sqlite> select * from primes;
2|1
3|1
4|1
5|1
sqlite> insert into primes(n) select n + 4 from primes;
sqlite> select * from primes;
2|1
3|1
4|1
5|1
6|1
7|1
8|1
9|1
```

The `update` statement sets all entries in certain columns of a table to new values for a subset of rows as indicated by an optional `where` clause. We can update all rows by omitting the optional `where` clause.

The `delete from` statement deletes a subset of rows of a table as indicated by an optional `where` clause. If we do not include a `where` clause, then we will delete all rows, but an empty table would remain in our database.

```
sqlite> update primes set prime = 0 where n > 2 and n % 2 = 0;
sqlite> update primes set prime = 0 where n > 3 and n % 3 = 0;
sqlite> select * from primes;
2|1
3|1
4|0
5|1
6|0
7|1
8|0
9|0
sqlite> delete from primes where prime = 0;
sqlite> select * from primes;
2|1
3|1
5|1
7|1
```

Continue: 4.4 Logic Programming