Combining Data

Objectives

- Explain the operation of a query that joins two tables.
- Explain how to restrict the output of a query containing a join to only include meaningful combinations of values.
- Write queries that join tables on equal keys.
- Explain what primary and foreign keys are, and why they are useful.
- Explain what atomic values are, and why database fields should only contain atomic values.

In order to submit her data to a web site that aggregates historical meteorological data, Gina needs to format it as latitude, longitude, date, quantity, and reading. However, her latitudes and longitudes are in the `Site` table, while the dates of measurements are in the `Visited` table and the readings themselves are in the `Survey` table. She needs to combine these tables somehow.

The SQL command to do this is `join`. To see how it works, let's start by joining the `Site` and `Visited` tables:

```
%load_ext sqlitemagic
```

```
%%sqlite survey.db
select * from Site join Visited;
```

```
DR-1 -49.85 -128.57 619 DR-1 1927-02-08
DR-1 -49.85 -128.57 622 DR-1 1927-02-10
DR-1 -49.85 -128.57 734 DR-3 1939-01-07
DR-1 -49.85 -128.57 735 DR-3 1930-01-12
DR-1 -49.85 -128.57 751 DR-3 1930-02-26
DR-1 -49.85 -128.57 752 DR-3 None
DR-1 -49.85 -128.57 837 MSK-4 1932-01-14
DR-1 -49.85 -128.57 844 DR-1 1932-03-22
DR-3 -47.15 -126.72 619 DR-1 1927-02-08
DR-3 -47.15 -126.72 622 DR-1 1927-02-10
DR-3 -47.15 -126.72 734 DR-3 1939-01-07
DR-3 -47.15 -126.72 735 DR-3 1930-01-12
DR-3 -47.15 -126.72 751 DR-3 1930-02-26
DR-3 -47.15 -126.72 752 DR-3 None
DR-3 -47.15 -126.72 837 MSK-4 1932-01-14
DR-3 -47.15 -126.72 844 DR-1 1932-03-22
MSK-4 -48.87 -123.4  619 DR-1 1927-02-08
MSK-4 -48.87 -123.4  622 DR-1 1927-02-10
MSK-4 -48.87 -123.4  734 DR-3 1939-01-07
MSK-4 -48.87 -123.4  735 DR-3 1930-01-12
MSK-4 -48.87 -123.4  751 DR-3 1930-02-26
MSK-4 -48.87 -123.4  752 DR-3 None
MSK-4 -48.87 -123.4  837 MSK-4 1932-01-14
MSK-4 -48.87 -123.4  844 DR-1 1932-03-22
```

`join` creates the cross product (../../gloss.html#cross-product) of two tables, i.e., it joins each record of one with each record of the other to give all possible combinations. Since there are three records in `Site` and eight in `Visited`, the join's output has 24 records. And since each table has three fields, the output has six fields.

What the join *hasn't* done is figure out if the records being joined have anything to do with each other. It has no way of knowing whether they do or not until we tell it how. To do that, we add a clause specifying that we're only interested in combinations that have the same site name:

```
%%sqlite survey.db
select * from Site join Visited on Site.name=Visited.site;
```

```
DR-1 -49.85 -128.57 619 DR-1 1927-02-08
DR-1 -49.85 -128.57 622 DR-1 1927-02-10
DR-1 -49.85 -128.57 844 DR-1 1932-03-22
DR-3 -47.15 -126.72 734 DR-3 1939-01-07
DR-3 -47.15 -126.72 735 DR-3 1930-01-12
DR-3 -47.15 -126.72 751 DR-3 1930-02-26
DR-3 -47.15 -126.72 752 DR-3 None
MSK-4 -48.87 -123.4  837 MSK-4 1932-01-14
```

`on` does the same job as `where`: it only keeps records that pass some test. (The difference between the two is that `on` filters records as they're being created, while `where` waits until the join is done and then does the filtering.) Once we add this to our query, the database manager throws away records that combined information about two different sites, leaving us with just the ones we want.

Notice that we used `table.field` to specify field names in the output of the join. We do this because tables can have fields with the same name, and we need to be specific which ones we're talking about. For example, if we joined the `person` and `visited` tables, the result would inherit a field called `ident` from each of the original tables.

We can now use the same dotted notation to select the three columns we actually want out of our join:

```
%%sqlite survey.db
select Site.lat, Site.long, Visited.dated
from   Site join Visited
on     Site.name=Visited.site;
```

```
-49.85 -128.57 1927-02-08
-49.85 -128.57 1927-02-10
-49.85 -128.57 1932-03-22
-47.15 -126.72 None
-47.15 -126.72 1930-01-12
-47.15 -126.72 1930-02-26
-47.15 -126.72 1939-01-07
-48.87 -123.4  1932-01-14
```

If joining two tables is good, joining many tables must be better. In fact, we can join any number of tables simply by adding more `join` clauses to our query, and more `on` tests to filter out combinations of records that don't make sense:

```
%%sqlite survey.db
select Site.lat, Site.long, Visited.dated, Survey.quant, Survey.readi
ng
from   Site join Visited join Survey
on     Site.name=Visited.site
and    Visited.ident=Survey.taken
and    Visited.dated is not null;
```

```
-49.85 -128.57 1927-02-08 rad  9.82
-49.85 -128.57 1927-02-08 sal  0.13
-49.85 -128.57 1927-02-10 rad  7.8
-49.85 -128.57 1927-02-10 sal  0.09
-47.15 -126.72 1939-01-07 rad  8.41
-47.15 -126.72 1939-01-07 sal  0.05
-47.15 -126.72 1939-01-07 temp -21.5
-47.15 -126.72 1930-01-12 rad  7.22
-47.15 -126.72 1930-01-12 sal  0.06
-47.15 -126.72 1930-01-12 temp -26.0
-47.15 -126.72 1930-02-26 rad  4.35
-47.15 -126.72 1930-02-26 sal  0.1
-47.15 -126.72 1930-02-26 temp -18.5
-48.87 -123.4  1932-01-14 rad  1.46
-48.87 -123.4  1932-01-14 sal  0.21
-48.87 -123.4  1932-01-14 sal  22.5
-49.85 -128.57 1932-03-22 rad  11.25
```

We can tell which records from `Site`, `Visited`, and `Survey` correspond with each other because those tables contain primary keys (../../gloss.html#primary-key) and foreign keys (../../gloss.html#foreign-key). A primary key is a value, or combination of values, that uniquely identifies each record in a table. A foreign key is a value (or combination of values) from one table that identifies a unique record in another table. Another way of saying this is that a foreign key is the primary key of one table that appears in some other table. In our database, `Person.ident` is the primary key in the `Person` table, while `Survey.person` is a foreign key relating the `Survey` table's entries to entries in `Person`.

Most database designers believe that every table should have a well-defined primary key. They also believe that this key should be separate from the data itself, so that if we ever need to change the data, we only need to make one change in one place. One easy way to do this is to create an arbitrary, unique ID for each record as we add it to the database. This is actually very common: those IDs have names like "student numbers" and "patient numbers", and they almost always turn out to have originally been a unique record identifier in some database system or other. As the query below demonstrates, SQLite automatically numbers records as they're added to tables, and we can use those record numbers in queries:

```
%%sqlite survey.db
select rowid, * from Person;
```

```
1 dyer     William  Dyer
2 pb       Frank    Pabodie
3 lake     Anderson Lake
4 roe      Valentina Roerich
5 danforth Frank    Danforth
```

Data Hygiene

Now that we have seen how joins work, we can see why the relational model is so useful and how best to use it. The first rule is that every value should be atomic (../../gloss.html#atomic-value), i.e., not contain parts that we might want to work with separately. We store personal and family names in separate columns instead of putting the entire name in one column so that we don't have to use substring operations to get the name's components. More importantly, we store the two parts of the name separately because splitting on spaces is unreliable: just think of a name like "Eloise St. Cyr" or "Jan Mikkel Steubart".

The second rule is that every record should have a unique primary key. This can be a serial number that has no intrinsic meaning, one of the values in the record (like the `ident` field in the `Person` table), or even a combination of values: the triple `(taken, person, quant)` from the `Survey` table uniquely identifies every measurement.

The third rule is that there should be no redundant information. For example, we could get rid of the `Site` table and rewrite the `Visited` table like this:

619 -49.85 -128.57 1927-02-08
622 -49.85 -128.57 1927-02-10
734 -47.15 -126.72 1939-01-07
735 -47.15 -126.72 1930-01-12
751 -47.15 -126.72 1930-02-26
752 -47.15 -126.72 null
837 -48.87 -123.40 1932-01-14
844 -49.85 -128.57 1932-03-22

In fact, we could use a single table that recorded all the information about each reading in each row, just as a spreadsheet would. The problem is that it's very hard to keep data organized this way consistent: if we realize that the date of a particular visit to a particular site is wrong, we have to change multiple records in the database. What's worse, we may have to guess which records to change, since other sites may also have been visited on that date.

The fourth rule is that the units for every value should be stored explicitly. Our database doesn't do this, and that's a problem: Roerich's salinity measurements are several orders of magnitude larger than anyone else's, but we don't know if that means she was using parts per million instead of parts per thousand, or whether there actually was a saline anomaly at that site in 1932.

Stepping back, data and the tools used to store it have a symbiotic relationship: we use tables and joins because it's efficient, provided our data is organized a certain way, but organize our data that way because we have tools to manipulate it efficiently if it's in a certain form. As anthropologists say, the tool shapes the hand that shapes the tool.

Challenges

1. Write a query that lists all radiation readings from the DR-1 site.
2. Write a query that lists all sites visited by people named "Frank".
3. Describe in your own words what the following query produces:

```
select Site.name from Site join Visited
on Site.lat<-49.0 and Site.name=Visited.site and Visited.dated>='1932-00-00'
;
```

Key Points

- Every fact should be represented in a database exactly once.
- A join produces all combinations of records from one table with records from another.
- A primary key is a field (or set of fields) whose values uniquely identify the records in a table.
- A foreign key is a field (or set of fields) in one table whose values are a primary key in another table.
- We can eliminate meaningless combinations of records by matching primary keys and foreign keys between tables.
- Keys should be atomic values to make joins simpler and more efficient.
