Filtering

Objectives

- Write queries that select records that satisfy user-specified conditions.
- Explain the order in which the clauses in a query are executed.

One of the most powerful features of a database is the ability to filter (../../gloss.html#filter) data, i.e., to select only those records that match certain criteria. For example, suppose we want to see when a particular site was visited. We can select these records from the `Visited` table by using a `where` clause in our query:

```
%load_ext sqlitemagic
```

```
%%sqlite survey.db
select * from Visited where site='DR-1';
```

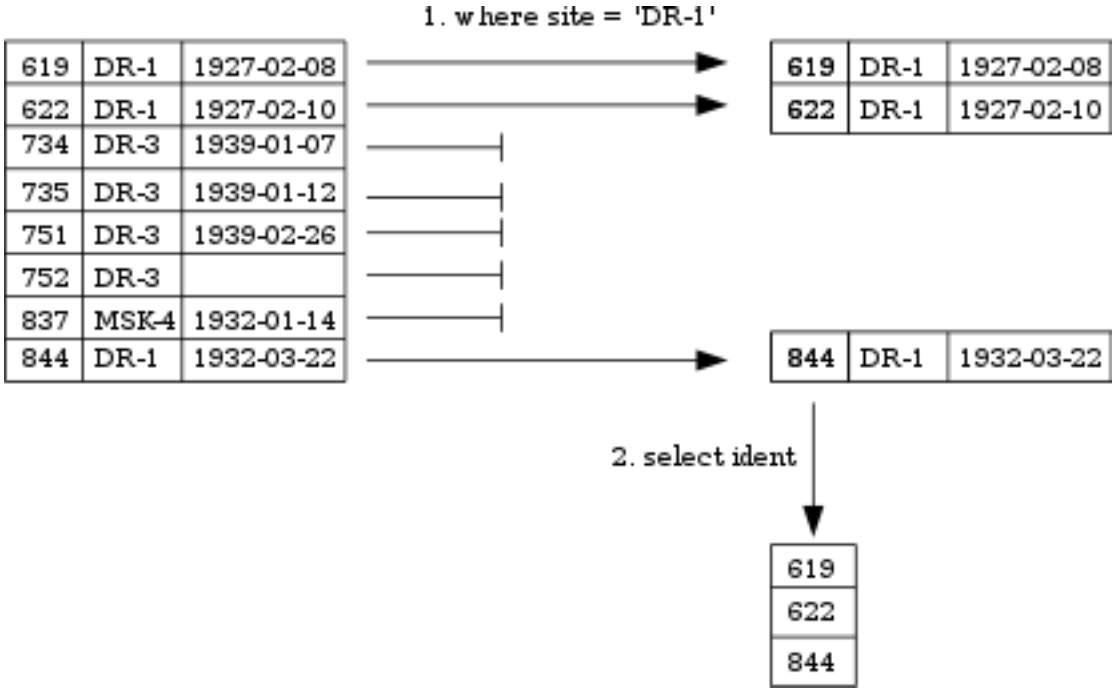
```
619 DR-1 1927-02-08
622 DR-1 1927-02-10
844 DR-1 1932-03-22
```

The database manager executes this query in two stages. First, it checks at each row in the `Visited` table to see which ones satisfy the `where`. It then uses the column names following the `select` keyword to determine what columns to display.

This processing order means that we can filter records using `where` based on values in columns that aren't then displayed:

```
%%sqlite survey.db
select ident from Visited where site='DR-1';
```

```
619
622
844
```



We can use many other Boolean operators to filter our data. For example, we can ask for all information from the DR-1 site collected since 1930:

```
%%sqlite survey.db
select * from Visited where (site='DR-1') and (dated>='1930-00-00');
```

```
844 DR-1 1932-03-22
```

(The parentheses around the individual tests aren't strictly required, but they help make the query easier to read.)

> Most database managers have a special data type for dates. In fact, many have two: one for dates, such as "May 31, 1971", and one for durations, such as "31 days". SQLite doesn't: instead, it stores dates as either text (in the ISO-8601 standard format "YYYY-MM-DD HH:MM:SS.SSSS"), real numbers (the number of days since November 24, 4714 BCE), or integers (the number of seconds since midnight, January 1, 1970). If this sounds complicated, it is, but not nearly as complicated as figuring out historical dates in Sweden (http://en.wikipedia.org/wiki/Swedish_calendar).

If we want to find out what measurements were taken by either Lake or Roerich, we can combine the tests on their names using `or`:

```
%%sqlite survey.db
select * from Survey where person='lake' or person='roe';
```

```
734 lake sal  0.05
751 lake sal  0.1
752 lake rad  2.19
752 lake sal  0.09
752 lake temp -16.0
752 roe  sal  41.6
837 lake rad  1.46
837 lake sal  0.21
837 roe  sal  22.5
844 roe  rad  11.25
```

Alternatively, we can use `in` to see if a value is in a specific set:

```
%%sqlite survey.db
select * from Survey where person in ('lake', 'roe');
```

```
734 lake sal  0.05
751 lake sal  0.1
752 lake rad  2.19
752 lake sal  0.09
752 lake temp -16.0
752 roe  sal  41.6
837 lake rad  1.46
837 lake sal  0.21
837 roe  sal  22.5
844 roe  rad  11.25
```

We can combine `and` with `or`, but we need to be careful about which operator is executed first. If we *don't* use parentheses, we get this:

```
%%sqlite survey.db
select * from Survey where quant='sal' and person='lake' or person='r
oe';
```

```
734 lake sal 0.05
751 lake sal 0.1
752 lake sal 0.09
752 roe  sal 41.6
837 lake sal 0.21
837 roe  sal 22.5
844 roe  rad 11.25
```

which is salinity measurements by Lake, and *any* measurement by Roerich. We probably want this instead:

```
%%sqlite survey.db
select * from Survey where quant='sal' and (person='lake' or person='
roe');
```

```
734 lake sal 0.05
751 lake sal 0.1
752 lake sal 0.09
752 roe  sal 41.6
837 lake sal 0.21
837 roe  sal 22.5
```

Finally, we can use `distinct` with `where` to give a second level of filtering:

```
%%sqlite survey.db
select distinct person, quant from Survey where person='lake' or pers
on='roe';
```

```
lake sal
lake rad
lake temp
roe  sal
roe  rad
```

But remember: `distinct` is applied to the values displayed in the chosen columns, not to the entire rows as they are being processed.

> What we have just done is how most people "grow" their SQL queries. We started with something simple that did part of what we wanted, then added more clauses one by one, testing their effects as we went. This is a good strategy—in fact, for complex queries it's often the *only* strategy—but it depends on quick turnaround, and on us recognizing the right answer when we get it.
> The best way to achieve quick turnaround is often to put a subset of data in a temporary database and run our queries against that, or to fill a small database with synthesized records. For example, instead of trying our queries against an actual database of 20 million Australians, we could run it against a sample of ten thousand, or write a small program to generate ten thousand random (but plausible) records and use that.

Challenges

1. Suppose we want to select all sites that lie more than 30° from the poles. Our first query is:

   ```
   select * from Site where (lat > -60) or (lat < 60);
   ```

   Explain why this is wrong, and rewrite the query so that it is correct.

2. Normalized salinity readings are supposed to be between 0.0 and 1.0. Write a query that selects all records from `Survey` with salinity values outside this range.

3. The SQL test `*column-name* like *pattern*` is true if the value in the named column matches the pattern given; the character '%' can be used any number of times in the pattern to mean "match zero or more characters".

Expression	Value
`'a' like 'a'`	True
`'a' like '%a'`	True
`'b' like '%a'`	False
`'alpha' like 'a%'`	True
`'alpha' like 'a%p%'`	True

The expression `*column-name* not like *pattern*` inverts the test. Using `like`, write a query that finds all the records in `Visited` that *aren't* from sites labelled 'DR-something'.

Key Points

- Use `where` to filter records according to Boolean conditions.
- Filtering is done on whole records, so conditions can use fields that are not actually displayed.
