XML

eXtensible Markup Language

XML package in R

- Handy functions for parsing XML
  - readHTMLTable: reads an HTML table into R
  - xmlParse: read an XML file into R
  - xmlValue: retrieve text content of a node
  - xmlSize: return the number of child nodes of a node
  - xmlSApply: applies the function to each child node of a node

- To read an XML file into R, use `xmlParse`

```
> doc = xmlParse(“plant.xml”)
```

and extract the root node using xmlRoot.

```
> catalog = xmlRoot(doc)
> class(catalog)
[1] "XMLNode"
```

To illustrate how we manipulate an XML object in R, we’ll take this data and reformat it into a data frame with one row for each plant.

`xmlParse` implements what is called the DOM (Document Object Model) parser. It reads the entire file into memory.

We don't have time to cover it, but you should be aware of another parsing model called SAX (Simple API for XML). It reads the document incrementally and is more memory efficient, but it is trickier to use.

The tree structure is represented in R as a list of lists.

We can access an element within a node (i.e., a child), using the usual `[[  ]]` indexing for lists.

```
> ## Look at the first plant node
> oneplant = catalog[[1]]
> class(oneplant)
[1] "XMLNode"
> print(oneplant)
<PLANT>
 <COMMON>Bloodroot</COMMON>
 <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
 <ZONE>4</ZONE>
 <LIGHT>Mostly Shady</LIGHT>
 <PRICE>$2.44</PRICE>
 <AVAILABILITY>031599</AVAILABILITY>
</PLANT>
```

Here is a leaf nodes
It contains the content.

- We can drill down further into the list:

```
> oneplant[['COMMON']]
<COMMON>Bloodroot</COMMON>
```

- Note that this doesn’t remove the markup. To do this, use the function `xmlValue`

```
> xmlValue(oneplant[['COMMON']])
[1] "Bloodroot"
> xmlValue(oneplant[['BOTANICAL']])
[1] "Sanguinaria canadensis"
```

- There are special XML versions of `lapply` and `sapply`, named `xmlApply xmlSApply`. Each takes an XMLNode object as its primary argument. They iterate over the node's children nodes, invoking the given function.
- Like `lapply, xmlApply` returns a list. Like `sapply`, `xmlSApply` returns a simpler data structure if possible.

To create the data frame:

1) What do you want to operate on (iterate over)?

2) What do you want to produce?

In our plants example, we can use xmlSApply to extract the common names of all the plants.

```
> common = xmlSApply(catalog, function(x){
+              xmlValue(x[['COMMON']])})
> head(common)
           PLANT                      PLANT                 PLANT
     "Bloodroot"                "Columbine"  "Marsh Marigold"
           PLANT                      PLANT                 PLANT
       "Cowslip"  "Dutchman's-Breeches"    "Ginger, Wild"
```

The elements of the root node are all
plant nodes,
like oneplant.

- Using the same strategy, we can create a full data frame with all variables.

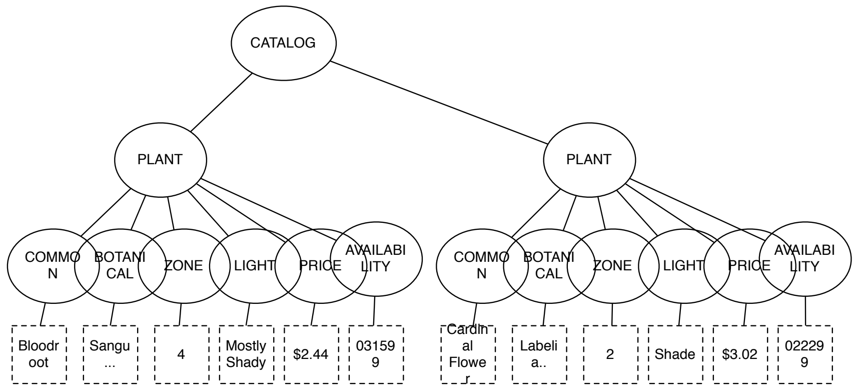
```
> getvar = function(x, var) xmlValue(x[[var]])
> varNames = names(catalog[[1]])
> res = lapply(varNames, function(var){
+           xmlSApply(catalog,getvar,var)})
> plants = data.frame(res)
```

What is this command doing?

XPath

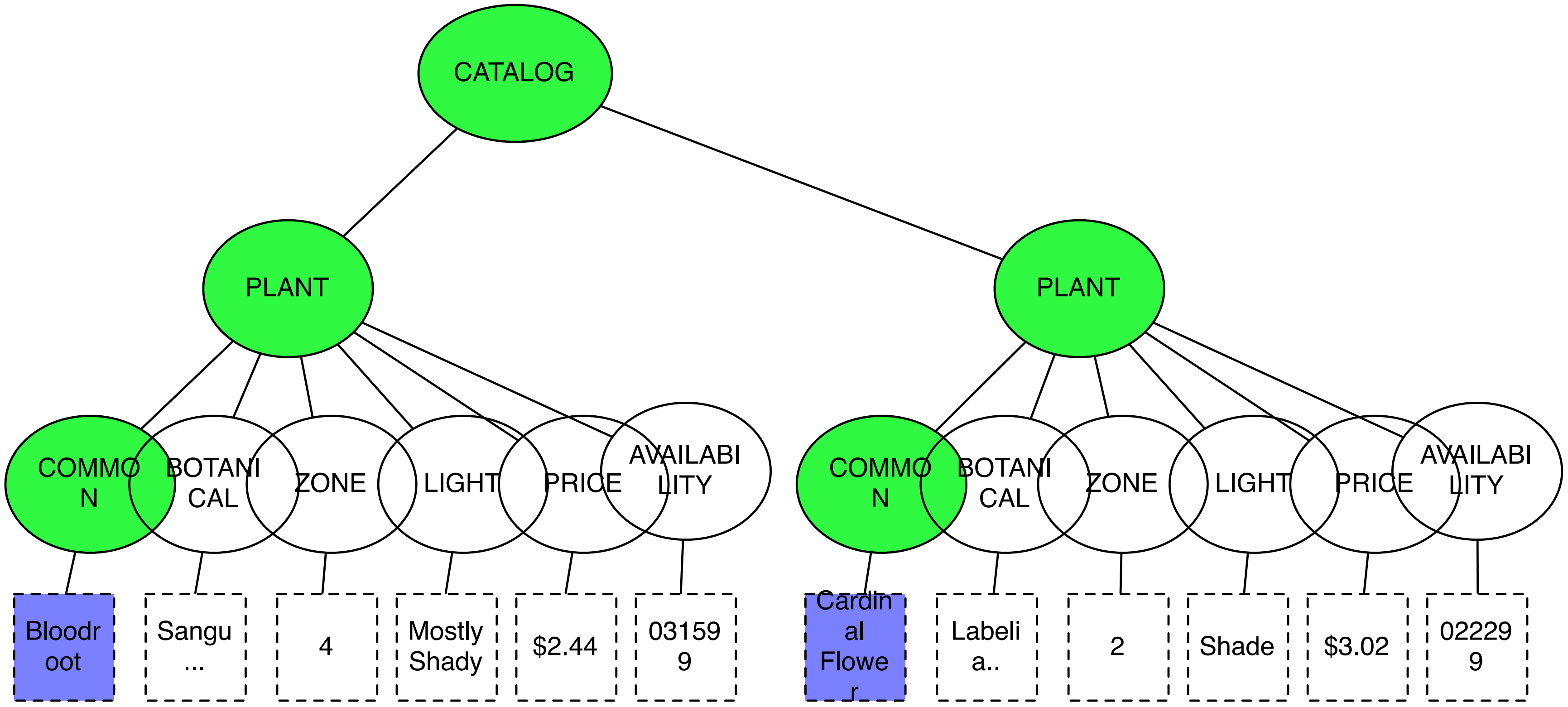
What is XPath

- Extraction tool designed for locating content in an XML file
- Uses the hierarchy of a well-formed XML document to specify the desired chunks to extract
- Syntax is similar to but more powerful than the way files are located in a hierarchy of directories in a computer file system



```
<PLANT>
 <COMMON>Bloodroot
 </COMMON>
 <BOTANICAL>Sanguinaria
canadensis
 </BOTANICAL>
 <ZONE>4</ZONE>
 <LIGHT>Mostly Shady
 </LIGHT>
 <PRICE>$2.44</PRICE>
 <AVAILABILITY>031599
 </AVAILABILITY>
</PLANT>
```

/CATALOG/PLANT/COMMON



Example

```
> common =
xpathSApply(catalog,
            "/CATALOG/PLANT/COMMON", xmlValue)

> head(common)
[1] "Bloodroot” "Columbine”              "Marsh Marigold"
[4] "Cowslip"   "Dutchman's-Breeches” "Ginger, Wild"
```

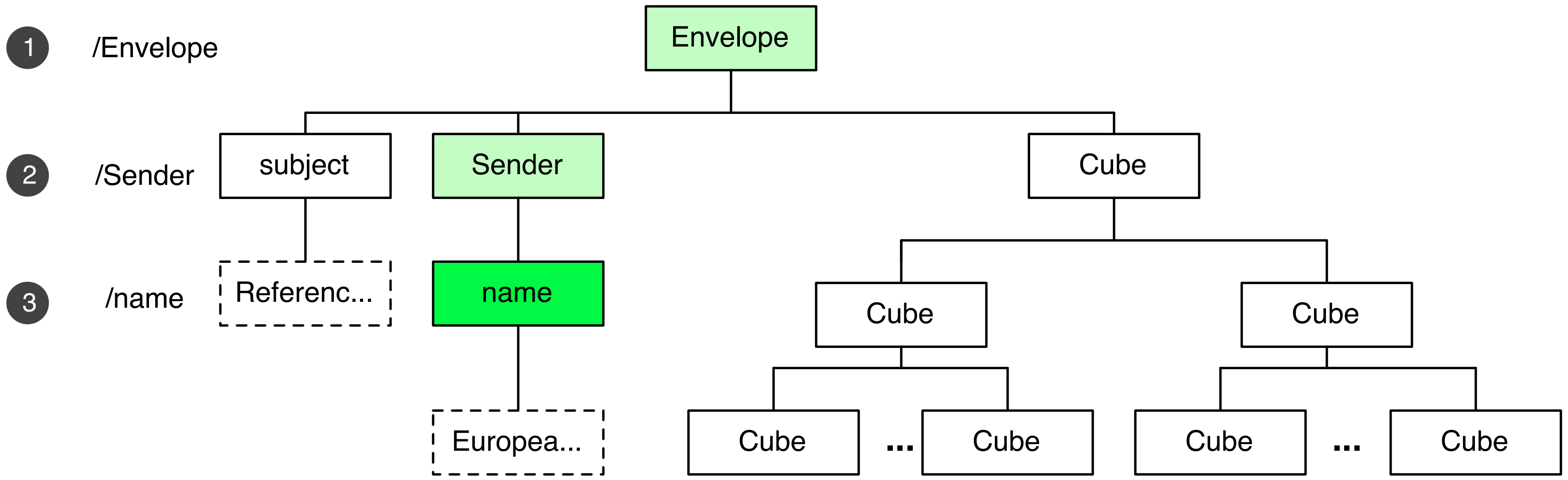
Example

```
> xpX =
   paste("/CATALOG/PLANT/", varNames, sep="")
> xpx[1]
[1] "/CATALOG/PLANT/COMMON"
> res =
  sapply(xpx,function(var)
         xpathSApply(catalog, var, xmlValue))
> res[1:2, 1:3]
[1,] "Bloodroot" "Sanguinaria canadensis" "4"
[2,] "Columbine" "Aquilegia canadensis"   "3"
```

```
<Envelope>
<subject>Reference rates</subject>
 <Sender>
  <name>European Central Bank</name>
 </Sender>
 <Cube>
  <Cube time="2008-04-21">
   <Cube currency="USD" rate="1.5898"/>
   <Cube currency="JPY" rate="164.43"/>
   <Cube currency="BGN" rate="1.9558"/>
   <Cube currency="CZK" rate="25.091"/>
  </Cube>
  <Cube time="2008-04-17">
   <Cube currency="USD" rate="1.5872"/>
   <Cube currency="JPY" rate="162.74"/>
   <Cube currency="BGN" rate="1.9558"/>
   <Cube currency="CZK" rate="24.975"/>
  </Cube>
 </Cube>
</Envelope>
```

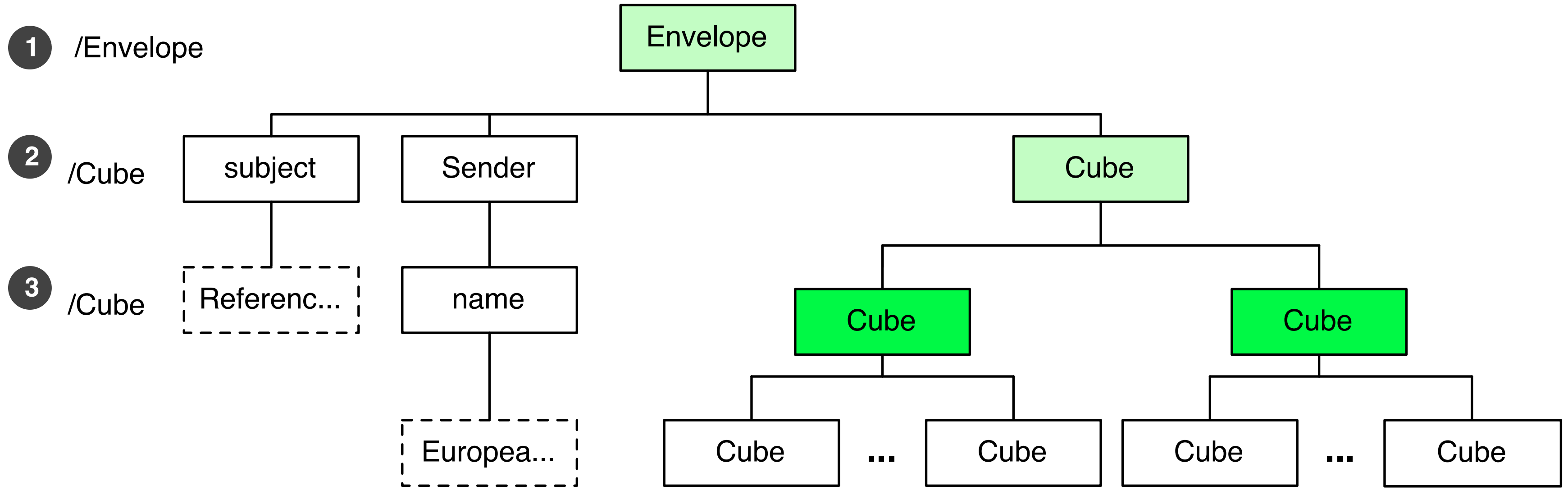
DRAW THE TREE FOR THIS DOCUMENT

/Envelope/Sender/name



1. The first location step identifies the root node, **<Envelope>**.
2. The next location step finds the **<Sender>** child of **<Envelope>**.
3. The third location step identifies **<Sender>**'s child called **<name>**.

/Envelope/Cube/Cube

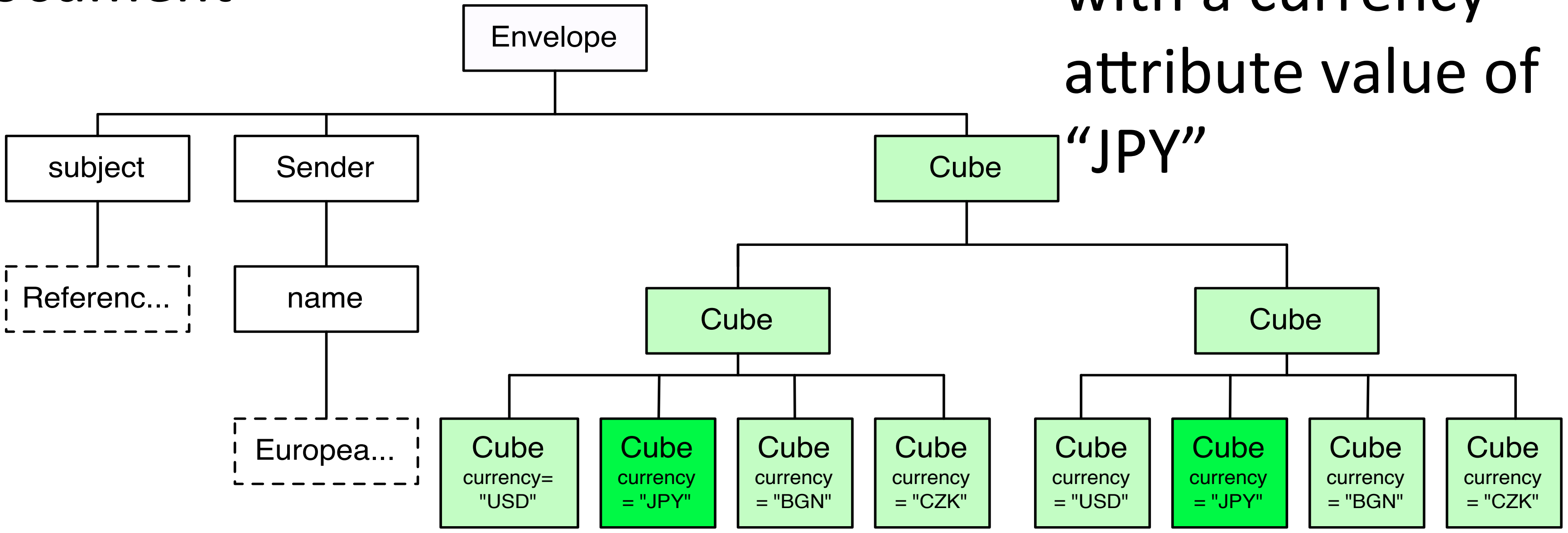


This XPath expression locates two sibling **<Cube>** nodes.

//Cube[@currency = "JPY"]

The expression: //Cube matches all Cube nodes found anywhere in the document

The **predicate** [@currency="JPY"] filters the matches to those elements with a currency attribute value of “JPY”



Your Turn

ISO Country Abbreviations

```
<ISO_3166-1_List_en  xml:lang="en">
  <ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>AFGHANISTAN
    </ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>AF
    </ISO_3166-1_Alpha-2_Code_element>
  </ISO_3166-1_Entry>
<ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>ALBANIA
    </ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>AL
    </ISO_3166-1_Alpha-2_Code_element>
  </ISO_3166-1_Entry>
```

DRAW the TREE

Write an Xpath expressions to locate name and abbreviations

World Health Factbook

```
<factbook>
....
<field dollars="false” unit="(deaths/1,000 live births)"    rankorder="1”
        name="Infant mortality rate" id="f2091">
<description>
This entry … number of deaths of infants under one year old in a given year
per 1,000 live births..
</description>
<rank number="121.63" dateEstimated="true"   dateLatest="2012-12-31”
        dateEarliest="2012-01-01" dateText="2012 est." country="af"/>
<rank number="109.98" dateEstimated="true" dateLatest="2012-12-31”
        dateEarliest="2012-01-01" dateText="2012 est." country="ng"/>
…
</field>
```

Write Xpath expression to locate country code and death rate

XPath syntax

- XPath is a language for locating content in XML documents.
- An XPath expression is a *location path* that is made up of *location steps* separated by /
- Each step has three parts: the axis, nodetest, and predicate
- Syntax:

```
axis::nodetest[predicate]
```

XPath syntax

```
axis::nodetest[predicate]
```

- The axis is the direction to look
- The nodetest is typically a node name that you wish to locate
- The optional predicate filters the qualifying nodes.

XPath syntax

- We will use only very simple Xpath expressions
- The axis will be either
  - “child”, which is the default and can be dropped, or
  - “descendant-or-self”, which says look anywhere down the tree and is abbreviated by “//”
  - “self” which is abbreviated with a .
- The nodetest will always be a node name
- The predicate is optional and will either be a number, [2] which asks for, e.g., the second node, or an attribute filter, e.g.,

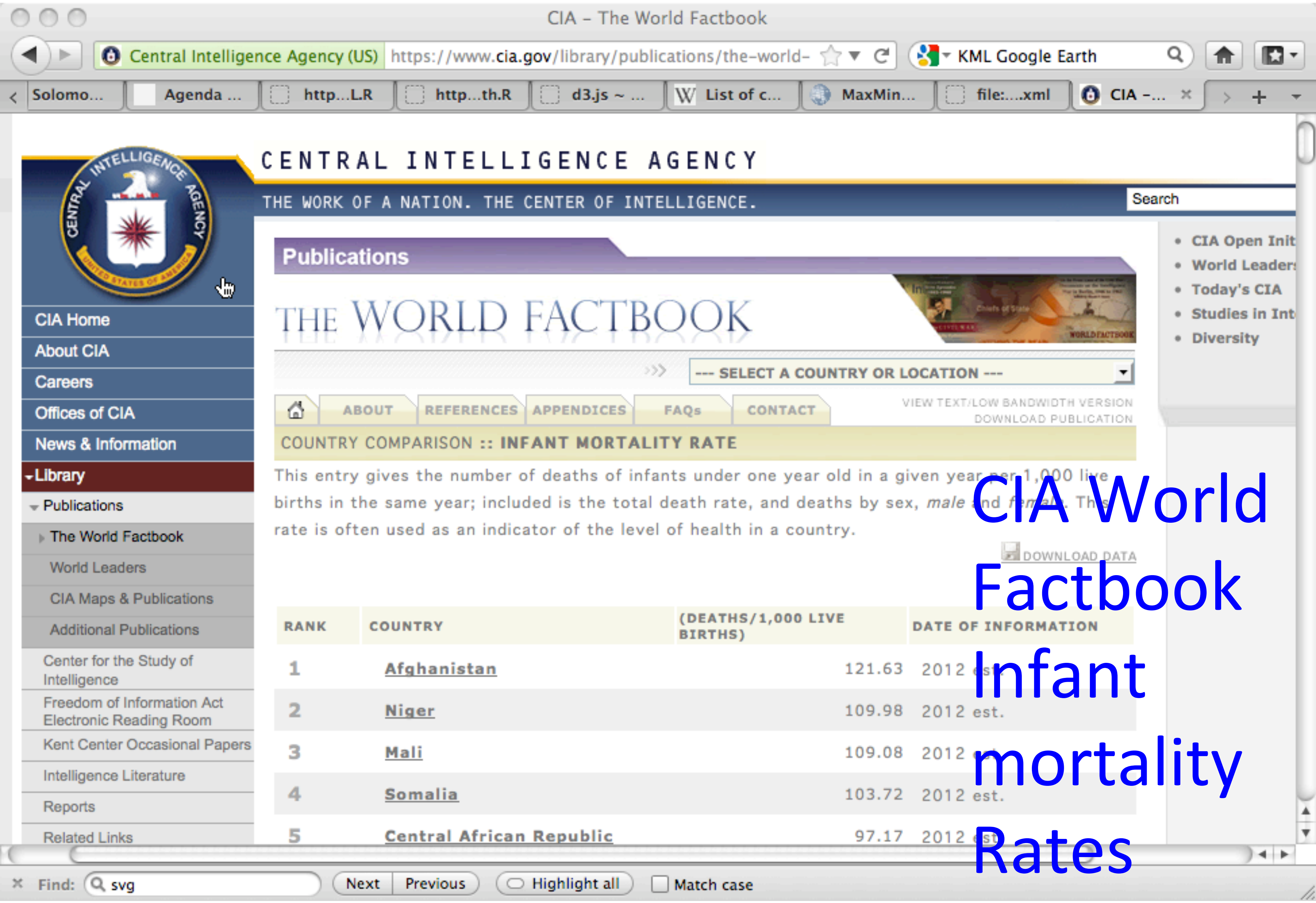
```
[@currency = “JPY” or @currency=“USD”]
```

Functions that take XPath expressions

- `getNodeSet(xmlTree, xpathExpression)` returns a list of XML nodes from `xmlTree` that satisfy the XPath expression.
- `xpathSApply(xmlTree, xpath, function)` the function is applied to those nodes in the XML tree that satisfy the Xpath expression. The return value is a vector when possible. `xpathApply` returns a list.

HW #6 MASH-UP

Create a mash-up on Google Earth



COUNTRY COMPARISON :: INFANT MORTALITY RATE

This entry gives the number of deaths of infants under one year old in a given year per 1,000 live births in the same year; included is the total death rate, and deaths by sex, *male* and *female*. This rate is often used as an indicator of the level of health in a country.

RANK	COUNTRY	(DEATHS/1,000 LIVE BIRTHS)	DATE OF INFORMATION
1	Afghanistan	121.63	2012 est.
2	Niger	109.98	2012 est.
3	Mali	109.08	2012 est.
4	Somalia	103.72	2012 est.
5	Central African Republic	97.17	2012 est.

CIA World Factbook Infant mortality Rates

Plain text file of country locations

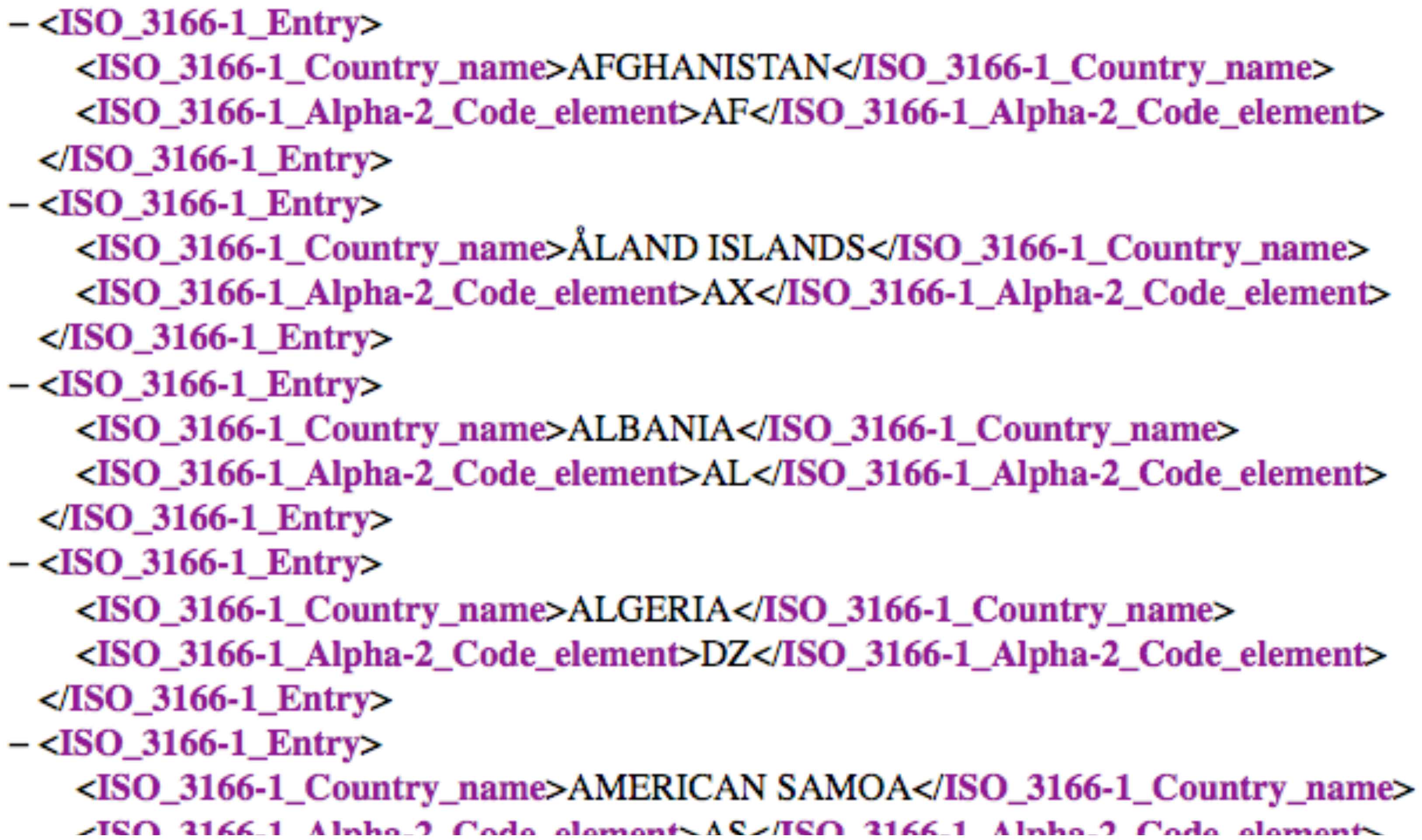
This page contains the average latitude and longitude for countries around the world.

Source: CIA World Factbook

```
"iso 3166 country","latitude","longitude"
AD,42.5000,1.5000
AE,24.0000,54.0000
AF,33.0000,65.0000
AG,17.0500,-61.8000
AI,18.2500,-63.1667
AL,41.0000,20.0000
AM,40.0000,45.0000
AN,12.2500,-68.7500
AO,-12.5000,18.5000
AP,35.0000,105.0000
AQ,-90.0000,0.0000
AR,-34.0000,-64.0000
AS,-14.3333,-170.0000
AT,47.3333,13.3333
AU,-27.0000,133.0000
```

Notice how the country ISO codes are used instead of country names

ISO country name – country code match



```
– <ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>AFGHANISTAN</ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>AF</ISO_3166-1_Alpha-2_Code_element>
  </ISO_3166-1_Entry>
– <ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>ÅLAND ISLANDS</ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>AX</ISO_3166-1_Alpha-2_Code_element>
  </ISO_3166-1_Entry>
– <ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>ALBANIA</ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>AL</ISO_3166-1_Alpha-2_Code_element>
  </ISO_3166-1_Entry>
– <ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>ALGERIA</ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>DZ</ISO_3166-1_Alpha-2_Code_element>
  </ISO_3166-1_Entry>
– <ISO_3166-1_Entry>
    <ISO_3166-1_Country_name>AMERICAN SAMOA</ISO_3166-1_Country_name>
    <ISO_3166-1_Alpha-2_Code_element>AS</ISO_3166-1_Alpha-2_Code_element>
```

- Although we can edit XML documents by hand, being able to do it programmatically is much more powerful.

- We will create a KML document that can be read in Google Earth. KML is a file format for geographic data, based on XML standards. See http://code.google.com/apis/kml/documentation/.

Sources

- Country population: CIA World Factbook
- Country infant mortality: CIA World Factbook
- Country location: plain text file
- Country name abbreviation map: ISO map
- Merge into one data frame
- Make a plot on Google Earth where circle size is proportional to country population, color is categorized by infant mortality, centers correspond to country latitude and longitude

Generating XML

XML package in R

Handy functions for creating XML

- `newXMLDoc`: create a new XML document
- `newXMLNode`: create a new XML Node
- `saveXML`: save the XML tree in a text file
- `parseXMLAndAdd`: parses character string of XML into an XML node and adds it to XMLInternalDocument (i.e. a tree)

Simple Example

```
doc = newXMLDoc()
root = newXMLNode("toplevel", doc = doc)
child1 = newXMLNode("level1", parent = root)
newXMLNode("level2", "This is the content",
           parent = child1)

> doc
<?xml version="1.0"?>
<toplevel>
  <level1>
    <level2>This is the content</level2>
  </level1>
</toplevel>
```

Simple Example

```
doc = newXMLDoc()
root = newXMLNode("toplevel", doc = doc)
child1 = newXMLNode("level1", parent =
root)
newXMLNode("level2", "This is the content",
              parent = child1)
saveXML(doc, file = "simple.xml")
```

We only need to assign to a variable nodes that we want to refer to as parents.

The names of the nodes in R (e.g. `root, child1`) are *not* part of the resulting XML document.

We can save the tree to a file

Exercise: First diagram the tree structure of this document, then create it from within R.

```
<?xml version="1.0"?>
<kml>
  <Document>
    <name>Earthquakes</name>
    <description>6+ Earthquakes, 1968-2008</description>
    <Folder>
     <name>Quakes</name>
     <Placemark>
      <Point>
        <coordinates>-124.95,41.04,0  </coordinates>
      </Point>
    </Placemark>
    <Placemark>
      <Point>
        <coordinates>-118.83,37.59,0  </coordinates>
      </Point>
    </Placemark>

... more Placemark nodes
    </Folder>
  </Document>
</kml>
```

Programming Languages we have seen so far

- R – uses control flow to describe a computation
- shell commands – command line interface to the operating system
- regular expressions – describes a pattern but not how to find it
- HTML – describes what should appear on a Web page but not how to render it
- Xpath – describes the location of content

Two types of programming:

- **Declarative programming**: describes a result; expresses what the program should do without specifying how to do it
- **Imperative programming:** describe the algorithm/steps of a computation; uses control flow, e.g., looping, assignment, conditional statements.

Which are imperative/declarative?

Imperative

- R
- shell

Declarative

- Regular expressions
- HTML
- XPath
- SQL

Relational Databases and SQL

A *database* is a collection of data with information about how the data are organized (meta-data). A *database server* is like a web server, but responds to requests for data rather than web pages.

We'll talk about *relational database management systems* (RDBMS) and how to communicate with them using the *structured query language* (SQL).

Why use a database?

- Coordinate synchronized access to data
- Change continually; give immediate access to live data
- Centralize data for backups
- Control access to the data

A RDBMS had three main parts

- Data definition
- Data access
- Privilege management

We'll concentrate on data access, assuming the database is already available and we have the needed privileges.

Topics:

- using SQL to extract info from RDBMSs
- relating these back to similar tasks in R
- using SQL from within R

There are tradeoffs in terms of what we choose to do using SQL and what we do in R.

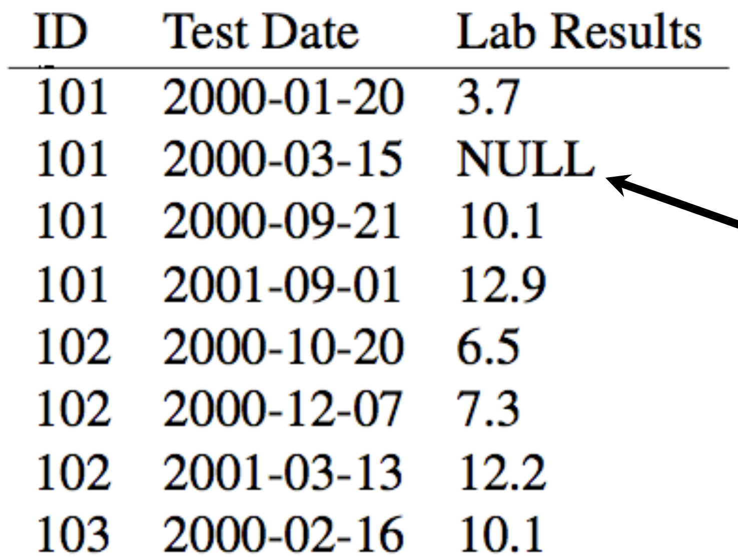
- A database is made up of one or more two dimensional *tables*, usually stored as files on the server.

- An important concept in the design of a database is *normalization*. The idea is to remove as much redundancy as possible when creating the tables. This is done by breaking the full dataset into separate tables.

- The "relational" in RDBMS comes from the fact that we then need to link the tables together.

- For now let's talk about a single table....

- A *table* is a rectangular arrangement of values, where a row represents a case, and a column represents a variable (just like a data frame in R).
- Another term for a table is a *relation*. The rows are referred to as *tuples* and the columns as *attributes*.



ID	Test Date	Lab Results
101	2000-01-20	3.7
101	2000-03-15	NULL
101	2000-09-21	10.1
101	2001-09-01	12.9
102	2000-10-20	6.5
102	2000-12-07	7.3
102	2001-03-13	12.2
103	2000-02-16	10.1

Missing value

- An *entity* is the general object of interest. For example, a lab test. Each row (tuple) is a particular occurrence of the entity. This means that rows in the table are unique.

- To identify each row, we use a *key*. A key is just an attribute or a combination of attributes that uniquely identifies the cases.

- In the lab test example, we need a *composite key* of both patient ID and date, since neither is necessarily unique.

- In R, the row names of a data frame play a similar role.

- SQL allows us to interactively *query* the database to reduce the data by subsetting, grouping, or aggregation.

- Each database program tends to have its own version of SQL, but they all support the same basic SQL statements. (We say statements rather than commands because SQL is referred to as a declarative rather than an imperative language.)

- The SQL statement for retrieving data is the SELECT statement. This operates on one or more tables. The result will always be another table.

We have a table called chips, with data about the CPU development of PCs over time

The simplest possible query gives back everything:

```
SELECT * FROM chips;
```

```
 processor   | date | transistors | microns | clockspeed | width | mips
-------------+------+-------------+---------+------------+-------+------
 8080        | 1974 |        6000 |       6 |          2 |     8 | 0.64
 8088        | 1979 |       29000 |       3 |          5 |    16 | 0.33
 80286       | 1982 |      134000 |     1.5 |          6 |    16 |    1
 80386       | 1985 |      275000 |     1.5 |         16 |    32 |    5
 80486       | 1989 |     1200000 |       1 |         25 |    32 |   20
 Pentium     | 1993 |     3100000 |     0.8 |         60 |    32 |  100
 PentiumII   | 1997 |     7500000 |    0.35 |        233 |    32 |  300
 PentiumIII  | 1999 |     9500000 |    0.25 |        450 |    32 |  510
 Pentium4    | 2000 |    42000000 |    0.18 |       1500 |    32 | 1700
```

- By convention, we display SQL statements in upper case. Statements are ended by a semicolon.

Attributes / Variables

- Recall that in R, we can select particular variables (columns) by name.

```
chips[ , c('mips', 'microns')]
```

- The order of the variable names determines the order in which they'll be returned in the resulting data frame.
- The corresponding SQL query is

```
SELECT mips, microns FROM chips;
```

SQL Syntax

- Similar to a sentence in English, except that there’s less flexibility in the order of the words.
- Sentence ends with a ;
- Use blanks and “,” and “=“ and () as delimiters
- We will only look at SELECT statements, which begin with the term SELECT

Examples of SELECT statements

```
SELECT * FROM chips;
SELECT mips, microns FROM chips;

SELECT * FROM chips
 WHERE processor = ‘Pentium’ OR
     processor = ‘PentiumII’;
```

Selecting Tuple/Row

In R we can select rows that match a condition:

```
chips[chips$processor == 'Pentium' |
      chips$processor == 'PentiumII',]
```

The corresponding SQL statement is

```
SELECT * FROM chips
 WHERE processor = 'Pentium' OR
   processor = 'PentiumII';
```

Note: Whitespace can be used freely in SQL statements. We often separate lines for clarity. The statement isn't evaluated until the semicolon is entered.

- Aside: In R, it usually doesn't matter whether you use single or double quotes to surround character strings. In SQL, the standard is to use single quotes, so we will do this throughout the notes for both R and SQL.

- The `WHERE` clause can also be used with other Boolean operators. The keyword NOT negates a condition, and parentheses can be used to clarify order of evaluation.

```
SELECT * FROM chips WHERE date > 1990;

SELECT * FROM chips WHERE NOT width = 8;

SELECT * FROM chips WHERE NOT (width = 8 OR
width = 16);
```

- In both R and SQL, we can do both types of subsetting at once.

- R:

```
chips[chips$processor == ‘Pentium’ |
          chips$processor == ‘PentiumII’,
        c(‘mips’, ‘microns’)]
```

- SQL:

```
SELECT mips, microns FROM chips
    WHERE processor = ‘Pentium’ OR
          processor = ‘PentiumII’;
```

General Syntax

SELECT attribute(s) FROM table(s)
 [WHERE constraints]; ← [optional]

How would we pull the years of all 32-bit processors that execute fewer than 250 million instructions per second (mips),

1) in R,

```
chips[ chips$mips < 250 & chips$width == 32,
       "date"]
```

2) in SQL?

```
SELECT date FROM chips
  WHERE mips < 250 AND width = 32;
```

SQL offers limited features for summarizing data -- some aggregate functions that operate over the rows of a table, and some mathematical functions that operate on individual values in a row.

The aggregate functions are

- COUNT - number of rows
- SUM - total of all values for an attribute
- AVG - average value for an attribute
- MIN - minimum value for an attribute
- MAX - maximum value for an attribute

SELECT attribute(s) FROM table(s) [WHERE constraints];

can also be *functions* of attributes

Exercise:

1) How many rows are in the chips table?

```
SELECT COUNT(date) FROM chips;
```

2) How many chips have attribute width equal to 32?

```
SELECT COUNT(date) FROM chips WHERE width = 32;
```

3) What is the average clock speed for the chips in question 2?

```
SELECT AVG(clockspeed) FROM chips WHERE width = 32;
```

Now answer the same questions assuming chips is a data frame in R.

```
nrow(chips);
sum(chips$width == 32);
mean(chips$clockspeed[chips$width == 32)
```

Additional clauses: GROUP BY

- The GROUP BY clause makes the aggregate functions in SQL more useful. It enables the aggregates to be applied to *subsets* of the rows in a table.

```
SELECT width, MAX(mips) FROM chips
    GROUP BY width;
```

```
 width | max
-------+------
     8 | 0.64
    16 |    1
    32 | 1700
(3 rows)
```

- More than one attribute can be included in the GROUP BY clause.

Additional clauses: `HAVING`

- The `WHERE` clause can't contain an aggregate function, but the `HAVING` clause can be used to refer to the groups to be selected.

```
SELECT width, MAX(mips) FROM chips
    GROUP BY width HAVING MAX(mips) >= 1;
```

```
 width | max
-------+------
    16 |    1
    32 | 1700
(2 rows)
```

- First the chips table is separated into sets of rows by width. For each set, `MAX(mips)` is calculated, and the set is discarded if `MAX(mips) < 1`. Finally, width and `MAX(mips)` are returned for each set.

A few other predicates and clauses

- DISTINCT - forces values of an attribute in the results table to have unique values
- NOT - negates conditions in WHERE or HAVING clause
- LIMIT - limits the number of rows returned

```
SELECT * FROM chips LIMIT 3;

SELECT DISTINCT width FROM chips;
```

Order of Execution

The order of execution of the clauses in a `SELECT` statement is as follows:

1. `FROM:` The working table is constructed.

2. `WHERE:` The `WHERE` clause is applied to each row of the table, and only the rows that test TRUE are retained.

3. `GROUP BY:` The results are broken into groups of rows all with the same value of the `GROUP BY` clause.

4. `HAVING`: The `HAVING` clause is applied to each group and only those that test `TRUE` are retained.

5. `SELECT:` The attributes not in the list are dropped, aggregates are calculated, and options `DISTINCT`, `ORDER BY` and `LIMIT` are applied.

Using SQL with R

The DBI and RSQLite libraries allow you to connect to an SQL database, submit a query, and receive the results as a data frame.

```
library(RSQLite)

# set up an interface to SQLite
M = dbDriver("SQLite")

# connect to particular DBMS (no security SQLite)
con = dbConnect(m, dbname="tfile")

# submit an SQL statement; returns a data frame
dbGetQuery(con, "SELECT * FROM chipsSQLite LIMIT
5;")
```

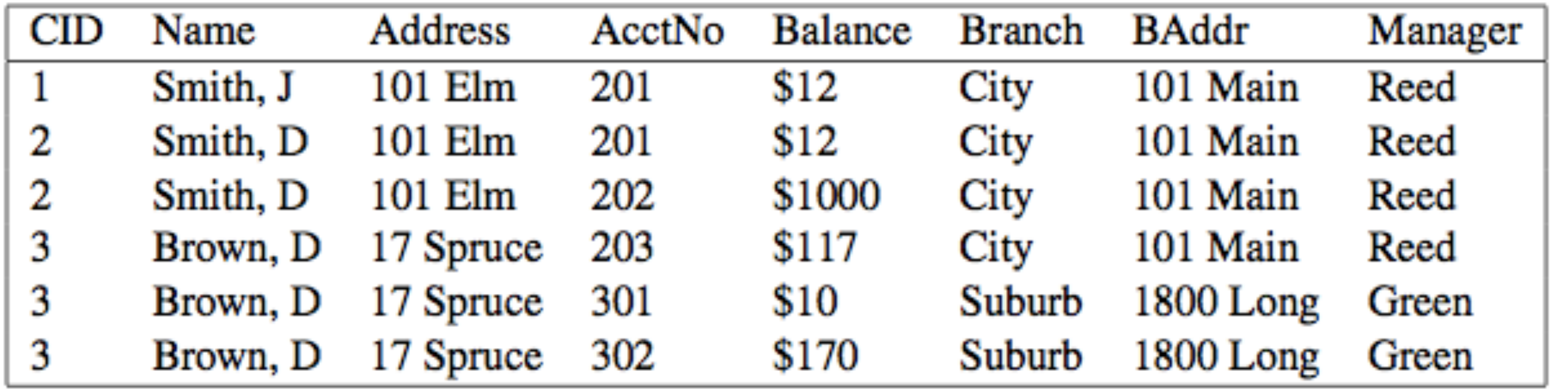
If the results are large, we don’t have to pull them all over into R at once.

```
# submit an SQL statement; but keep results in SQL
rs = dbSendQuery(con, "SELECT * FROM chipsSQLite;")

# Retrieve the first 5 rows in the results
fetch(rs, n = 5)

# Retrieve the next 3 rows
fetch(rs, n = 3)

# Retrieve the remaining rows
fetch(rs, n = -1)

# Close the query
dbClearResult(rs)
```

We can of course import the whole table, then extracting what we want using R commands.

But, it may be more efficient to use SELECT to extract and import only the results with which we want to work

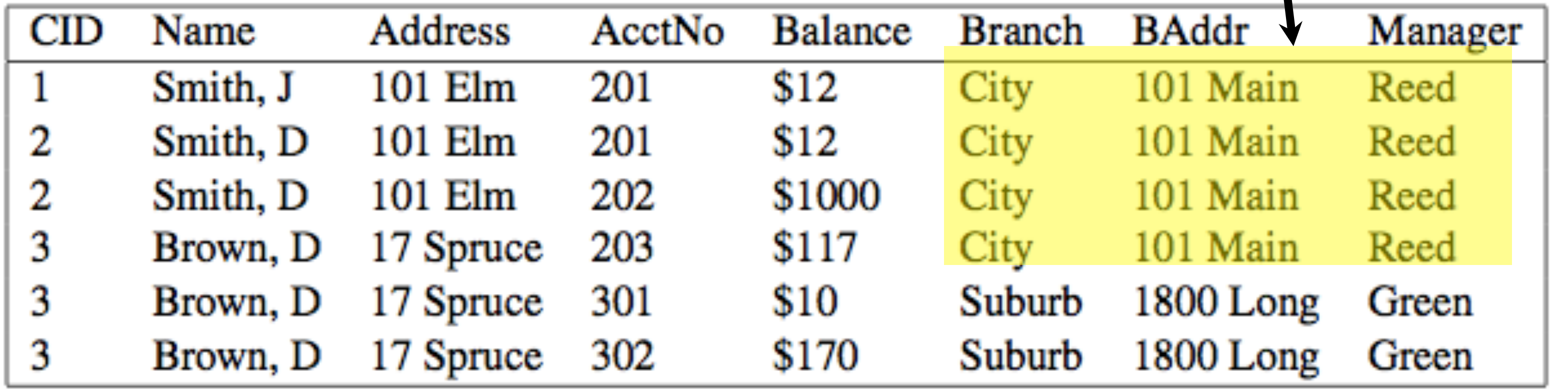
Multiple Tables



CID	Name	Address	AcctNo	Balance	Branch	BAddr	Manager
1	Smith, J	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	202	$1000	City	101 Main	Reed
3	Brown, D	17 Spruce	203	$117	City	101 Main	Reed
3	Brown, D	17 Spruce	301	$10	Suburb	1800 Long	Green
3	Brown, D	17 Spruce	302	$170	Suburb	1800 Long	Green

- Where do you see redundancy?
- What might an entity be? (e.g. lab test)

BANK BRANCH

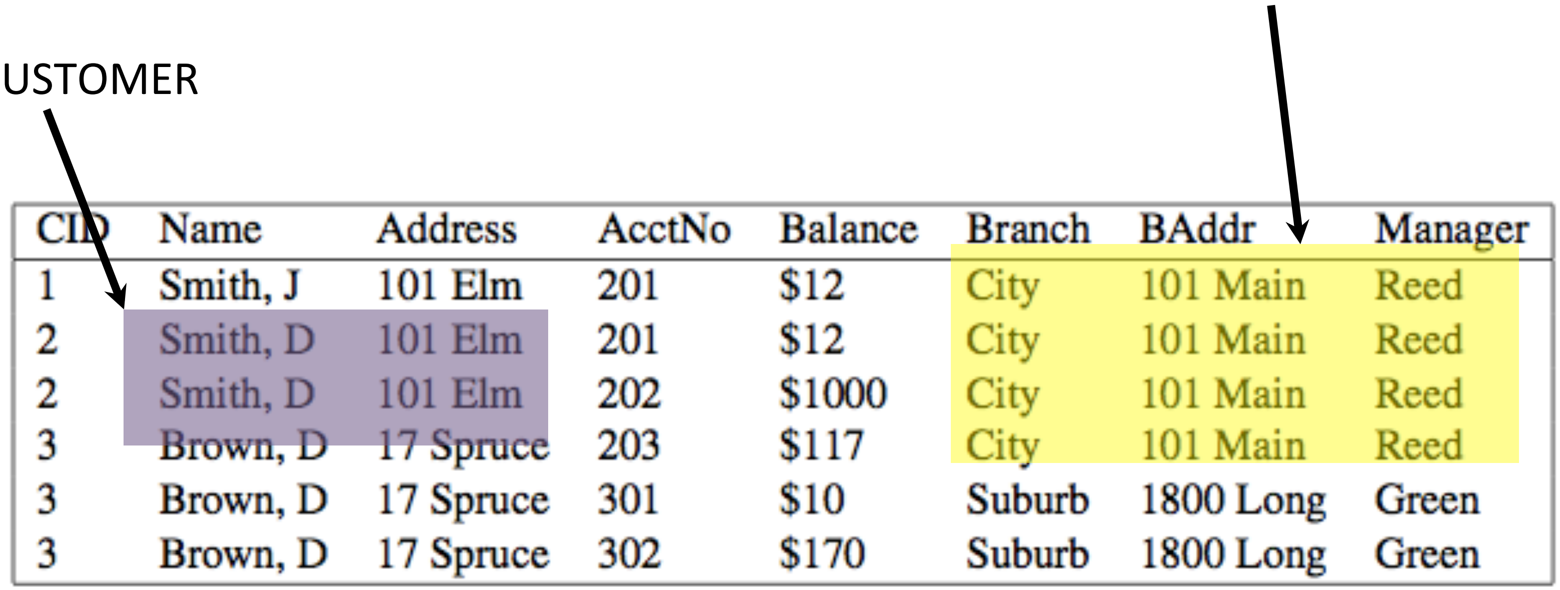


CID	Name	Address	AcctNo	Balance	Branch	BAddr	Manager
1	Smith, J	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	202	$1000	City	101 Main	Reed
3	Brown, D	17 Spruce	203	$117	City	101 Main	Reed
3	Brown, D	17 Spruce	301	$10	Suburb	1800 Long	Green
3	Brown, D	17 Spruce	302	$170	Suburb	1800 Long	Green

- Where do you see redundancy?
- What might an entity be? (e.g. lab test)

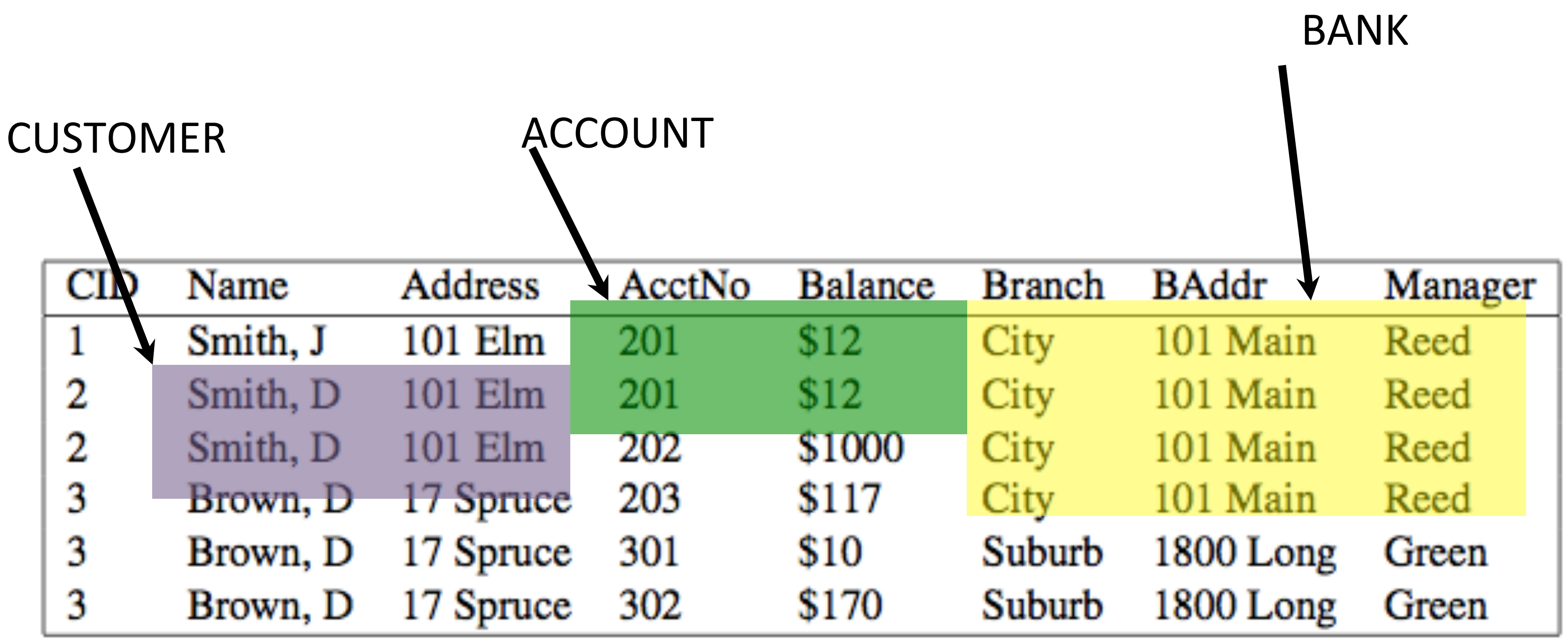
BANK BRANCH

CUSTOMER



CID	Name	Address	AcctNo	Balance	Branch	BAddr	Manager
1	Smith, J	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	202	$1000	City	101 Main	Reed
3	Brown, D	17 Spruce	203	$117	City	101 Main	Reed
3	Brown, D	17 Spruce	301	$10	Suburb	1800 Long	Green
3	Brown, D	17 Spruce	302	$170	Suburb	1800 Long	Green

- Where do you see redundancy?
- What might an entity be? (e.g. lab test)


BANK
CUSTOMER
ACCOUNT

CID	Name	Address	AcctNo	Balance	Branch	BAddr	Manager
1	Smith, J	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	201	$12	City	101 Main	Reed
2	Smith, D	101 Elm	202	$1000	City	101 Main	Reed
3	Brown, D	17 Spruce	203	$117	City	101 Main	Reed
3	Brown, D	17 Spruce	301	$10	Suburb	1800 Long	Green
3	Brown, D	17 Spruce	302	$170	Suburb	1800 Long	Green

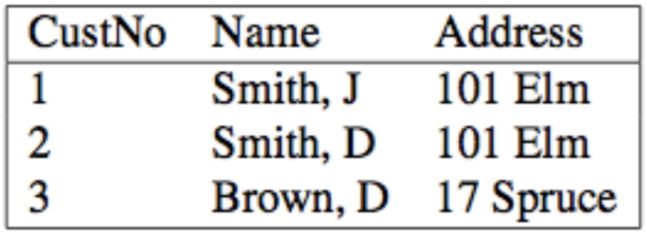
- Where do you see redundancy?
- What might an entity be? (e.g. lab test)

Entities

- Customer:
  - Customers can have more than one account
- Account:
  - Accounts can be owned by more than one customer;
  - An account is in one branch
- Branch:
  - Branches can have many accounts

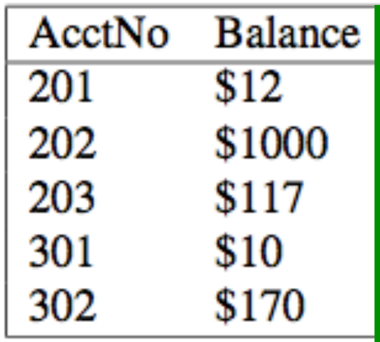
- Here are tables for each of the three entities we identified.

Customers Table



CustNo	Name	Address
1	Smith, J	101 Elm
2	Smith, D	101 Elm
3	Brown, D	17 Spruce

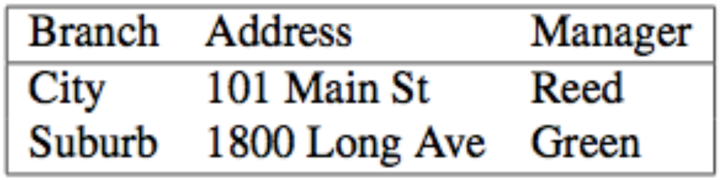
Accounts Table



AcctNo	Balance
201	$12
202	$1000
203	$117
301	$10
302	$170

Is anything missing?

Branches Tal



Branch	Address	Manager
City	101 Main St	Reed
Suburb	1800 Long Ave	Green

We have information about:
Customers
Accounts
Branch

Missing Information

- Relations:
  - Customers have accounts
  - Accounts are located in Branches

- How do we bring this information into the database?
  - Add branch information to the account table
  - Create a new table that holds account, customer pairs

- Customers and their bank accounts are linked via the registration table – this is a relation:

Customers Table

CustNo	Name	Address
1	Smith, J	101 Elm
2	Smith, D	101 Elm
3	Brown, D	17 Spruce

Accounts Table

AcctNo	Balance	Branch
201	$12	City
202	$1000	City
203	$117	City
301	$10	Suburb
302	$170	Suburb

Notice:
Account 201 appears twice
Customer 3 appears three times

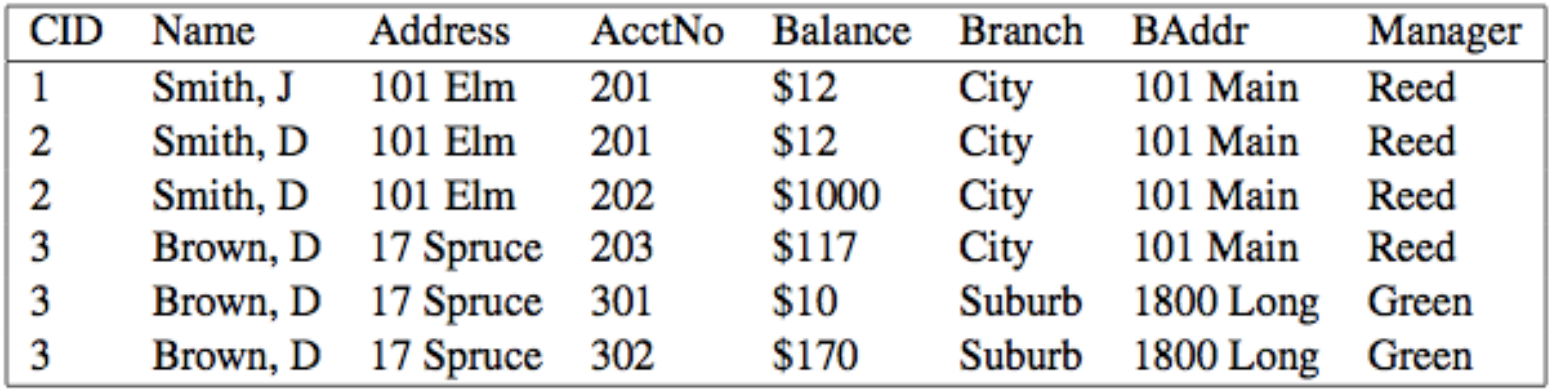
Registration Table

CID	AcctNo
1	201
2	201
2	202
3	203
3	301
3	302

Branches Table

Branch	Address	Manager
City	101 Main St	Reed
Suburb	1800 Long Ave	Green

- We could put this information in one big table, but this would introduce redundancy.



CID	Name	Address	AcctNo	Balance	Branch	BAddr	Manager
1	Smith, J	101 Elm	201	\$12	City	101 Main	Reed
2	Smith, D	101 Elm	201	\$12	City	101 Main	Reed
2	Smith, D	101 Elm	202	\$1000	City	101 Main	Reed
3	Brown, D	17 Spruce	203	\$117	City	101 Main	Reed
3	Brown, D	17 Spruce	301	\$10	Suburb	1800 Long	Green
3	Brown, D	17 Spruce	302	\$170	Suburb	1800 Long	Green

- There can be speed and memory advantages to working with the smaller tables, extracting just the information we need.

Examples of extractions

Warmup

- The names and addresses of all customers.

```
SELECT Name,Addr FROM Customers;
      Name    Address
1 Smith, J    101 Elm
2 Smith, D    101 Elm
3 Brown, D 17 Spruce
```

Warmup

- All of the attributes in the Registration table for accounts belonging to Customer 3.

```
SELECT * FROM Registration WHERE CID = 3;
```

```
 row_names CID AcctNo
1  4          3    203
2  5          3    301
3   6         3    302
```

Warmup

- Account numbers for accounts that don’t meet minimum $100 balance in City branch.

```
SELECT AcctNo FROM Accounts
   WHERE Branch = 'City'
            AND Balance < 100;
```

```
    AcctNo
1      201
```

- The total balance for all accounts belonging to a branch.

```
SELECT Branch, SUM(Balance)
       FROM Accounts
       GROUP BY Branch;
  Branch SUM(Balance)
1   City         1129
2 Suburb          180
```

- The total overdrawn amount at a branch, for those branches which have an account with at least $100 overdrawn.

```
SELECT Branch, SUM(Balance) FROM Accounts
  WHERE Balance < 0
  GROUP BY Branch
  HAVING Min(Balance) < -100;
[1] Branch       SUM(Balance)
 <0 rows> (or 0-length row.names)
```

- Suppose we want the name(s) of the owner(s) for every account
- The customer name is in Customers
- The account number is in Registration
- For each account in Registration, we want the associated customer names.
- We need to joint two tables to get this information

Joining tables

```
SELECT Name, AcctNo
  FROM Customers,Registration
  WHERE Customers.CustNo = Registration.CID;
```

```
      Name   AcctNo
1 Smith, J     201
2 Smith, D     201
3 Smith, D     202
4 Brown, D     203
5 Brown, D     301
6 Brown, D     302
```

- Combining two tables in this way is called an *inner join.*
- Note how we refer to columns within a given table using the form tableName.column ("dot notation"). This isn't strictly necessary when there's only one possible match, but it improves clarity.

- We can use AS to rename tables or columns. This can save a lot of typing.

```
SELECT Name, AcctNo AS Account
  FROM Customers AS C, Registration AS R
  WHERE C.CustNo = R.CID;
```

```
      Name Account
1 Smith, J     201
2 Smith, D     201
3 Smith, D     202
4 Brown, D     203
5 Brown, D     301
6 Brown, D     302
```

- The customer name, account number, and balance of the account for all accounts.

```
SELECT C.Name, A.AcctNo, A.Balance
  FROM Customers C, Registration R,
       Accounts A
  WHERE C.CustNo = R.CID AND
        A.AcctNo = R.AcctNo;
```

```
  Name      AcctNo Balance
1 Smith, J     201      12
2 Smith, D     201      12
3 Smith, D     202    1000
4 Brown, D     203     117
5 Brown, D     301      10
6 Brown, D     302     170
```

- Give the total balance for each customer with multiple accounts.

```
SELECT CID, SUM(Balance)
   FROM Registration R, Accounts A
   WHERE A.AcctNo = R.AcctNo
   GROUP BY R.CID HAVING COUNT(*) > 1;
```

```
   CID  SUM(Balance)
1    2          1012
2    3           297
```

Other types of Databases

- NoSQL
  - data bases that don’t use SQL
  - Not relational
  - JSON – style data structures
- NoSQL RDBMS
  - Data base that doesn’t use SQL, but is relational
  - Shell based
- Example: MongoDB (from "hu**mongo**us") is a scalable, high-performance