Lecture 14: Oh mighty Isis!
Last time

We started with a review of some of the mechanics behind SSH, specifically public-key authentication -- It was something we should have mentioned a loooong time ago

We were forced to bring it up last time because we also talked about code sharing via GitHub, another dangling thread

We then finished looking at the last few introductory data types in R, namely lists and dataframes and factors...
Today

The goal is to provide you with at least a basic introduction to some of the programming constructions you’ll be using in R should you decide to implement Shazam in R

We’ll also talk about databases and in particular access through R to SQL databases -- Our main example will be, for its simplicity, SQLite
Just Add Urbanism
cityLAB Symposium

Monday
November 15 | 6:30 pm

The radical scalar shifts from Google Earth to Street View, animation’s wild temporal formats, the aggregate perception of online, tiled photographs, and the ability to explore space in gaming—these means to represent urbanism as not just a context but a pretext for architecture were formerly, and literally, unimaginable. Amidst an ever-proliferating number of techniques to describe spatial experience, this symposium looks to design disciplines outside architecture and planning that have demonstrated the ability to suggest projective urban thinking as well as to engage new audiences.

Presentations:
Rodolphe el-Khoury
Penelope Dean
Mark Linder

Participants:
Benjamin Bratton
Diane Favro
Ron Frankel
Mario Gandelsonas
Mark Hansen
John May
Michael Osman
Jason Payne
Peter Zellner

Moderators:
Dana Cuff
Roger Sherman

Lectures and exhibitions are open to the public.

Perloff Hall is located on the UCLA Campus.
Perloff Hall, M-F, 9am-5pm
Lectures take place at 6:30pm in Perloff Hall, Decafé
Parking is available in Lot 3 for $10
Check the website for confirmation of all programs at www.aud.ucla.edu
Info: 310.267.4704
The campus map is available at http://www.aud.ucla.edu/pdfs/ucla-map.pdf
Let’s step back for a moment...

Now that we’ve had a look at some basic data types (vectors, matrices, lists, data frames and factors), let’s talk a bit more about the structure of the language

So far we have hints about how to glue things together, whether it be computations (defining functions) or more elaborate objects
As we have seen, the **interactive interpreter** in R is much like that for the Unix shell or Python -- **You type in commands and something happens**

Elementary commands consist of **expressions or assignments** (in the former case a computation is performed and the result prints to the screen; in the latter case, the value is caught and assigned to a variable)

Typically, we have one line per command, although you can string multiple lines together separated by a `;`
Commands

As you work, you create objects that get stored in your workspace; recall that this is just the entry `.GlobalEnv` in your search path.

When evaluating expressions, R might have to search for an object and it looks along your search path; it starts with your workspace and moves out.
Commands

As with Unix (although we haven’t really tried shell scripting) and Python, we can either type commands into the R interpreter, or we can store them in a file and read them in.

Whether you type them in or you source time in, the expressions are executed in the “context” of your main workspace -- Any objects you create are stored there.
% cat cmd.R

ihash <- function(r1,r2,dtr) dtr*2^16 + (r2-1)*2^8 + (r1-1)

print(ihash(200,135,201))

% R

R version 2.10.1 (2009-12-14)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> source("cmd.R")
[1] 13207239
Aside: Interacting with the filesystem

R has a series of convenience functions for interacting with the operating system.

You can check on or change your current working directory; list the files there; copy, rename and delete files.

While there are plenty of such functions, you might need to “call out” to the operating system; this is done with the system() function.

```r
> getwd()
[1] "/Users/cocteau/stat202/lecture7"

> list.files()
[1] "la.csv" "lecture7.key"

> file.exists("la.csv")
[1] TRUE

> file.exists("eq.csv")
[1] FALSE

> file.copy("la.csv","eq.csv")
[1] TRUE

> list.files()
[1] "cmd.R" "remove_me.R"

> file.remove("remove_me.R")
[1] TRUE

> list.files()
[1] "cmd.R"

# the following directs the output into an r character vector

> system("ls",intern=T)
[1] "cmd.R"
```
So far...

So far, then, despite this (largely covered “for the record”) detour into saving commands in a file, we have mainly interacted with R by **typing expressions at the prompt**; they are then parsed and evaluated

- We have printed or exhibited objects,
- Identified subsets of objects like vectors, matrices, arrays, data frames and lists,
- Performed arithmetic and evaluated special operators,
- Created new objects by assignment, and
- Invoked functions

As we move from tool-users to tool-builders, we are going to start to add to the language, **creating our own objects and implementing our own procedures**
A function

When we started thinking about hash functions, we decided that we’d have to perform this computation many times and that it might make sense to create a function to perform this operation.

\[ \text{ihash} \leftarrow \text{function}(r1, r2, dtr) \ dtr \times 2^{16} + (r2 - 1) \times 2^8 + (r1 - 1) \]

\[ \text{ihash}(200, 135, 201) \]

[1] 13207239
Functions in R

The definition of a function will typically have the following structure (or syntax):

1. The "reserved" word function

2. A comma-separated list of arguments enclosed in parentheses

3. A single expression or a sequence of expressions contained in curly braces

When R executes a function definition, it creates an object with three parts: A header describing the function’s formal arguments; a single expression or a sequence of expressions comprising the function’s body; and the so-called environment in which the function was defined (more on that later)
Functions in R

In the case of our hash function, we have chosen to give it a name -- As any other named object in R, we can then ask it some basic questions, or see its contents by typing its name

```r
> ihash <- function(r1,r2,dtr) dtr*2^16 + (r2-1)*2^8 + (r1-1)
> class(ihash)
[1] "function"

> ihash
function(r1,r2,dtr) dtr*2^16 + (r2-1)*2^8 + (r1-1)
```
Calling a function: (i) Arguments

When you call our function distance, R finds the definition of the function and then assigns the actual arguments you provided to the formal arguments listed in the function’s definition.

This is done first by name (if the actual arguments were given names) and then by position -- the following calls all produce the same results:

> ihash(201,135,216)

> ihash(r1=201,r2=135,dtr=216)

> ihash(135,216,r1=201)
Calling a function: (i) Arguments

When we call a function, we provide it with data (values like 23 or “a cat”; the names of variables in our workspace or a package or some other location; and even expressions) -- R then matches the data we provide with the formal argument list using three rules

**Exact matching**: For each named argument we supply in the function call, R will search the list of formal arguments for one with exactly the same name

**Partial matching**: Next, matches are made if the name of a supplied argument agrees with the beginning of one of the (previously unmatched) formal arguments

**Positional matching**: Finally, formal arguments are matched in order (left to right) with the unnamed arguments we supplied in the function call
Calling functions: (i) Arguments

The **formal arguments** of a function represent the data we would like to operate on -- They are combined into a **comma-separated** list that can consists of

1. A symbol or name;

2. A statement of the form ‘name=expression’ to specify **default values** for the argument; or

3. A special formal catch-all argument ‘...’ that is used to represent **any number of supplied arguments not named in the formal argument list**; it can be used when a function acts on any number of objects or when you want to pass variables to functions called later
Calling a function: (i) Arguments

Notice that you can pass functions as arguments, just as you would any other object -- It is this facility with functions that makes R somewhat unique, and provides it with a great deal of expressiveness when it comes to statistical applications.

For example, apply is often called with unnamed or orphan functions that can be defined in the call to apply -- here we consider computing the minimum of the columns of some data matrix.

```r
> data <- matrix(runif(1000),ncol=10)
> mins <- apply(data,1,function(x) min(x))
> mins[1:4]
[1] 0.06679730 0.06145933 0.06611873 0.03366189
```

In this case, we apply the unnamed function to the rows (remember 1 means rows) of the matrix data; each row is passed as the argument x to the unnamed function.
Calling a function: (i) Arguments

Consider, for example, the function we've been calling to construct new matrices -- Every argument is given a default value!

```r
> matrix
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
{
  data <- as.vector(data)
  if (missing(nrow))
    nrow <- ceiling(length(data)/ncol)
  else if (missing(ncol))
    ncol <- ceiling(length(data)/nrow)
  .Internal(matrix(data, nrow, ncol, byrow, dimnames))
}
<environment: namespace:base>
```
Calling a function: (ii) Returning values

Here we create our first long(ish) function `dihash` that given a hash, **returns the three values specifying a segment** -- The argument is `h` which has no default value

```r
> dihash <- function(h){

    r1 <- (h %% 2^8) + 1
    h <- (h-r1+1)/2^8
    r2 <- (h %% 2^8) + 1
    dtr <- (h-r2+1)/2^8

    return(c(r1,r2,dtr))

}
```

In this example, we make an explicit call to the function `return()` which, as you might guess, **signals a stop to the computations** executed by our function, **returns** a vector of indices and **hands control back** to the person or function that called it.
Calling a function: (ii) Returning values

In general, a series of valid R expressions contained in curly braces, {}, defines a logical block of code -- the expressions can be separated by newlines or by semicolons

```r
> objects()
character(0)

> { x <- 1:3 ; y <- 12 ; z <- y-x }
> objects()
[1] "x" "y" "z"

> b <- { x <- 1:3 ; y <- 12 ; z <- y-x }
> b
[1] 11 10  9

> objects()
[1] "b" "x" "y" "z"
```

The value of a block of code is the result of the last expression executed, a value we can, if we like, assign to a variable
Calling a function: (ii) Returning values

Functions **always return values**; they do so either **explicitly** with a call to `return()` or **implicitly**, using the value of the last expression evaluated by the function.

When we execute a function and do not assign its output (the returned value) to a new variable, it will often just print to the screen; sometimes we don’t see value because the function returns **NULL** (a reserved word representing a null object in R) or the function’s author has used `invisible()`, a variant of `return()`.

As its name suggests, `invisible()` will not print to the screen...

```r
> ihash <- function(r1, r2, dtr)
    invisible(dtr*2^16 + (r2-1)*2^8 + (r1-1))
> ihash(201, 135, 216)

> ihash <- function(r1, r2, dtr) dtr*2^16 + (r2-1)*2^8 + (r1-1)
> ihash(201, 135, 216)
[1] 14190280
```
Calling a function: (iii) Avoiding side-effects

In most cases, **R makes a copy of the data** you supply to a function so that while operations that take place in the body of the function **won’t change your original data** (this is referred to as **pass-by-value semantics** -- as opposed to the pass-by-reference construction we saw in Python)

Similarly, **variables defined in the body of a function are local to that function** and do not interfere (unless you explicitly request them to) with variables of the same name in your workspace -- more on this later

In general, **R makes it difficult for a function to produce so-called side effects** other than the value it returns (although a few functions like `options()` and `par()` are used exclusively for their side effects -- what are these functions? how do you find out? -- as is assignment)
Calling a function: (iv) Default values

As R evaluates the expressions in the body of a function, when it first encounters one of its formal arguments, it will check the list of matched expressions for a value and use it (this is referred to as lazy evaluation -- more on this later)

If it can’t find a match, it will check to see if there is a default value specified in the function’s definition (something we’ll get to later) -- if it still can’t figure out what to do, it will exit the function with an error
Control-flow structures

Finally, we review a couple of the structures that R has for iterating (for- and while-loops) and for conditional evaluation (if-else statements) -- In each case, we’ll see that the body of these expressions is bracketed by { }’s as opposed to the indentation syntax used by Python
Conditional evaluation

As we’ve seen in Python, we often find ourselves in a position conditional evaluation, and almost every language you will come across implements this idea in some way -- in R, its basic syntax is given by

```r
if (condition1) {
    statement1
} else if (condition2) {
    statement2
} else {
    statement3
}
```

Where we have defined two new reserved words, if and else
Conditional evaluation

The operation of this construction is as you might expect: First the condition is evaluated; if it is TRUE, we evaluate statement1, otherwise we move onto condition2; if it's true we evaluate statement2 and otherwise we move on to statement3 and actually, the else clause is optional

If the result of the condition is numeric and not logical, a 0 is treated as FALSE and any other value is TRUE -- If the result of the condition is not numeric or logical, or if it is an NA, you will get an error
Boolean expressions

The operators & and | are vectorized in the sense that they apply to vectors elementwise; R provides another set of operators, && and || that are not vectorized (and will ignore anything but the first element of a vector if you provide them with more)

These operators are often used in if/else constructions (along with commands any() and all()) that take logical vectors as their input and, as their names suggest, evaluate to TRUE if any elements of the vector are TRUE (any) or if all of the elements in the vector are TRUE (all)
Boolean expressions

&& and || also employ **short-circuit semantics**, meaning, they evaluate only as much as they need to determine the value of an expression -- A && B for example will return FALSE if A is FALSE, no matter what can be said about B and A | | B will return TRUE immediately if A is TRUE

```r
globalFunc := function()
  print("hi")
else
  print("bye")

globalFunc()
```
More control-flow

Looping is another construction that lets us repeat a block of code a fixed number of times (for-loops) or until a certain logical condition is satisfied (while-loops) -- These constructions are illustrated on the next page

Note, however, that because many of R’s operations are vectorized, you should think before you loop -- For example, you should not loop over the elements of vectors to execute simple mathematical expressions
For-loops

The syntax of a for-loop is as follows

```r
for (name in vector) {
    statement
}
```

Here, `for` and `in` are reserved words in the language; with each pass through the loop, **the variable name is assigned the next value from vector** -- In many examples you iterate over a sequence of integers, say, 1:100, although there’s no reason why you couldn’t loop over the lower case letters from a-z

```r
for(x in letters){
    print(paste("the letter is",x))
}
```
While-loops

The syntax of a while-loop is as follows:

```plaintext
while ( condition ) {
    statement
}
```

Each iteration of this loop begins by evaluating the condition; as long as it is TRUE, execution will continue -- Which might make you a little nervous about loops that never end!
Break-ing out of a loop

Like Python, R provides the functions `break()` and `continue()` that let you exit a loop (or one iteration) prematurely -- By invoking `break()` you leave the loop entirely, while `continue()` stops the current iteration and places you at the “top” of the next pass

Murrell’s text discusses most of these constructions in some detail; and I also refer you to the texts by Dalgaard and Murdoch -- We will return to the formalities of function evaluation in a later lecture, but this should get you going for now
Databases

For the remainder of this class we’re going to talk about how you can use R to interact with a database -- We’ll focus today mainly on relational databases (as opposed to the NoSQL options like MongoDB)

The data we’ll use come from an MIT project studying social interactions using smartphones as autonomous data collectors...
Reality Mining

Reality Mining is a project started at MIT that promised a new kind of sociological research -- It started with the idea that just about everyone has or will have a mobile phone.

In our first lecture we discussed the fact that these devices can act as a kind of wearable sensor, collecting time- and location-stamped sounds, images, text and even (with custom applications) survey responses (“tags”).

Unlike the Participatory Sensing work we have discussed, Reality Mining depends on passive monitoring of your phone and its various systems -- The Media Lab group created a small piece of monitoring software that runs on certain mobile phones (since that point Nokia has a project called EveryBit that allows you to capture, well, every bit on the phone).
Experimenting with Context, Content, and Community to Improve Relevancy

Richard A. Hankins, Esa Etelaiperä, David Racz
Nokia Research Center - Palo Alto
{rich.hankins,esa.etelaiperadavid.racz}@nokia.com

Motivation

We have become increasingly dependent on search engines and other information services for the discovery, organization, and presentation of information. Unfortunately, the relevancy of the results returned by these services has plateaued. The key to improving relevancy may lie with incorporating knowledge about the context and communities in which content is created and consumed. However, many fundamental questions remain; for example, what aspects of context, content and community are most useful for improving relevancy? And what is the cost to privacy? To answer these questions we plant to collect and analyze a large amount of data from many different users.

Mobile devices are ideally positioned to capture the context, content and community data required to make potential breakthroughs in relevancy. Phones already capture much of the context and community of users, and more context is being generated on phones everyday. The C3 (Context, Content, and Community) group at Nokia Research Center - Palo Alto is developing the infrastructure for large-scale collection and analysis of this data. This will be a unique platform for information-relevancy research, data mining, algorithm development, and community-service experimentation and deployment.

![Services API](image)

Figure 1: Services API

Large-Scale Experimentation

Participants in our large-scale user studies will run special client software in the background on each of their mobile devices. This software will periodically collect all content, context, and community data from the device, and then push it into our Data Aggregation Center (DAC). A special data management software layer, called the DAC Services API, will provide access control to this data to all services. This process is illustrated in Figure 1.

To access participants’ data, researchers will either leverage existing services or create their own through the DAC Services API. As illustrated in Figure 1, standard services will include data visualization tools, data mining tools, search, and a data broker (a trusted intermediary that connects data suppliers to consumers). The actual amount and type of data available to each researcher will depend upon the security preference of each participant as well as the researcher’s own permissions. Of course, each participant will have full access to their own data and possibly to a subset of other users’ data, depending on explicit sharing permissions.

Examples of content and community data we plan to collect include: SMS messages, movies, pictures, address books, calendars, call logs, and message logs. Examples of context data include: battery strength, charger status, idle time, and currently active applications. We also plan to collect a large set of radio-related context, including GPS data, all visible cell IDs and strengths, and all visible Bluetooth and WLAN MAC addresses. The software service responsible for collecting data from users is called EveryBit, and is described next.

EveryBit is a community-focused, web-based filesystem for mobile devices. It supports efficient data archiving, search, and retrieval, and also allows users to easily publish and share their data with others. The potential benefits to the user include:

- Community Sharing - Facilitates communities where users can efficiently share (potentially) large amounts of data amongst other devices and other users.
- Unlimited Storage - Lowers the barrier to content creation on the phone.
- Persistence - Provides a reliable data store in case of device failure, loss, etc.

The motivation for providing this service is simple: We need to bootstrap our data collection effort, so we will offer the service for free in exchange for collecting users’ context information.

Current Status

The EveryBit service will soon be distributed to select participants within the NRC-Palo Alto group. After this initial field test, a larger test is planned for all of Nokia Research. Upon initiating this second field test, we plan to begin collaborating with groups inside and outside of Nokia on designing experiments, collecting data, and analyzing results.
Machine Perception and Learning of Complex Social Systems

Reality Mining defines the collection of machine-sensed environmental data pertaining to human social behavior. This new paradigm of data mining makes possible the modeling of conversation context, proximity sensing, and temporospatial location throughout large communities of individuals. Mobile phones (and similarly innocuous devices) are used for data collection, opening social network analysis to new methods of empirical stochastic modeling.

The original Reality Mining experiment is one of the largest mobile phone projects attempted in academia. Our research agenda takes advantage of the increasingly widespread use of mobile phones to provide insight into the dynamics of both individual and group behavior. By leveraging recent advances in machine learning we are building generative models that can be used to predict what a single user will do next, as well as model behavior of large organizations.

We have captured communication, proximity, location, and activity information from 100 subjects at MIT over the course of the 2004-2005 academic year. This data represents over 350,000 hours (~40 years) of continuous data on human behavior. Such rich data on complex social systems have implications for a variety of fields. The research questions we are addressing include:

- How do social networks evolve over time?
- How entropic (predictable) are most people's lives?
- How does information flow?
- Can the topology of a social network be inferred from only proximity data?
- How can we change a group's interactions to promote better functioning?

If you have a Nokia Symbian Series 60 Phone (such as the Nokia 6600) with a data plan, you can participate. Additionally, we have cleaned the 2004-2005 data of identifiable information and are making it available to other researchers within the academic community. Both the mobile phone application and the resultant dataset can be downloaded here.
Reality Mining

When using a mobile phone, your location is roughly known (even without using a GPS device) because you “associate” with a particular cell tower (the one with the strongest signal) and the position of cell towers are known by the service provider.

More advanced techniques use the signal strength from several nearby towers to perform a kind of triangulation to resolve your position -- If you have a smartphone, so-called WiFi finger printing looks at the wireless networks that are visible to you to provide a finer estimate of your location.

In addition, many mobile phones have the ability to “see” other local wireless devices via the Bluetooth protocol; this means that it is possible to passively track social interactions and exhibit daily patterns of contact.
Bluetooth

This protocol was introduced by Ericsson in 1994 and this year it is predicted to hit 95% penetration for mobile internet devices -- it has been reported that the “installed base” of Bluetooth devices was nearly 2 billion.

You might be own Bluetooth wireless headset, keyboard, or mouse; the protocol was initially designed to form ad hoc networks so that nearby devices could cooperate in some sense.

One feature of this protocol is something called device discovery; a Bluetooth phone can identify information on other Bluetooth devices within 5-10 meters.
Reality Mining

In addition, mobile phones provide information about users’ communication patterns: Who did you call? Who did you text?

Here, for example, is a representation of user locations with links indicating current calls.

We can also collect information about the device itself; Is it idle? Is the battery charging?
Reality Mining

We'll start by looking at just one part of the data, the call history for one participant for the full 9 month period -- We have isolated just a few variables to keep things simple.

The data have been exported in a CSV file (comma separated values), a plain text data format that uses commas to indicated different fields -- Each row represents a different call and each column records aspects of the call (start and end times, duration, direction, etc.).

You can scp the data from homework.stat202a.org (the file is located at /data/reality/calls.csv) or you can download it from the Stat 202a web site at:

http://www.stat.ucla.edu/~cocteau/stat202a/calls.csv
# on homework.stat202a.org

% wc /data/reality/calls.csv
   60780  244870 4717342 /data/reality/one_call.csv

% head /data/reality/calls.csv

"endtime","starttime","phonenumber_oid","contact","description","direction","duration"
"2004-08-03 19:07:26","2004-08-03 19:07:26",3,-1,"Packet Data","Outgoing",0
"2004-08-03 19:07:26","2004-08-03 19:07:26",3,-1,"Packet Data","Outgoing",0
"2004-08-03 19:07:26","2004-08-03 19:07:26",3,-1,"Packet Data","Outgoing",0
"2004-08-03 19:07:26","2004-08-03 19:07:26",3,-1,"Packet Data","Outgoing",0

# each row refers to a different call and each of the 7 columns records a
different feature or aspect of that call -- this CSV file has interpretable
(for the most part) labels for each column that help us get a sense of what
data are contained here
Reading the data into R

If you are running R on homework.stat202a.org you can load the data with the command

```r
> calls <- read.csv("/data/reality/calls.csv")
```

Or if you would like to work from your own machine, you can open a `url` connection and load it into R from the web

```r
> loc <- url("http://www.stat.ucla.edu/~cocteau/stat202a/calls.csv")
> calls <- read.csv(loc)
```
```r
> head(calls)

<table>
<thead>
<tr>
<th></th>
<th>endtime</th>
<th>starttime</th>
<th>phonenumber_oid</th>
<th>contact</th>
<th>description</th>
<th>direction</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2004-08-03 19:07:26</td>
<td>2004-08-03 19:07:26</td>
<td>3</td>
<td>-1</td>
<td>Packet Data</td>
<td>Outgoing</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2004-08-03 19:07:26</td>
<td>2004-08-03 19:07:26</td>
<td>3</td>
<td>-1</td>
<td>Packet Data</td>
<td>Outgoing</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2004-08-03 19:07:26</td>
<td>2004-08-03 19:07:26</td>
<td>3</td>
<td>-1</td>
<td>Packet Data</td>
<td>Outgoing</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2004-08-03 16:37:52</td>
<td>2004-08-03 16:26:53</td>
<td>1959</td>
<td>95</td>
<td>Voice Call</td>
<td>Outgoing</td>
<td>659</td>
</tr>
<tr>
<td>5</td>
<td>2004-08-03 19:07:26</td>
<td>2004-08-03 19:07:26</td>
<td>3</td>
<td>-1</td>
<td>Packet Data</td>
<td>Outgoing</td>
<td>0</td>
</tr>
</tbody>
</table>

> class(calls)
[1] "data.frame"

> dim(calls)
[1] 60779     7

> class(calls$duration)
[1] "integer"

> class(calls$description)
[1] "factor"

> levels(calls$description)
[1] "Packet Data" "Short Message" "Voice Call"
```
The data

While we will focus on a single participant, the Reality Mining study consisted of tracking **100 people for 9 months** using specially prepared Nokia 6600 smart phones -- 75 of the users were students or faculty in the Media Lab, while the remaining 25 come from the Sloan Business School.

In addition to call logs, the researchers recorded the identities of nearby Bluetooth devices, the cell towers each user “associated” with, and statistics related to application usage and phone status.

The data are distributed freely in the form of a **relational database**...
What is a database?

- an organized body of related information

- A database is a collection of information stored in a computer in a systematic way, such that a computer program can consult it to answer questions. The software used to manage and query a database is known as a database management system (DBMS). The properties of database systems are studied in information science.

- Data stored on computer files or on CD-ROM. A database may contain bibliographic, textual or numeric data. The data are usually structured so that they may be searched in a number of ways. A variety of databases is accessible via this website.

- A database is an organised collection of information records that can be accessed electronically.

- is an organized collection of information stored on a computer.

- A database is a collection of data that is organized so that its contents can easily be accessed, managed and updated.

- A collection of information that has been systematically organized for easy access and analysis. Databases typically are computerized.

- A collection of information arranged into individual records to be searched by computer.

- Any organized collection of information; it may be paper or electronic.

- a standardized collection of information in computerized format, searchable by various parameters; in libraries often refers to electronic catalogs and indexes.

- A collection of electronic records having a standardized format and using specific software for computer access.

- A collection of information organized and presented to serve a specific purpose. A computerized database is an updated, organized file of machine readable information that is rapidly searched and retrieved by computer.

- A set of data that is structured and organized for quick access to specific information.
Databases “out there”

In industry, in large (and even not-so-large) applications, you will be called upon to access data from databases -- These are often commercial systems from vendors like Oracle or even SAS

The data from the Reality Mining project were provided to us in the form of a MySQL database...
MySQL

At a technical level, MySQL is a multithreaded, multi-user, structured query language (SQL) database management system (DBMS)

It is distributed freely under the GPL (GNU Public License, talked about software licenses previously)

It is owned and sponsored by a for-profit company that sells support and service contracts, as well as commercially-licensed copies of MySQL
Data models

To ground our discussion somewhat, a database provides a kind of **organizational structure** to the data it contains and defines **operations that can be performed** on the data.

We can think of a data frame in R as having a simple (the simplest) "**flat**" **data model** -- It is essentially a two-dimensional array of data elements where all members of a given column have similar values, and members of each row are related in some way.
The relational model

In a broad sense, a relational database contains multiple, linked tables, each similar to the flat, single-table model; the word relation is borrowed from mathematics where it represents an abstraction that we can think of as a table.

In addition to working with many tables at once, a relational database is places restrictions on each table’s content.

Each table in a database has a unique name; each column within a table has a unique name (within the table); and each column also has a data type associated with it and all the values in a single column must be of that type.

As we saw last time, the last restriction also applies to data frames in R -- Each column is a vector and vectors can have only one data type.
The relational model

Formally, an **entity** is an abstraction of **an object, event or concept** represented in the database -- The columns of each table are referred to as **attributes**; its rows are called **tuples**; and links between entities encode their **relationships** (We use the term **instance** when referring to a particular occurrence of the object)

Next, generic integrity constraints are imposed that apply to the way we **associate elements in one table to those in another** -- This is a abstract right now, but we’ll get back to it

Finally, the model includes a series of **operators through which users query the database**, deriving new tables (relations) from old ones -- For example, we might extract a subset of a given table, perform simple computations, or merge the data in several tables
The relational model

In 1968, E. F. Codd, a researcher at IBM and a mathematician by training realized that a mathematical abstraction could be used to “inject some solid principles and rigor” into the field of database management.

His original definition of the so-called relational model appeared in an IBM research report in 1969; it consisted of three components, related to the structure, integrity and manipulation of data.
The relational model

As we noted, this model can be described mathematically -- In some sense it is an idealization, “an abstract machine”

This is distinct from its implementations (technologies like SQL or specific products like MySQL or SQLite or Oracle) that may or may not adhere to the principles of the abstract model

And this is the balance we are going to strike today -- We want you to know that there is a very rational system at the core of the tools we will be talking about, but we don’t really have the space to delve into it deeply

Instead, we will take a somewhat pragmatic approach, and prepare you to interact with a technology and some specific DBMS products, leaving the more elegant model to a second course
Benefits of a database

A database management system (DBMS) is a piece of software that implements this abstraction or data model for us -- Often these systems can be large, providing access to hundreds or thousands of people, but they can also be quite small (there is a version of SQLite on your desktop, for example)

Large production DBMS have advantages in that the data they contain can be shared and accessed by many users; it is also possible to regulate the kind of access granted to each user, introducing a layer of security

In these cases, redundancy is reduced in the sense that not everyone has to have their own private copy of the data; as a corollary we also have the opportunity to reduce inconsistencies in the data and maintain better control over data integrity

Finally, the act of creating a database and deciding on how data are to be represented often forces discussions by the users about what services they require from the database; it also allows for the introduction of data standards which could enable the interchange of data with other systems and organizations
But why not just load the data into R?

First, as a main memory program, there’s a definite limit to the size of your data -- Granted computer memory keeps growing, but our capacity to dream up ways to fill it certainly keeps pace.

Because data are kept in memory and only written to disk when you exit your session (or you issue an explicit command to save objects), others cannot regularly see the changes you are making to data -- Put another way, R doesn’t support concurrent access to its data.

The format for data stored in R is somewhat specialized and is not directly readable by other systems -- Of course, you can output it as comma separated values or something more portable...
Reality Mining

In the Reality Mining database, there are several tables related to the objects under study -- In some cases data have been removed to protect the privacy of the participants

- **cellname**: oid, name, person_oid, celltower_oid
- **celltower**: oid, name
- **device**: oid, macaddr, person_oid, name
- **person**: name, password, email, phonenumber_oid,...
In addition, there are several tables related to events that were captured during the monitoring period -- Events include using a particular cell tower, calling someone or being near a Bluetooth device

**callspan**: oid, endtime, starttime, person_oid, phonenumber_oid, callid, contact, description, direction, duration, number, status, remote

**cellspan**: oid, endtime, starttime, person_oid, celltower_oid

**devicespan**: oid, endtime, starttime, person_oid, device_oid
Reality Mining

Here are some entries in the `celltower` table -- note that it has just two columns

<table>
<thead>
<tr>
<th>oid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1343</td>
<td>53411, AT&amp;T/Cingu</td>
</tr>
<tr>
<td>1551</td>
<td>13, AT&amp;T Wirel</td>
</tr>
<tr>
<td>1552</td>
<td>2832, AT&amp;T Wirel</td>
</tr>
<tr>
<td>1559</td>
<td>2083, AT&amp;T Wirel</td>
</tr>
<tr>
<td>1842</td>
<td>40811, TMO</td>
</tr>
<tr>
<td>1843</td>
<td>40311, TMO</td>
</tr>
<tr>
<td>1844</td>
<td>40332, TMO</td>
</tr>
<tr>
<td>1845</td>
<td>40333, TMO</td>
</tr>
<tr>
<td>1861</td>
<td>40763, TMO</td>
</tr>
<tr>
<td>1863</td>
<td>48732, TMO</td>
</tr>
</tbody>
</table>

The entities here are, well, separate cell towers, and each tower has a name given by the operator (some are T-Mobile and some are AT&T Wireless) -- One could easily imagine this table also including information about the location of the tower, say latitude and longitude of the station
Reality Mining

Each table in a database has a **primary key**; this is a unique identifier for each row in the table -- For the `cellname` table, the key is just the column labeled ‘oid’ which is a running count or row ID

It is possible to create a primary key by combining the values of two or more columns to form what is called a composite primary key

Now, consider the `cellname` table -- It provides naming information about cell towers also, but this time the names are given by participants in the Reality Mining project
Reality Mining

The cellname table, for example includes the following entries

+-----+-----------------+------------+---------------+
| oid | name            | person_oid | celltower_oid |
+-----+-----------------+------------+---------------+
| 643 | ML              | 29         | 3393          |
| 644 | Office          | 29         | 19290         |
| 645 | Greg's apt      | 29         | 19291         |
| 646 | Greg's apt      | 29         | 4373          |
| 647 | Greg's apt      | 29         | 4377          |
| 648 | Jon's apartment | 29         | 3442          |
| 649 | Red hat         | 29         | 3427          |
| 650 | Nathan's apt    | 29         | 3428          |
| 651 | Porter sq t     | 29         | 19292         |
| 652 | Home            | 29         | 1861          |
+-----+-----------------+------------+---------------+

The **entity** referred to in each row is a **named location** -- A participant was asked to describe where they were (free-text like “Home” or “Office” or “Nathan’s apt”) when they were “associated” with a particular cell tower.

(Note that because we have limited the data to the contributions from a single person, all these locations were identified and given names by the same participant, number 29)
Reality Mining

The relational model allows us to link data between tables -- The `celltower_oid` referred to in the `cellname` table, corresponds to the primary key `oid` in the `celltower` table and we see that the particular tower labeled “Home” is owned by T-Mobile -- We refer to the `celltower_oid` column in the `cellname` table as a **foreign key**

<table>
<thead>
<tr>
<th>cellname table</th>
<th>celltower table</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-------------+-----------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oid</td>
</tr>
<tr>
<td>+-------------+-----------------+-------------------+-------------------+-------------------+-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>643</td>
</tr>
<tr>
<td></td>
<td>644</td>
</tr>
<tr>
<td></td>
<td>645</td>
</tr>
<tr>
<td></td>
<td>646</td>
</tr>
<tr>
<td></td>
<td>647</td>
</tr>
<tr>
<td></td>
<td>648</td>
</tr>
<tr>
<td></td>
<td>649</td>
</tr>
<tr>
<td></td>
<td>650</td>
</tr>
<tr>
<td></td>
<td>651</td>
</tr>
<tr>
<td>652</td>
<td>Home</td>
</tr>
</tbody>
</table>
| +-------------+-----------------+-------------------+-------------------+-------------------+-------------------+
Reality Mining

Foreign keys express relationships between entities stored in a database; the so-called cardinality of these relationships can be one-to-one, many-to-one or many-to-many.

Because a single cell tower could service multiple places, the relationship with the celltower table is many-to-one (and is represented by placing a foreign key in the table for cell names -- the many -- that refers to the primary key in the table for cell towers -- the one).
Reality Mining

The `cellspan` table, on the other hand, expresses a many-to-many relationship

<table>
<thead>
<tr>
<th>oid</th>
<th>endtime</th>
<th>starttime</th>
<th>person_oid</th>
<th>celltower_oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1097404</td>
<td>2004-07-26 21:01:34</td>
<td>2004-07-26 21:00:13</td>
<td>29</td>
<td>1552</td>
</tr>
</tbody>
</table>

The entity referred to in each row is an “association event” -- Whenever a participant (here just participant 29) uses or associates with a particular cell tower, the beginning and end times are recorded as well as the id of the participant and the id of the tower (both foreign keys that refer to primary keys in the person and celltower tables, respectively)

This table is an example of a many-to-many relationship -- Each participant will use many cell towers and each cell tower will be used by many participants
The relational model

When creating a database, a fair bit of thought has to go into how many tables, what kind of information is stored in each table, and how they are linked together.

Murrell discusses several different strategies (at a very high level) for designing a database including data integrity and so-called database normalization.

The latter refers to a process whereby we reduce redundancy in a database (not storing the same information in many tables) and ensuring that the links between tables make sense.
Relational databases

DBMSs are optimized to handle a set of queries made by users; these queries essentially let us make new tables out of old one by *subsetting*, *merging* or performing *simple computations*

A **query language** allows users to interact with the database, reducing the data and summarizing it before retrieving the results

The **Structured Query Language** (SQL) is widely used, and is supported by most commercial databases
An quick look

On your lab computers and on homework (and you can download versions for Windows) you have access to a fairly powerful DBMS, SQLite -- It has a command line interface or shell that you can use to interact with a database

We'll also see how we can access a database from within R, allowing us to load tables (as data frames) from a DBMS so that we can act on it further -- Fitting a model, producing graphics, and storing the results back into the database

SQLite has a small command set and otherwise accepts SQL expressions...
Welcome.

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

Sponsors

Ongoing development and maintenance of SQLite is sponsored in part by SQLite Consortium members, including:

Oracle - Software. Hardware. Complete.

Mozilla - Working to preserve choice and innovation on the internet.

Bloomberg - A world leader in financial-information technology.

Current Status

- Version 3.7.3 of SQLite is recommended for all new development. Upgrading from version 3.7.2 is optional. Upgrading from all other SQLite versions is recommended.

Common Links

- Features
- Frequently Asked Questions
- Well-known Users
- Getting Started
- SQL Syntax
  - Pragmas
  - SQL functions
  - Date & time functions
  - Aggregate functions
- C/C++ Interface Spec
  - Introduction
  - List of C-language APIs
.bail ON|OFF Stop after hitting an error. Default OFF
.databases List names and files of attached databases
.dump ?TABLE? ... Dump the database in an SQL text format
.echo ON|OFF Turn command echo on or off
.exit Exit this program
.explain ON|OFF Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF Turn display of headers on or off
.help Show this message
.import FILE TABLE Import data from FILE into TABLE
.indices TABLE Show names of all indices on TABLE
.mode MODE ?TABLE? Set output mode where MODE is one of:
  csv  Comma-separated values
  column Left-aligned columns. (See .width)
  html  HTML <table> code
  insert SQL insert statements for TABLE
  line One value per line
  list Values delimited by .separator string
  tabs  Tab-separated values
  tcl  TCL list elements
.nullvalue STRING Print STRING in place of NULL values
.output FILENAME Send output to FILENAME
.output stdout Send output to the screen
.prompt MAIN CONTINUE Replace the standard prompts
.quit Exit this program
.read FILENAME Execute SQL in FILENAME
.schema ?TABLE? Show the CREATE statements
.separator STRING Change separator used by output mode and .import
.show Show the current values for various settings
.tables ?PATTERN? List names of tables matching a LIKE pattern
.timeout MS Try opening locked tables for MS milliseconds
.width NUM NUM ... Set column widths for "column" mode
Let’s start with the data for a single Reality Mining participant (our friend number 29) -- SQLite stores its data in a single file which, for the following few slides you can access on homework.stat202a.org

% cp /data/reality/singleRM.db .

or download from the Stat 202a web site

http://www.stat.ucla.edu/~cocteau/stat202a/singleRM.db
% sqlite3 singleRM.db

SQLite version 3.4.0
Enter ".help" for instructions

sqlite> .mode column
sqlite> .header on
sqlite> SELECT * FROM cellspan LIMIT 10;

<table>
<thead>
<tr>
<th>oid</th>
<th>endtime</th>
<th>starttime</th>
<th>person_oid</th>
<th>celltower_oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1097404</td>
<td>2004-07-26 21:01:34</td>
<td>2004-07-26 21:00:13</td>
<td>29</td>
<td>1552</td>
</tr>
</tbody>
</table>
SQL: Example

The SELECT statement is used to retrieve data from a database -- You specify the table you want to draw from and various conditions you want to impose to extract a subset of the data

```
SELECT * FROM cellname;
```

This will return the complete cellname table; you can add “LIMIT 10” to reduce the amount of output
% sqlite3 singleRM.db

SQLite version 3.4.0
Enter ".help" for instructions

sqlite> .mode column
sqlite> .header on
sqlite> SELECT * FROM cellspan LIMIT 10;

<table>
<thead>
<tr>
<th>oid</th>
<th>endtime</th>
<th>starttime</th>
<th>person_oid</th>
<th>celltower_oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1097404</td>
<td>2004-07-26 21:01:34</td>
<td>2004-07-26 21:00:13</td>
<td>29</td>
<td>1552</td>
</tr>
</tbody>
</table>
SQL: Example

SQLite has various commands that lets you examine the structure of a database; `.table` lets you list out the tables and `.schema` gives you the structure of the tables (listing their columns, the associated data types)

We give a couple examples here but reserve a fuller discussion of SQL in a moment when we introduce how R plays in this world...
sqlite> .tables

acthiyspan  cellname      celltower     device        person
callspan    cellspan      coverspan     devicespan    phonenumber

sqlite> .schema cellname

CREATE TABLE `cellname` (
  `oid` integer NOT NULL primary key autoincrement,
  `name` char(255) NOT NULL default '',
  `person_oid` integer NOT NULL default '0',
  `celltower_oid` integer NOT NULL default '0');

sqlite> SELECT name FROM cellname LIMIT 5;

name
----------
ML
Office
Greg's apt
Greg's apt
Greg's apt

sqlite> SELECT * FROM cellname WHERE name = "Home";

<table>
<thead>
<tr>
<th>oid</th>
<th>name</th>
<th>person_oid</th>
<th>celltower_oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>652</td>
<td>Home</td>
<td>29</td>
<td>1861</td>
</tr>
<tr>
<td>719</td>
<td>Home</td>
<td>29</td>
<td>6013</td>
</tr>
<tr>
<td>724</td>
<td>Home</td>
<td>29</td>
<td>6012</td>
</tr>
</tbody>
</table>
In general, we could consider commands of the form

```
SELECT column(s) FROM relation(s)
[WHERE constraints];
```

In the previous example, we looked for cell towers that were labeled as “Home”

SQL also allows for simple computations on the database (averages, finding a minimum or a maximum of a column) or counting the number of elements in a table -- Rather than go into details here, let’s bring R into the mix...
Database access through R

There are three ingredients to accessing a relational database in R

**The driver:** This facilitates the communication between the R session and the database management system (DBMS) by defining, say, data type mappings

**The connection:** This object encapsulates the actual connection with the DBMS and carries out the requested queries

**The result:** This tracks the status of a query, such as the number of rows that have been fetched and whether or not the query has completed.
# if you are working on the homework machine SQLite is already installed
# as is the R package for communicating with it -- to do this on your own
# machine you will have to install both

> library(RSQLite)
Loading required package: DBI

# the driver...
> drv <- dbDriver("SQLite")

# ... and the connection
> con <- dbConnect(drv,"singleRM.db")

# now issue some commands... what tables do we have?
> dbListTables(con)
[1] "activityspan"  "callspan"     "cellname"     "cellspan"
[2] "celltower"     "coverspan"    "device"       "devicespan"
[3] "person"        "phonenumber" "sqlite_sequence"

# what are the columns in, say, the celltower table?
> dbListFields(con,"celltower")
[1] "oid"  "name"
# next, recall the fields in the cellname table
> dbListFields(con,"cellname")

[1] "oid"           "name"          "person_oid"    "celltower_oid"

# let's bring these data into R
> cellnames <- dbReadTable(con,"cellname")  # read the complete table into R

# the data appear as a data frame...
> class(cellnames)
[1] "data.frame"

> dim(cellnames)
[1] 87  4

> names(cellnames)
[1] "oid"           "name"          "person_oid"    "celltower_oid"

> head(cellnames)
          oid name person_oid celltower_oid
1  643   ML         29         3393
2  644  Office         29         19290
3  645 Greg's apt         29         19291
4  646 Greg's apt         29          4373
5  647 Greg's apt         29          4377
6  648  Jon's apartment         29          3442

# and when we are done working, we close up shop...
> dbDisconnect(con)
> dbUnloadDriver(drv)
Relational databases

As we mentioned before, DBMSs are optimized to handle a set of queries made by users -- Again, these queries essentially let us make new tables out of old by subsetting, merging or performing simple computations

The Structured Query Language (SQL) is widely used, and is supported by most commercial databases...
SQL

The SELECT statement is used to retrieve data from a database -- You specify the table you want to draw from and various conditions you want to impose to extract a subset of the data

```
SELECT * FROM callspan;
```

This will return the complete `callspan` table and is equivalent to calling `dbReadTable` (Note that our SQL statements are always in caps -- The language is not case sensitive, but we do this for readability)

We can add a LIMIT clause to this command so that we don't have to watch thousands of data points flash by -- To accomplish the same thing in R we'd wrap our expressions in a call to `head()`

```
SELECT * FROM callspan LIMIT 10;
```

In the next two slides, we'll assume that the data frame `calls` has been created using the expressions on the following slide....
drv <- dbDriver("SQLite")
con <- dbConnect(drv,"singleRM.db")

dbListTables(con)
[1] "activityspan"    "callspan"        "cellname"        "cellspan"
[5] "celltower"        "coverspan"       "device"          "devicespan"
[9] "person"           "phonenumber"     "sqlite_sequence"

calls <- dbReadTable(con,"callspan")
> names(calls)
[1] "oid" "endtime" "starttime" "person_oid"
[5] "phonenumber_oid" "callid" "contact" "description"
[9] "direction" "duration" "number" "status"
[13] "remote"
SQL

We can understand some of what SELECT does by comparing it to subsetting on data frames -- For example, compare

```
SELECT description, direction FROM callspan;
```

and an operation on our R data frame

```
calls[,c("description","direction")]
```
We can understand some of what SELECT does by comparing it to subsetting on data frames -- For example, compare

```
SELECT direction, duration FROM callspan
    WHERE description = "Voice Call";
```

and an operation on our R data frame

```
calls[calls$description == "Voice Call",
    c("direction", "duration")]
```
> x1 <- calls[calls$description == "Voice Call", c("direction", "duration")]

# to issue an SQL query from R, we provide a single string -- here we
# use the paste() function to help divide the portions of the query into
# readable chunks...
> x2 <- dbGetQuery(con, 
    paste('SELECT direction, duration FROM callspan,' 
    'WHERE description = "Voice Call";'))

# and compare...

> all(x1==x2)
[1] TRUE

# we can also just send the query and pull pieces of data...
> z <- dbSendQuery(con, 'SELECT * FROM callspan;')
> group1 <- fetch(z, n=5)
> group1

    oid             endtime           starttime         person_oid phonenumber_oid
group1
1 370982 2004-08-03 19:07:26 2004-08-03 19:07:26        29               3
2 370983 2004-08-03 19:07:26 2004-08-03 19:07:26        29               3
3 370984 2004-08-03 19:07:26 2004-08-03 19:07:26        29               3
4 370985 2004-08-03 16:37:52 2004-08-03 16:26:53        29        1959
5 370986 2004-08-03 19:07:26 2004-08-03 19:07:26        29               3

callid contact description direction duration number       status remote
1  304      -1 Packet Data  Outgoing        0        Disconnected
2  304      -1 Packet Data  Outgoing        0        Disconnected
3  304      -1 Packet Data  Outgoing        0        Disconnected
4  300      95  Voice Call  Outgoing      659
5  304      -1 Packet Data  Outgoing        0        Disconnected
We can understand some of what SELECT does by comparing it to subsetting on data frames -- For example, compare

```
SELECT description, direction, duration FROM callspan
WHERE contact IN (29,32);
```

and an operation on our R data frame

```
calls[calls$contact %in% c(29,78),
     , c("description","direction","duration")]
```
SQL

The general form of this command is

```
SELECT column(s) FROM relation(s)
  [WHERE constraints];
```

For example, we get closer to our familiar subsetting with statements like

```
SELECT * FROM callspan
  WHERE duration > 1200 AND contact = 78;
```
SQL Review

SQL is primarily designed for data retrieval -- it is not a computational language nor is it a statistical language.

It does, however, contain certain summarizing capabilities in the form of functions that can be applied over rows in a table -- the GROUP BY can be used to identify subsets to apply these over also.

- **COUNT**: returns the number of tuples (rows)
- **SUM**: the total for the attribute
- **AVG**: the average for the attribute
- **MIN**: the minimum across the attribute
- **MAX**: the maximum across the attribute
These additional clauses can be added to the statement, to function like our various flavors of apply

```
SELECT phonenumber_oid, AVG(duration)
FROM callspan GROUP BY phonenumber_oid;
```
We can take it a step further with the following construction

```sql
SELECT phonenumber_oid, AVG(duration)
FROM callspan
GROUP BY phonenumber_oid
HAVING MAX(duration) < 10000;
```

We've taken so much time to build up this expression because certain computations like these can be performed much more quickly in the database -- We would rather avoid pulling hundreds of thousands or millions of records into R just to turn around and take a sum
```r
> system.time(
  y <- dbGetQuery(con,'SELECT * FROM callspan')
  x1 <- aggregate(y$duration,by=list(y$direction,y$description),FUN=sum)
)

    user  system elapsed
     0.000   0.000   0.395

> system.time(
  x2 <- dbGetQuery(con,
    paste('SELECT description, direction, SUM(duration)',
      'FROM callspan',
      'GROUP BY description,direction'))
)

    user  system elapsed
     0.000   0.000   0.369

# here the data are small enough that the differences between the two
# approaches are minimal -- with larger data sets this is certainly not
# the case and the latter is favored over the former (on the full reality mining
# data set, the first approach took 5-10 seconds whereas the latter took
# less than 1)
```
SQL Review

So far we have introduced the Structured Query Language (SQL) and, in particular, discussed the `SELECT` statement that is used to retrieve information from relational databases

The general form of the `SELECT` statement is

```
SELECT column(s)
FROM relation(s)
[WHERE constraints];
```

and we have presented a number of different constructions for setting constraints; we compared these specifically to the subsetting rules we are familiar with in R
Relational databases

When we bring data into R from a DMBS, we convert data from the database into formats that R understands -- While we know about the kinds of data R handles, SQLite only knows about NULL (a DBMS version of NA), integer, real text and “blob” (meaning the data are just stored as bytes without any other structure)

Other DBMS have more elaborate primitive types -- MySQL for example has special types to handle times

<table>
<thead>
<tr>
<th>BOOLEAN</th>
<th>DATE</th>
<th>CHAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>DATETIME</td>
<td>TEXT</td>
</tr>
<tr>
<td>FLOAT</td>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>TIME</td>
<td>YEAR</td>
</tr>
</tbody>
</table>

http://dev.mysql.com/doc/refman/5.0/en/string-type-overview.html
Multiple relations

Consider the following questions we might ask of the data:

1. Find the amount of time participants spend near each other

2. Find the amount of time they are associated with cell towers they have identified as “Work”

Notice that in these cases, we have to combine data contained in multiple tables -- This is an important component of the relational model
Joining relations

Recall that keys uniquely identify the rows (tuples) in a table (relation) -- We use these keys to merge (join) data in different tables (relations)

In the case of the Reality Mining database, each table has a column labeled oid that represents its primary key -- In this case it’s nearly trivial in the sense that the oid is basically a row number

Suppose we want to get a table of all the times our participant 29 was at “Home”, or, rather, associated with a cell tower they labeled as “Home” -- This means we need to combine elements from the cellspan and cellname tables
> cellnames <- dbGetQuery(con,"SELECT * FROM cellname LIMIT 5;")
> celltimes <- dbGetQuery(con,"SELECT * FROM cellspan LIMIT 5;")

> cellnames

<table>
<thead>
<tr>
<th>oid</th>
<th>name</th>
<th>person_oid</th>
<th>celltower_oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>643</td>
<td>ML</td>
<td>29</td>
<td>3393</td>
</tr>
<tr>
<td>644</td>
<td>Office</td>
<td>29</td>
<td>19290</td>
</tr>
<tr>
<td>645</td>
<td>Greg's apt</td>
<td>29</td>
<td>19291</td>
</tr>
<tr>
<td>646</td>
<td>Greg's apt</td>
<td>29</td>
<td>4373</td>
</tr>
<tr>
<td>647</td>
<td>Greg's apt</td>
<td>29</td>
<td>4377</td>
</tr>
</tbody>
</table>

> celltimes

<table>
<thead>
<tr>
<th>oid</th>
<th>endtime</th>
<th>starttime</th>
<th>person_oid</th>
<th>celltower_oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1097404</td>
<td>2004-07-26 21:01:34</td>
<td>2004-07-26 21:00:13</td>
<td>29</td>
<td>1552</td>
</tr>
</tbody>
</table>
# a cross join of cellspan and cellname makes a table representing the cartesian product of the rows of the two tables...

```r
> x <- dbGetQuery(con, paste(
  'SELECT starttime, endtime, name FROM cellspan,cellname'))
> dim(x)
[1] 5630292 3

> dbGetQuery(con,'SELECT COUNT(*) from cellspan')
COUNT(*)
1 64716

> dbGetQuery(con,'SELECT COUNT(*) from cellname')
COUNT(*)
1 87

> 87*64716
[1] 5630292

# now limit this to just those rows where the celltower_oid match, in effect adding name information to the span data -- this is also called an “inner join”

> x <- dbGetQuery(con, paste(
  'SELECT starttime, endtime, name FROM cellspan,cellname',
  'WHERE cellspan.celltower_oid = cellname.celltower_oid'))
> head(x)
            starttime     endtime   name
3 2004-07-26 21:00:13 2004-07-26 21:01:34 S best
```
# now do the same thing but use "aliases" that make the statement a little
# cleaner to read... and only keep those events where the celltower was "Home"

```r
> x <- dbGetQuery(con, paste(
    'SELECT s.starttime, s.endtime, n.name FROM cellspan s, cellname n',
    'WHERE s.celltower_oid = n.celltower_oid',
    'AND n.name = "Home"'))
```

```r
> head(x)
       starttime             endtime name
1 2004-08-01 17:43:31 2004-08-01 17:45:39 Home
3 2004-08-01 18:27:42 2004-08-01 19:20:26 Home
5 2004-08-01 23:34:12 2004-08-02 00:33:14 Home
6 2004-08-01 23:42:46 2004-08-02 00:33:14 Home
```

```r
> x <- dbGetQuery(con, paste(
    'SELECT s.starttime, s.endtime, n.name',
    'FROM cellspan s',
    'INNER JOIN cellname n ON s.celltower_oid = n.celltower_oid',
    'WHERE n.name = "Home"'))
```
Sub-queries

If we only wanted information from cellspan that corresponded to events that involved cell towers having provider-given names we might care about, we could use a sub-query (this example is strained even to state it, but the expression we present is powerful)

This construction allows us to assemble information from another table temporarily -- These kinds of constructions can be nested beyond the simple application in the next example
# here we use a subquery to specify a couple of cell towers by name
# (this example is getting somewhat artificial but the ability to do
# this is what we're illustrating here)

```r
> x <- dbGetQuery(con, paste(
  'SELECT starttime, endtime',
  'FROM cellspan',
  'WHERE celltower_oid IN',
  '(SELECT oid from celltower',
  'WHERE name IN ("40311, TMO", "40311, T - Mobile");'))

> x

<table>
<thead>
<tr>
<th>starttime</th>
<th>endtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004-08-03 00:56:05</td>
<td>2004-08-03 00:59:46</td>
</tr>
<tr>
<td>2004-08-03 01:00:27</td>
<td>2004-08-03 01:00:50</td>
</tr>
<tr>
<td>2004-08-29 00:32:11</td>
<td>2004-08-29 00:34:47</td>
</tr>
<tr>
<td>2004-08-29 00:35:25</td>
<td>2004-08-29 00:37:25</td>
</tr>
<tr>
<td>2004-08-29 00:37:42</td>
<td>2004-08-29 00:37:57</td>
</tr>
<tr>
<td>2004-08-29 00:38:19</td>
<td>2004-08-29 00:40:02</td>
</tr>
<tr>
<td>2004-08-29 00:40:11</td>
<td>2004-08-29 00:40:39</td>
</tr>
<tr>
<td>2004-08-29 00:40:59</td>
<td>2004-08-29 00:41:38</td>
</tr>
<tr>
<td>2004-08-29 00:41:58</td>
<td>2004-08-29 00:43:14</td>
</tr>
<tr>
<td>2004-08-29 00:43:24</td>
<td>2004-08-29 00:45:18</td>
</tr>
<tr>
<td>2004-09-12 12:02:16</td>
<td>2004-09-12 12:10:43</td>
</tr>
<tr>
<td>2004-09-12 12:11:00</td>
<td>2004-09-12 12:11:25</td>
</tr>
<tr>
<td>2004-09-12 12:12:57</td>
<td>2004-09-12 12:14:26</td>
</tr>
<tr>
<td>2004-09-12 12:15:18</td>
<td>2004-09-12 12:16:49</td>
</tr>
</tbody>
</table>
```
Finally, storing data

While our emphasis so far has been on data extraction, with a SQLite database file, you can add data as you like -- This will prove handy, say, for storing hashes from your Shazam example (for those of you using R)

On the next two slides we show how you can populate a table and then use a simple SQL query to extract a particular hash value...
> library(RSQLite)

> drv <- dbDriver("SQLite")
> con <- dbConnect(drv,"mystuff.db")   # creates a file mystuff.db

> dbListTables(con)
character(0)

# now simulate a data frame that has time stamps and hashes (lame)
> m = data.frame(time=sample(1:256,100000,rep=T),hash=sample(1:2^8,100000,rep=T))

> head(m)
  time hash
  1  154   28
  2  101   11
  3  208  102
  4   97  218
  5  153  191
  6  189   86

# and load the table into the db

> dbWriteTable(con,"songs",m)
[1] TRUE

> dbListTables(con)
[1] "songs"

> system("ls -l mystuff.db")
-rw-r--r-- 1 cocteau cocteau 1822720 2010-11-10 20:38 mystuff.db
# finally, let's query the db for one particular hash value...

> qh <- 168

> x <- dbGetQuery(con, paste('SELECT * FROM songs WHERE hash = ', qh))

> dim(x)
[1] 434  3

> head(x)
row_names time hash
1   368  130  168
2   479  252  168
3   686  161  168
4  1075   89  168
5  1157  184  168
6  1518   26  168
Fast

That was a whirlwind introduction to SQL -- We focused mainly on the aspects of DBMS that would get you up and running as quickly as possible.

Keep in mind that there are multiple classes required of DBMS researchers in CS programs -- So this is by no means all there is to say about any of the topics we've touched on.