Introduction to Data Science: Data Representation Models

Héctor Corrada Bravo

University of Maryland, College Park, USA

2020-02-10

Overview

Principles of preparing and organizing data in a way that is amenable for analysis.

Overview

Principles of preparing and organizing data in a way that is amenable for analysis.

Data representation model: collection of concepts that describes how data is represented and accessed.

Overview

Principles of preparing and organizing data in a way that is amenable for analysis.

Data representation model: collection of concepts that describes how data is represented and accessed.

Thinking abstractly of data structure, beyond a specific implementation, makes it easier to share data across programs and systems, and integrate data from different sources.

Overview

- **Structure**: We have assumed that data is organized in rectangular data structures (tables with rows and columns)
- **Semantics**: We have discussed the notion of *values*, *attributes*, and *entities*.

Overview

- **Structure**: We have assumed that data is organized in rectangular data structures (tables with rows and columns)
- **Semantics**: We have discussed the notion of *values*, *attributes*, and *entities*.

So far, *data semantics*: a dataset is a collection of *values*, numeric or categorical, organized into *entities* (*observations*) and *attributes* (*variables*).

Overview

- **Structure**: We have assumed that data is organized in rectangular data structures (tables with rows and columns)
- **Semantics**: We have discussed the notion of *values*, *attributes*, and *entities*.

So far, *data semantics*: a dataset is a collection of *values*, numeric or categorical, organized into *entities* (*observations*) and *attributes* (*variables*).

Each *attribute* contains values of a specific measurement across *entities*, and *entities* collect all measurements across *attributes*.

Overview

In the database literature, we call this exercise of defining structure and semantics as *data modeling*.

Overview

In the database literature, we call this exercise of defining structure and semantics as *data modeling*.

In this course we use the term *data representational modeling*, to distinguish from *data statistical modeling*.

Data representational modeling

- **Data model**: A collection of concepts that describes how data is represented and accessed
- **Schema**: A description of a specific collection of data, using a given data model

Data representational modeling

- Modeling Constructs: A collection of concepts used to represent the structure in the data.

Typically we need to represent types of *entities*, their *attributes*, types of *relationships* between *entities*, and *relationship attributes*

Data representational modeling

- Integrity Constraints: Constraints to ensure data integrity (i.e., avoid errors)

Data representational modeling

- Integrity Constraints: Constraints to ensure data integrity (i.e., avoid errors)
- Manipulation Languages: Constructs for manipulating the data

Data representational modeling

We desire that models are:

- sufficiently *expressive* so they can capture real-world data well,
- *easy to use*,
- lend themselves to defining computational methods that have good performance.

Data representational modeling

Some examples of data models are

- Relational, Entity-relationship model, XML...
- Object-oriented, Object-relational, RDF...
- Current favorites in the industry: JSON, Protocol Buffers, Avro, Thrift, Property Graph

Data representational modeling

- **Data independence:** The idea that you can change the representation of data w/o changing programs that operate on it.
- **Physical data independence:** I can change the layout of data on disk and my programs won't change
    - index the data
    - partition/distribute/replicate the data
    - compress the data
    - sort the data

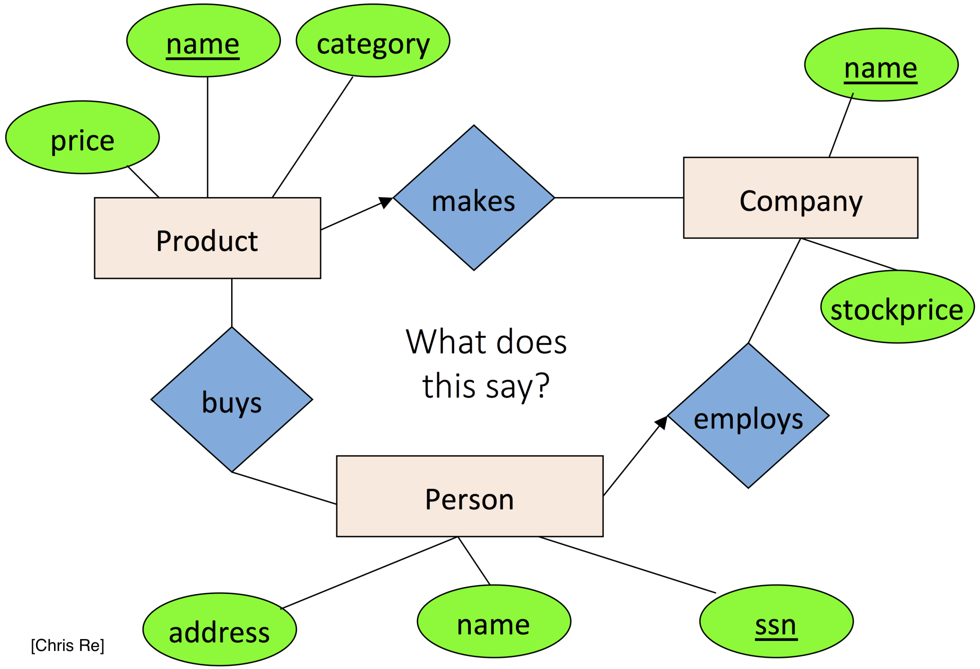
The Entity-Relationship and Relational Models

Modeling constructs:

- *entities* and their *attributes*
- *relationships* and *relationship attributes*.

Entities are objects represented in a dataset: people, places, things, etc.

Relationships model just that, relationships between entities.

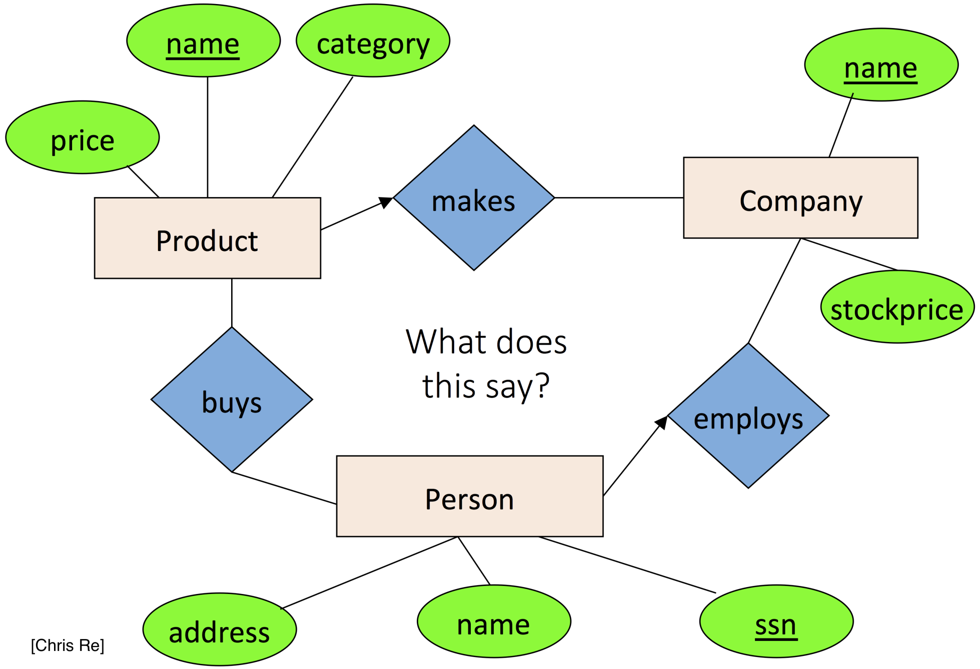


[Chris Re]

The Entity-Relationship and Relational Models

Diagrams:

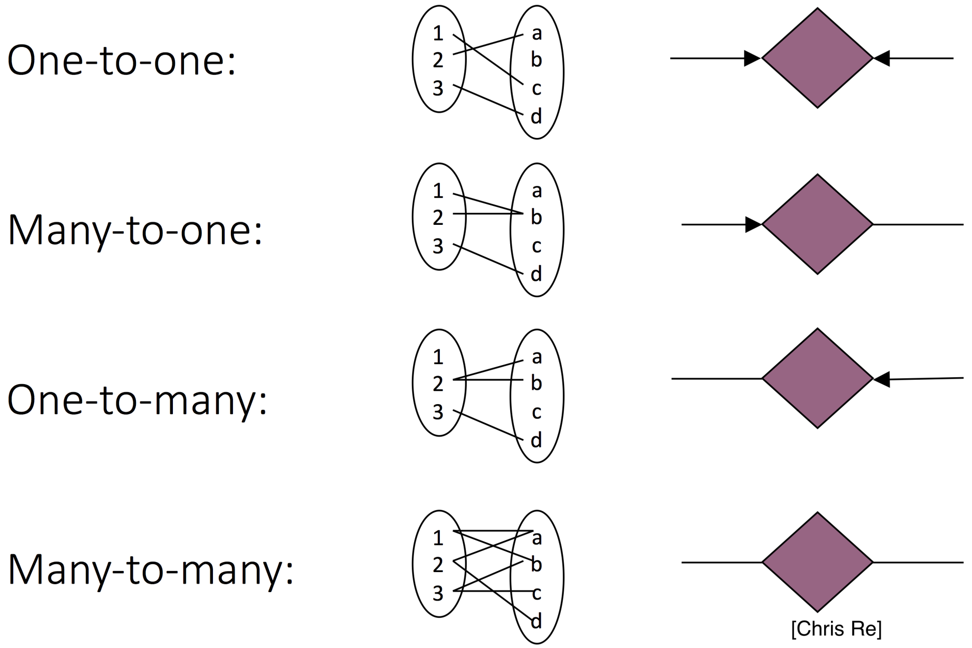
- rectangles are *entitites*
- diamonds and edges indicate *relationships*
- Circles describe either entity or relationship *attributes*.



[Chris Re]

The Entity-Relationship and Relational Models

Arrows are used indicate multiplicity of relationships



The Entity-Relationship and Relational Models

Relationships are defined over *pairs* of entities.

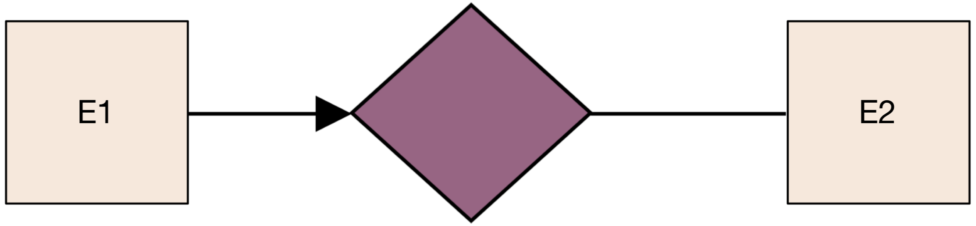
Relationship R over sets of entities E_1 and E_2 is defined over the *cartesian product* $E_1 \times E_2$.

For example: if $e_1 \in E_1$ and $e_2 \in E_2$, then $(e_1, e_2) \in R$.

The Entity-Relationship and Relational Models

Arrows specify how entities participate in relationships.

For example: this specifies that entities in E_1 appear in *only one* relationship pair.



That is, there is a single entity $e_2 \in E_2$ such that $(e_1, e_2) \in R$.

The Entity-Relationship and Relational Models

In databases and general datasets we work on, both Entities and Relationships are represented as *Relations* (tables).

The Entity-Relationship and Relational Models

In databases and general datasets we work on, both Entities and Relationships are represented as *Relations* (tables).

Such that a *unique* entity/relationship is represented by a single tuple (the list of attribute values that represent an entity or relationship).

The Entity-Relationship and Relational Models

In databases and general datasets we work on, both Entities and Relationships are represented as *Relations* (tables).

Such that a *unique* entity/relationship is represented by a single tuple (the list of attribute values that represent an entity or relationship).

How can we ensure *uniqueness* of entities?

The Entity-Relationship and Relational Models

In databases and general datasets we work on, both Entities and Relationships are represented as *Relations* (tables).

Such that a *unique* entity/relationship is represented by a single tuple (the list of attribute values that represent an entity or relationship).

How can we ensure *uniqueness* of entities?

keys are an essential ingredient to uniquely identify entities and relationships in tables.

Formal introduction to keys

- Attribute set K is a **superkey** of relation R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
  - Example: `{SSN}` and `{SSN,name}` are both superkeys of *person*

Formal introduction to keys

- Attribute set K is a **superkey** of relation R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
  - Example: `{SSN}` and `{SSN,name}` are both superkeys of *person*
- Superkey K is a **candidate key** if K is minimal
  - Example: `{SSN}` is a candidate key for *person*

Formal introduction to keys

- Attribute set K is a **superkey** of relation R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
  - Example: `{SSN}` and `{SSN,name}` are both superkeys of *person*
- Superkey K is a **candidate key** if K is minimal
  - Example: `{SSN}` is a candidate key for *person*
- One of the candidate keys is selected to be the **primary key**
  - Typically one that is small and immutable (doesn't change often)
  - Primary key typically highlighted in ER diagram

Formal introduction to keys

- **Foreign key**: Primary key of a relation that appears in another relation
  - `{SSN}` from *person* appears in *employs*
  - *person* called referenced relation
  - *employs* is the referencing relation

Formal introduction to keys

- **Foreign key**: Primary key of a relation that appears in another relation
  - `{SSN}` from *person* appears in *employs*
  - *person* called referenced relation
  - *employs* is the referencing relation

- **Foreign key constraint**: the tuple corresponding to that primary key must exist
  - Imagine:
    - Tuple: `('123-45-6789', 'Apple')`in *employs*
    - But no tuple corresponding to '123-45-6789' in *person*
  - Also called referential integrity constraint

Tidy Data

We use the term *Tidy Data* to refer to datasets that are represented in a form that is amenable for manipulation and statistical modeling.

It is very closely related to the concept of *normal forms* in the ER model and the process of *normalization* in the database literature.

Tidy Data

Here we assume we are working in the ER data model represented as *relations*: rectangular data structures where

1. Each attribute (or variable) forms a column
2. Each entity (or observation) forms a row
3. Each type of entity (observational unit) forms a table

Tidy Data

Here is an example of a tidy dataset: One entity per row, a single attribute per column. Only information about flights included.

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_
2013	1	1	517	515	2	830	
2013	1	1	533	529	4	850	

Structure Query Language

The Structured-Query-Language (SQL) is the predominant language used in database systems.

It is tailored to the Relational data representation model.

SQL is a declarative language, we don't write a *procedure* to compute a relation, we *declare* what the relation we want to compute looks like.

Structure Query Language

The basic construct in SQL is the so-called `SFW` construct: *select-from-where* which specifies:

- *select*: which attributes you want the answer to have
- *from*: which relation (table) you want the answer to be computed from
- *where*: what conditions you want to be satisfied by the rows (tuples) of the answer

Structure Query Language

E.g.: movies produced by Disney in 1990: note the *rename*

```sql
select m.title, m.year

from movie m

where m.studioname = 'disney' and m.year = 1990
```

Structure Query Language

The **select** clause can contain expressions (this is paralleled by the `mutate` operation we saw previously)

- `select title || ' (' || to_char(year) || ')' as titleyear`
- `select 2014 - year`

Structure Query Language

The **where** clause support a large number of different predicates and combinations thereof (this is parallel to the `filter` operation)

- `year between 1990 and 1995`
- `title like 'star wars%'` `title like 'star wars _'`

Structure Query Language

We can include ordering, e.g., find distinct movies sorted by title

```sql
select distinct title

from movie

where studioname = 'disney' and year = 1990

order by title;
```

Structure Query Language

Group-by and summarize

SQL has an idiom for grouping and summarizing

E.g., compute the average movie length by year

```
select name, avg(length)

from movie

group by year
```

Two-table operations

So far we have looked at data operations defined over single tables and data frames.

In this section we look at efficient methods to combine data from multiple tables.

The fundamental operation here is the `join`, which is a workhorse of database system design and impementation.

Two-table operations

The `join` operation:

Combines rows from two tables to create a new single table

Based on matching criteria specified over attributes of each of the two tables.

Two-table operations

Consider a database of `flights` and `airlines`:

```
flights
```

```
## # A tibble: 336,776 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_time
##     <int> <int> <int>    <int>          <int>     <dbl>    <int>
##   1  2013     1     1      517            515         2      830
##   2  2013     1     1      533            529         4      850
##   3  2013     1     1      542            540         2      923
##   4  2013     1     1      544            545        -1     1004
##   5  2013     1     1      554            600        -6      812
```

Two-table operations

```
airlines
```

```
## # A tibble: 16 x 2
##    carrier name
##    <chr>   <chr>
##  1 9E      Endeavor Air Inc.
##  2 AA      American Airlines Inc.
##  3 AS      Alaska Airlines Inc.
##  4 B6      JetBlue Airways
##  5 DL      Delta Air Lines Inc.
##  6 EV      ExpressJet Airlines Inc.
##  7 F9      Frontier Airlines Inc.
```

Two-table operations

Here, we want to add airline information to each flight.

Join the attributes of the respective airline from the `airlines` table with the `flights` table based on the values of attributes `flights$carrier` and `airlines$carrier`.

Two-table operations

Every row of `flights` with a specific value for `flights$carrier`, is joined with the the corresponding row in `airlines` with the same value for `airlines$carrier`.

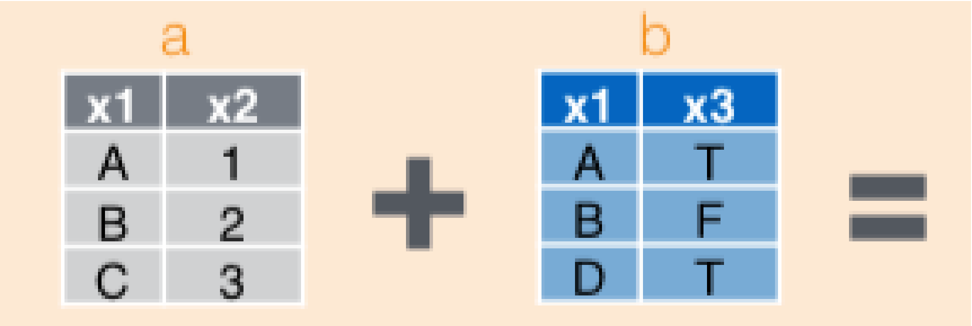
Two-table operations

There are multiple ways of performing this operation that differ on how non-matching observations are handled.

Two-table operations

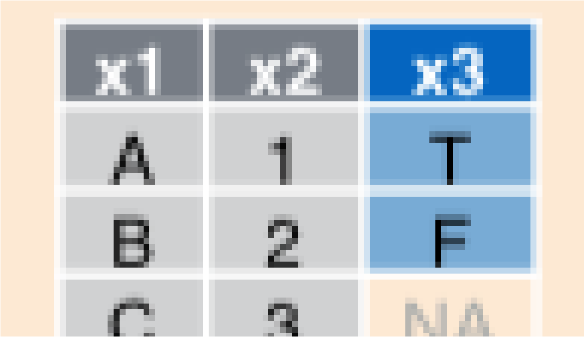
Left Join

In a `left join`, all observations on left operand (LHS) are retained:



x1	x2
A	1
B	2
C	3

x1	x3
A	T
B	F
D	T



x1	x2	x3
A	1	T
B	2	F
C	3	NA

Two-table operations

Other operations:

- *right join*: all observations in RHS are retained
- *outer join*: all observations are retained (full join)
- *inner join*: only matching observations are retained

Details in lecture notes

Two-table operations

Join conditions

All join operations are based on a matching condition:

```
flights %>%
  inner_join(airlines, by="carrier")
```

specifies to join observations where `flights$carrier` equals `airlines$carrier`.

Two-table operations

In this case, where no conditions are specified using the `by` argument:

```
flights %>%
  left_join(airlines)
```

a *natural join* is perfomed. In this case all variables with the same name in both tables are used in join condition.

Two-table operations

You can also specify join conditions on arbitrary attributes using the `by` argument.

```
flights %>%
  left_join(airlines, by=c("carrier" = "name"))
```

Two-table operations

SQL Constructs: Multi-table Queries

Key idea:

- Do a join to combine multiple tables into an appropriate table
- Use **SFW** constructs for single-table queries

Two-table operations

SQL Constructs: Multi-table Queries

Key idea:

- Do a join to combine multiple tables into an appropriate table
- Use **SFW** constructs for single-table queries

For the first part, where we use a join to get an appropriate table, the general SQL construct includes:

- The name of the first table to join
- The *type* of join to do
- The name of the second table to join

Two-table operations

```sql
select title, year, me.name as producerName

from movies m join movieexec me

where m.producer = me.id;
```

Entity Resolution and Record Linkage

Often, we will be faced with the problem of *data integration*:

- combine two (or more) datasets from different sources
- that may contain information about the same *entities*.

Entity Resolution and Record Linkage

Often, we will be faced with the problem of *data integration*:

- combine two (or more) datasets from different sources
- that may contain information about the same *entities*.

But,... the *attributes* in the two datasets may not be the same,

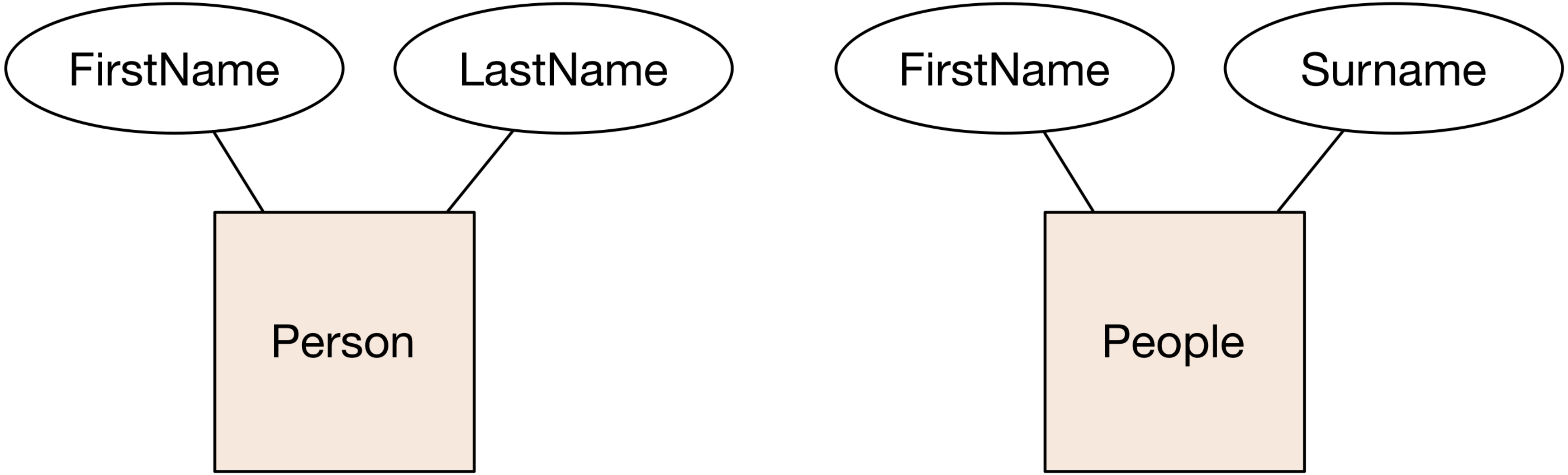
Entity Resolution and Record Linkage

Often, we will be faced with the problem of *data integration*:

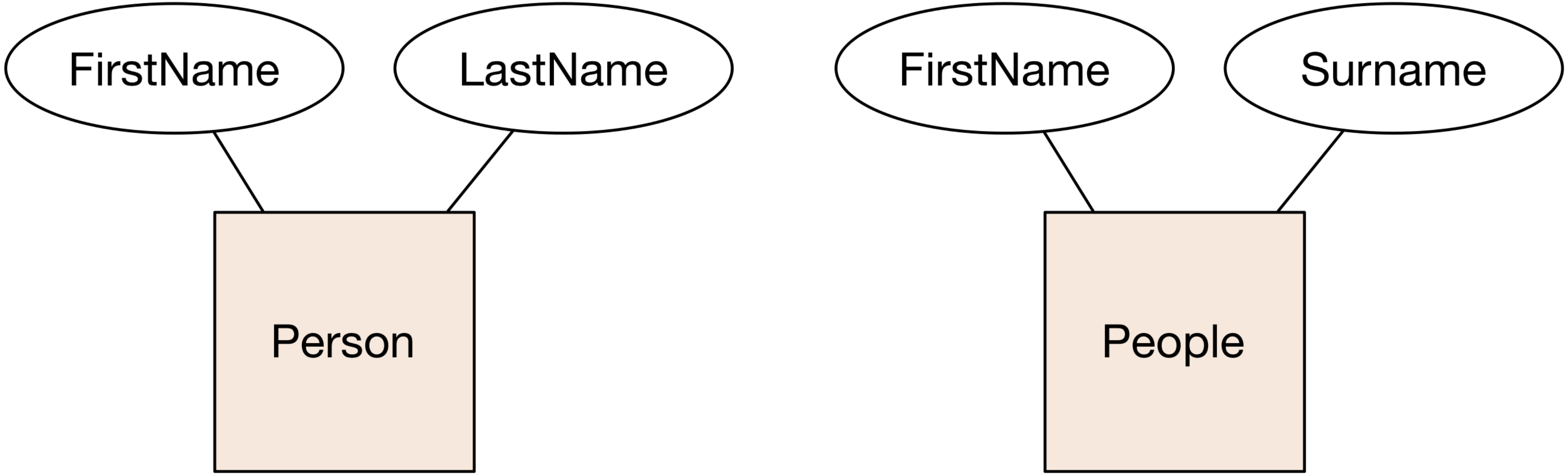
- combine two (or more) datasets from different sources
- that may contain information about the same *entities*.

But,... the *attributes* in the two datasets may not be the same,

Entity Resolution and Record Linkage



Entity Resolution and Record Linkage



<John, Katz>

<Johnathan, Katz>
<Johnathan, Kats>

Entity Resolution and Record Linkage

These are examples of a general problem referred to as **Entity Resolution** and **Record Linkage**.

Entity Resolution and Record Linkage

Problem Definition

Given: Entity sets E_1 and E_2,

Find: Linked entities (e_1, e_2) with $e_1 \in E_1$ and $e_2 \in E_2$.

Entity Resolution and Record Linkage

One approach: similarity function

- Define a *similarity* function between entities e_1 and e_2
- Link entities with high similarity.

Entity Resolution and Record Linkage

Define similarity as an *additive* function over some set of shared attributes A:

$$s(e_1, e_2) = \sum_{j \in A} s_j(e_1[j], e_2[j])$$

with s_j a similarity function defined for *each* attribute j,

Entity Resolution and Record Linkage

Example attribute functions

Categorical attribute: pairs of entities with the same value are more similar to each other than pairs of entities with different values. E.g.,

$$s_j(e_1[j], e_2[j]) = \begin{cases} 1 & \text{if } e_1[j] == e_2[j] \\ 0 & \text{o.w.} \end{cases}$$

Entity Resolution and Record Linkage

Example attribute functions

Continuous attribute: pairs of entities with values that are *close* to each other are more similar than pairs of entities with values that are *farther* to each other.

Note that to specify *close* or *far* we need to introduce some notion of *distance*. We can use Euclidean distance for example,

$$d_j(e_1[j], e_2[j]) = (e_1[j] - e_2[j])^2;\ s_j(e_1[j], e_2[j]) = e^{-d_j(e_1[j], e_2[j])}$$

Entity Resolution and Record Linkage

Example attribute functions

Text attributes: based on *edit distance* between strings rather than Euclidean distance. We can use domain knowledge to specify similarity.

For example, fact that `John` and `Johnathan` are similar requires domain knowledge of common usage of English names.

Solving the resolution problem

Need a rule to match entities we think are linked.

This depends on assumptions we make about the dataset, similar to assumptions we made when performing joins.

Solving the resolution problem

Model the entity resolution problem as an *optimization* problem:

maximize *objective function* (based on similarity)

over possible sets V of *valid* pairs (e_1, e_2), where set V constraints pairs based on problem-specific assumptions.

$$R = \arg\max_V \sum_{(e_1,e_2)\in V} s(e_1, e_2)$$

Solving the resolution problem

Many-to-one resolutions

Constrain sets V to represent many-to-one resolutions.

Thus, entities in e_1 can only appear once in pairs in V, but entities e_2 may appear more than once.

In this case, we can match (e_1, e_2) where

$$e_2 = \arg\max_{e \in E_2} s(e_1, e)$$

Solving the resolution problem

One-to-one resolutions

Suppose we constrain sets V to those that represent one-to-one resolutions:

If $(e_1, e_2) \in V$ then e_1 and e_2 appear in only one pair in V.

In this case, we have a harder computational problem. In fact, this is an instance of the *maximum bipartite matching problem*, and would look at network flow algorithms to solve.

Solving the resolution problem

Other constraints

We can add additional constraints to V to represent other information we have about the task.

A common one would be to only allow pairs $(e_1, e_2) \in V$ to have similarity above some threshold t. I.e., $(e_1, e_2) \in V$ only if $s(e_1, e_2) \geq t$.

Solving the resolution problem

Discussion

The procedure outlined above is an excellent first attempt to solve the Entity Resolution problem.

This is a classical problem in Data Science for which a variety of approaches and methods are in use.

Database Query Optimization

Earlier we made the distinction that SQL is a *declarative* language rather than a *procedural* language.

A reason why data base systems rely on a declarative language is that it allows the system to decide how to *evaluate* a query *most efficiently*.

Database Query Optimization

Consider a Baseball database where we have two tables `Batting` and `Master`

what is the maximum batting "average" for a player from the state of California?

```
select max(1.0 * b.H / b.AB) as best_ba
from Batting as b join Master as m on b.playerId = m.playerId
where b.AB >= 100 and m.birthState = "CA"
```

Database Query Optimization

Now, let's do the same computation using `dplyr` operations:

Version 1:

```
Batting %>%
  inner_join(Master, by="playerID") %>%
  filter(AB >= 100, birthState == "CA") %>%
  mutate(AB=1.0 * H / AB) %>%
  summarize(max(AB))
```

```
##     max(AB)
## 1 0.4057018
```

Database Query Optimization

Version 2:

```
Batting %>%
  filter(AB >= 100) %>%
  inner_join(
    filter(Master, birthState == "CA")) %>%
  mutate(AB = 1.0 * H / AB) %>%
  summarize(max(AB))
```

```
##     max(AB)
## 1 0.4057018
```

Database Query Optimization

Which should be most efficient? Think about a simple *cost* model. The costliest operation here is the join between two tables.

function $\textsc{InnerJoin}(T1, T2, A)$
    $R \leftarrow \emptyset$
    for all row $r1 \in T1$ **do**
        for all row $r2 \in T2$ **do**
            if $r1[A] == r2[A]$ **then** $R \leftarrow (r1, r2) \bigcup R$
            end if
        end for
    end for
    return R
end function

Database Query Optimization

What is the cost of this algorithm? $|T1| \times |T2|$.

For the rest of the operations, let's assume we perform this with a single pass through the table.

For example, we assume that `filter(T)` has cost $|T|$.

Database Query Optimization

Let's write out the cost of each of the two pipelines.

```
Batting %>%
  inner_join(Master, by="playerID") %>% # cost: |Batting| x |Master|
  filter(AB >= 100, birthState == "CA") %>% # cost: |R1|
  mutate(AB=1.0 * H / AB) %>% # cost: |R|
  summarize(max(AB)) # cost: |R|
```

Database Query Optimization

Cost of version 1 is

$$|\text{Batting}| \times |\text{Master}| + |R1| + 2|R|$$

$R1$: inner join between `Batting` and `Master` R: is $R1$ filtered to rows with `AB >=100 & birthState == "CA"`.

In this example: 2.08e+09

Database Query Optimization

Now, let's look at the second version.

```
Batting %>%
  filter(AB >= 100) %>% # cost: |Batting|
  inner_join(
    Master %>% filter(birthState == "CA") # cost: |Master|
  ) %>% # cost: |B1| x |M1|
  mutate(AB = 1.0 * H / AB) %>% # cost |R|
  summarize(max(AB)) # cost |R|
```

Database Query Optimization

Cost of version 2 is $|\text{Batting}| \times |\text{Master}| + |B1| \times |M1| + 2|R|$

$B1$: `Batting` filtered to include only rows with `AB >= 100` $M2$: `Master` filtered to include `birthState == "CA"`.

In our example: 8.95e+07

Database Query Optimization

Version 1 (join tables before filtering) is 23 times costlier.

When using SQL in a database system we only write the one query describing our desired result,

With the *procedural* (`dplyr`) we need to think which of the two versions is more efficient.

Database Query Optimization

Database systems use *query optimization* to decide how to evaluate queries efficiently.

The goal of query optimization is to decide the most efficient query *plan* to use to evaluate a query out of the many possible candidate plans it could use.

It needs to solve two problems: search the space of possible plans, approximate the *cost* of evaluating a specific plan.

Database Query Optimization

Think of the two procedural versions above as two candidate plans that the DB system *could* use to evaluate the query.

Query optimzation *approximates* what it would cost to evaluate each of the two plans and decides to use the most efficient plan.

Semi-structured Data Representation Model

The Entity-Relational data model we have described so far is mostly defined for *structured data*: where a specific and consistent schema is assumed.

Data models like XML and JSON are instead intended for *semi-structured* data.

Semi-structured Data Representation Model

XML: eXtensible Markup Language

Data models like XML rely on flexible, self-describing schemas:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Edited by XMLSpy -->
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
```

Semi-structured Data Representation Model

JSON: Javascript Object Notation

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
```

Semi-structured Data Representation Model

This is the format most contemporary data REST APIs use to transfer data. For instance, here is part of a JSON record from a Twitter stream:


```
{
  "created_at":"Sun May 05 14:01:34+00002013",
  "id":331046012875583488,
  "id_str":"331046012875583488",
  "text":"\u0425\u043e\u0447\u0443, \u0447\u0442\u043e\u0431 \u0442\u044b \u0441\u0434\u0434\u0
  "source":"\u003ca href=\"http:\/\/twitterfeed.com\"rel=\"nofollow\"\u003etwitterfeed\u0
  "in_reply_to_user_id_str":null,
  "user":{
    "id":548422428,
```

Summary

We have looked at specifics of **Data Representation Modeling**

- Entity Relationship and Relational Models
- Definition of *Tidy Data*
- Joining tables
- Entity Resolution
- Models for semi-structured data