

# **15-388/688 - Practical Data Science: Relational Data**

J. Zico Kolter  
Carnegie Mellon University  
Fall 2019

# Outline

Overview of relational data

Entity relationships

Pandas and SQLite

Joins

# Outline

Overview of relational data

Entity relationships

Pandas and SQLite

Joins

# The basic relation (i.e. the table)

The term technical term “relation” can be interchanged with the standard notion we have of “tabular data,” say an instance of a “Person” relation

ID	Last Name	First Name	Role
1	Kolter	Zico	Instructor
2	Manek	Gaurav	TA
3	Rice	Leslie	TA
4	Peres	Filipe	TA
5	Gates	William	Student
6	Musk	Elon	Student

# The basic relation (i.e. the table)

The term technical term “relation” can be interchanged with the standard notion we have of “tabular data,” say an instance of a “Person” relation

ID	Last Name	First Name	Role
1	Kolter	Zico	Instructor
2	Manek	Gaurav	TA
3	Rice	Leslie	TA
4	Peres	Filipe	TA
5	Gates	William	Student
6	Musk	Elon	Student

Rows are called *tuples (or records)*, represent a single instance of this relation, and *must be unique*

# The basic relation (i.e. the table)

The term technical term “relation” can be interchanged with the standard notion we have of “tabular data,” say an instance of a “Person” relation

ID	Last Name	First Name	Role
1	Kolter	Zico	Instructor
2	Manek	Gaurav	TA
3	Rice	Leslie	TA
4	Peres	Filipe	TA
5	Gates	William	Student
6	Musk	Elon	Student

Columns are called *attributes*, specify some element contained by each of the tuples

# Multiple tables and relations

**Person**

ID	Last Name	First Name	Role ID
1	Kolter	Zico	1
2	Manek	Gaurav	2
3	Rice	Leslie	2
4	Peres	Filipe	2
5	Gates	William	3
6	Musk	Elon	3

**Role**

ID	Name
1	Instructor
2	TA
3	Student

# Primary keys

**Person**

ID	Last Name	First Name	Role ID
1	Kolter	Zico	1
2	Manek	Gaurav	2
3	Rice	Leslie	2
4	Peres	Filipe	2
5	Gates	William	3
6	Musk	Elon	3

**Role**

ID	Name
1	Instructor
2	TA
3	Student

**Primary key:** *unique* ID for every tuple in a relation (i.e. every row in the table), each relation must have exactly one primary key



# Foreign keys

**Person**

ID	Last Name	First Name	Role ID
1	Kolter	Zico	1
2	Manek	Gaurav	2
3	Rice	Leslie	2
4	Peres	Filipe	2
5	Gates	William	3
6	Musk	Elon	3

**Role**

ID	Name
1	Instructor
2	TA
3	Student

A **foreign key** is an attribute that points to the primary key of *another* relation

If you delete a primary key, need to delete all foreign keys pointing to it

# Indexes (not indices)

Indexes are created as ways to “quickly” access elements of a table

For example, consider finding people with last name “Gates”: no option but just scan through the whole dataset:  $O(n)$  operation

ID	Last Name	First Name	Role ID
1	Kolter	Zico	1
2	Manek	Gaurav	2
3	Rice	Leslie	2
4	Peres	Filipe	2
5	Gates	William	3
6	Musk	Elon	3

# Indexes

**Person**

Location	ID	Last Name	First Name	Role ID
0	1	Kolter	Zico	1
100	2	Manek	Gaurav	2
200	3	Rice	Leslie	2
300	4	Peres	Filipe	2
400	5	Gates	William	3
500	6	Musk	Elon	3

**Last Name Index**

Last Name	Location
Gates	400
Kolter	0
Manek	100
Musk	500
Peres	300
Rice	200

Think of an index as a separate *sorted* table containing the indexed column and the tuple location: searching for value takes  $O(\log n)$  time

In practice, use data structure like a B-tree or several others

# Indexes

The primary key always has an index associated with it (so you can think of primary keys themselves as always being a fast way to access data)

Indexes don't have to be on a single column, can have an index over multiple columns (with some ordering)

# Outline

Overview of relational data

Entity relationships

Pandas and SQLite

Joins

# Entity relationships

Several types of inter-table relationships

1. One-to-one
2. (One-to-zero/one)
3. One-to-many (and many-to-one)
4. Many-to-many

These relate one (or more) rows in a table with one (or more) rows in another table, via a foreign key

Note that these relationships are really between the “entities” that the tables represent, but we won’t formalize this beyond the basic intuition

# One-to-many relationship

We have already seen a one-to-many relationship: one **role** can be shared by many **people**, denoted as follows



**Person**

ID	Last Name	First Name	Role ID
1	Kolter	Zico	1
2	Manek	Gaurav	2
3	Rice	Leslie	2
4	Peres	Filipe	2
5	Gates	William	3
6	Musk	Elon	3

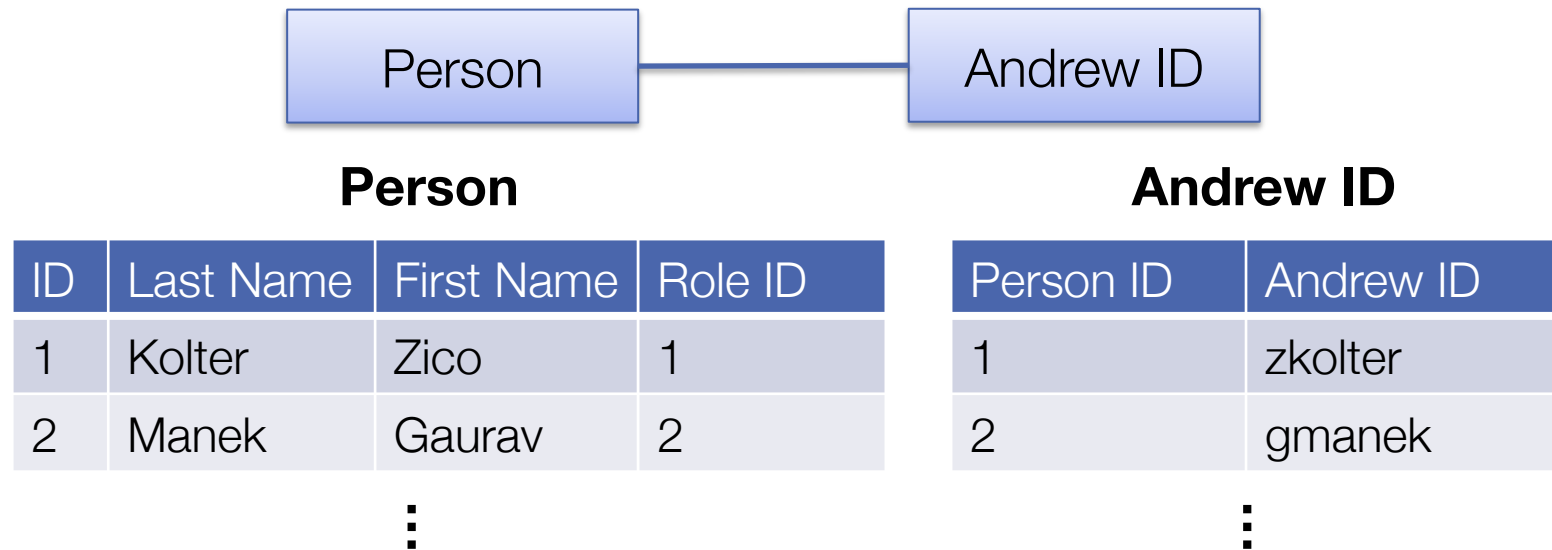
**Role**

ID	Name
1	Instructor
2	TA
3	Student

# One-to-one relationships

In a true one-to-one relationship spanning multiple tables, each row in a table has *exactly* one row in another table

Not very common to break these across multiple tables, as you may as well just add another attribute to an existing table, but it is possible

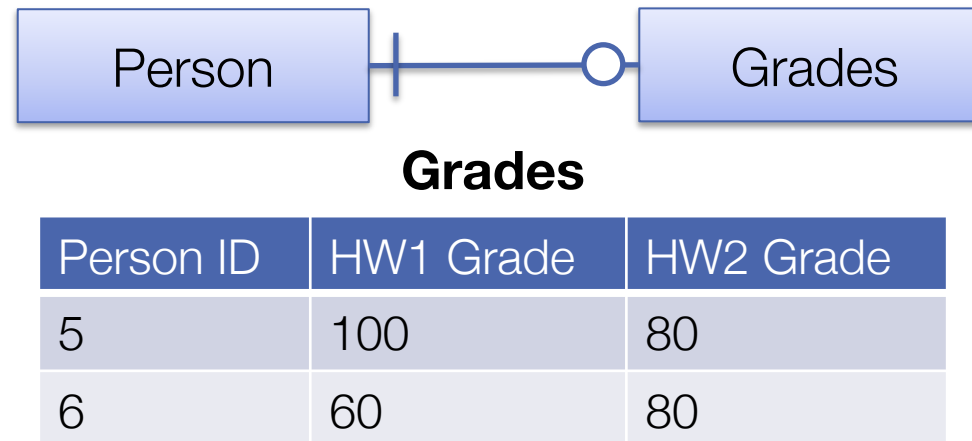




# One-to-zero/one relationships

More common in databases is to find “one-to-zero/one” relationships broken across multiple tables

Consider adding a “Grades” table to our database: each person can have at most one tuple in the grades table



Bars and circles denote “mandatory” versus “option” relationships (we won’t worry about these, just know that there is notation for them)

# Many-to-many relationships

Creating a grades table as done before is a bit cumbersome, because we need to keep adding columns to the table, null entries if someone doesn't do the homework

Alternatively, consider adding two tables, a “homework” table that represents information about each homework, and an associative table that links homeworks to people

**Homework**

ID	Name	388 Points	688 Points
1	HW 1	65	35
2	HW 2	75	25

**Person Homework**

Person ID	HW ID	Score
5	1	100
5	2	80
6	1	60
6	2	80

# Associative tables

**Person Homework**

Person ID	HW ID	Score
5	1	100
5	2	80
6	1	60
6	2	80

What is the primary key of this table? What are foreign keys?

Which indexes would you want to create on this table?

# Poll: Primary key of associative table

What should the primary key be for this table?

**Person Homework**

Person ID	HW ID	Score
5	1	100
5	2	80
6	1	60
6	2	80

1. Person ID
2. HW ID
3. (Person ID, HW ID)
4. (Person ID, HW ID, Score)

# Many-to-many relationships

Setups like this encode many-to-many relationships: each person can have multiple homeworks, and each homework can be done by multiple people



We could also write this in terms of relationships specified by the associative table, but this is not really correct, as it is mixing up the underlying relationships with how they are stored in a database



# Outline

Overview of relational data

Entity relationships

Pandas and SQLite

Joins

# Pandas

Pandas is a “Data Frame” library in Python, meant for manipulating in-memory data with row and column labels (as opposed to, e.g., matrices, that have no row or column labels)

Pandas is *not* a relational database system, but it contains functions that mirror some functionality of relational databases

We’re going to cover Pandas in more detail in other portions of the class, but just discuss basic functionality for now

# Pandas examples

Create a DataFrame with our Person example:

```
import pandas as pd

df = pd.DataFrame([(1, 'Kolter', 'Zico'),
                   (2, 'Manek', 'Gaurav'),
                   (3, 'Rice', 'Leslie'),
                   (4, 'Peres', 'Filipe'),
                   (5, 'Gates', 'Bill'),
                   (6, 'Musk', 'Elon')],
                  columns=["Person ID", "Last Name", "First Name"])
df.set_index("Person ID", inplace=True)
```

In [4]:

1

df

Out[4]:

Person ID	Last Name	First Name
1	Kolter	Zico
2	Manek	Gaurav
3	Rice	Leslie
4	Peres	Filipe
5	Gates	Bill
6	Musk	Elon



# Some important notes

As mentioned, Pandas is *not* a relational data system, in particular it has no notion of primary keys (but it does have indexes)

Operations in Pandas are typically *not* in place (that is, they return a new modified DataFrame, rather than modifying an existing one)

Use the “inplace” flag to make them done in place

If you select a single row or column in a Pandas DataFrame, this will return a “Series” object, which is like a one-dimensional DataFrame (it has only an index and corresponding values, not multiple columns)

# Some common Pandas commands

```
# read CSV file into DataFrame
df = pd.read_csv(filename)

# get first five rows of DataFrame
df.head()

# index into a dataframe
# df.loc[rows, columns] and df.iloc[row numbers, column numbers]
df.loc[:, "Last Name"]           # Series of all last names
df.loc[:, ["Last Name"]]         # DataFrame with one column
df.loc[[1,2], :]                 # DataFrame with only ids 1,2
df.loc[1, "Last Name"] = "Kilter" # Set an entry in a DataFrame
df.loc[7, :] = ("Moore", "Andrew") # Add a new entry with index=7
df.iloc[0,0]                     # Index rows and columns by zero-index
```

We're going to cover more next lecture in conjunction with visualization

# SQLite

An actual relational database management system (RDBMS)

Unlike most systems, it is a *serverless* model, applications directly connect to a file

Allows for simultaneous connections from many applications to the same database file (but not quite as much concurrency as client-server systems)

All operations in SQLite will use SQL (Structured Query Language) command issued to the database object

You can enforce foreign keys in SQLite, but we won't bother

# Creating a database and table

You can create a database and connect using this boilerplate code:

```
import sqlite3
conn = sqlite3.connect("people.db")
cursor = conn.cursor()

# do your stuff

conn.close()
```

Create a new table:

```
cursor.execute("""
CREATE TABLE role (
    id INTEGER PRIMARY KEY,
    name TEXT
)""")
```

# Creating a new table and inserting data

Insert data into the table:

```
cursor.execute("INSERT INTO role VALUES (1, 'Instructor')")
cursor.execute("INSERT INTO role VALUES (2, 'TA')")
cursor.execute("INSERT INTO role VALUES (3, 'Student')")
conn.commit()
```

Delete items from a table:

```
cursor.execute("DELETE FROM role WHERE id == 3")
conn.commit()
```

Note: if you don't call commit, you can undo with `conn.rollback()`

# Querying all data from a table

Read all the rows from a table:

```
for row in cursor.execute('SELECT * FROM role'):  
    print row
```

Read table directly into a Pandas DataFrame:

```
pd.read_sql_query("SELECT * FROM role", conn, index_col="id")
```

name	
id	
1	Instructor
2	TA
3	Student

# Outline

Overview of relational data

Entity relationships

Pandas and SQLite

**Joins**

# Joins

Join operations merge multiple tables into a single relation (can be then saved as a new table or just directly used)

Four typical types of joins:

1. Inner
2. Left
3. Right
4. Outer

You join two tables *on* columns from each table, where these columns specify which rows are kept



# Example: joining Person and Grades

Consider joining two tables, Person and Grades, on ID / Person ID

**Person**

ID	Last Name	First Name	Role ID
1	Kolter	Zico	1
2	Manek	Gaurav	2
3	Rice	Leslie	2
4	Peres	Filipe	2
5	Gates	William	3
6	Musk	Elon	3

**Grades**

Person ID	HW1 Grade	HW2 Grade
5	100	80
6	60	80
100	100	100

# Inner join (usually what you want)

Join two tables where we only return the rows where the two joined columns contain the *same value*

ID	Last Name	First Name	Role ID	HW1 Grade	HW2 Grade
5	Gates	William	3	100	80
6	Musk	Elon	3	60	80

Only these two rows have an entry in “Person” *and* an entry in “Grades”

# Inner join in Pandas/SQLite

```
# Pandas way
df_person = pd.read_sql_query("SELECT * FROM person", conn)
df_grades = pd.read_sql_query("SELECT * FROM grades", conn)
df_person.merge(df_grades, how="inner",
                left_on="id", right_on="person_id")

# SQLite way
cursor.execute("SELECT * FROM person, grades WHERE
               person.id == grades.person_id")
```

In Pandas, you can also join on index using `right_index/left_index` parameters

There is also the `join` call in Pandas, which is a bit more limited (always assumes right is joined on index, left not on index)

# Left joins

Keep all rows of the left table, add entries from right table that match the corresponding columns

Example: left join Person and Grades on ID, Person ID

ID	Last Name	First Name	Role ID	HW1 Grade	HW2 Grade
1	Kolter	Zico	1	NULL	NULL
2	Manek	Gaurav	2	NULL	NULL
3	Rice	Leslie	2	NULL	NULL
4	Peres	Filipe	2	NULL	NULL
5	Gates	William	3	100	80
6	Musk	Elon	3	60	80

# Left join in Pandas and SQLite

```
# Pandas way
df_person.merge(df_grades, how="left",
                left_on="id", right_on="person_id")

# SQLite way
cursor.execute("SELECT * FROM person LEFT JOIN grades ON
               person.id == grades.person_id")
```

	id	last_name	first_name	person_id	hw1_grade	hw2_grade
0	1	Kolter	Zico	NaN	NaN	NaN
1	2	Manek	Gaurav	NaN	NaN	NaN
2	3	Rice	Leslie	NaN	NaN	NaN
3	4	Peres	Filipe	NaN	NaN	NaN
4	5	Gates	Bill	5.0	85.0	95.0
5	6	Musk	Elon	6.0	80.0	60.0

# Right join

Like a left join but with the roles of the tables reversed

ID	Last Name	First Name	Role ID	HW1 Grade	HW2 Grade
5	Gates	William	3	100	80
6	Musk	Elon	3	60	80
100	NULL	NULL	NULL	100	100

```
# Pandas way  
df_person.merge(df_grades, how="right",  
                 left_on="id", right_on="person_id")  
  
# Not supported in SQLite
```

# Outer join

Return all rows from both left and right join

ID	Last Name	First Name	Role ID	HW1 Grade	HW2 Grade
1	Kolter	Zico	1	NULL	NULL
2	Manek	Gaurav	2	NULL	NULL
3	Rice	Leslie	2	NULL	NULL
4	Peres	Filipe	2	NULL	NULL
5	Gates	William	3	100	80
6	Musk	Elon	3	60	80
100	NULL	NULL	NULL	100	100

```
# Pandas way  
df_person.merge(df_grades, how="outer",  
                 left_on="id", right_on="person_id")
```