The Tabular Data Model

and SQL

Univ.AI

Outline

- What Grammar of Data do tables follow? How is this grammar implemented in Pandas and SQL?
- Creation and alteration in the grammae of data
- The grammar as related to combining tables

A simple but powerful model of a table

- A collection of tables related to each other through common data values.
- Rows represent attributes of something
- Everything in a column is values of *one* attributes
- A cell is expected to be atomic
- Tables are related to each other if they have columns called keys which represent the same values

Contributors

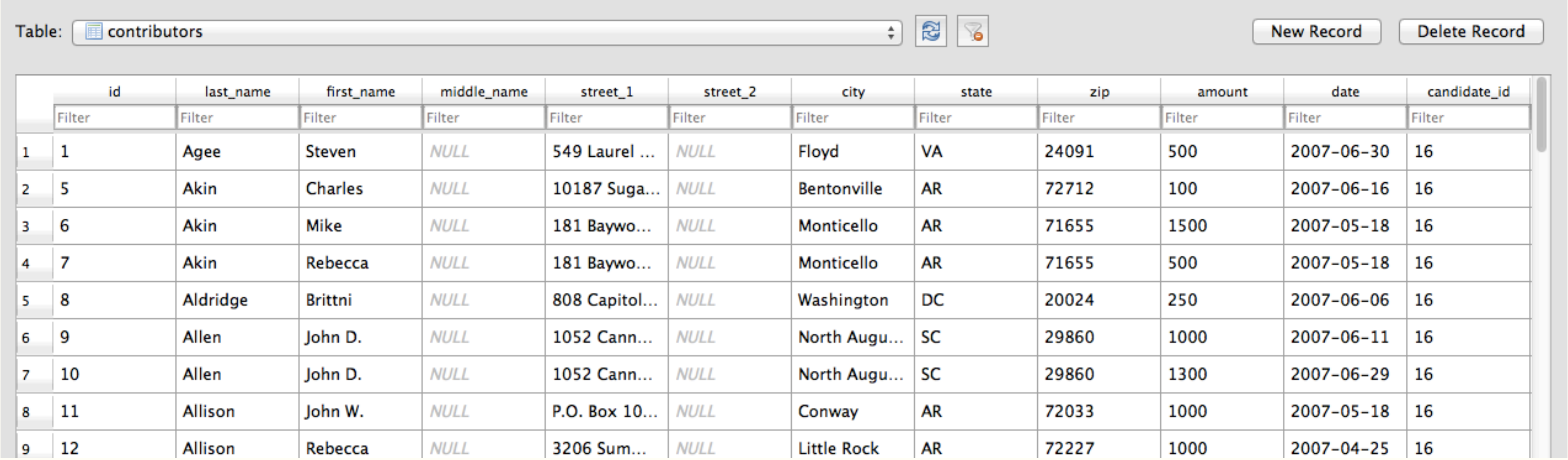


Table: contributors

	id	last_name	first_name	middle_name	street_1	street_2	city	state	zip	amount	date	candidate_id
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Agee	Steven	NULL	549 Laurel ...	NULL	Floyd	VA	24091	500	2007-06-30	16
2	5	Akin	Charles	NULL	10187 Suga...	NULL	Bentonville	AR	72712	100	2007-06-16	16
3	6	Akin	Mike	NULL	181 Baywo...	NULL	Monticello	AR	71655	1500	2007-05-18	16
4	7	Akin	Rebecca	NULL	181 Baywo...	NULL	Monticello	AR	71655	500	2007-05-18	16
5	8	Aldridge	Brittni	NULL	808 Capitol...	NULL	Washington	DC	20024	250	2007-06-06	16
6	9	Allen	John D.	NULL	1052 Cann...	NULL	North Augu...	SC	29860	1000	2007-06-11	16
7	10	Allen	John D.	NULL	1052 Cann...	NULL	North Augu...	SC	29860	1300	2007-06-29	16
8	11	Allison	John W.	NULL	P.O. Box 10...	NULL	Conway	AR	72033	1000	2007-05-18	16
9	12	Allison	Rebecca	NULL	3206 Sum...	NULL	Little Rock	AR	72227	1000	2007-04-25	16

Typing our tables

```sql
DROP TABLE IF EXISTS "candidates";
DROP TABLE IF EXISTS "contributors";

CREATE TABLE "candidates" (
    "id" INTEGER PRIMARY KEY  NOT NULL ,
    "first_name" VARCHAR,
    "last_name" VARCHAR,
    "middle_name" VARCHAR,
    "party" VARCHAR NOT NULL
);
```

```sql
CREATE TABLE "contributors" (
    "id" INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL,
    "last_name" VARCHAR,
    "first_name" VARCHAR,
    "middle_name" VARCHAR,
    "street_1" VARCHAR,
    "street_2" VARCHAR,
    "city" VARCHAR,
    "state" VARCHAR,
    "zip" VARCHAR,
    "amount" INTEGER,
    "date" DATETIME,
    "candidate_id" INTEGER NOT NULL,
    FOREIGN KEY(candidate_id) REFERENCES candidates(id)
);
```

Grammar of Data

Formalized by Hadley Wickham in `dplyr`[1].

1. provide simple verbs for simple things. These are functions corresponding to common data manipulation tasks

2. the backend does not matter. Here we constrain ourselves to Pandas and SQL in sqlite

3. multiple backends implemented in Pandas, Spark, Impala, Pig, dplyr, ibis, blaze

[1] Hadley Wickham: https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html

Why bother with SQL and the grammar

- learn how to do core data manipulations, no matter what the system
- relational databases critical for tables that dont fit in memory.
- SQL is declarative, the query optimizer decides how to make the query.
- dbs like Cloudera, Azure Data Lake implement the SQL lingua-franca or similar over clusters, multiple databases

You can always find out what these core verbs are in any system!

VERB	**dplyr**	**pandas**	**SQL**
QUERY/SELECTION	`filter()` (and `slice()`)	`query()` (and `loc[]`, `iloc[]`)	`SELECT WHERE`
SORT	`arrange()`	`sort_values()`	`ORDER BY`
SELECT-COLUMNS/ PROJECTION	`select()` (and `rename()`)	`[]`(__getitem__) (and `rename()`)	`SELECT` COLUMN
SELECT-DISTINCT	`distinct()`	`unique()`,`drop_duplicates()`	`SELECT DISTINCT` COLUMN
ASSIGN	`mutate()` (and `transmute()`)	`assign`	`ALTER/UPDATE`
AGGREGATE	`summarise()`	`describe()`, `mean()`, `max()`	None, `AVG()`,`MAX()`
SAMPLE	`sample_n()` and `sample_frac()`	`sample()`	implementation dep, use `RAND()`
GROUP-AGG	`group_by/summarize`	`groupby/agg`, `count`, `mean`	`GROUP BY`
DELETE	?	`drop`/masking	`DELETE/WHERE`

SQLITE

Sqlite is a on-file database, as opposed to other common databases such as Oracle and Postgres, which run as different processes on your system. Sqlite is great for on-disk large databases which wont fit into memory.

Its also built into Python, but to use the command line tool, I recommend you install it: https://www.sqlite.org/download.html. I also recommend you download and install the sqlite browser: http://sqlitebrowser.org .

Python implements a standard database API over all databases. Its called DBAPI2. It works across many SQL databases, including Sqlite. There is an even higher level API available, called SQLAlchemy.

Connect to and populate the database

```
def get_db(dbfile):
    # get a connection
    sqlite_db = sq3.connect(dbfile)
    return sqlite_db

def init_db(dbfile, schema):
    """Creates the database tables."""
    db = get_db(dbfile)
    # execute some SQL code
    c = db.cursor()
    c.executescript(schema)
    # make a commit
    db.commit()
    return db

db=init_db("/tmp/cancont.db", schema)
# populate the database
dfcand.to_sql("candidates", db,
    if_exists="append", index=False)
dfcwci.to_sql("contributors", db,
    if_exists="append", index=False)
```

Using the sqlite dbapi2:

```
# get a pointer to rows of results
c=db.cursor()
# you can also use db directly
c.execute("SELECT * FROM candidates;")
# you can fetch all, or some at a time
# here we fetch all
c.fetchall()
```

```
[(16, 'Mike', 'Huckabee', None, 'R'),
 (20, 'Barack', 'Obama', None, 'D'),
 (22, 'Rudolph', 'Giuliani', None, 'R'),
 (24, 'Mike', 'Gravel', None, 'D'),
 (26, 'John', 'Edwards', None, 'D'),
 (29, 'Bill', 'Richardson', None, 'D'),
 ....
 (41, 'Fred', 'Thompson', 'D.', 'R')]
```

Operation	Pandas	SQL
QUERY	dfcwci.query("state=='VA' & amount < 400")	SELECT * FROM contributors WHERE state='VA' AND amount < 400;
SORT	dfcwci.sort_values("amount")	SELECT * FROM contributors ORDER BY amount;
SELECT-COLUMNS	dfcwci[['first_name', 'amount']]	SELECT first_name, amount FROM contributors;
SELECT-DISTINCT	dfcwci[['last_name','first_name']].drop_duplicates()	SELECT DISTINCT last_name, first_name FROM contributors;

ASSIGN	dfcwci['name']=dfcwci['last_name']+", "+dfcwci['first_name']	ALTER TABLE contributors ADD COLUMN name;
		UPDATE contributors SET name = ? WHERE id = ?;

AGGREGATE	dfcwci.amount.max()	SELECT MAX(amount) FROM contributors;
GROUP-AGG	dfcwci.groupby("state").sum()	SELECT state,SUM(amount) FROM contributors GROUP BY state;
DELETE	dfcwci=dfcwci[dfcwci.last_name!='Ahrens']	DELETE FROM contributors WHERE last_name="Ahrens";
DELETE TABLE	del dfcwci	DELETE FROM contributors;
CREATE TABLE	dfcwci=pd.read_csv("data/contributors_with_candidate_id.txt")	INSERT INTO candidates (id, first_name, last_name, middle_name, party) VALUES (?,?,?,?,?);

Some examples of the verbs

```
# combining multiple selects
dfcwci[(dfcwci.state=='VA') & (dfcwci.amount < 400)]
SELECT * FROM contributors WHERE state='VA' AND amount < 400;
# array membership
dfcwci[dfcwci.state.isin(['VA','WA'])]
SELECT * FROM contributors WHERE state IN ('VA','WA');
# project onto specific columns
dfcwci[['first_name', 'amount']]
SELECT first_name, amount FROM contributors;
# aggregation
dfcwci.amount.max()
SELECT MAX(amount) FROM contributors;
```

The Split-Apply-Combine (GROUP-AGG) pattern

1. split the data into groups
2. based on some criteria apply a function to each group independently
3. combinine the results into a data structure

```
SELECT state, MEAN(amount)
    FROM contributors GROUP BY state;
```

```
dfcwci.groupby("state")['amount'].mean()
```

```
state
AK       403.333333
AR      1183.333333
AZ       120.000000
CA      -217.988261
CO     -1455.750000
CT      2300.000000
DC      -309.982000
...
```

Creation and Alteration

Inserting data into SQLITE

```
sql_template = """
INSERT INTO candidates (id, first_name, last_name, middle_name, party) \
    VALUES (?,?,?,?,?);
"""

with open("data/candidates.txt") as fd:
    slines =[l.strip().split('|') for l in fd.readlines()]
    for line in slines[1:]:
        theid, first_name, last_name, middle_name, party = line
        valstoinsert = (int(theid), first_name, last_name, middle_name, party)
        db.cursor().execute(sql_template, valstoinsert)

db.commit()
```

What is this `commit()`?

Things can go wrong. Software fails. Hardware fails. Netorks fail. Multiple clients may update tables.

You dont want a bank to only do a withdrawal but not a deposit on a money transfer! You want a **transaction**.

A transaction groups reads and writes together into ONE operation. Either all happen and succeed, or none do. The database will **rollback** any partial changes. `db.commit()` signals a transaction.

The acronym ACID was coined in 1983 by Theo Härder and Andreas Reuter to describe transactions:

- **A** : Atomic, the above all or none guarantee
- **C** : Consistency means "good state". In bank transfer, a half done tranfer is not consistent. This definition is application specific.
- **I** : Isolation means that concurrently executing transactions dont interfere with each other: one happens first, and the other only sees things after the first ransaction executed
- **D** : Durability : after commit, data wont be lost.

Altering tables by adding columns

```
dfcwci['name']=dfcwci['last_name']+", "+dfcwci['first_name'] # pandas

alt="ALTER TABLE contributors ADD COLUMN name;" # SQL
db.cursor().execute(alt)
db.commit()

alt2="UPDATE contributors SET name = ? WHERE id = ?;" # SQL
for ele in out2:
    db.cursor().execute(alt2, ele)
db.commit()
```

Updating existing columns

```
dfcwci.loc[dfcwci.state=='VA', 'name']="junk" # pandas
```

```
upd="UPDATE contributors SET name = 'junk' WHERE state = 'VA';"
db.cursor().execute(upd) # SQL
db.commit()
```

Combining Tables

RELATIONSHIPS

- we usually need to combine data from multiple sources
- different systems have diffferent ways, but most copy SQL
- Example: sub-select:

```
obamaid=dfcand.query("last_name=='Obama'")['id'].values[0]
dfcwci.query("candidate_id==%i" % obamaid)
```

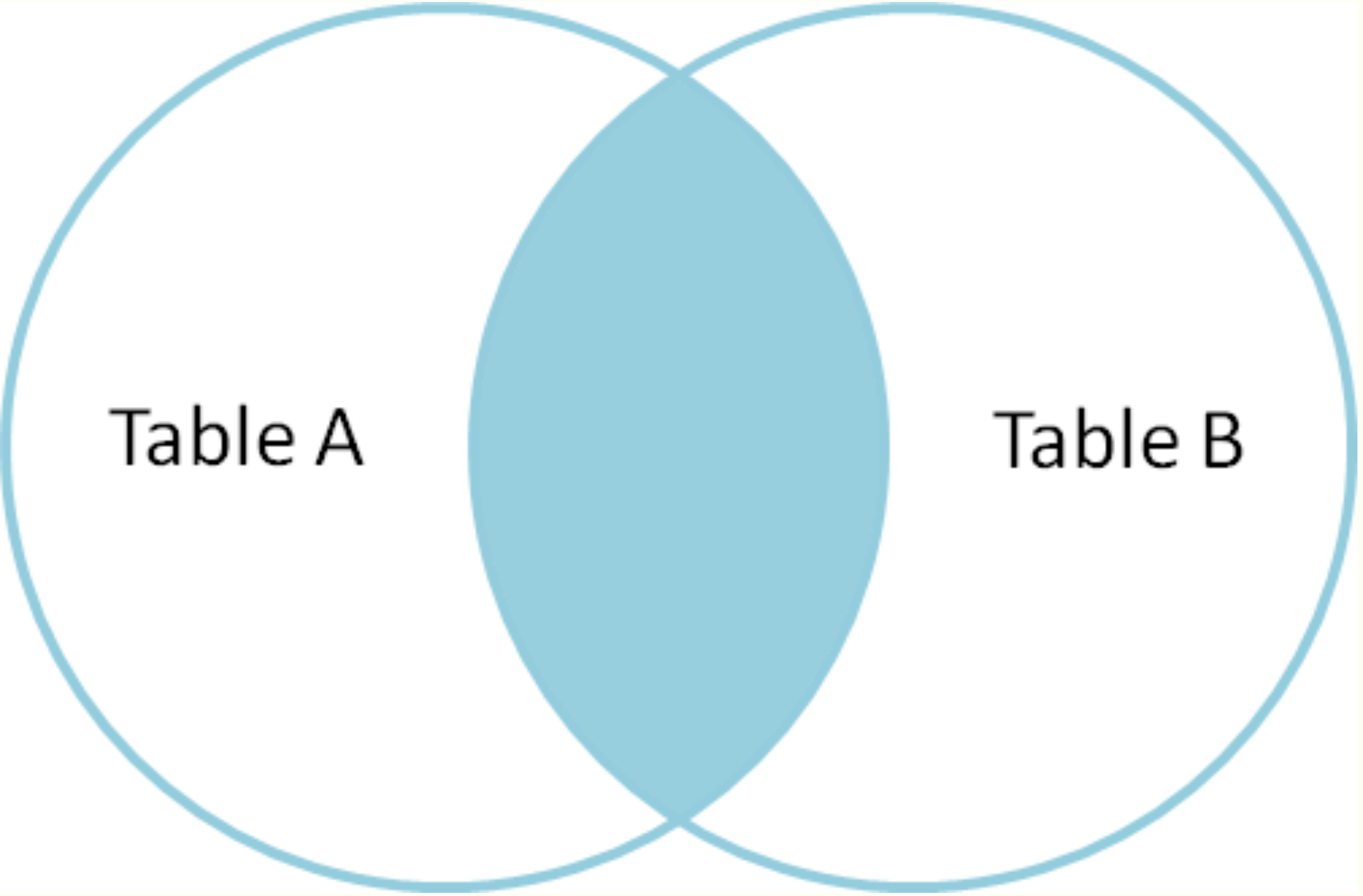
```
SELECT * FROM contributors WHERE
    candidate_id = (SELECT id from candidates WHERE last_name = 'Obama');
```

Two table grammar: JOINS

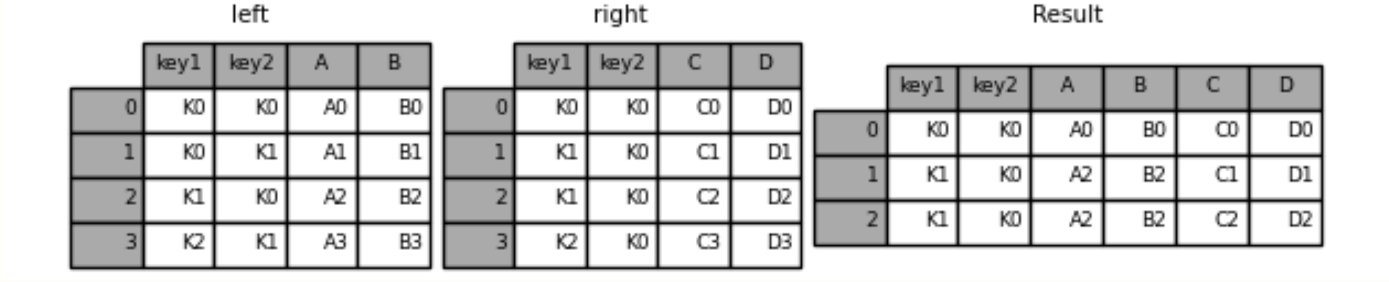
MUTATING JOINS: add new variables to one table from matching rows in another. Types of join are:

1. inner join,
2. left(outer) join
3. right(outer) join, and
4. full(outer) join

INNER JOINS



Combine 2 tables on a common key value. 90% of the time this is an explicit inner join



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

PANDAS:

```
dfcwci.merge(dfcand, left_on="candidate_id", right_on="id")
```

SQL:

```
SELECT * FROM
    contributors JOIN candidates
ON contributors.candidate_id = candidates.id;
```

Why are these joins important?

Denormalized data is very important for data analysis. Why?

Foreign keys carry no semantic meaning and models need to be fed data rather than pointers.

If we have many tables with information about items, we want to join them rather than keep the pointers.

This wide, denormalized form can be fed to models.

Close the database connection

```
db.close()
```