Piazza signup:
http://piazza.com/mit/spring2022/6s079

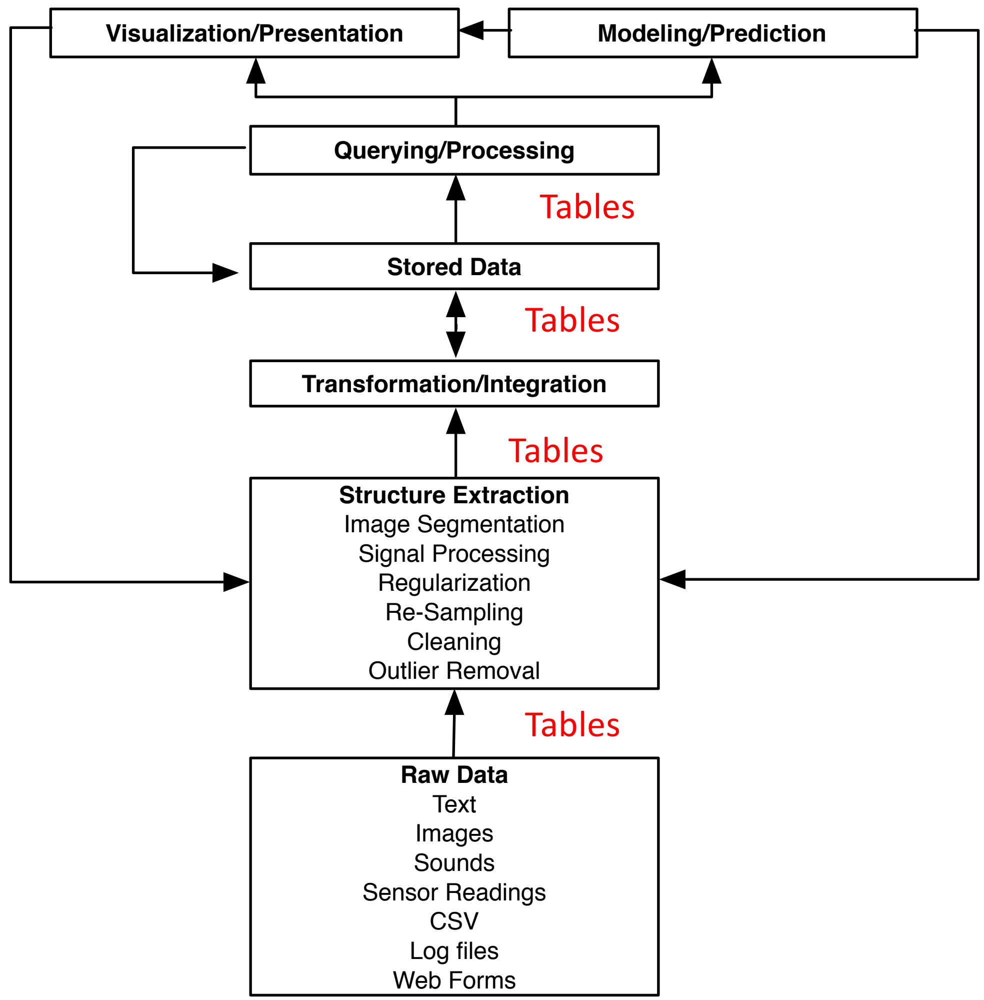
http://dsg.csail.mit.edu/6.S079/

6.S079 Lecture 2

Sam Madden

Key ideas:
Tabular data & relational model
Relational algebra & SQL

Data Science Pipeline



Tables Are Everywhere

- Most data is published in tabular form
- E.g., Excel spreadsheets, CSV files, databases

- Going to spend next few lectures talking about working with tabular data

- Focus on “relational model” used by databases and common programming abstractions like Pandas in Python.

Getting Tables Right is Subtle

- What makes a table or set of tables “good”?

- **Consistent**
  - E.g., values in each column are the same type
- **Compact**
  - Information is not repeated
- **Easy-to-use**
  - In a format that programming tools can ingest
- **Well-documented**
  - E.g., column names make sense, documentation tells you what each value means

Spreadsheets ➔ Bad Data Hygiene

Using properly structured relations & databases encourage a consistent, standardized way to publish & work with data

Lake Lanier Water Quality Trend Monitoring

Samples taken: October 7, 2007

Field Measurements

Station	Name	Time	Air Temp °C	Water Temp °C	pH	Conduct. micromhos/cm	Cond @25°C micromhos/cm	D.O. mg/l	Comments
1	Balus Cr.	1200	26	19	7.39	106	118	8.2	p. cloudy
2	Flat Cr.	1315	27	24	7.28	1244	1267	7.4	p. cloudy
3	Limestone Cr.	1130	25	20	7.16	123	138	8.5	p. cloudy
4	Chatt. R.	1100	24	21	7.11	48	50	7.5	p. cloudy
5	Little R.	1040	24	19	7.22	60	67	7.1	clear
6	Wahoo Cr.	0945	20	18	7.12	60	70	7.0	clear
7	Squirrel Cr.	1005	23	20	7.08	73	82	8.5	clear
8	Chestatee R.	0920	19	20	7.24	41	45	7.9	p. cloudy
9	Six Mile Cr.	1405	28	20	6.96	189	207	7.9	p. cloudy
10	Buford Dam Splwy	1440	29	10	6.42	36	49	4.5	p. cloudy
11	Bolling Bridge	1345	27	24	7.27	47	47	7.9	p. cloudy

Lab Measurements

Station	Name	Fecal cfb/100ml	BOD_5 mg/l	TSS mg/l	Turb NTU	Hardness mg/l $CaCO_3$	Alkalinity mg/l $CaCO_3$	COD mg/l
1	Balus Cr.	880	1.9	0.6	2.2	44	43	3.4
2	Flat Cr.	80	1.9	0.6	0.8	217	54	12.3
3	Limestone Cr.	100	2.0	1.2	3.3	54	54	7.9
4	Chatt. R.	60	2.1	14.8	12.5	14	15	6.9
5	Little R.	300	1.9	11.4	12.5	17	23	5.9
6	Wahoo Cr.	1270	1.9	9.2	16.0	20	26	8.4
7	Squirrel Cr.	870	2.0	11.2	5.8	27	33	7.4
8	Chestatee R.	190	1.7	3.0	5.0	13	15	6.4
9	Six Mile Cr.	1400	1.7	1.8	2.7	47	19	2.0
10	Buford Dam Splwy	8	1.7	1.8	4.7	14	15	2.5
11	Bolling Bridge	0	1.5	2.2	2.5	13	16	3.9

Station	Name	NO_2+NO_3 mg/l	NH_4 mg/l	Tot N mg/l	Tot P mg/l
1	Balus Cr.	0.6634	0.0099	1.1524	0.0041
2	Flat Cr.	17.0169	0.0222	23.9789	0.0263
3	Limestone Cr.	0.4982	0.0169	23.3754	0.0071
4	Chatt. R.	0.4082	0.0438	10.3025	0.0207
5	Little R.	0.7740	0.0283	5.5969	0.0115
6	Wahoo Cr.	0.2170	0.0423	1.9598	0.0489
7	Squirrel Cr.	0.2525	0.0642	5.2055	0.0717
8	Chestatee R.	0.1755	0.0159	1.9598	0.0153
9	Six Mile Cr.	8.3309	0.0178	18.9063	0.0151
10	Buford Dam Splwy	0.2991	0.0629	5.9394	0.0017
11	Bolling Bridge	0.0147	0.0074	1.7477	0.0067

9-09-07 | 9-30-07 | **10-07-07** | 10-30-07 | 11-11-07 | 12-01-07 | 12-10-07

bandfan.com

Tabular Representation

"Relations"

Named, typed columns

Members

ID Primary key	Name	Birthday	Address	Email
1	Sam	1/1/2000	32 Vassar St	srmadden
2	Tim	1/2/1980	46 Pumpkin St	timk

Unique records

Schema: the names and types of the fields in a table
Tuple: a single record

Unique identifier for a row is a *key*
A minimal unique non-null identifier is a *primary key*

bandfan.com

Tabular Representation

Members

ID Primary key	Name	Birthday	Address	Email
1	Sam	1/1/2000	32 Vassar St	srmadden
2	Tim	1/2/1980	46 Pumpkin St	timk

Bands

ID Primary key	Name	Genre
1	Nickelback	Terrible
2	Creed	Terrible
3	Limp Bizkit	Terrible

How to capture relationship between bandfan members and the bands?

Types of Relationships

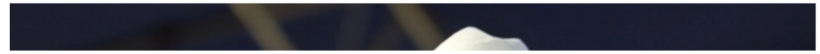
- One to one: each band has a genre
- One to many: bands play shows, one band per show *
- Many to many: members are fans of multiple bands

* Of course, shows might only multiple bands – this is a design decision

Tim the Superfan

Chad Kroeger of Nickelback

2. Nickelback



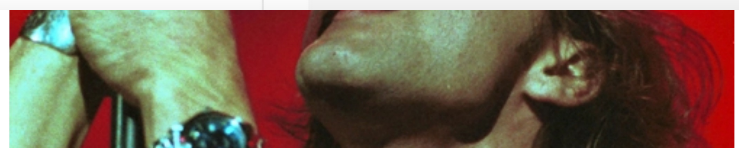
Politics

Who is holding the signs in Iowa that say Ted Cruz likes Nickelback?

By Katie Zezima

January 23, 2016

ANKENY, Iowa - Sen. Ted Cruz (R-Tex.) has been dogged on the campaign trail here in Iowa by a curious protester: a young man holding a sign that states, "Ted Cruz likes Nickelback."



It's no surprise that Creed won this poll. It wasn't even close. This is a band so hated that their own fans sued them after a famously

Representing Fandom Relationship – Try 1

Member-band-fans

FanID	Name	Birthday	Address	Email	BandID	BandName	Genre
2	Tim	1/2/1980	46 Pumpkin St	timk	1	Nickelback	Terrible
2	Tim	1/2/1980	46 Pumpkin St	timk	2	Creed	Terrible
2	Tim	1/2/1980	46 Pumpkin St	timk	3	Limp Bizkit	Terrible

What's wrong with this representation?

Representing Fandom Relationship – Try 1

Member-band-fans

FanID	Name	Birthday	Address	Email	BandID	BandName	Genre
2	Tim	1/2/1980	46 Pumpkin St	timk	1	Nickelback	Terrible
2	Tim	1/2/1980	46 Pumpkin St	timk	2	Creed	Terrible
2	Tim	1/2/1980	46 Pumpkin St	timk	3	Limp Bizkit	Terrible
1	Sam	1/1/2000	32 Vassar St	srmadden	NULL	NULL	NULL

Adding NULLs is messy because it again introduces the possibility of missing data

Representing Fandom Relationship – Try 1

Member-band-fans

FanID	Name	Birthday	Address	Email	BandID	BandName	Genre
2	Tim	1/2/1980	46 Pumpkin St	timk	1	Nickelback	Terrible
2	Tim	1/2/1980	46 Pumpkin St	timk	2	Creed	Terrible
2	Tim	1/2/1980	46 Pumpkin St	timk	3	Limp Bizkit	Terrible
1	Sam	1/1/2000	32 Vassar St	srmadden	NULL	NULL	NULL
3	Markos	1/1/2005	77 Mass Ave	markakis	2	Creed	~~Terrible~~ -- Awful

Duplicated data

- Wastes space
- Possibility of inconsistency

Representing Fandom Relationship – Try 2

Columns that reference keys in other tables are *Foreign keys*

Member-band-fans

FanID	Name	Birthday	Address	Email	BandID
2	Tim	1/2/1980	46 Pumpkin St	timk	1
2	Tim	1/2/1980	46 Pumpkin St	timk	2
2	Tim	1/2/1980	46 Pumpkin St	timk	3

Problem solved?

Still have redundancy

Bands

BandID	Name	Genre
1	Nickelback	Terrible
2	Creed	Terrible
3	Limp Bizkit	Terrible

Representing Fandom Relationship – Try 3

"Normalized"

Members

FanID	Name	Birthday	Address	Email
2	Tim	1/2/1980	46 Pumpkin St	timk
1	Sam	1/1/2000	32 Vassar St	srmadden

Bands

BandID	Name	Genre
1	Nickelback	Terrible
2	Creed	Terrible
3	Limp Bizkit	Terrible

Member-Band-Fans

FanID	BandID
2	1
2	2
2	3

Relationship table

Some members can be a fan of no bands

No duplicates

One-to-Many Relationships

Bands

ID	Name	Genre
1	Nickelback	Terrible
2	Creed	Terrible
3	Limp Bizkit	Terrible

Shows

ID	Location	Date
1	Gillette	4/5/2020
2	Fenway	5/1/2020
3	Agganis	6/1/2020

How to represent the fact that each show is played by one band?

One-to-Many Relationships

Bands

ID	Name	Genre
1	Nickelback	Terrible
2	Creed	Terrible
3	Limp Bizkit	Terrible

Add a band columns to shows

Shows

ID	Location	Date	BandId
1	Gillette	4/5/2020	1
2	Fenway	5/1/2020	1
3	Agganis	6/1/2020	2

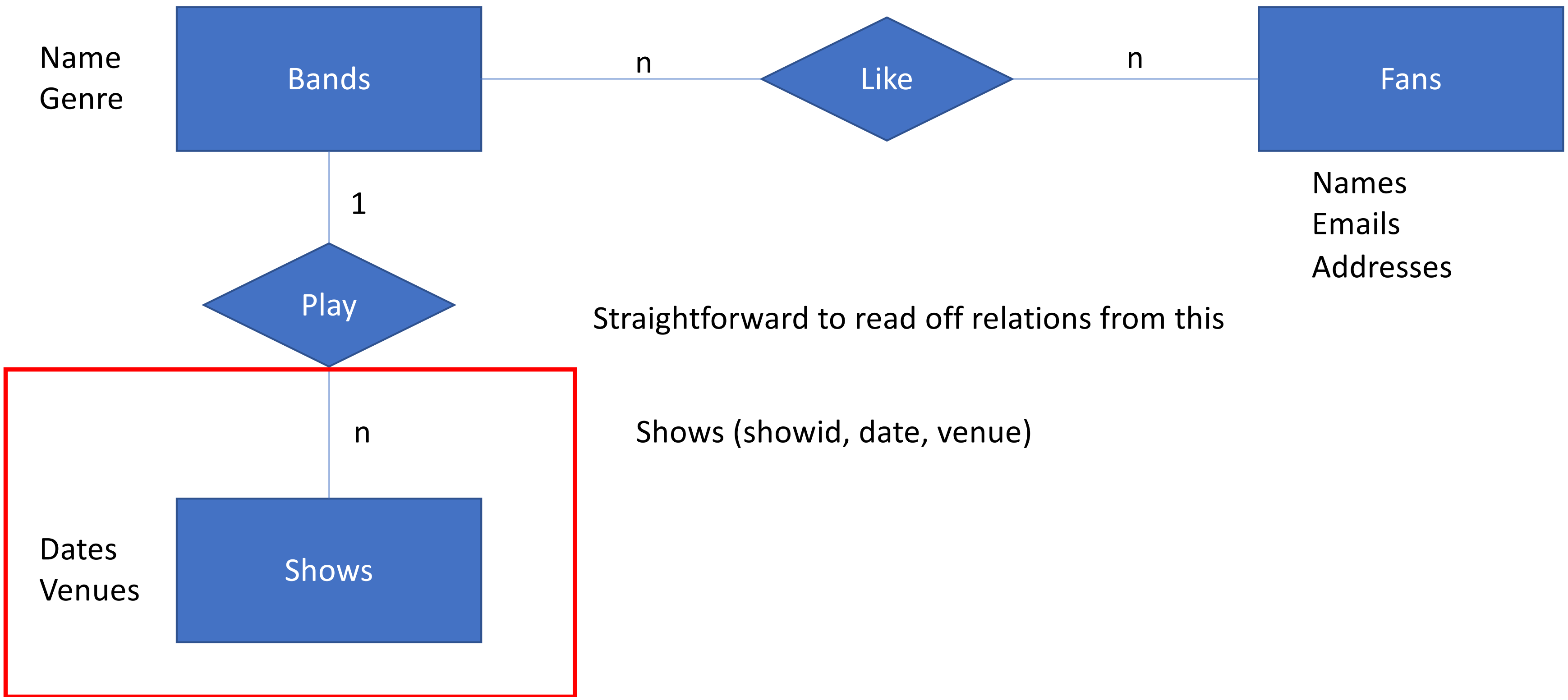
Each band can play multiple shows

Some bands can play no shows

General Approach

- For many-to-many relationships, create a relationship table to eliminate redundancy
- For one-to-many relationships, add a reference column to the table “one” table
  - E.g., each show has one band, so add to the shows table
- Note that deciding which relationships are 1/1, 1/many, many/many is up to the designer of the database
  - E.g., could have shows with multiple bands!

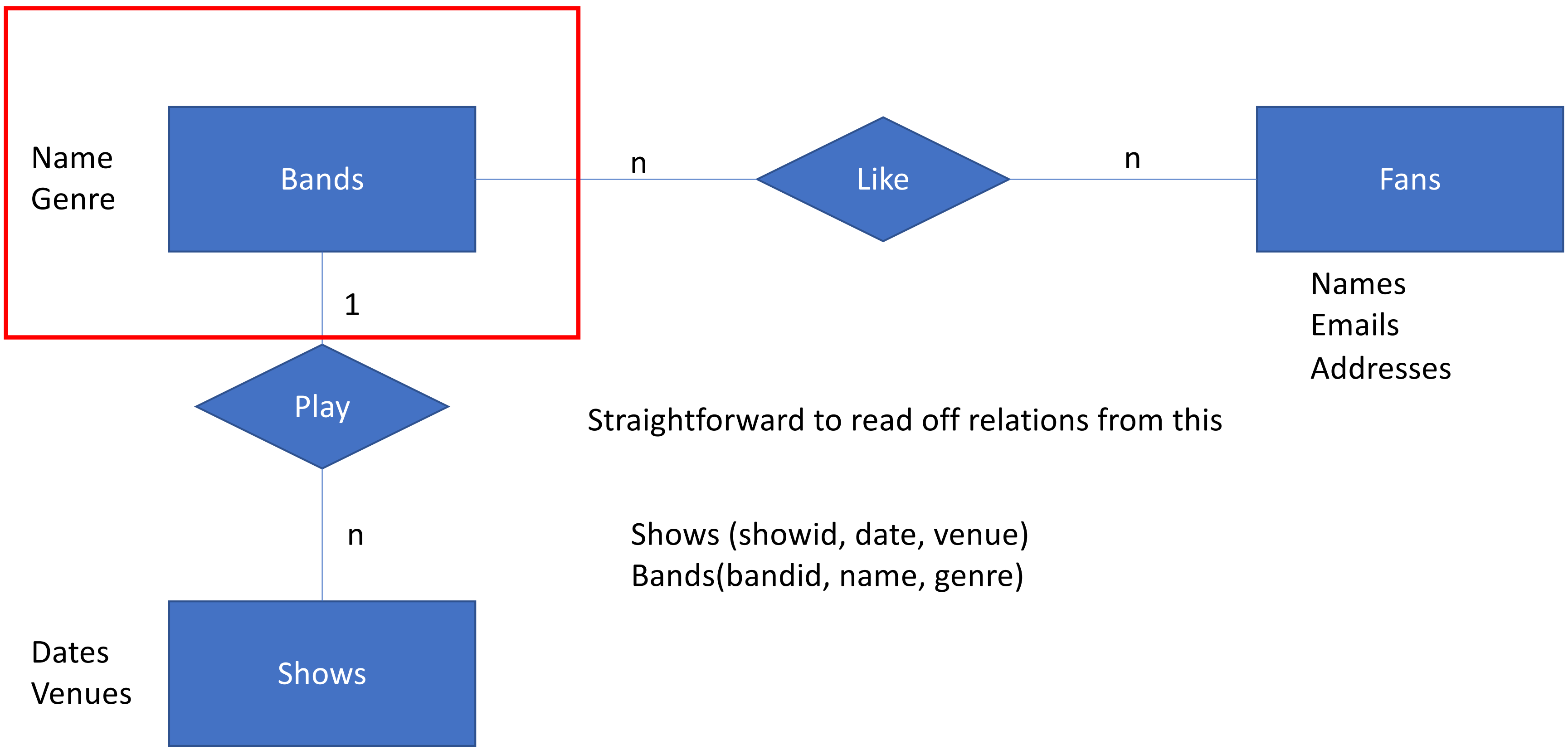
Entity Relationship Diagrams



Straightforward to read off relations from this

Shows (showid, date, venue)

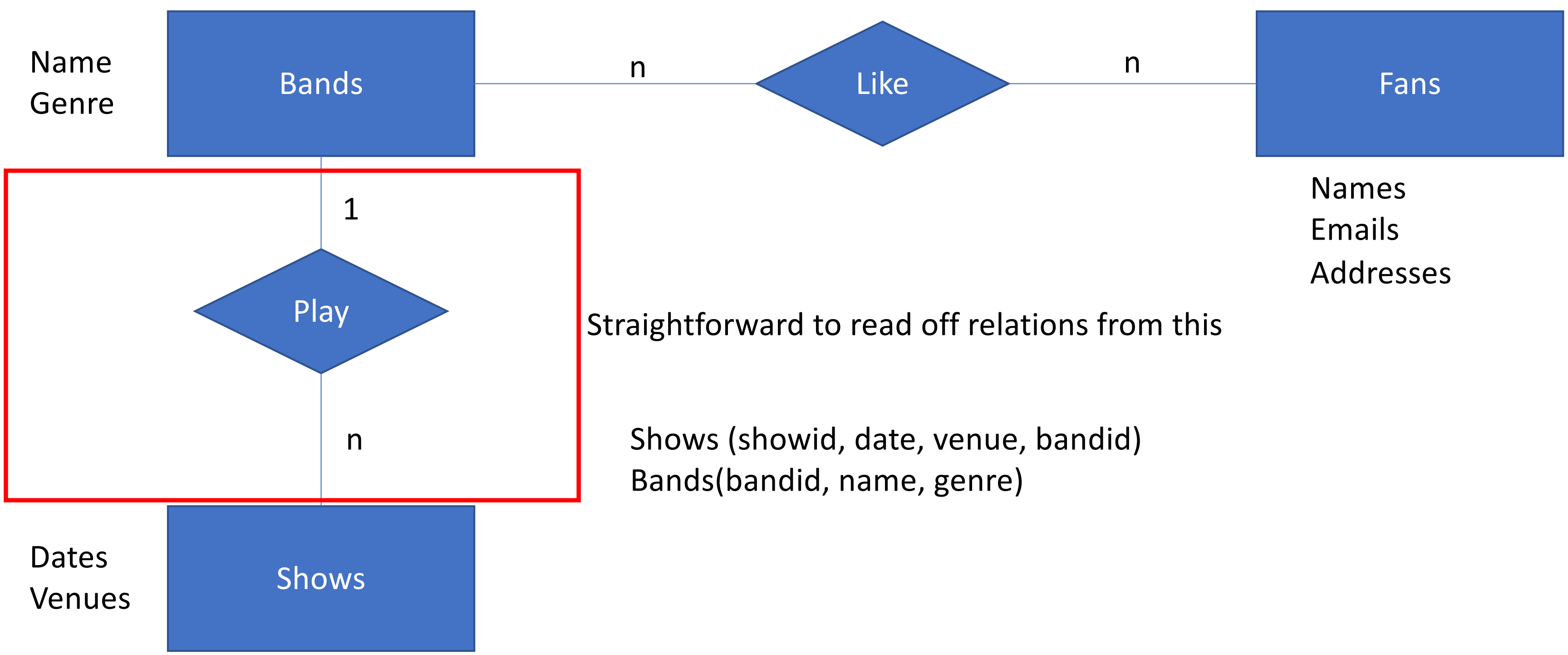
Entity Relationship Diagrams



Straightforward to read off relations from this

Shows (showid, date, venue)
Bands(bandid, name, genre)

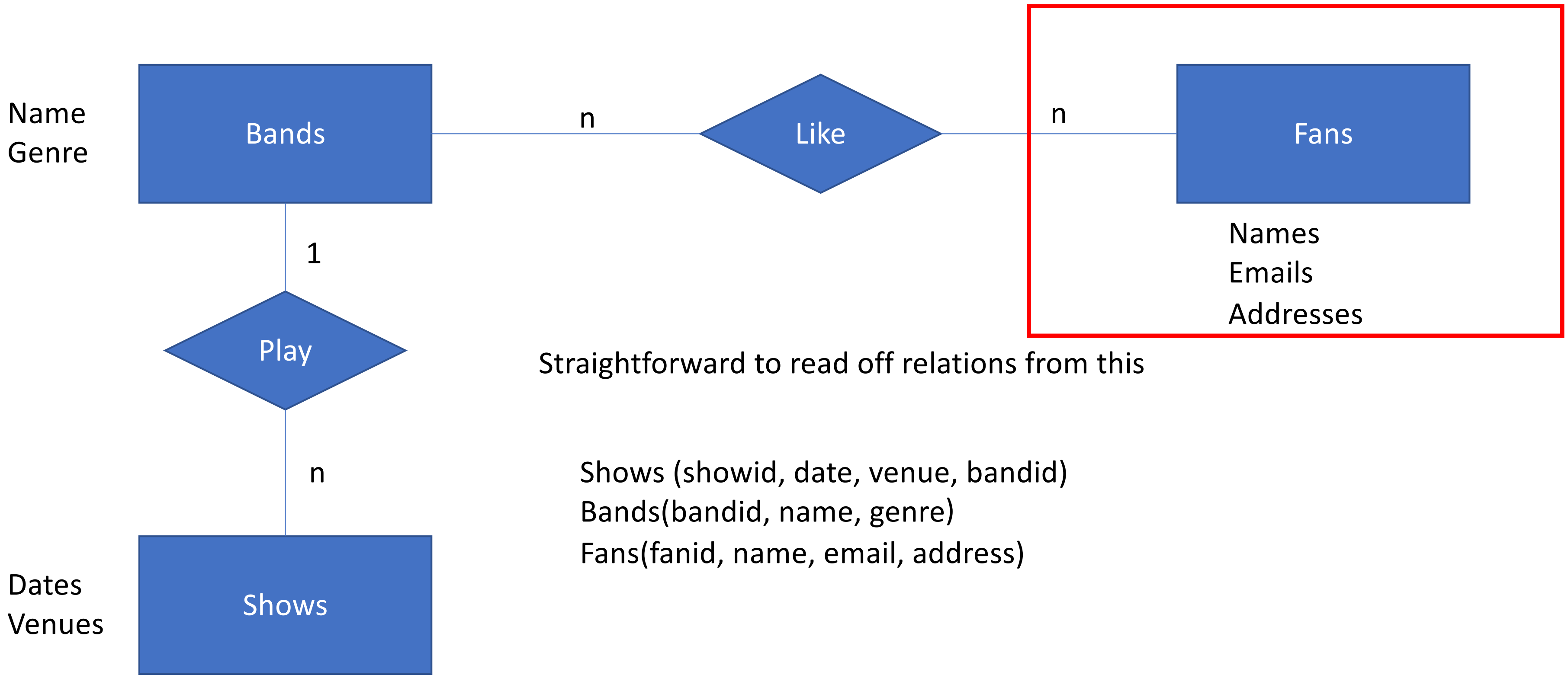
Entity Relationship Diagrams



Straightforward to read off relations from this

Shows (showid, date, venue, bandid)
Bands(bandid, name, genre)

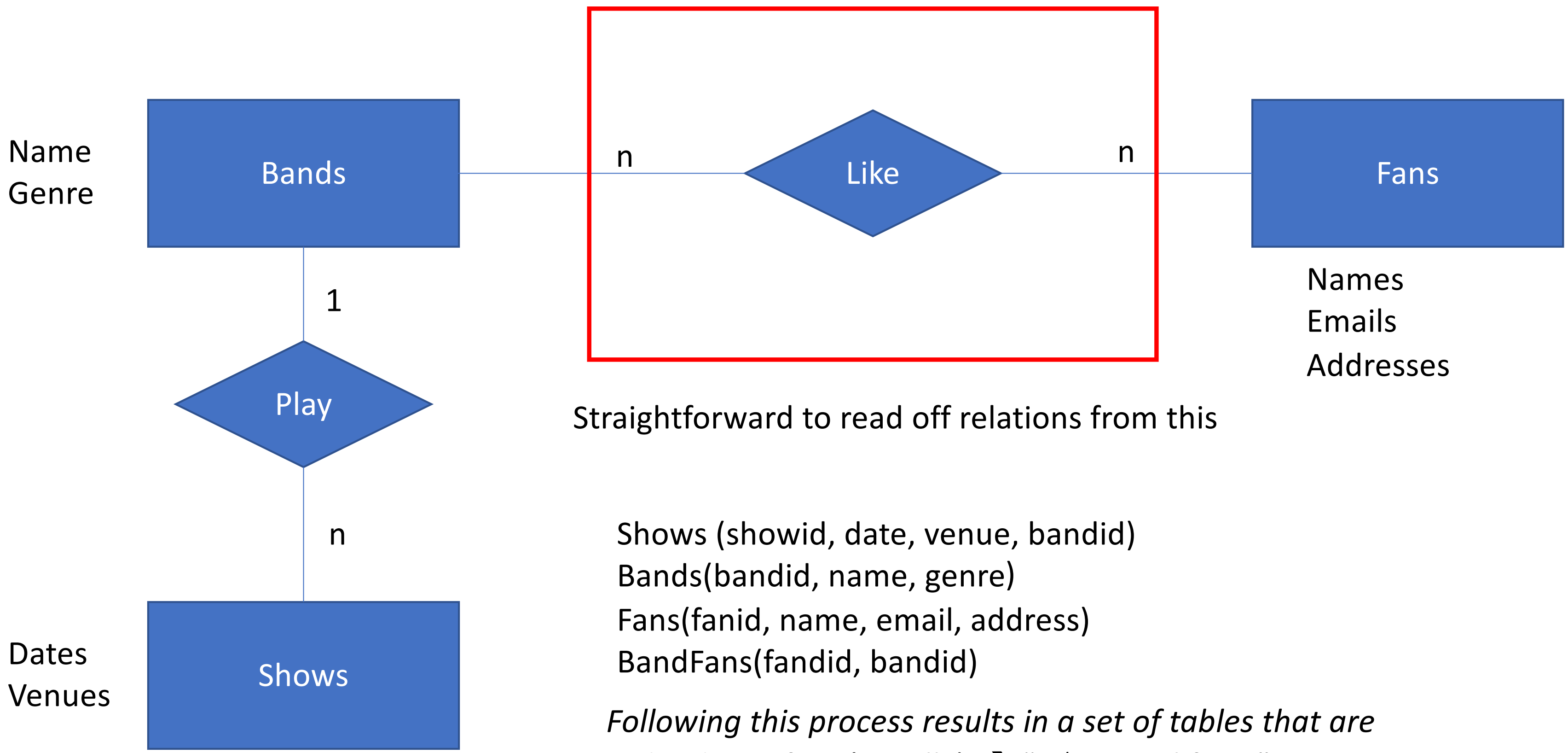
Entity Relationship Diagrams



Straightforward to read off relations from this

Shows (showid, date, venue, bandid)
Bands(bandid, name, genre)
Fans(fanid, name, email, address)

Entity Relationship Diagrams

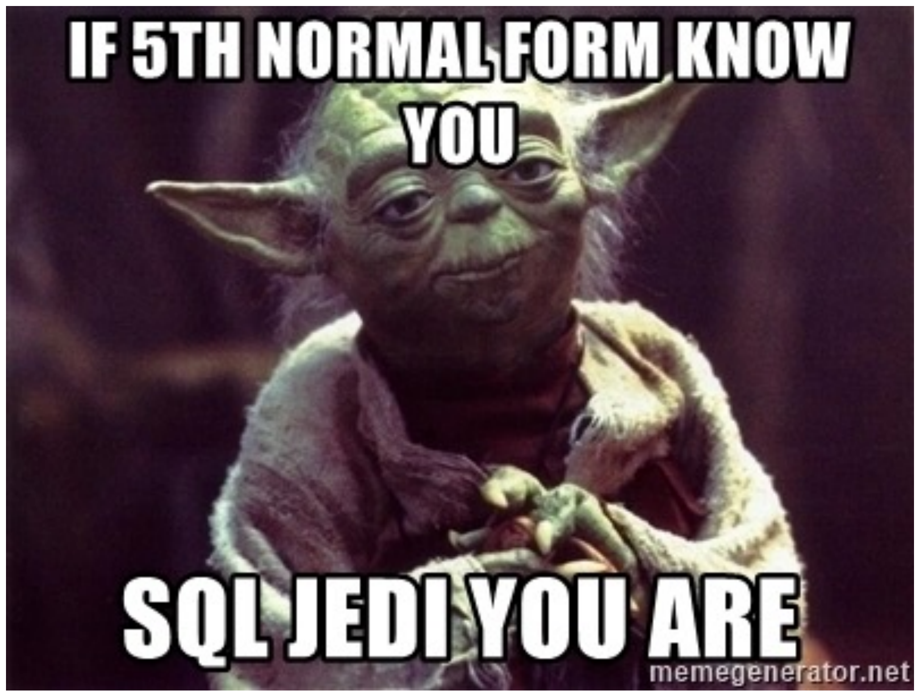


Straightforward to read off relations from this

Shows (showid, date, venue, bandid)
Bands(bandid, name, genre)
Fans(fanid, name, email, address)
BandFans(fandid, bandid)

Following this process results in a set of tables that are redundancy free (usually) ➔ "3rd normal form"

Now you know 90% of what you need to know about database design



Study Break

- Patient database
- Want to represent patients at hospitals with doctors
- Patients have names, birthdates
- Doctors have names, specialties
- Hospitals have names, addresses
- One doctor can treat multiple patients, each patient has one doctor *1-to-many*
- Each patient in one hospital, hospitals have many patients
- Each doctor can work at many hospitals *many-to-many*

Write out schema that captures these relationships, including primary keys and foreign keys

Sol'n

Underline indicates key *1-to-many*

- Patients (<u>pid</u>, name, bday, did references doctors.did, *hid references hospitals.hid*)
- Doctors (<u>did</u>, name, specialty)
- Hospital (<u>hid</u>, name, addr)
- DoctorHospitals(<u>did</u>,<u>hid</u>) *many-to-many*

Operations on Relations

- Can write programs that iterate over and operate on relations
- But there are a very standard set of common operations we might want to perform
  - Filter out rows by conditions (“select”)
  - Connect rows in different tables (“join”)
  - Select subsets of columns (“project”)
  - Compute basic statistics (“aggregate”)

- **Relational algebra** is a formalization of such operations
  - Relations are unordered tables without duplicates (sets)
  - Algebra ➔ operations are closed, i.e., all operations take relations as input and produce relations as output
    - Like arithmetic over $\mathbb{R}$

- A “database” is a set of relations

Relational Algebra

- Projection (π(T,c1, …, cn)) – select a subset of columns c1 .. cn
- Selection (σ(T, pred)) – select a subset of rows that satisfy pred
- Cross Product (T1 x T2) – combine two tables
- Join (T1, T2, pred) = σ(T1 x T2, pred) $\bowtie$ *(T1, T2, pred)*

Plus set operations (Union, Difference, etc)

All ops are set oriented (tables in, tables out)

Join as Cross Product

Bands

bandid	name
1	Nickelback
2	Creed
3	Limp Bizkit

Shows

showid	...	bandid
1		1
2		1
3		2
4		3

Find shows by Creed

```
σ (
    ⋈(
       bands,
       shows,
       bands.bandid=shows.bandid
    ),
    name='Creed'
)
```

Bandid	bandid	Band	...
1	1	Nickelback	
2	1	Creed	
3	1	Limp Bizkit	
1	2	Nickelback	
2	2	Creed	
3	2	Limp Bizkit	
1	3	Nickelback	
2	3	Creed	
3	3	Limp Bizkit	
1	4	Nickelback	
2	4	Creed	
3	4	Limp Bizkit	

Real implementations do not ever materialize the cross product

Join as Cross Product

Bands

bandid	name
1	Nickelback
2	Creed
3	Limp Bizkit

Shows

showid	…	bandid
1		1
2		1
3		2
4		3

Find shows by Creed

σ (
  ⋈(
    bands,
    shows,
    bands.bandid=shows.bandid
  ),
  name='Creed'
)

1. bandid=bandid

Bandid	bandid	Band
1	1	Nickelback
~~2~~	~~1~~	~~Creed~~
~~3~~	~~1~~	~~Limp Bizkit~~
~~1~~	~~2~~	~~Nickelback~~
2	2	Creed
~~3~~	~~2~~	~~Limp Bizkit~~
~~1~~	~~3~~	~~Nickelback~~
~~2~~	~~3~~	~~Creed~~
3	3	Limp Bizkit
~~1~~	~~4~~	~~Nickelback~~
~~2~~	~~4~~	~~Creed~~
~~3~~	~~4~~	~~Limp Bizkit~~

Join as Cross Product

Bands

bandid	name
1	Nickelback
2	Creed
3	Limp Bizkit

Shows

showid	…	bandid
1		1
2		1
3		2
4		3

Find shows by Creed

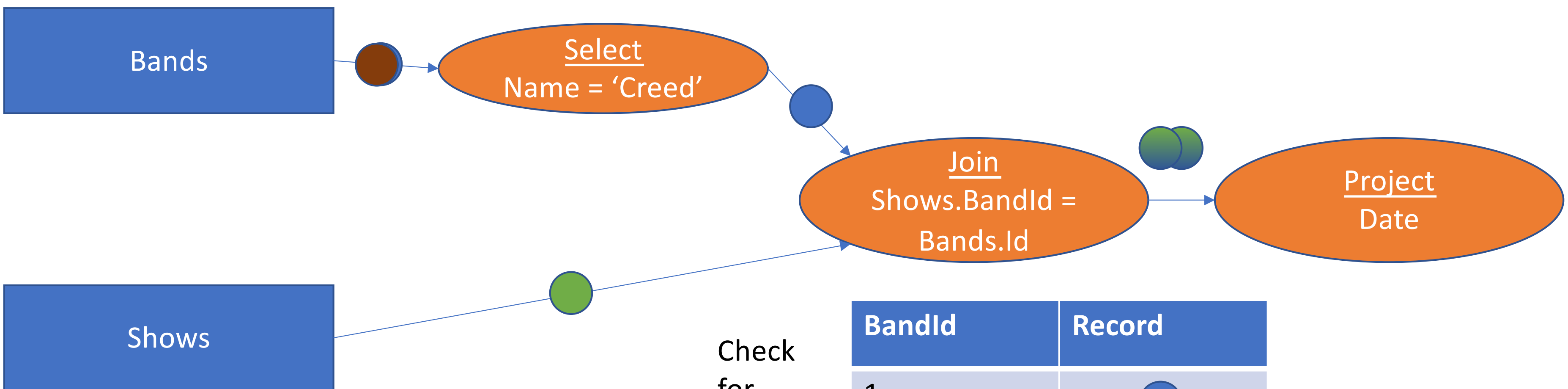
σ (
    ⋈(
        bands,
        shows,
        bands.bandid=shows.bandid
    ),
    name='Creed'
)

1. bandid=bandid
2. name = 'Creed'

Do you think this is how databases actually execute joins?

Bandid	bandid	Band
~~1~~	~~1~~	~~Nickelback~~
~~2~~	~~1~~	~~Creed~~
~~3~~	~~1~~	~~Limp Bizkit~~
~~1~~	~~2~~	~~Nickelback~~
2	2	Creed
~~3~~	~~2~~	~~Limp Bizkit~~
~~1~~	~~3~~	~~Nickelback~~
~~2~~	~~3~~	~~Creed~~
~~3~~	~~3~~	~~Limp Bizkit~~
~~1~~	~~4~~	~~Nickelback~~
~~2~~	~~4~~	~~Creed~~
~~3~~	~~4~~	~~Limp Bizkit~~

Data Flow Graph Representation of Algebra



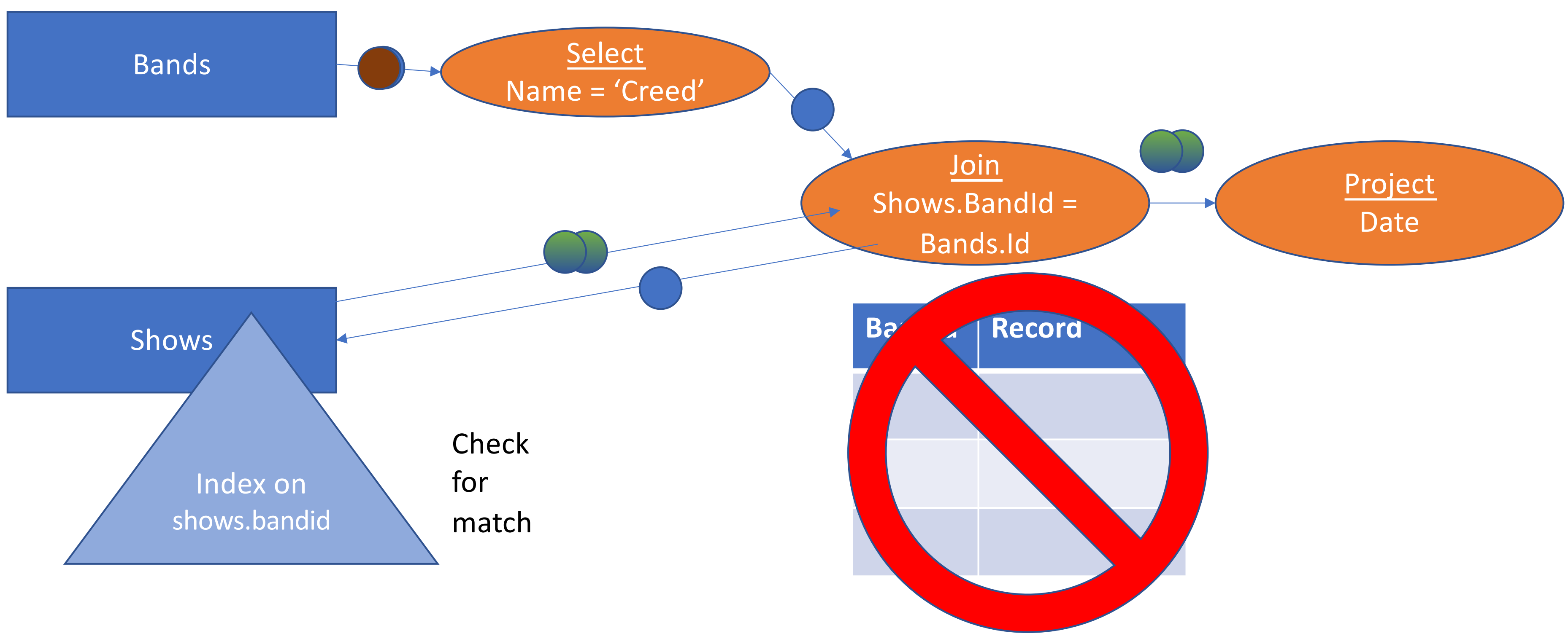
Check
for
match

BandId	Record
1	

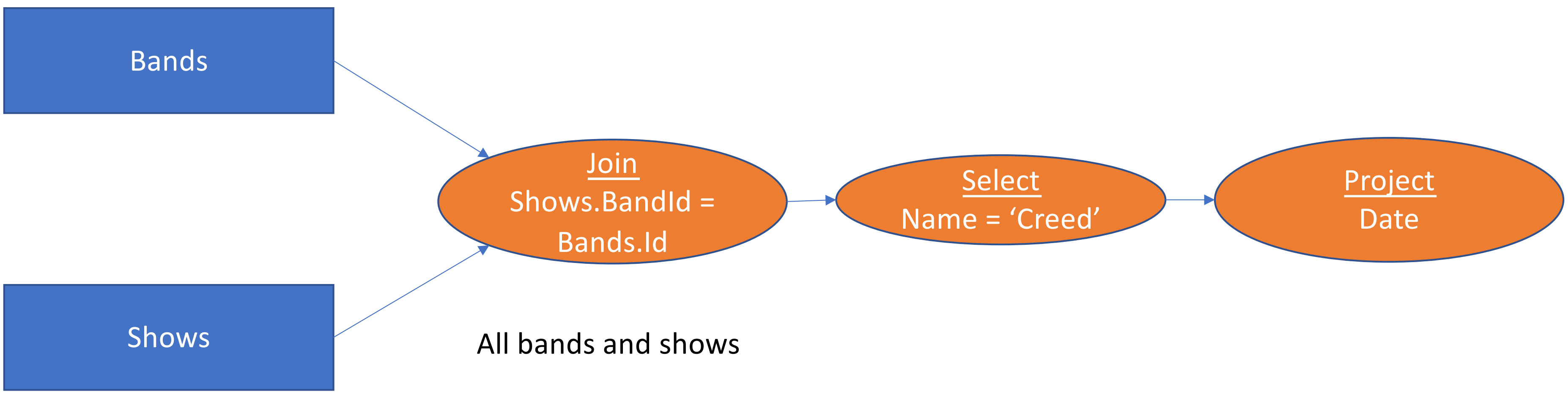
Imagine records flowing out of tables from left to right

Many possible implementations

Suppose we have an *index* on shows: e.g., we store it sorted by band id



Equivalent Representation



Which is better? Why?

Study Break

- Write relational algebra for “Find the bands Tim likes”, using projection, selection, and join

Members

FanID	Name	Birthday	Address	Email

Bands

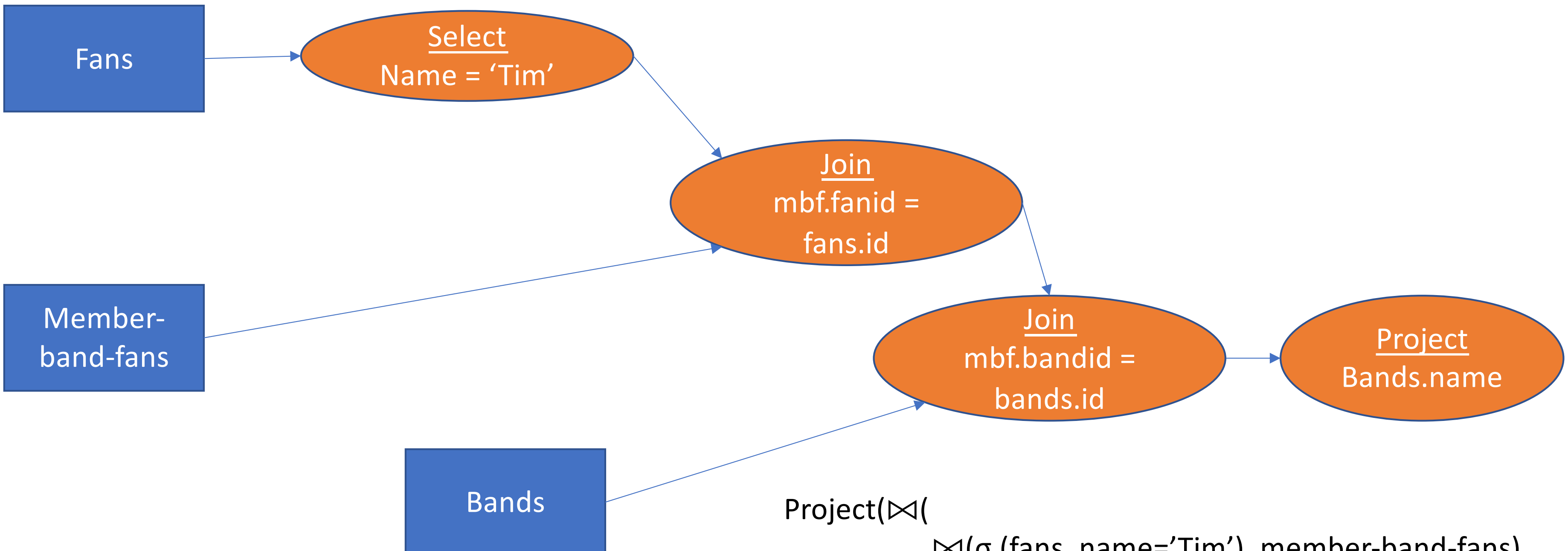
BandID	Name	Genre

Member-Band-Fans

FanID	BandID

- **Projection** (π(T,c1, …, cn)) -- select a subset of columns c1 .. cn
- **Selection** (sel(T, pred)) -- select a subset of rows that satisfy pred
- Cross Product (T1 x T2) -- combine two tables
- **Join** (T1, T2, pred) = sel(T1 x T2, pred)

Find the bands Tim likes



Project(⋈(
⋈(σ (fans, name='Tim'), member-band-fans),
Bands
),
Bands.name))

Multiple Joins

- Note that with multiple joins there are an exponential number of orderings, all of which are equivalent

- E.g., (member-band-fans ⋈ bands) ⋈ fans
  (member-band-fans ⋈ fans) ⋈ bands
  (fans ⋈ bands) ⋈ member-band-fans *Cross product*
- With n tables, n!/2 orderings (assuming a ⋈ b is same as b ⋈ a)

Relational Identities

- Join reordering
  - $(a \bowtie b) \bowtie c = (a \bowtie c) \bowtie b$
- Selection pushdown
  - $\sigma (a \bowtie b) = \sigma(a) \bowtie \sigma(b)$
- These are important when executing SQL queries

SQL

High level programming language based on relational model

Declarative: ”Say what I want, not how to do it”

Let’s look at some examples and come back to this

E.g., programmers doesn’t need to know what operations the database executes to find a particular record

Band Schema in SQL

Varchar is a type, meaning a variable length string

```
CREATE TABLE bands (id int PRIMARY KEY, name varchar, genre varchar);

CREATE TABLE fans (id int PRIMARY KEY, name varchar, address varchar);

CREATE TABLE band_likes(fanid int REFERENCES fans(id),
                        bandid int REFERENCES bands(id));
```

REFERENCEs is a *foreign key*

SQL

- Find the genre of Justin Bieber

```
SELECT genre
FROM bands
WHERE name = 'Justin Bieber'
```

Find the Beliebers

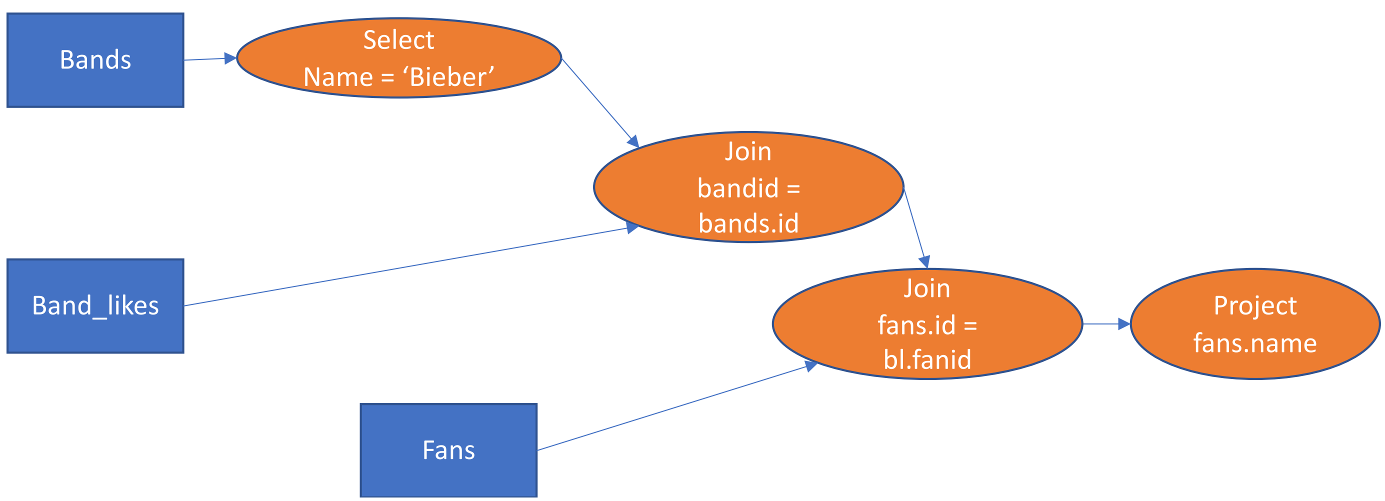
SELECT fans.name

FROM bands

JOIN band_likes bl ON bl.bandid = bands.id *Connect band_likes to bands*

JOIN fans ON fans.id = bl.fanid *Connect fans to band_likes*

WHERE bands.name = 'Justin Bieber'



The fact that the bands – bands_likes join comes first does not imply it will be executed first!

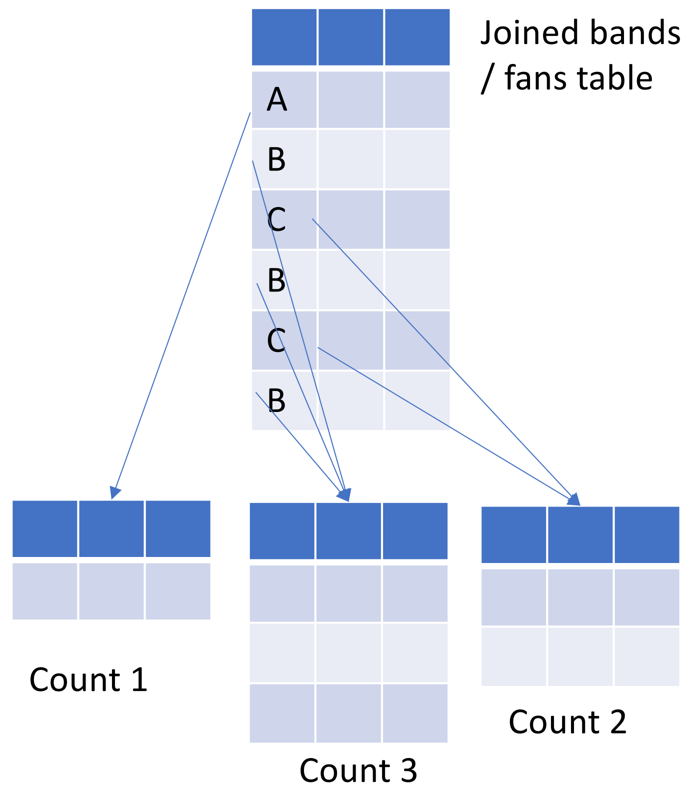
"Declarative" in the sense that the programmer doesn't need to worry about this, or the specifics of how the join will be executed

Find how many fans each band has

```
SELECT bands.name,
       count(*)
FROM bands
JOIN band_likes bl ON bl.bandid = bands.id
JOIN fans ON fans.id = bl.fanid
GROUP BY bands.name;
```

Get the number of bands each fan likes

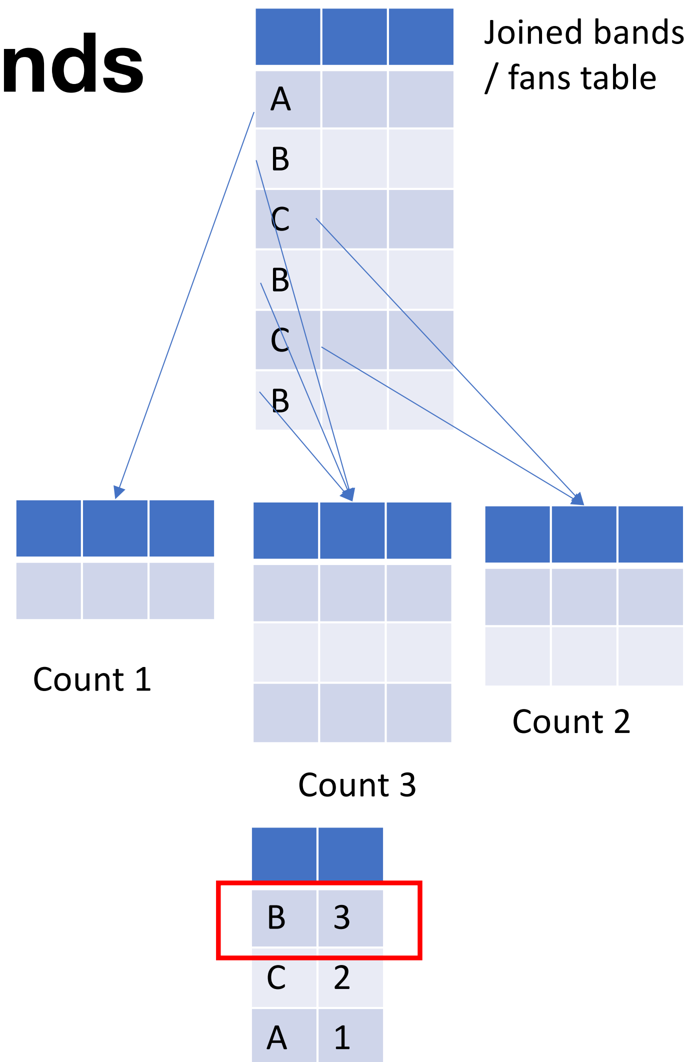
Partition the table by fan name



Find the fan of the most bands

```
SELECT fans.name,
      count(*)
FROM bands
JOIN band_likes bl ON bl.bandid = bands.id
JOIN fans ON fans.id = bl.fanid
GROUP BY fans.name
ORDER BY count(*) DESC LIMIT 1;
```

Sort from highest to lowest and output the top fan



SQL Properties

- **Declarative** – many possible implementations, we don’t have to pick
  - E.g., even for a simple selection, may be:
    - 1) Iterating over the rows
    - 2) Keeping table sorted by primary key and do binary search
    - 3) Keep the data in some kind of a tree (index) structure and do logarithmic search
  - Many more options for joins
  - Not the topic of this course!
- **Physical data independence**
  - As a programmer, you don’t need to understand how data is physically stored
  - E.g., sorted, indexed, unordered, etc
- Keeps programs **simple**, but leads to performance complexity

SQL can get complex

```
with one_phone_tags as (
   select tag_mac_address
   from mapmatch_history
   where uploadtime > '9/1/2021'::date and uploadtime < '9/10/2021'::date
   and json_extract_path_text(device_config,'manufacturer') = 'Apple’
   group by 1
   having count(distinct device_config_hint) = 1
),
ios15_tags as (
select  json_extract_path_text(device_config,'version_release') os_version,
        json_extract_path_text(device_config,'model') model_number,
        tag_mac_address
   from mapmatch_history
   where uploadtime >= '10/11/2021'::date
   and json_extract_path_text(device_config,'manufacturer') = 'Apple'
   and tag_mac_address in (select tag_mac_address from one_phone_tags)
  and substring(os_version, 1, 2) = '15'
   group by 1,2,3
),
ios14_tags as (
select  json_extract_path_text(device_config,'version_release') os_version,
        json_extract_path_text(device_config,'model') model_number,
        tag_mac_address
   from mapmatch_history
   where uploadtime >= '9/15/2021'::date and uploadtime <= '9/20/2021'::date
   and json_extract_path_text(device_config,'manufacturer') = 'Apple'
   and tag_mac_address in (select tag_mac_address from one_phone_tags)
  and substring(os_version, 1, 2) = '14'
   group by 1,2,3 ),
ios15_trip_stats as (
   select tag_mac_address, count(*) ios15_num_trips,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end)
ios15_num_trips_no_phone,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end) /
count(*)::float ios15_frac_none,
   from triplog_trips join ios15_tags using(tag_mac_address)
   where created_date >= '10/11/2021'::date
   and trip_start_ts >= '10/09/2021'::date
   and substring(model_number, 1, 8) = 'iPhone13'
   group by tag_mac_address
   having count(*) > 0
),
ios14_trip_stats as (
   select tag_mac_address, count(*) ios14_num_trips,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end)
ios14_num_trips_no_phone,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end) /
count(*)::float ios14_frac_none,
   from triplog_trips join ios14_tags using(tag_mac_address)
   where created_date >= '9/15/2021'::date and created_date <= '9/20/2021'::date
   and trip_start_ts >= '9/13/2021'::date and trip_start_ts <= '9/20/2021'::date
   and substring(model_number, 1, 8) = 'iPhone13'
   group by tag_mac_address
   having count(*) > 0
)
select
tag_mac_address,ios14_num_trips,ios14_num_trips_no_phone,ios14_frac_none,
     ios15_num_trips,ios15_num_trips_no_phone,ios15_frac_none
     from ios15_trip_stats join ios14_trip_stats using(tag_mac_address)
```

Tuning Example: Beliebers

- Find fans of Justin Bieber

```
SELECT fans.name
FROM bands
JOIN band_likes bl ON bl.bandid = bands.id
JOIN fans ON fans.id = bl.fanid
WHERE bands.name = 'Justin Bieber'
```

How might we make this query faster?

```
create  index band_names_index on bands(name);
```

Next Time

- Fancier SQL
- Performance Tuning

- Relational algebra in pandas / python