6.S079 Lecture 2
Sam Madden

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

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


An SQL query walks into a bar and sees two tables. He walks up to them and says "Can I join you?"
Data Science Pipeline

- Visualization/Presentation
- Modeling/Prediction
- Querying/Processing
- Stored Data
- Transformation/Integration
  - Structure Extraction
    - Image Segmentation
    - Signal Processing
    - Regularization
    - Re-Sampling
    - Cleaning
    - Outlier Removal
- Raw Data
  - Text
  - Images
  - Sounds
  - Sensor Readings
  - CSV
  - Log files
  - Web Forms

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

"Relations"

<table>
<thead>
<tr>
<th>ID</th>
<th>Primary key</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Sam</td>
<td>1/1/2000</td>
<td>32 Vassar St</td>
<td>srmadden</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
</tr>
</tbody>
</table>

Named, typed columns

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**
# Tabular Representation

## Members

<table>
<thead>
<tr>
<th>ID</th>
<th>Primary key</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Sam</td>
<td>1/1/2000</td>
<td>32 Vassar St</td>
<td>srmadden</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
</tr>
</tbody>
</table>

## Bands

<table>
<thead>
<tr>
<th>ID</th>
<th>Primary key</th>
<th>Name</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
</tbody>
</table>

*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
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

<table>
<thead>
<tr>
<th>FanID</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
<th>BandID</th>
<th>BandName</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
</tbody>
</table>

What’s wrong with this representation?
## Representing Fandom Relationship – Try 1

### Member-band-fans

<table>
<thead>
<tr>
<th>FanID</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
<th>BandID</th>
<th>BandName</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
<tr>
<td>1</td>
<td>Sam</td>
<td>1/1/2000</td>
<td>32 Vassar St</td>
<td>srmadden</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Adding NULLs is messy because it again introduces the possibility of missing data.
## Representing Fandom Relationship – Try 1

### Member-band-fans

<table>
<thead>
<tr>
<th>FanID</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
<th>BandID</th>
<th>BandName</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
<tr>
<td>1</td>
<td>Sam</td>
<td>1/1/2000</td>
<td>32 Vassar St</td>
<td>smaddem</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>Markos</td>
<td>1/1/2005</td>
<td>77 Mass Ave</td>
<td>markakis</td>
<td>2</td>
<td>Creed</td>
<td>Terrible Awful</td>
</tr>
</tbody>
</table>

Duplicated data
- Wastes space
- Possibility of inconsistency
Representing Fandom Relationship – Try 2

### Member-band-fans

<table>
<thead>
<tr>
<th>FanID</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
<th>BandID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
<td>3</td>
</tr>
</tbody>
</table>

### Bands

<table>
<thead>
<tr>
<th>BandID</th>
<th>Name</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
</tbody>
</table>

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

*Problem solved?*
*Still have redundancy*
## Representing Fandom Relationship – Try 3

“Normalized”

### Members

<table>
<thead>
<tr>
<th>FanID</th>
<th>Name</th>
<th>Birthday</th>
<th>Address</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tim</td>
<td>1/2/1980</td>
<td>46 Pumpkin St</td>
<td>timk</td>
</tr>
<tr>
<td>1</td>
<td>Sam</td>
<td>1/1/2000</td>
<td>32 Vassar St</td>
<td>srmadden</td>
</tr>
</tbody>
</table>

### Bands

<table>
<thead>
<tr>
<th>BandID</th>
<th>Name</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
</tbody>
</table>

### Member-Band-Fans

<table>
<thead>
<tr>
<th>FanID</th>
<th>BandID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

*Relationship table*

Some members can be a fan of no bands

No duplicates
One-to-Many Relationships

<table>
<thead>
<tr>
<th>Bands</th>
<th>ID</th>
<th>Name</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shows</th>
<th>ID</th>
<th>Location</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Gillette</td>
<td>4/5/2020</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Fenway</td>
<td>5/1/2020</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Agganis</td>
<td>6/1/2020</td>
</tr>
</tbody>
</table>

How to represent the fact that each show is played by one band?
One-to-Many Relationships

<table>
<thead>
<tr>
<th>Bands</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Name</td>
<td>Genre</td>
</tr>
<tr>
<td>1</td>
<td>Nickelback</td>
<td>Terrible</td>
</tr>
<tr>
<td>2</td>
<td>Creed</td>
<td>Terrible</td>
</tr>
<tr>
<td>3</td>
<td>Limp Bizkit</td>
<td>Terrible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shows</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Location</td>
<td>Date</td>
</tr>
<tr>
<td>1</td>
<td>Gillette</td>
<td>4/5/2020</td>
</tr>
<tr>
<td>2</td>
<td>Fenway</td>
<td>5/1/2020</td>
</tr>
<tr>
<td>3</td>
<td>Agganis</td>
<td>6/1/2020</td>
</tr>
</tbody>
</table>

Add a band columns to shows

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

Name
Genre

Bands

Like

n

n

Fans

Names
Emails
Addresses

Play

n

1

Dates
Venues

Shows

Straightforward to read off relations from this

Shows (showid, date, venue)
Bands(bandid, name, genre)
Entity Relationship Diagrams

Bands
- Name
- Genre

Fans
- Names
- Emails
- Addresses

Shows
- Dates
- Venues

Like

Straightforward to read off relations from this

Shows (showid, date, venue, bandid)
Bands(bandid, name, genre)
Entity Relationship Diagrams

Bands

- Name
- Genre

Fans

- Names
- Emails
- Addresses

Shows

- Dates
- Venues

Like

- n

Play

- 1

Straightforward to read off relations from this

Shows (showid, date, venue, bandid)
Bands (bandid, name, genre)
Fans (fanid, name, email, address)
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
• Each patient in one hospital, hospitals have many patients
• Each doctor can work at many hospitals

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

Underline indicates key

• Patients (pid, name, bday, did references doctors.did, hid references hospitals.hid)
• Doctors (did, name, specialty)
• Hospital (hid, name, addr)
• DoctorHospitals(did,hid)  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 \( \rightarrow \) 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 ($\pi(T,c_1, \ldots, c_n)$) – select a subset of columns $c_1 \ldots c_n$
- Selection ($\sigma(T, \text{pred})$) – select a subset of rows that satisfy $\text{pred}$
- Cross Product ($T_1 \times T_2$) – combine two tables
- Join ($T_1, T_2, \text{pred} = \sigma(T_1 \times T_2, \text{pred}) = \Join(T_1, T_2, \text{pred})$)

Plus set operations (Union, Difference, etc)

All ops are set oriented (tables in, tables out)
Join as Cross Product

Find shows by Creed

$$\sigma ( \ni ( \text{bands, shows, } \text{bands.bandid=shows.bandid}) , \text{name='Creed'})$$

Real implementations do not ever materialize the cross product
Join as Cross Product

**Bands**

<table>
<thead>
<tr>
<th>bandid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nickelback</td>
</tr>
<tr>
<td>2</td>
<td>Creed</td>
</tr>
<tr>
<td>3</td>
<td>Limp Bizkit</td>
</tr>
</tbody>
</table>

**Shows**

<table>
<thead>
<tr>
<th>showid</th>
<th>...</th>
<th>bandid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Find shows by Creed

\[
\sigma (\ni (bandid=bandid), \text{name='Creed'})
\]
## Join as Cross Product

### Bands

<table>
<thead>
<tr>
<th>bandid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nickelback</td>
</tr>
<tr>
<td>2</td>
<td>Creed</td>
</tr>
<tr>
<td>3</td>
<td>Limp Bizkit</td>
</tr>
</tbody>
</table>

### Shows

<table>
<thead>
<tr>
<th>showid</th>
<th>...</th>
<th>bandid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

### Find shows by Creed

\[ \sigma (\exists (\text{bands, shows, bands.bandid}=\text{shows.bandid}, \text{name}=\text{'Creed'}) ) \]

1. bandid=bandid
2. name = ‘Creed’

---

### SQL

\[ \sigma (\exists (\text{bands, shows, bands.bandid}=\text{shows.bandid}, \text{name}=\text{'Creed'}) ) \]

---

**Do you think this is how databases actually execute joins?**
Data Flow Graph Representation of Algebra

Imagine records flowing out of tables from left to right

<table>
<thead>
<tr>
<th>BandId</th>
<th>Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
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?

Bands

Shows

Join
Shows.BandId = Bands.Id

Select
Name = ‘Creed’

Project
Date

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

**Projection** ($\pi(T, c_1, ..., c_n)$) -- select a subset of columns $c_1 .. c_n$

**Selection** ($\sigma(T, \text{pred})$) -- select a subset of rows that satisfy pred

**Cross Product** ($T_1 \times T_2$) -- combine two tables

**Join** ($T_1, T_2, \text{pred}$) = $\sigma(T_1 \times T_2, \text{pred})$
Find the bands Tim likes
Multiple Joins

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

• E.g., (member-band-fans ▶️ fans) ▶️ 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

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));
SQL

- Find the genre of Justin Bieber

```sql
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
JOIN fans ON fans.id = bl.fanid
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