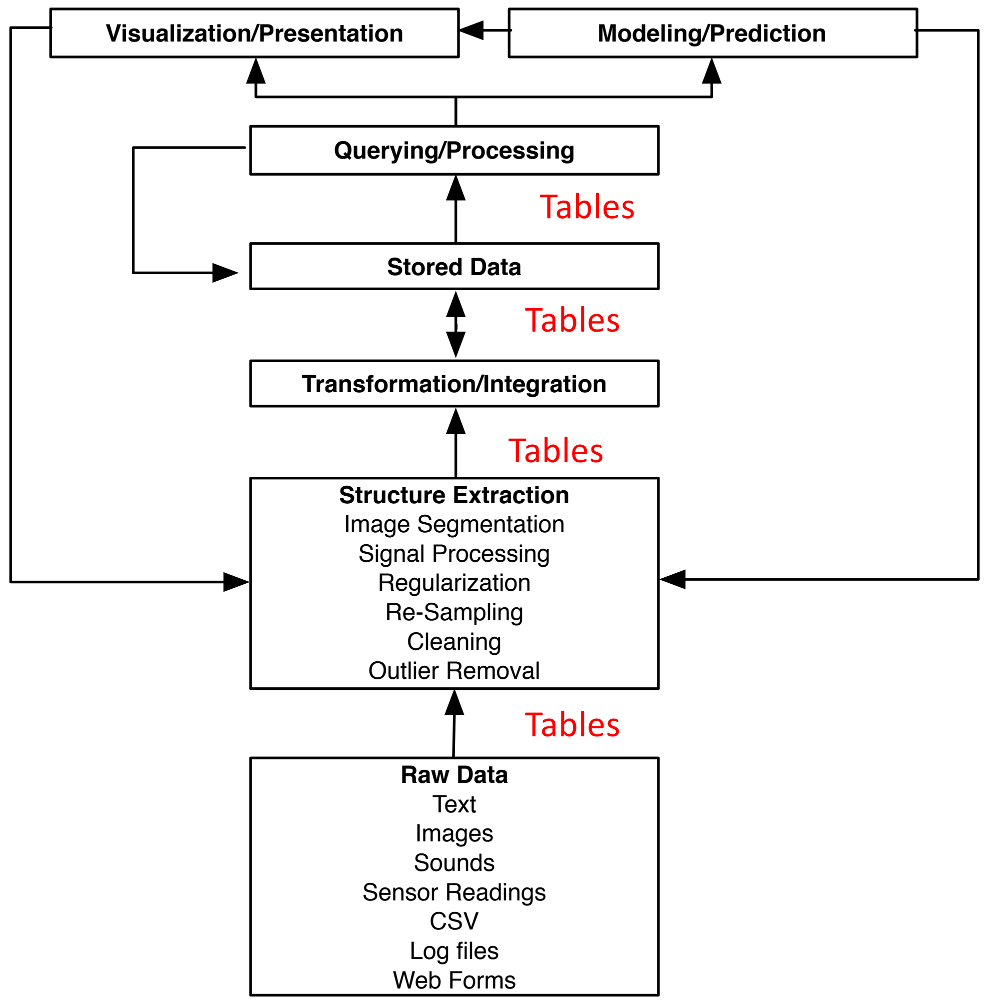
http://dsg.csail.mit.edu/6.S079/

Programming with Data Bootcamp: Lecture 1

Slides courtesy of Sam Madden /
Tim Kraska (6.S079)

Key ideas:
Tabular data & relational model
Relational algebra & SQL
Indexes and performance tuning

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

	A	B	C	D	E	F	G	H	I	J
1			Lake Lanier Water Quality Trend Monitoring							
2				Samples taken:		October 7, 2007				
3										
4					Field Measurements					
5				Air	Water		Conduct.	Cond @25°C		
6	Station	Name	Time	Temp °C	Temp °C	pH	nicromhos/cm	nicromhos/cm	D.O. mg/l	Comments
7	1	Balus Cr.	1200	26	19	7.39	106	118	8.2	p. cloudy
8	2	Flat Cr.	1315	27	24	7.28	1244	1267	7.4	p. cloudy
9	3	Limestone Cr.	1130	25	20	7.16	123	138	8.5	p. cloudy
10	4	Chatt. R.	1100	24	21	7.11	48	50	7.5	p. cloudy
11	5	Little R.	1040	24	19	7.22	60	67	7.1	clear
12	6	Wahoo Cr.	0945	20	18	7.12	60	70	7.0	clear
13	7	Squirrel Cr.	1005	23	20	7.08	73	82	8.5	clear
14	8	Chestatee R.	0920	19	20	7.24	41	45	7.9	p. cloudy
15	9	Six Mile Cr.	1405	28	20	6.96	189	207	7.9	p. cloudy
16	10	Buford Dam Splw	1440	29	10	6.42	36	49	4.5	p. cloudy
17	11	Bolling Bridge	1345	27	24	7.27	47	47	7.9	p. cloudy
18										
19										
20					Lab Measurements					
21			Fecal	BOD_5	TSS		Hardness	Alkalinity	COD	
22	Station	Name	cfb/100ml	mg/l	mg/l	Turb NTU	mg/l $CaCO_3$	mg/l $CaCO_3$	mg/l	
23	1	Balus Cr.	880	1.9	0.6	2.2	44	43	3.4	
24	2	Flat Cr.	80	1.9	0.6	0.8	217	54	12.3	
25	3	Limestone Cr.	100	2.0	1.2	3.3	54	54	7.9	
26	4	Chatt. R.	60	2.1	14.8	12.5	14	15	6.9	
27	5	Little R.	300	1.9	11.4	12.5	17	23	5.9	
28	6	Wahoo Cr.	1270	1.9	9.2	16.0	20	26	8.4	
29	7	Squirrel Cr.	870	2.0	11.2	5.8	27	33	7.4	
30	8	Chestatee R.	190	1.7	3.0	5.0	13	15	6.4	
31	9	Six Mile Cr.	1400	1.7	1.8	2.7	47	19	2.0	
32	10	Buford Dam Splw	8	1.7	1.8	4.7	14	15	2.5	
33	11	Bolling Bridge	0	1.5	2.2	2.5	13	16	3.9	
34										
35			NO_2+NO_3	NH_4	Tot N	Tot P				
36	Station	Name	mg/l	mg/l	mg/l	mg/l				
37	1	Balus Cr.	0.6634	0.0099	1.1524	0.0041				
38	2	Flat Cr.	17.0169	0.0222	23.9789	0.0263				
39	3	Limestone Cr.	0.4982	0.0169	23.3754	0.0071				
40	4	Chatt. R.	0.4082	0.0438	10.3025	0.0207				
41	5	Little R.	0.7740	0.0283	5.5969	0.0115				
42	6	Wahoo Cr.	0.2170	0.0423	1.9598	0.0489				
43	7	Squirrel Cr.	0.2525	0.0642	5.2055	0.0717				
44	8	Chestatee R.	0.1755	0.0159	1.9598	0.0153				
45	9	Six Mile Cr.	8.3309	0.0178	18.9063	0.0151				
46	10	Buford Dam Splw	0.2991	0.0629	5.9394	0.0017				
47	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

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	$\boxed{\text{Terrible}}$
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

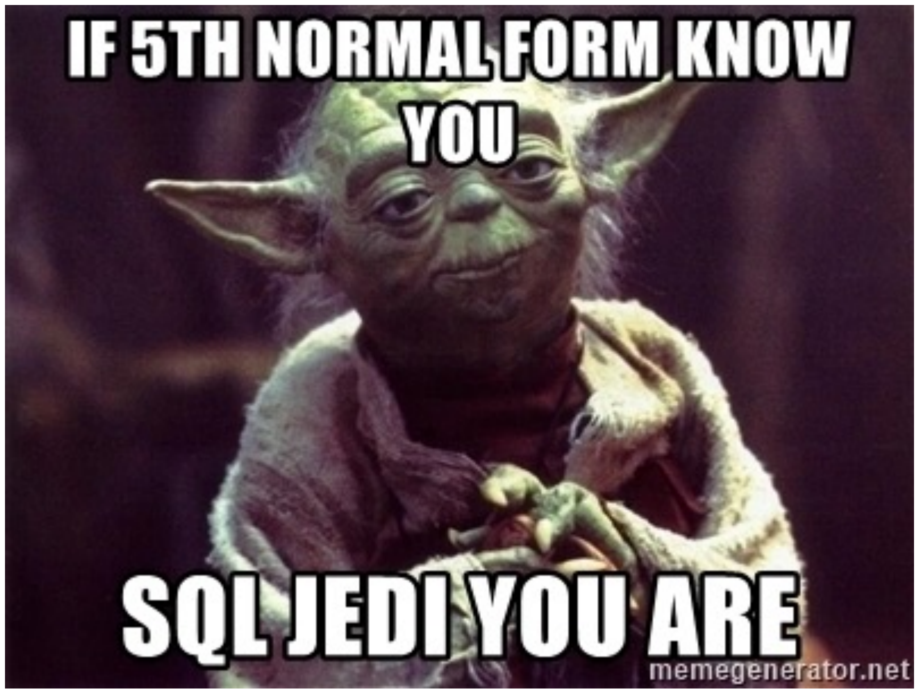
Each band can play multiple shows

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!

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

  1-to-many
- 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 *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 (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 ➔ 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	name	...
1	1	Nickelback	
2	1	Creed	
3	1	Limp Bizkit	
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	

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	name
1	1	Nickelback
~~2~~	~~1~~	~~Creed~~
~~3~~	~~1~~	~~Limp Bizkit~~
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

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

bandid	bandid	name
~~1~~	~~1~~	~~Nickelback~~
~~2~~	~~1~~	~~Creed~~
~~3~~	~~1~~	~~Limp Bizkit~~
~~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~~

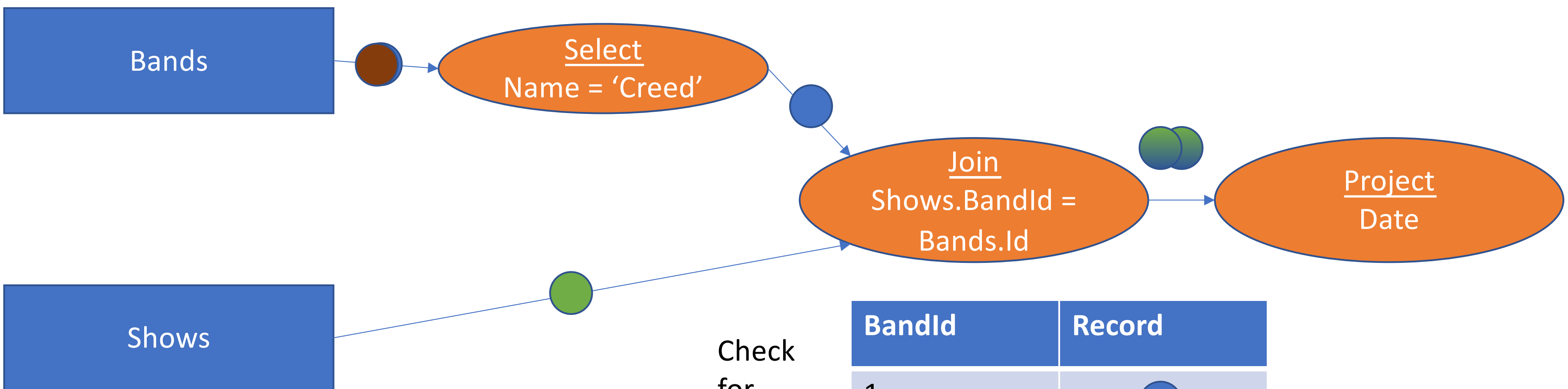
Find shows by Creed

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?

Data Flow Graph Representation of Algebra



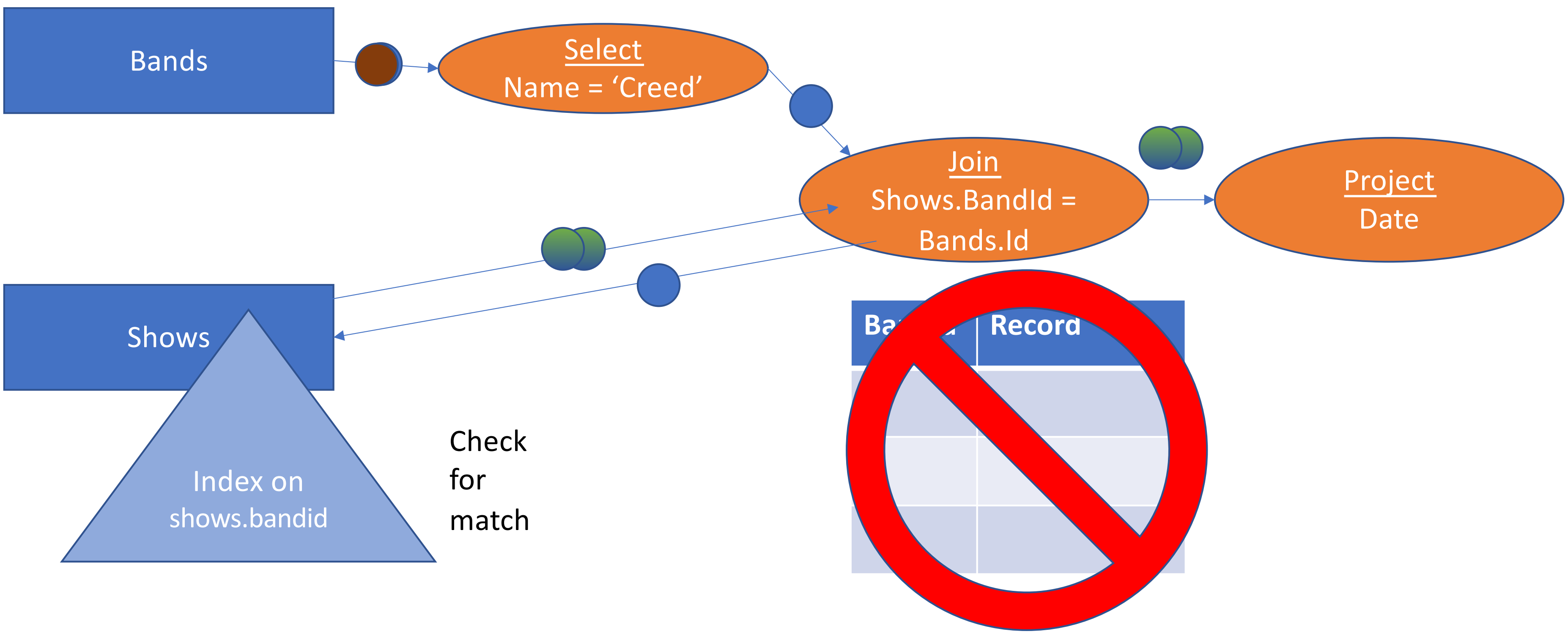
Check for match

BandId	Record
1	

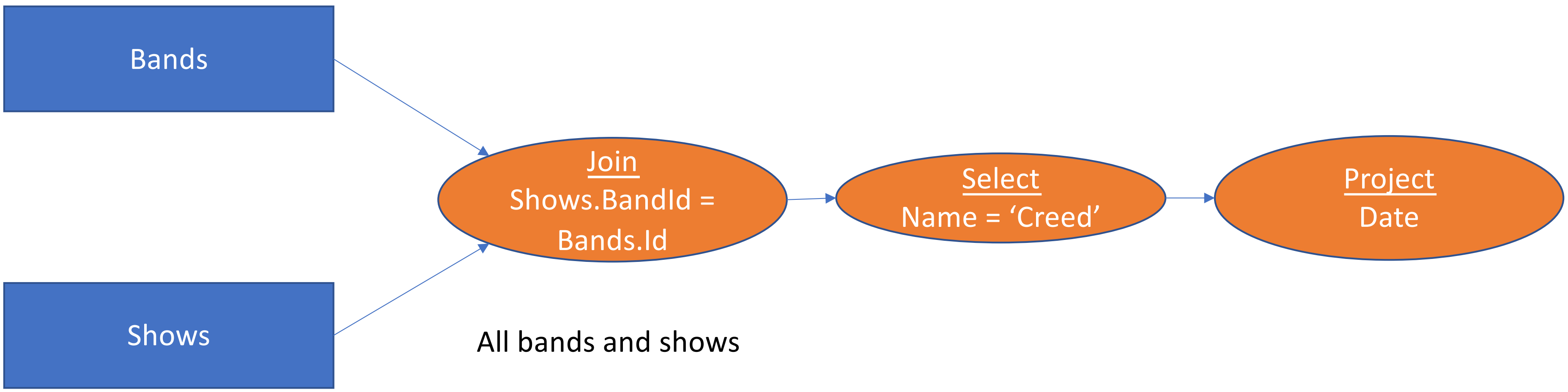
Imagine records flowing out of tables from left to right

Many possible implementations

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



Equivalent Representation



All bands and shows

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

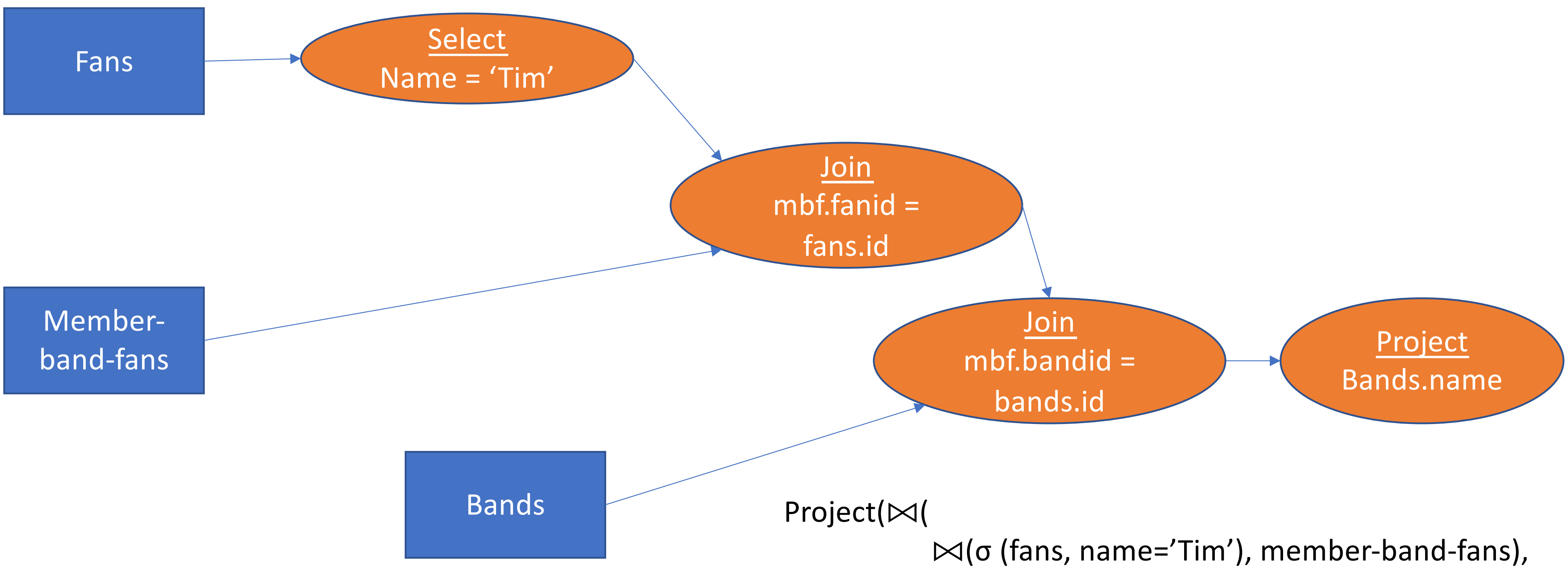
BandID	Name	Genre

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))
```

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

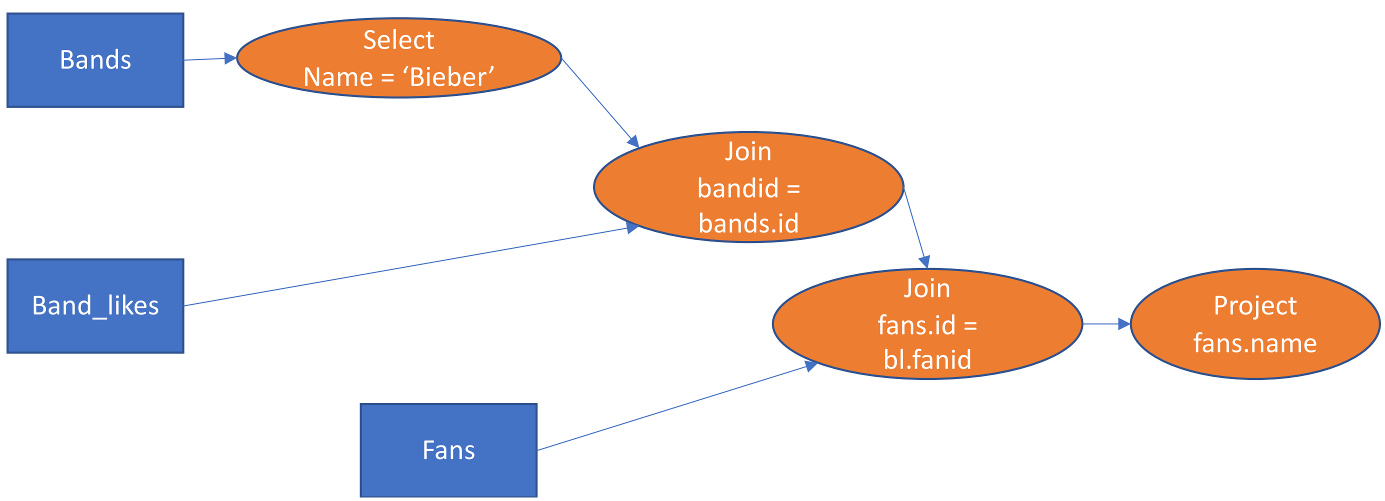
SELECT fans.name

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!

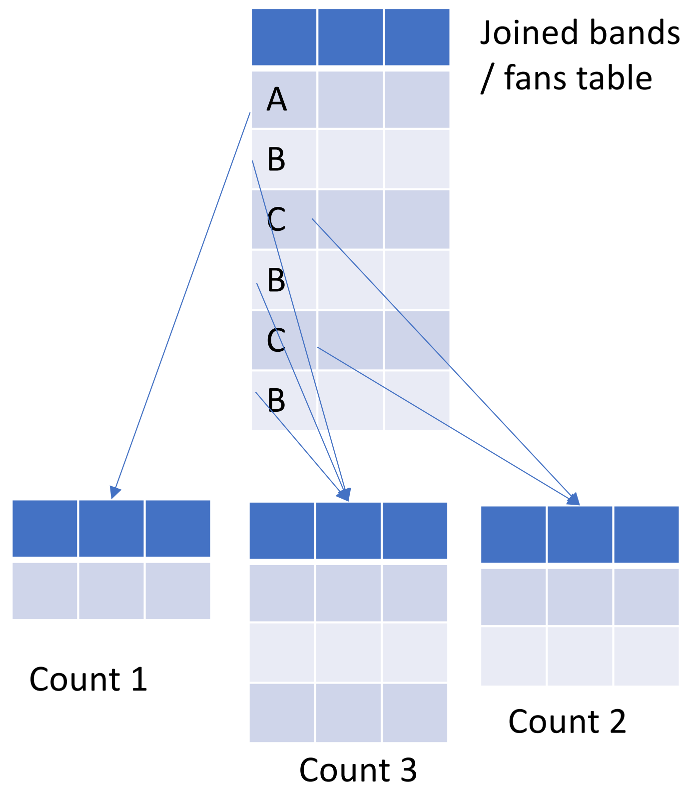
“Declarative” in the sense that the programmer doesn’t need to worry about this, or the specifics of how the join will be executed

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

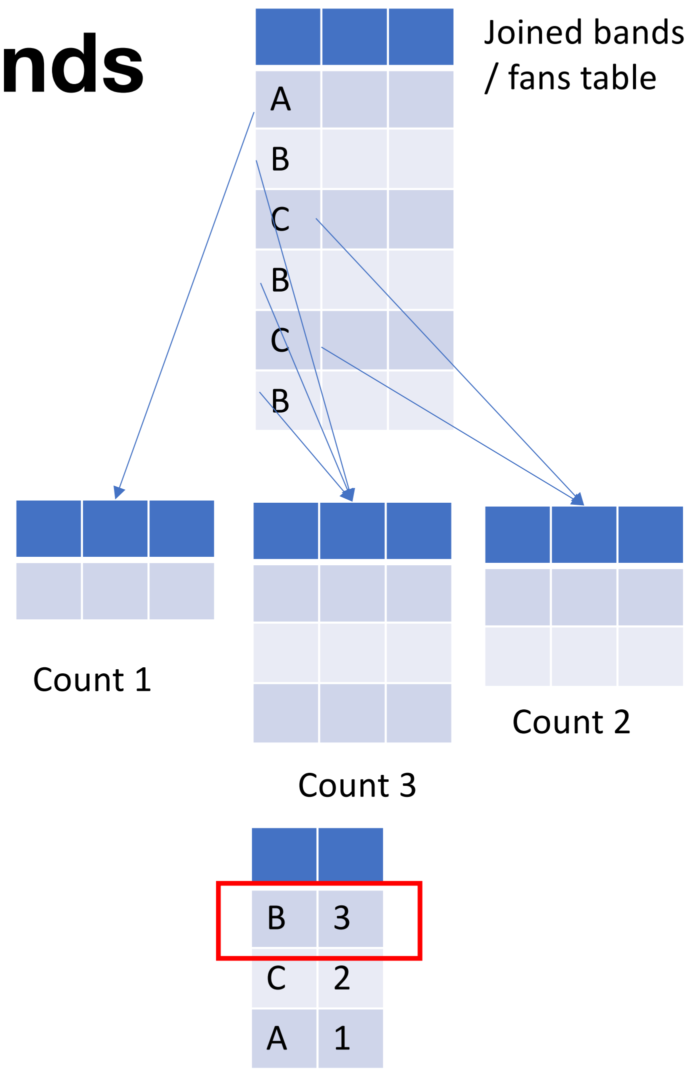
Partition the table by fan name



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)
```

Dates of ‘slipknot’ shows

```
SELECT date
FROM shows JOIN bands ON show_bandid  = bandid
WHERE name = ‘slipknot’
```

Alternately

```
SELECT date
FROM shows, bands
WHERE show_bandid  = bandid
AND name = ‘slipknot’
```

Bands: bandid, name, genre
Shows: showid, show_bandid, date, venue
Fans: fanid, name, birthday
BandFans: bf_bandid, bf_fanid

Aliases and Ambiguity

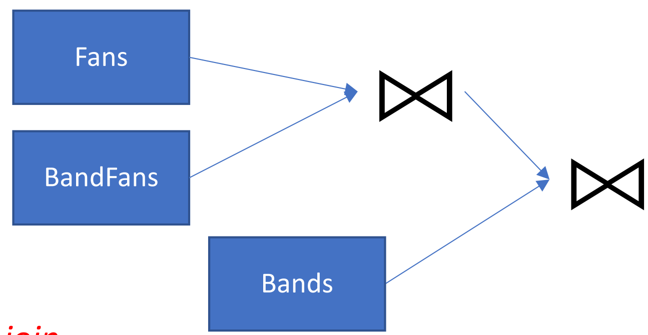
Bands: <u>bandid</u>, name, genre
Shows: <u>showid</u>, show_bandid, date, venue
Fans: <u>fanid</u>, name, birthday
BandFans: <u>bf_bandid</u>, <u>bf_fanid</u>

- Fans who like ‘slipknot’

SELECT name
FROM fans JOIN bandfans ON bf_fanid = fanid
JOIN bands on bf_bandid = bandid
WHERE name = ‘slipknot’

Unclear which “name” we are referring to

3 table join
(fans $\bowtie$ bandfans) $\bowtie$ bands



This doesn’t work. Why?

Aliases and Ambiguity

Bands: <u>bandid</u>, name, genre
Shows: <u>showid</u>, show_bandid, date, venue
Fans: <u>fanid</u>, name, birthday
BandFans: <u>bf_bandid</u>, <u>bf_fanid</u>

- Fans who like 'slipknot'
- Solution: disambiguate which table we are referring to

Declare 'f' and 'b' as aliases for fans and bands

SELECT ~~name~~ f.name
FROM fans f JOIN bandfans ON bf_fanid = fanid
JOIN bands b on bf_bandid = bandid
WHERE ~~name~~ b.name = 'slipknot'

Aggregation

- Find the number of fans of each band

```
SELECT bands.name,count(*)
FROM bands JOIN bandfans ON bandid=bf_bandid
GROUP BY bands.name
```

- What about bands with 0 fans?

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't join with any row in T2
  - For those rows, fields of T2 are NULL

<u>Example:</u>

SELECT bands.name, MAX(bf_fanid)

FROM bands LEFT JOIN bandfans

ON bandid=bf_bandid

GROUP BY bands.name

name	bandid
slipknot	1
limp bizkit	2
mariah carey	3

bf_bandid	bf_fanid
1	1
2	2
2	3

name	MAX
slipknot	1
limp bizkit	3
mariah carey	NULL

What about COUNT?

Can also use "RIGHT JOIN" and "OUTER JOIN" to get all rows of T2 or all rows of both T1 and T2

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't satisfy pred
  - For those rows, fields of T2 are NULL

Example:

SELECT bands.name, COUNT(*)

FROM bands LEFT JOIN bandfans

ON bandid=bf_bandid

GROUP BY bands.name

name	bandid
slipknot	1
limp bizkit	2
mariah carey	3

bf_bandid	bf_fanid
1	1
2	2
2	3

name	COUNT
slipknot	1
limp bizkit	2
mariah carey	1

Not what we wanted!

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't satisfy pred
  - For those rows, fields of T2 are NULL

Example:

SELECT bands.name, COUNT(bf_bandid)

FROM bands LEFT JOIN bandfans

ON bandid=bf_bandid

GROUP BY bands.name

name	bandid
slipknot	1
limp bizkit	2
mariah carey	3

bf_bandid	bf_fanid
1	1
2	2
2	3

name	COUNT
slipknot	1
limp bizkit	2
mariah carey	0

COUNT(*) counts all rows including NULLs, COUNT(col) only counts rows with non-null values in col

Self Joins

- Fans who like ‘slipknot’ and ‘limp bizkit’

```
SELECT f.name
FROM fans f JOIN bandfans ON bf_fanid = fanid
JOIN bands b on bf_bandid = bandid
WHERE b.name = 'slipknot' AND b.name = 'limp bizkit'
```

Doesn’t work!

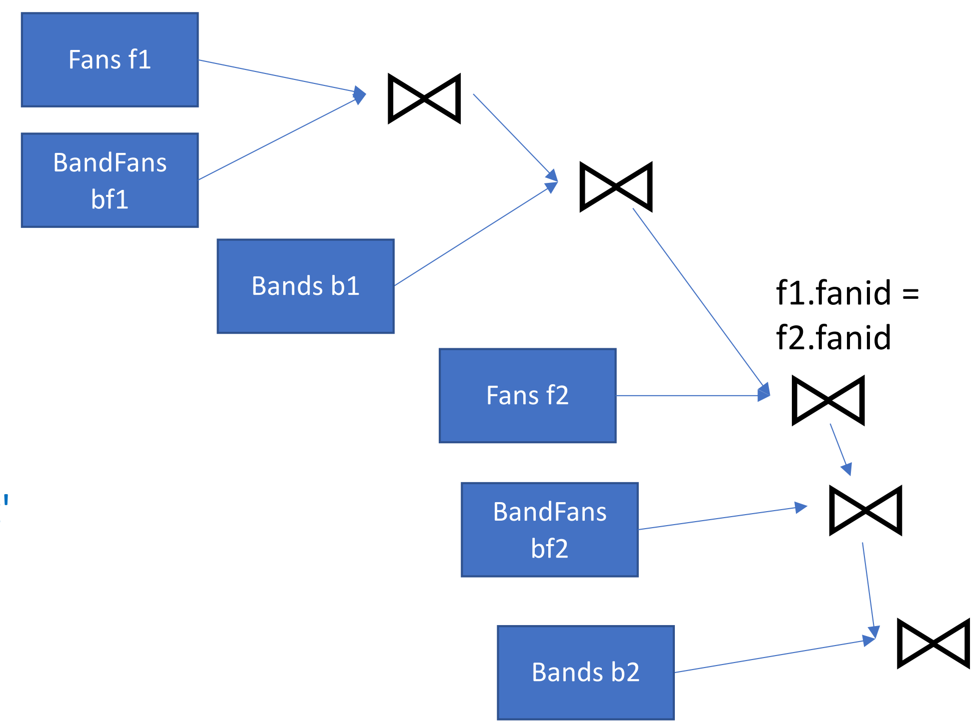
OR b.name = ‘limp bizkit’?

Also doesn’t work!

Self Joins

- Fans who like ‘slipknot’ and ‘limp bizkit’
- Need to build two tables, one of ‘slipknot’ fans and one of ‘limp bizkit’ fans, and intersect them

```
SELECT f1.name
FROM fans f1 JOIN bandfans bf1 ON bf_fanid = fanid
JOIN bands b1 on bf_bandid = bandid
JOIN fans f2 ON f1.fanid = f2.fanid
JOIN bandfans bf2 ON bf2.bf_fanid = f2.fanid
JOIN bands b2 ON b2.bandid = bf2.bf_bandid
WHERE b1.name = 'slipknot' AND b2.name = 'limp bizkit'
```



Nested Queries

```
SELECT fans1.name
FROM (
    SELECT fanid, f.name
    FROM fans f JOIN bandfans ON bf_fanid = fanid
    JOIN bands b ON bf_bandid = bandid
    WHERE b.name  = 'slipknot’) AS fans1,
JOIN (
    SELECT fanid, f.name
    FROM fans f JOIN bandfans ON bf_fanid = fanid
    JOIN bands b ON bf_bandid = bandid
    WHERE b.name  = 'limp bizkit’) AS fans2
 ON fans1.fanid = fans2.fanid
```

Every query is a relation (table)

Generally anywhere you can use a table, you can use a query!

Simplify with Common Table Expressions (CTEs)

```
WITH fans1 AS
    (SELECT fanid, f.name
    FROM fans f JOIN bandfans ON bf_fanid = fanid
    JOIN bands b ON bf_bandid = bandid
    WHERE b.name  = 'slipknot'),
fans2 AS
    (SELECT fanid, f.name
    FROM fans f JOIN bandfans ON bf_fanid = fanid
    JOIN bands b ON bf_bandid = bandid
    WHERE b.name  = 'limp bizkit')
SELECT fans1.name
FROM fans1 JOIN fans2 ON fans1.fanid = fans2.fanid
```

CTEs work better than nested expressions when the CTE needs to be referenced in multiple places

Study Break

- Write a SQL query to find all the bands who have fans who are fans of 'limp bizkit'
  - I.e.:
    - Mary is a fan of limp bizkit and korn
    - Tim is a fan of creed and justin Bieber
    - Sam is a fan of limp bizkit and nickelback
    - Janelle is a fan of nickelback and slipknot

Should return korn and nickelback

Bands: bandid, name, genre
Shows: showid, show_bandid, date, venue
Fans: fanid, name, birthday
BandFans: bf_bandid, bf_fanid

```sql
WITH lb_fans AS
( SELECT bf_fanid fanid
  FROM bandfans
  JOIN bands ON bandid = bf_bandid
  WHERE bands.name = 'limp bizkit'
)
SELECT bands.name
FROM bandfans
JOIN lb_fans ON bf_fanid = fanid
JOIN bands ON bf_bandid = bandid
```

fanid	name
1	mary
2	tim
3	sam
4	janelle

fanid
1
3

lb_fans

Need to eliminate duplicates
Filter out limp bizkit

bf_bandid	bf_fanid
2	1
3	1
5	2
6	2
2	3
4	3
1	4
4	4

bands
limp bizkit
korn
limp bizkit
nickelback

bandid	name
1	slipknot
2	limp bizkit
3	korn
4	nickelback
5	creed
6	Justin bieber

Solution

```
WITH lb_fans AS
( SELECT bf_fanid fanid
  FROM bandfans
  JOIN bands ON bandid = bf_bandid
  WHERE bands.name = 'limp bizkit'
)
SELECT DISTINCT bands.name
FROM bandfans
JOIN lb_fans ON bf_fanid = fanid
JOIN bands ON bf_bandid = bandid
WHERE bands.name != 'limp bizkit'
```

Take a Break

Database Tuning Primer

- Sometimes queries don't run as fast as you would like
- Need to "tune" the database to run faster

- Unlike SQL, most tuning is very specific to the database you are using
  - Many different databases out there, e.g., MySQL, Postgres, Oracle, SQLite, SQLServer (aka AzureDB), Redshift, Snowflake, etc

- Before we explore some of the most common ways to tune, let's understand why a query may be slow

If you want to understand this in more detail, take 6.814/6.830!

Analytics vs Transactions

- **Analytics** is more typical of data science
  - E.g., dashboards or ad-hoc queries looking at trends and aggregates
  - Queries often read a significant amount of data (> 1% of DB?)
  - Updates are infrequent / batch
  - Focus is on minimizing the amount of data we need to read, and ensuring sufficient memory/resources for expensive operations like sorts & joins

Focus in this class

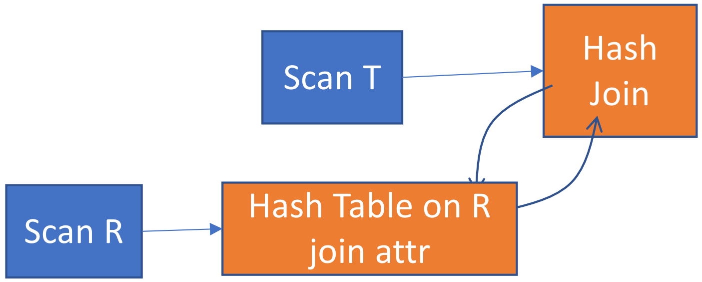
- **Transactions** are common in websites, other online applications
  - Create, Read, Update, Delete (CRUD) workload
  - Less complex queries (often "key/value" is sufficient)
  - Requires mechanisms to prevent concurrent updates to same data
  - Focus is on eliminating contention in these mechanisms, ensuring queries are indexed

Where Does Time Go?

- In analytics applications, CPU + I/O dominate
- CPU: instructions to compute results
  - Most typically the time to join tables
- I/O: transferring data from disk
  - Most typically reading data from tables or moving data to / from memory when results don't fit into RAM

Example

- Joining a 1 GB table T to a 100 MB table R
- 10 Bytes / record (so T = 100M records, R = 10M records)
- System can process 100M records / sec
- Disk can read 100 MB/sec
- 200 MB of memory



- Executing join:
  - Load R into a hash table (1 sec I/O + 0.1 sec to process 10M records)
  - Scan through T, looking up each record in hash table (10 sec I/O, + 1 sec to process 100M records)
  - Total time 12.1 sec

Tuning Goal

- Reduce the number of and size of records read and processed
- Ensure that we have sufficient memory for joins and other operations
  - If neither join result can fit into memory system falls back on much slower implementations that shuffle data to / from disk
  - Surprisingly, database systems still answer queries when tables are larger than memory!
    - Fall back on “external” implementations

How Can We Make This Faster?

- Goal: Reduce amount of data read
- What about just scanning bands rows that correspond to ‘limp bizkit’?
  - Index on bands.name
- Could we just scan the bandfans rows that correspond to ‘limp bizkit’?
  - Index on bandfans.bandid

Creating An Index

- CREATE INDEX band_name ON bands(name);
- CREATE INDEX bf_index ON bandfans(bf_bandid);

B-Tree Index Example (B=2)

”Heap File”
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

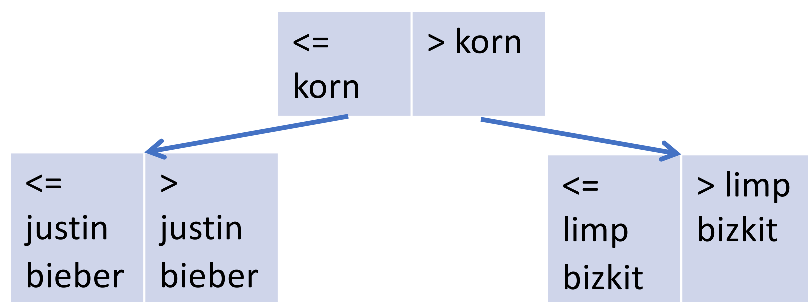
B-Tree Index Example (B=2)

<= korn	> korn

"Heap File"
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

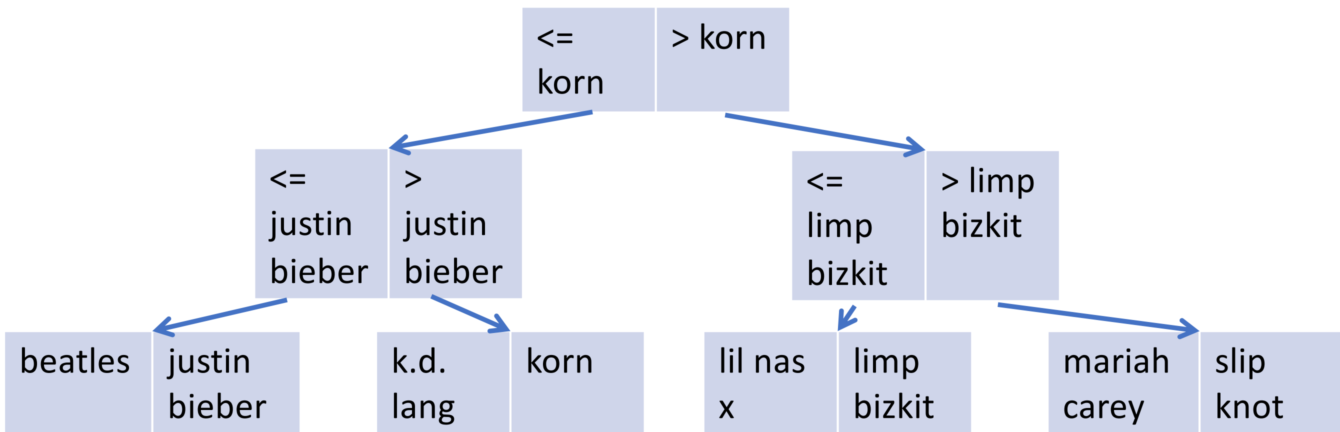
B-Tree Index Example (B=2)



”Heap File”
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

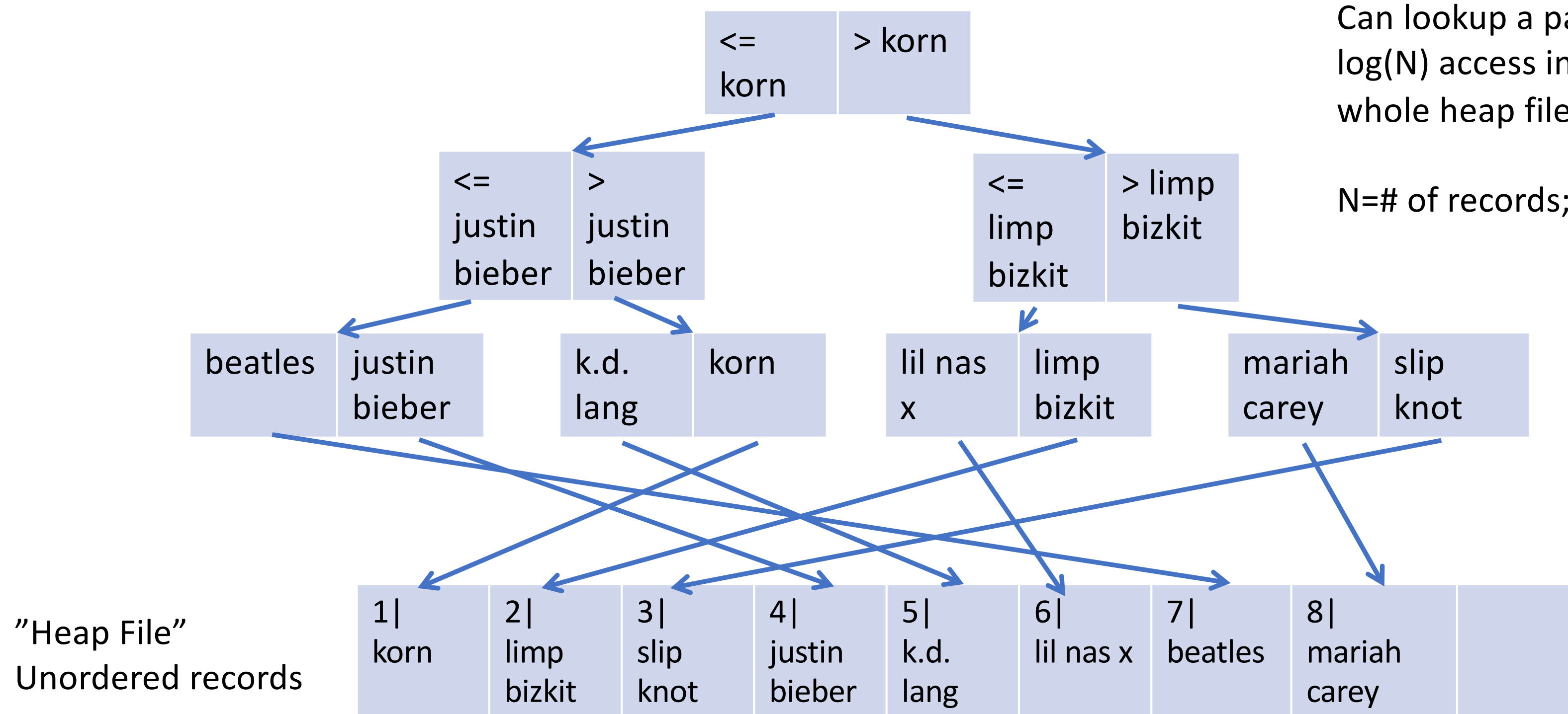
B-Tree Index Example (B=2)



"Heap File"
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

B-Tree Index Example (B=2)

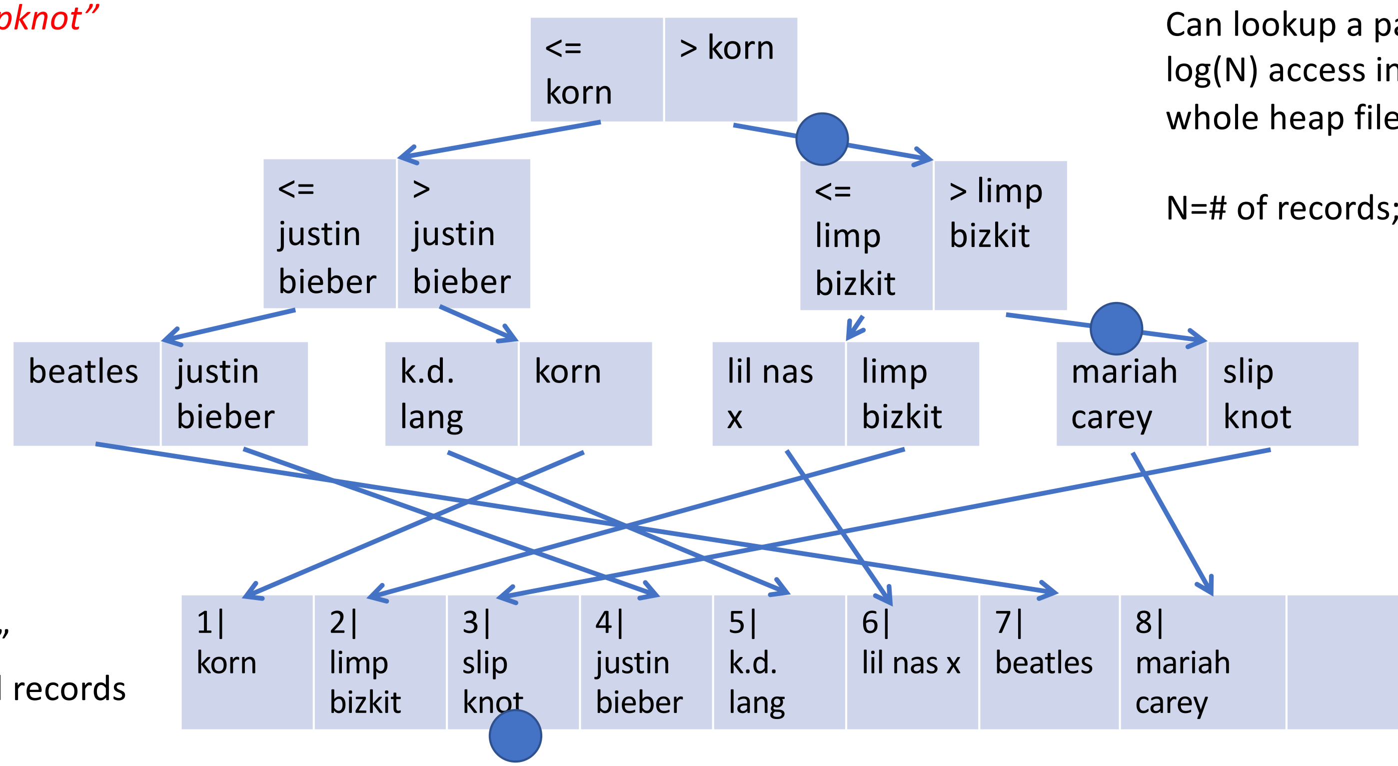


Can lookup a particular record in log(N) access instead of scanning whole heap file

N=# of records; base of log is B

B-Tree Index Example (B=2)

Find "slipknot"



"Heap File"
Unordered records

Can lookup a particular record in log(N) access instead of scanning whole heap file

N=# of records; base of log is B

Pros and Cons of Indexing

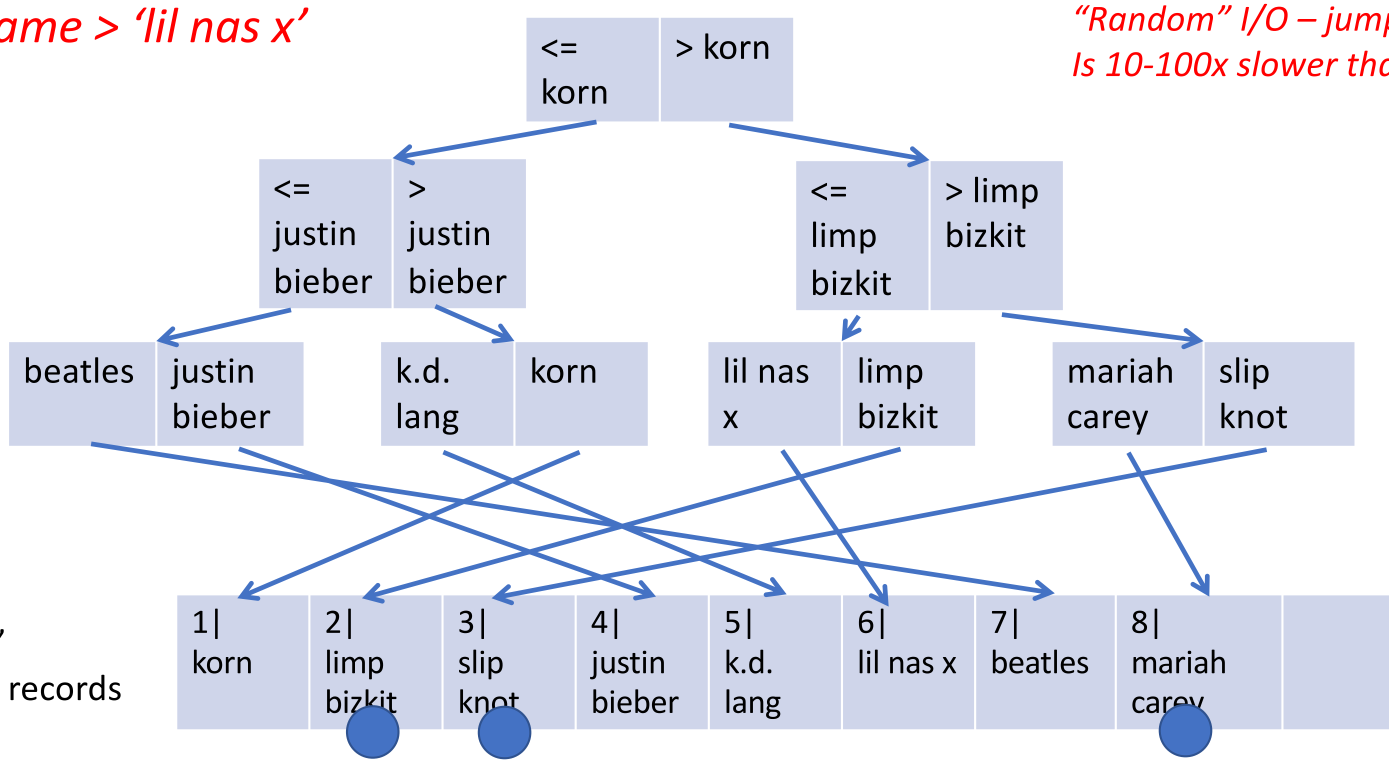
- Pros:
  - Reduces time to lookup specific records

- Cons:
  - Uses space
  - Increases insert time
  - If heap file isn't ordered on index, may not speed up I/O

B-Tree Index Example (B=2)

Find name > ‘lil nas x’

“Random” I/O – jumping around on disk
Is 10-100x slower than reading in order



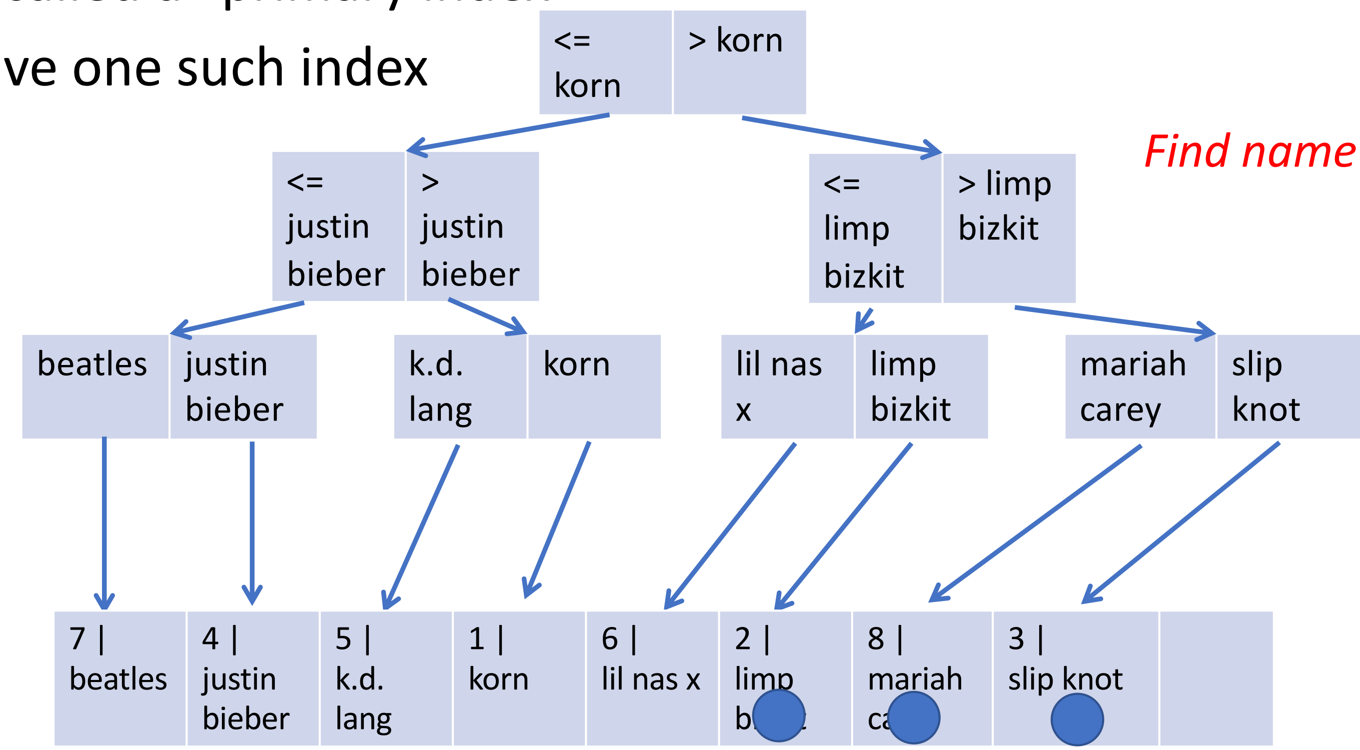
”Heap File”
Unordered records

“Clustering” a B-Tree

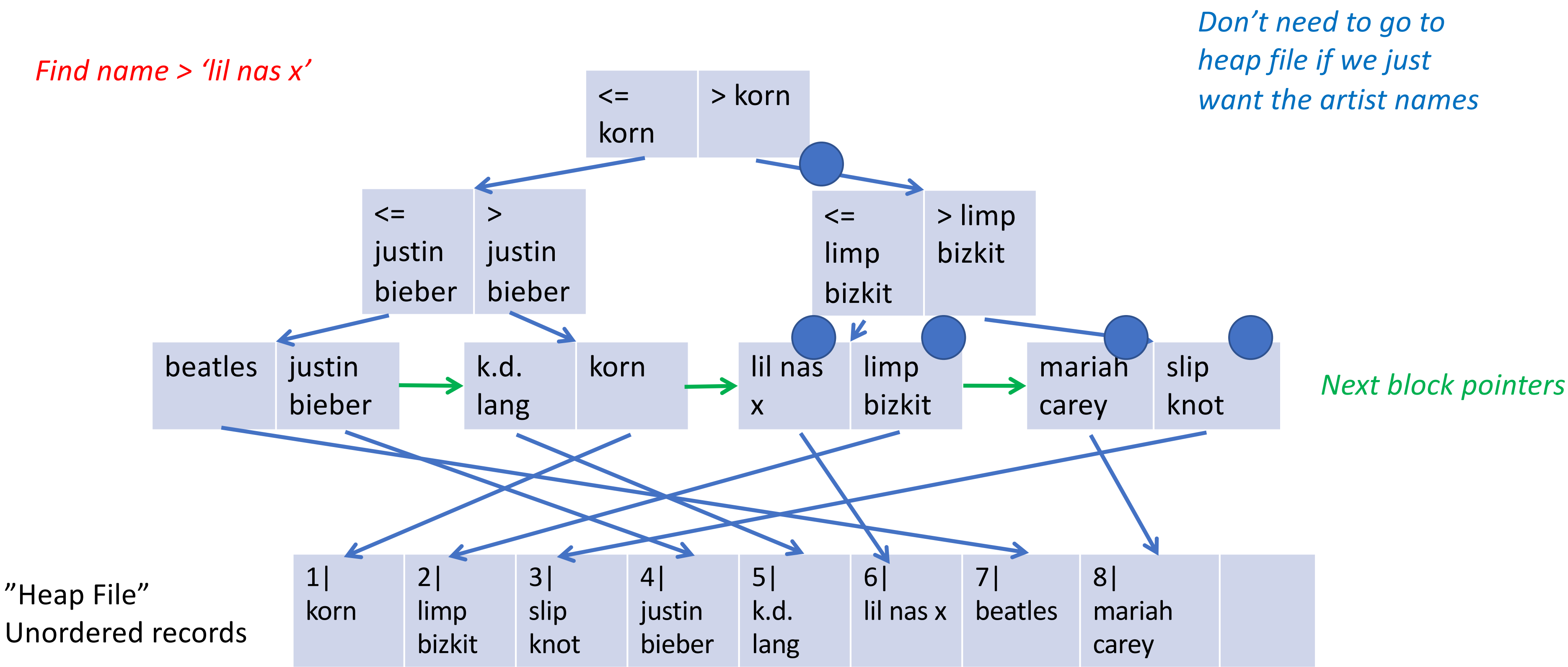
- Records are in order of index
- Alternately called a “primary index”
- Can only have one such index

How this is done is DB specific.

Find name > ‘lil nas x’



Index-Only Scans



Postgres

create index bf_index on bandfans(bf_bandid);

EXPLAIN SELECT count(*)
FROM bandfans JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'

```
Aggregate  (cost=2162.44..2162.45 rows=1 width=8)
  ->  Nested Loop  (cost=0.42..2162.36 rows=30 width=0)
        ->  Seq Scan on bands  (cost=0.00..1918.84 rows=3 width=4)
              Filter: ((name)::text = 'limp bizkit'::text)
        ->  Index Only Scan using bf_index on bandfans  (cost=0.42..56.17 rows=2500 width=4)
              Index Cond: (bf_bandid = bands.bandid)
```

Find limp bizkit record by scanning bands

Postgres

create index bf_index on bandfans(bf_bandid);

Estimated cost 2000 vs 12000
Actual 8ms vs 80ms

EXPLAIN SELECT count(*)
FROM bandfans JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'

```
Aggregate  (cost=2162.44..2162.45 rows=1 width=8)
  ->  Nested Loop  (cost=0.42..2162.36 rows=30 width=0)
        ->  Seq Scan on bands  (cost=0.00..1918.84 rows=3 width=4)
              Filter: ((name)::text = 'limp bizkit'::text)
        ->  Index Only Scan using bf_index on bandfans  (cost=0.42..56.17 rows=2500 width=4)
              Index Cond: (bf_bandid = bands.bandid)
```

For each limp bizkit record (3 estimated)

Do an index only scan to count the number of fans

Postgres

create index bf_index on bandfans(bf_bandid);
create index band_name on bands(name);

EXPLAIN SELECT count(*)
FROM bandfans JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'

Estimated cost 260 vs 2000 vs 12000
Actual .5 ms vs 8 ms vs 80 ms

160x speedup!

```
Aggregate  (cost=259.94..259.95 rows=1 width=8)
  ->  Nested Loop  (cost=0.72..259.87 rows=30 width=0)
        ->  Index Scan using band_name on bands  (cost=0.29..16.34 rows=3 width=4)
              Index Cond: ((name)::text = 'limp bizkit'::text)
        ->  Index Only Scan using bf_index on bandfans  (cost=0.42..56.17 rows=2500 width=4)
              Index Cond: (bf_bandid = bands.bandid)
```

Use index to directly lookup 'limp bizket'

Today's Reading

- Critique of SQL
- Some specific complaints about, e.g.,
  - json and windowing support
  - Verbose join syntax
  - Pitfalls around, e.g., subqueries
- More generally:
  - Lack of standards for extensions, e.g., new types or procedural support
  - New features, e.g., json and windows, are added via new syntax, rather than libraries as in most languages
    - Massive spec, very complex to support, huge burden on developers

Recap: Some Common Data Access Themes

- SQL provides a powerful set-oriented way to get the data you want
- Joins are the crux of data access and primary performance concern
- To speed up queries, “read what you need”
  - Indexing & Index-only Scans
  - Predicate pushdown
    - E.g., using an index to find ‘limp bizkit’ records
  - Column-orientation
    - More on this later – we can physically organize data to avoid reading parts of records we don’t need