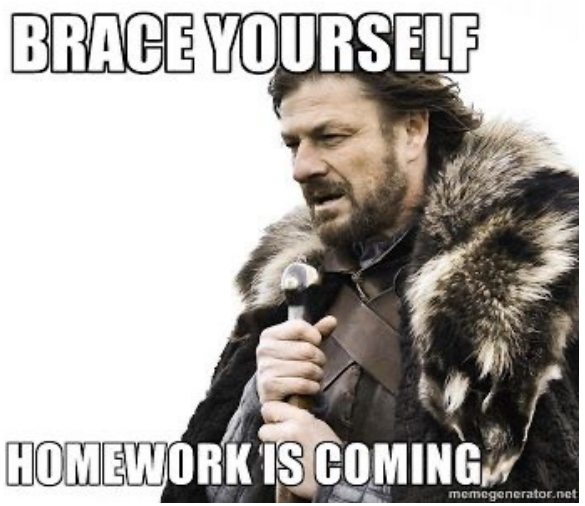
Piazza signup:
http://piazza.com/mit/spring2022/6s079

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

# 6.S079
# Lecture 3

Sam Madden




**Key ideas:**
More SQL
Indexes and performance tuning

**Lab 0 Due**
**Lab 1 Next Week**

# Recap: SQL Syntax and Joins

- Bands schema
  - **Bands**: bandid, name, genre
  - **Shows**: showid, show_bandid REFERENCES bands.bid, date, venue
  - **Fans**: fanid, name, birthday
  - **BandFans**: bf_bandid REFERENCES bands.bandid, bf_fanid REFERENCES fans.fanid

# 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

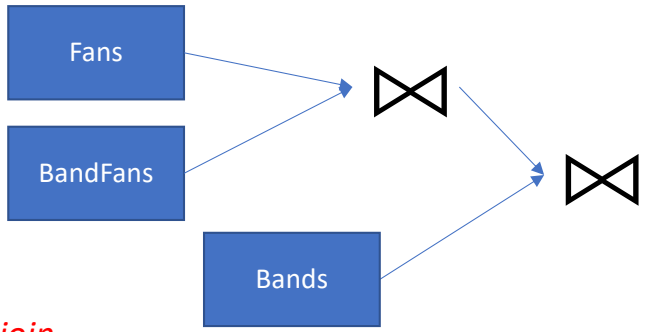
# Aliases and Ambiguity

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

- Fans who like ‘slipknot’

*Unclear which “name” we are referring to*

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




*3 table join*
*(fans ⋈ bandfans) ⋈ bands*

*This doesn’t work. Why?*

# Aliases and Ambiguity

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

- 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'

# Clicker / SQL Comprehension

- Fill in the blank to complete this query to "find shows by slipknot after Jan 1 2022"? (Assume syntax for dates is correct)

SELECT date, venue FROM _____ WHERE name = 'slipknot' AND date > '1/1/2022'

1. show, bands
2. shows JOIN bands ON showid = show_bandid
3. shows JOIN bands ON bandid = show_bandid
4. shows JOIN bands ON bandid = showid

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

# 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

Example:

```
SELECT bands.name, MAX(bf_fanid)
FROM bands LEFT JOIN bandfans
ON bandid=bf_bandid
GROUP BY bands.name
```

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

| 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?*

# 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!*

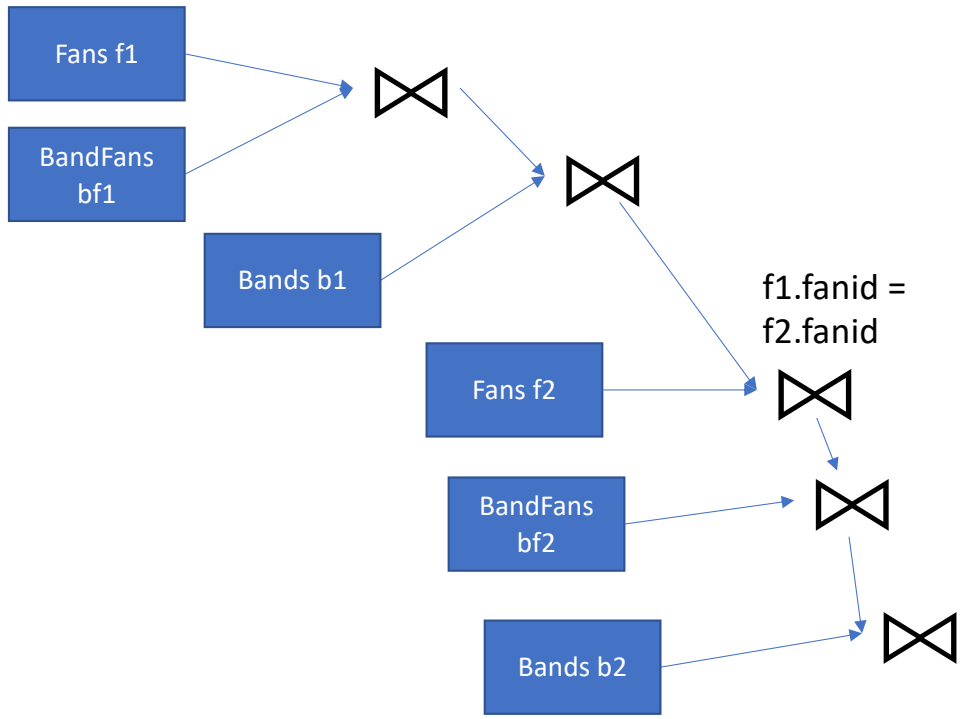
OR b.name = ‘limp bizkit’?

*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

```
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 |

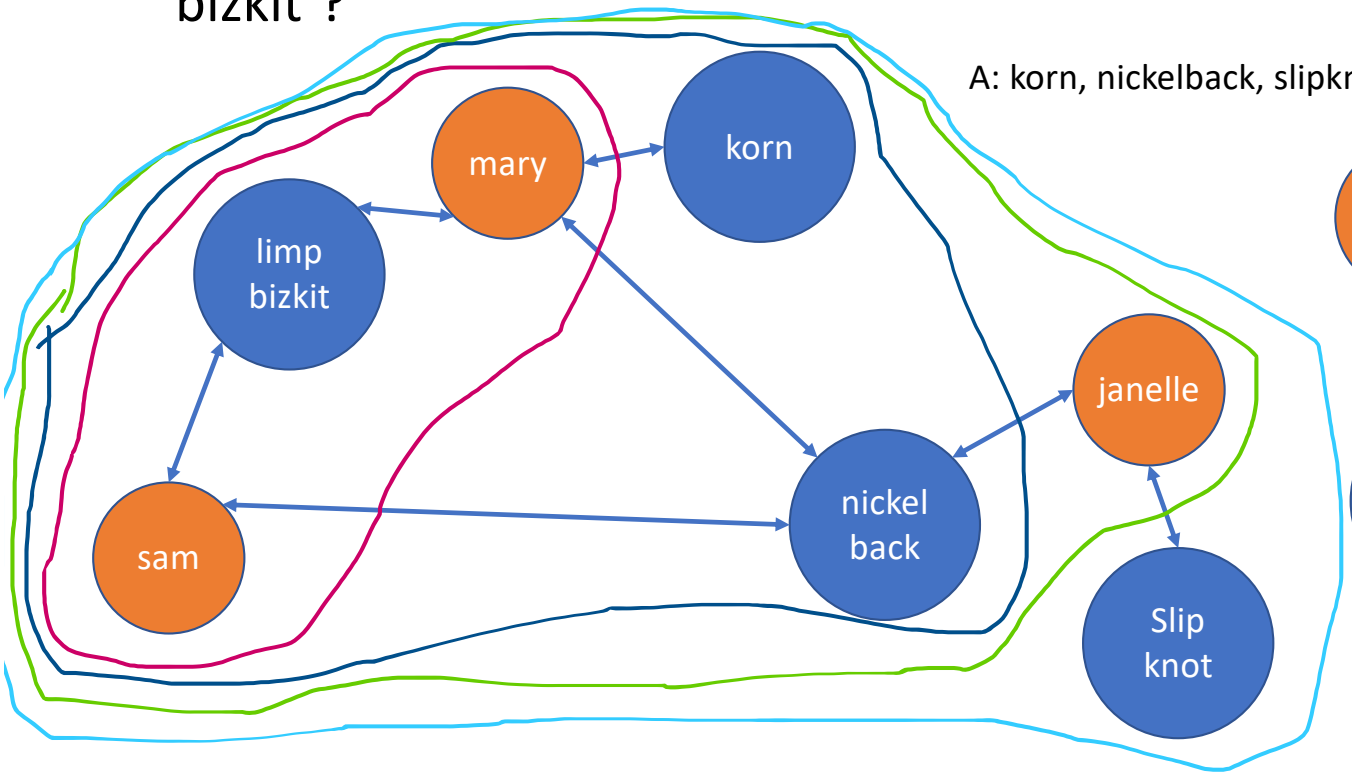
| bandid | name |
|---|---|
| 1 | slipknot |
| 2 | limp bizkit |
| 3 | korn |
| 4 | nickelback |
| 5 | creed |
| 6 | Justin bieber |

# Solution

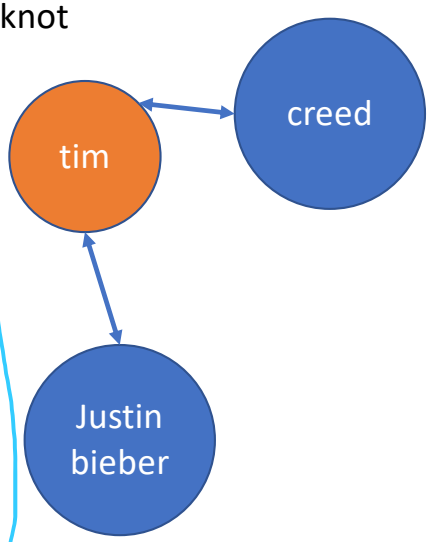
```
WITH lb_fans AS
( SELECT bf_fanid fanid
   FROM bandfans
   JOIN bands ON bandid = bf_bandid
   WHERE bands.name = 'limp bizkit'
)
SELECT DISTINCT bands.name
FROM bandfans
JOIN lb_fans ON bf_fanid = fanid
JOIN bands ON bf_bandid = bandid
WHERE bands.name != 'limp bizkit'
```

# Recursive Queries

- Suppose we want to find all bands connected to a fan who likes ‘limp bizkit’?




A: korn, nickelback, slipknot

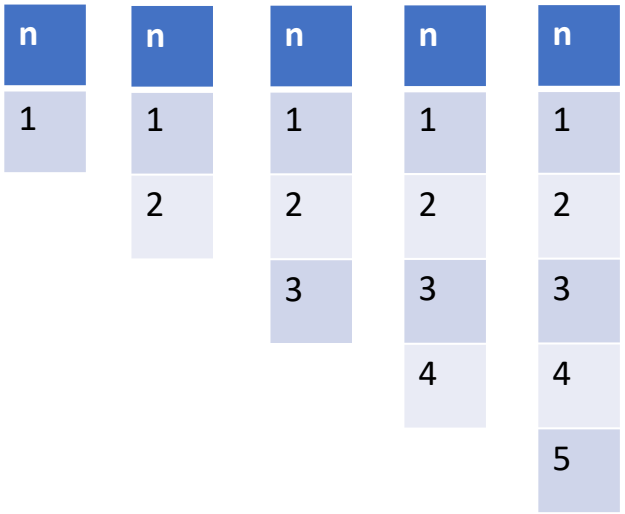



*Challenge: each successive join follows one set of edges. Size of graph is unbounded!*

# Recursive Queries

- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

```
WITH RECURSIVE t(n) AS
( VALUES (1)
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```



# Recursive Queries

- Suppose we want to find all bands connected to a fan who likes ‘limp bizkit’?

```
WITH recursive lb_fan_bands as (
   SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid)
)
SELECT distinct name FROM lb_fan_bands
JOIN bands ON bf_bandid = bandid
WHERE name != 'limp bizkit'
```

*Tricky – add new fans of bands we already found and new bands liked by fans we already found*

# Recursion Example

- Limp bizkit is band 2

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

Base case

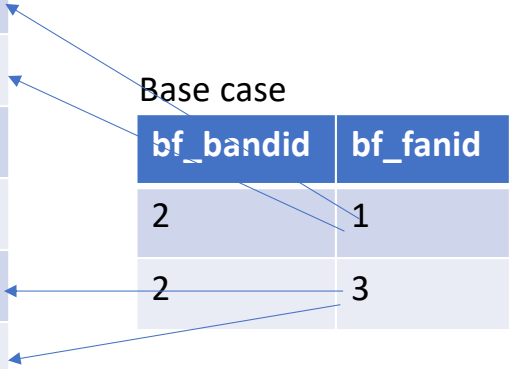
| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

# Recursion Example

- Limp bizkit is band 2

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



Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

Iter 1

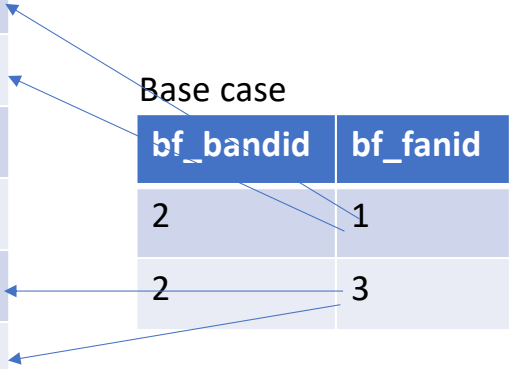
| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

# Recursion Example

- Limp bizkit is band 2

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



Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

Iter 1

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

# Recursion Example

- Limp bizkit is band 2

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

*Base case already got these rows*

Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

Iter 1

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

# Recursion Example

- Limp bizkit is band 2

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

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

*Iter 1 already got these fans*

Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

Iter 1

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

Iter 2

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

# Recursion Example

- Limp bizkit is band 2

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

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

Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

Iter 1

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

Iter 2

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

*One new fan found*

# Recursion Example

- Limp bizkit is band 2

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

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

Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

Iter 1

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |

Iter 2

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |
| 4 | 4 |

# Recursion Example

- Limp bizkit is band 2

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

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

Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

*One new band found*

Iter 2

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |
| 4 | 4 |

Iter 3

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |
| 4 | 4 |

# Recursion Example

- Limp bizkit is band 2

```
WITH recursive lb_fan_bands as (
SELECT bf_fanid, bf_bandid
   FROM bandfans
   JOIN bands on bf_bandid = bandid
   WHERE bands.name = 'limp bizkit'
   UNION
   SELECT bandfans.bf_fanid, bandfans.bf_bandid
   FROM bandfans JOIN lb_fan_bands
   ON (lb_fan_bands.bf_fanid = bandfans.bf_fanid
      OR lb_fan_bands.bf_bandid = bandfans.bf_bandid))
```

*At this point all bands have been found! Recursion stops when no new records found.*

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

Base case

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |

Iter 2

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |
| 4 | 4 |

Iter 3

| bf_bandid | bf_fanid |
|---|---|
| 2 | 1 |
| 2 | 3 |
| 3 | 1 |
| 4 | 3 |
| 4 | 4 |
| 1 | 4 |

# 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*

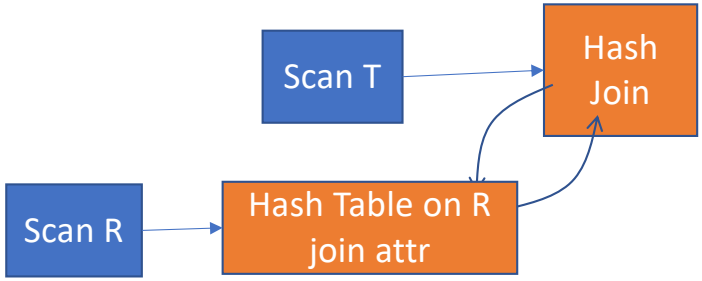
- **Transactions** are common in websites, other online applications
  - Create, Read, Update, Delete (CRUD) workload
  - Less complex queries (often “key/value” is sufficient)
  - Requires mechanisms to prevent concurrent updates to same data
  - Focus is on eliminating contention in these mechanisms,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

# Bandfans example

- Created a larger fake version of bandfans
  - 1M likes
  - 800 fans
  - 100K bands

# Understanding Database Plans

- Most database systems provide an "explain" command that shows how it executes a query

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

*This query takes 80ms to execute*
*Not slow, but this isn't a large DB, and could be painful if we have to run many times*

Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

*Parse tree*
*Read bottom up*

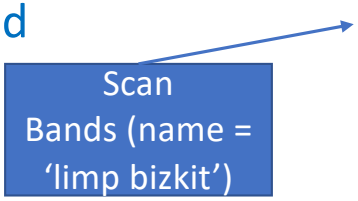
# Understanding Database Plans

- Most database systems provide an “explain” command that shows how it executes a query

  **EXPLAIN** SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

  WHERE name = 'limp bizkit'




Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

# Understanding Database Plans

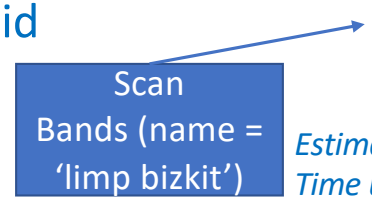
- Most database systems provide an “explain” command that shows how it executes a query

  **EXPLAIN** SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

  WHERE name = 'limp bizkit'

Example: POSTGRES




*Estimated time and number of rows*
*Time units are arbitrary*
*Two numbers: time to produce 1st record*
*.. time to produce last record*

*Here time is a combination of CPU + I/O*

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

# Understanding Database Plans

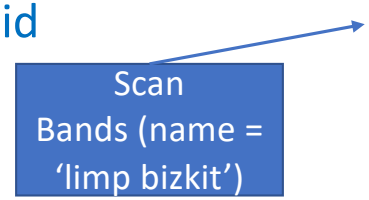
- Most database systems provide an “explain” command that shows how it executes a query

  **EXPLAIN** SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

  WHERE name = 'limp bizkit'

Example: POSTGRES




```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

*Most expensive steps*

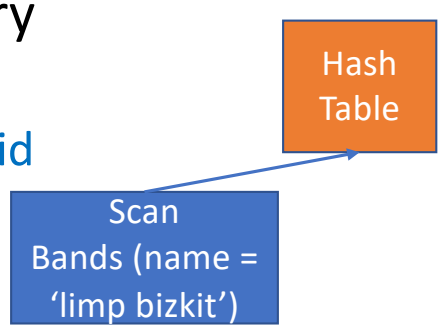
# Understanding Database Plans

- Most database systems provide an “explain” command that shows how it executes a query

  **EXPLAIN** SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

  WHERE name = 'limp bizkit'




Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

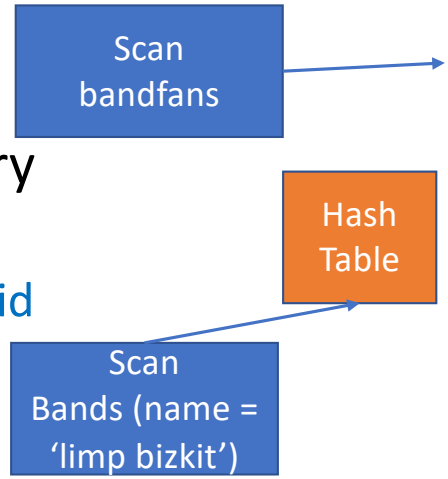
# Understanding Database Plans

- Most database systems provide an “explain” command that shows how it executes a query

  **EXPLAIN** SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

  WHERE name = 'limp bizkit'




Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

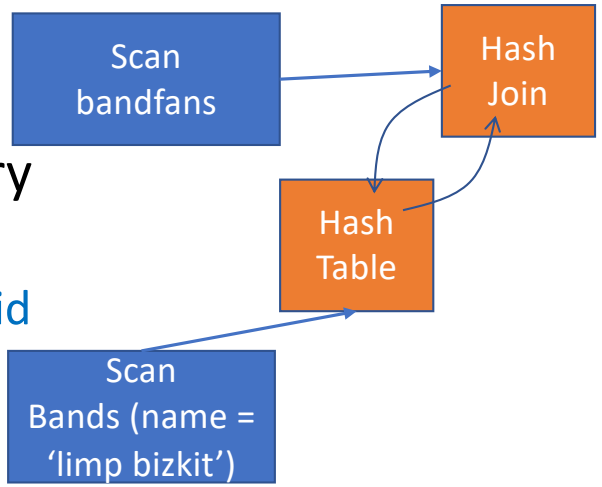
# Understanding Database Plans

- Most database systems provide an “explain” command that shows how it executes a query

  **EXPLAIN** SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

  WHERE name = 'limp bizkit'




Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

# 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 | |
|---|---|---|---|---|---|---|---|---|

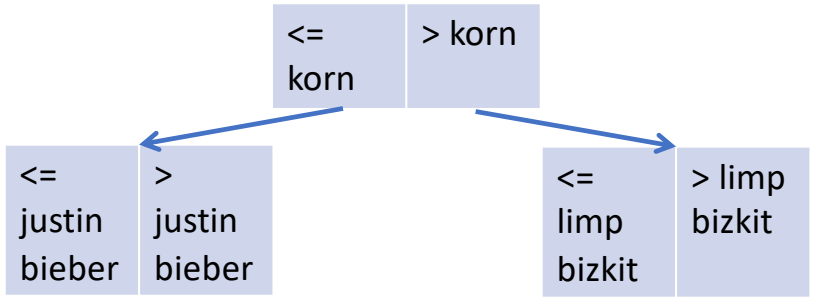
# B-Tree Index Example (B=2)

| <= 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 | |
|---|---|---|---|---|---|---|---|---|

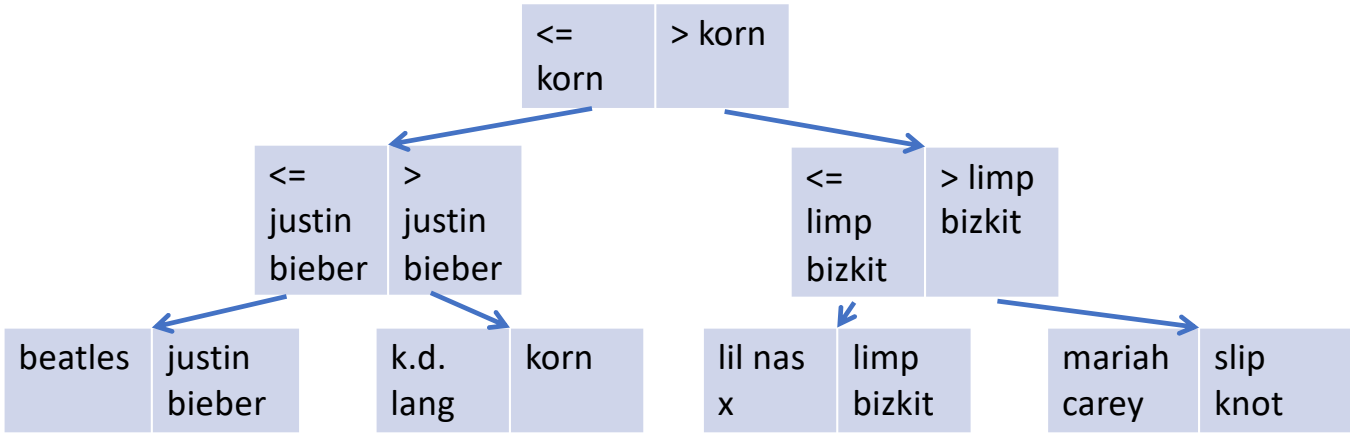
# B-Tree Index Example (B=2)




”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 | |

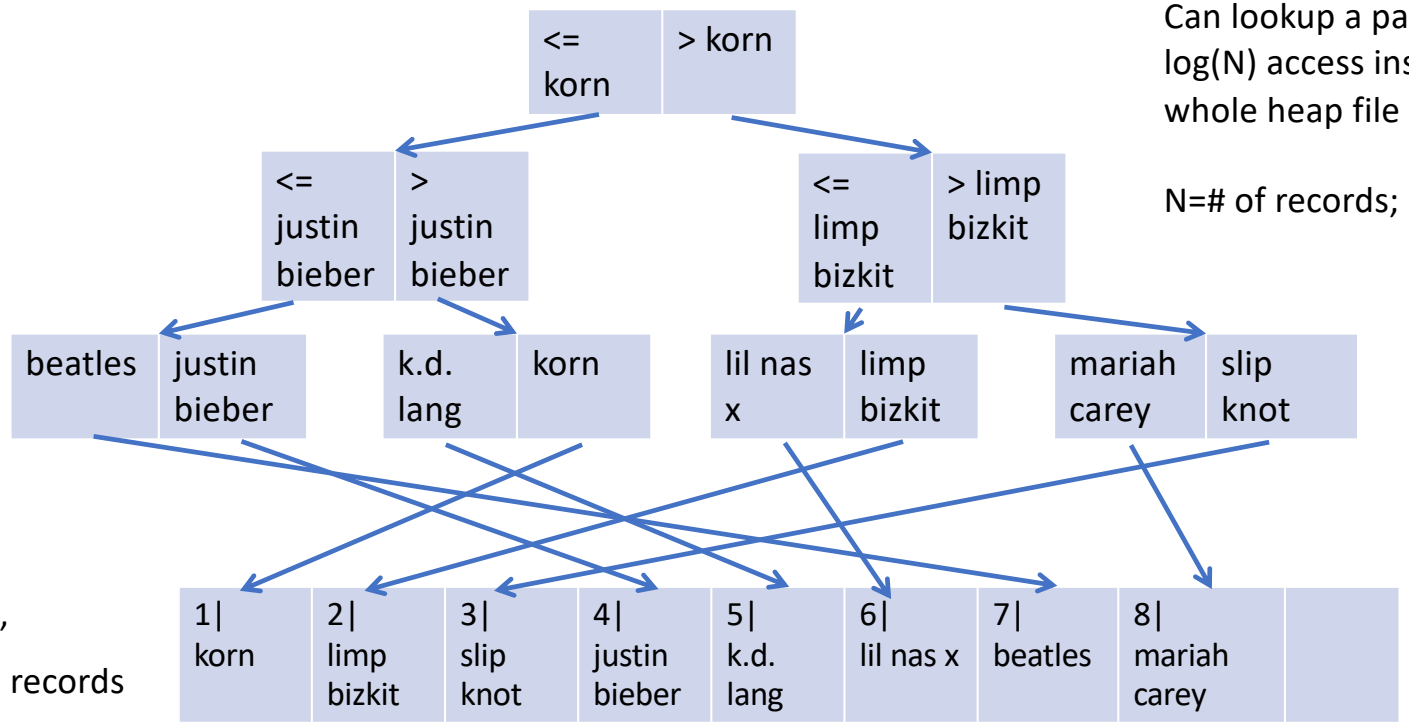
# B-Tree Index Example (B=2)




”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)



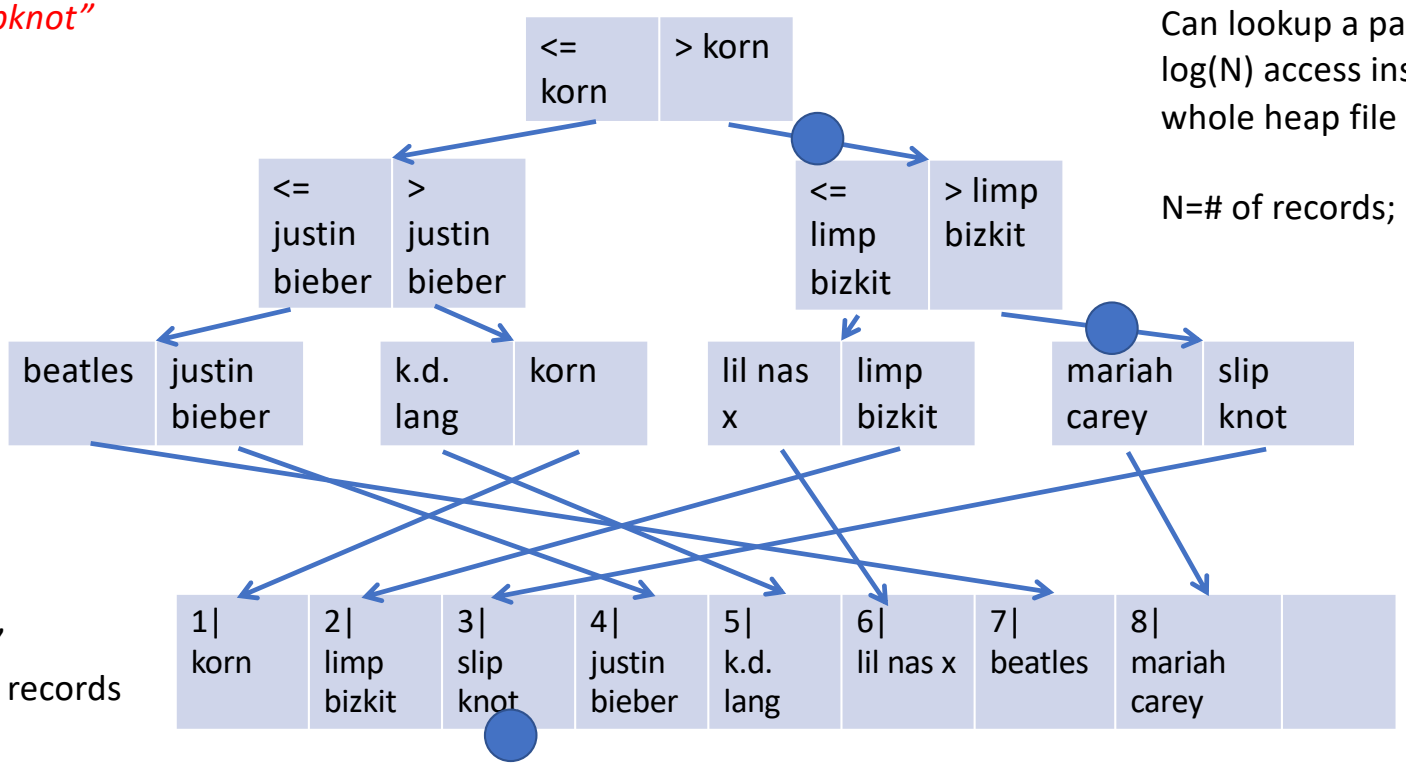

Can lookup a particular record in log(N) access instead of scanning whole heap file

N=# of records; base of log is B

”Heap File”
Unordered records

# B-Tree Index Example (B=2)

*Find "slipknot"*




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

N=# of records; base of log is B

"Heap File"
Unordered records

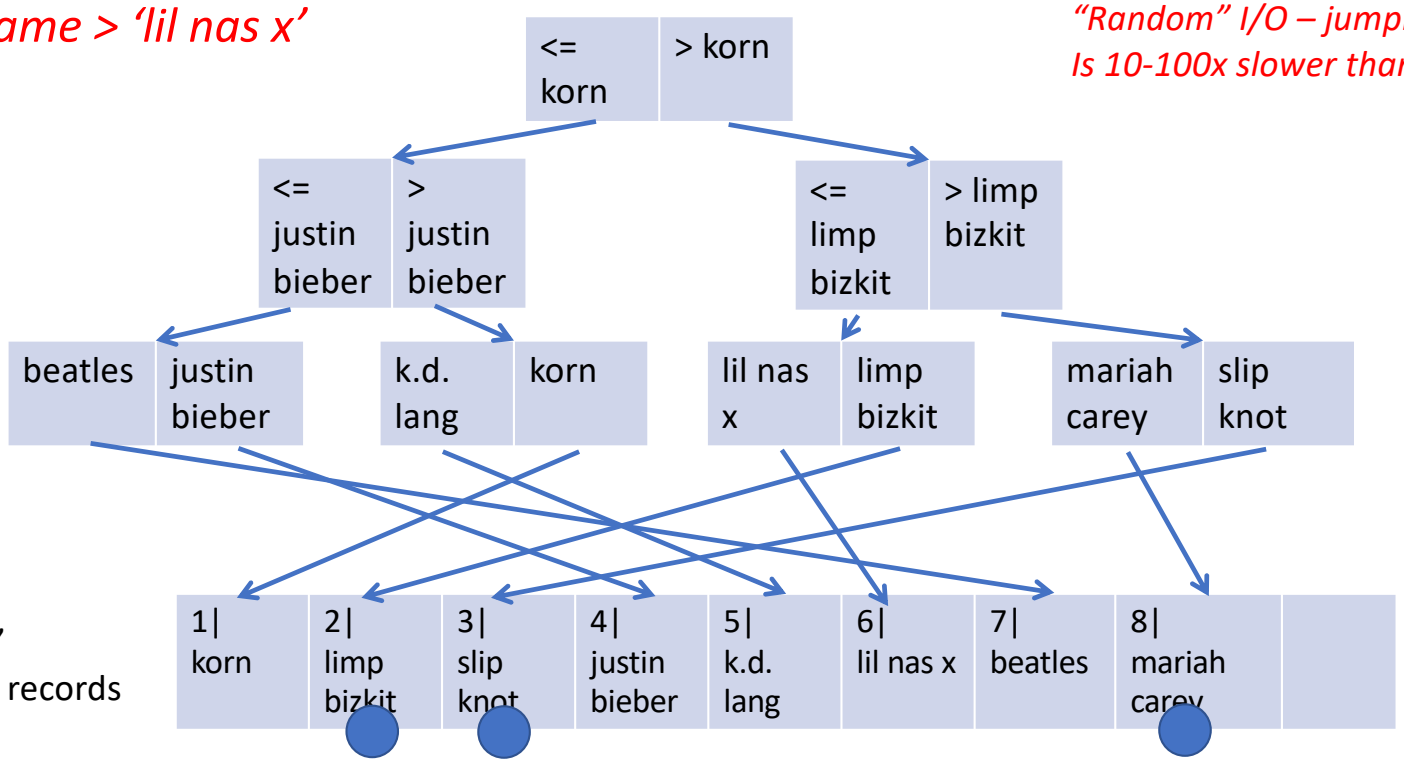
# Pros and Cons of Indexing

- 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*

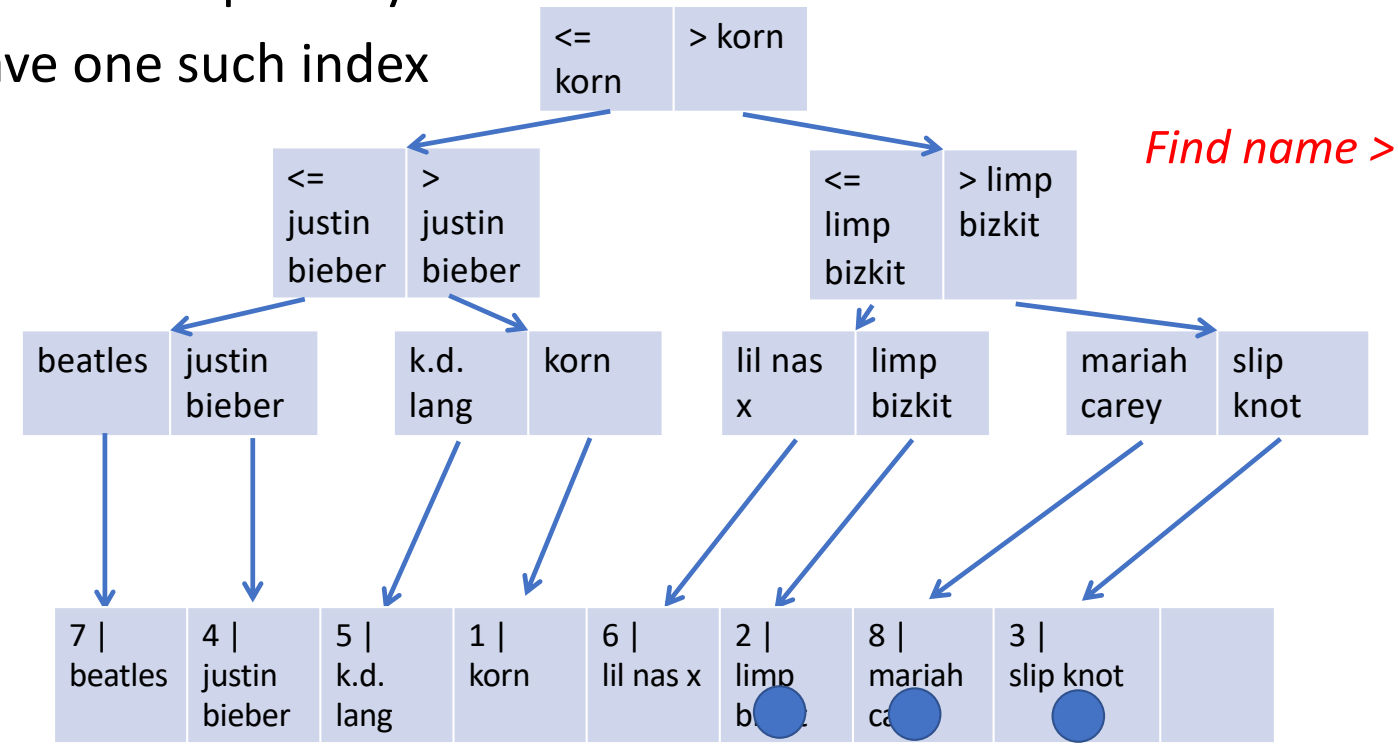



"Heap File"
Unordered records

# “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.

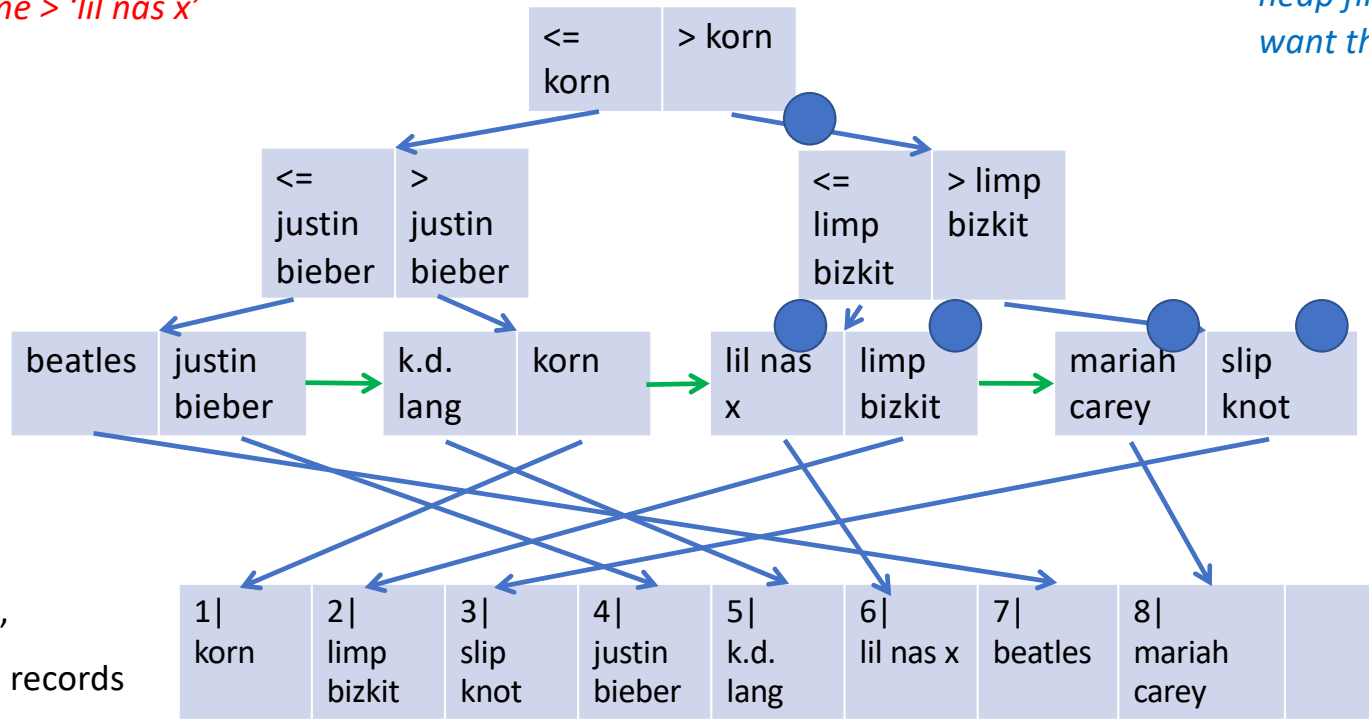



*Find name > ‘lil nas x’*

# Index-Only Scans

*Find name > 'lil nas x'*

*Don't need to go to heap file if we just want the artist names*




*Next block pointers*

"Heap File"
Unordered records

# 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!*

*Use index to directly lookup ‘limp bizket’*

```
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)
```

# 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

# Next Time

- Pandas / Python
- When to use SQL vs Python